```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Licensed under the Apache License, Version 2.0 (the "License");

Image captioning with visual attention

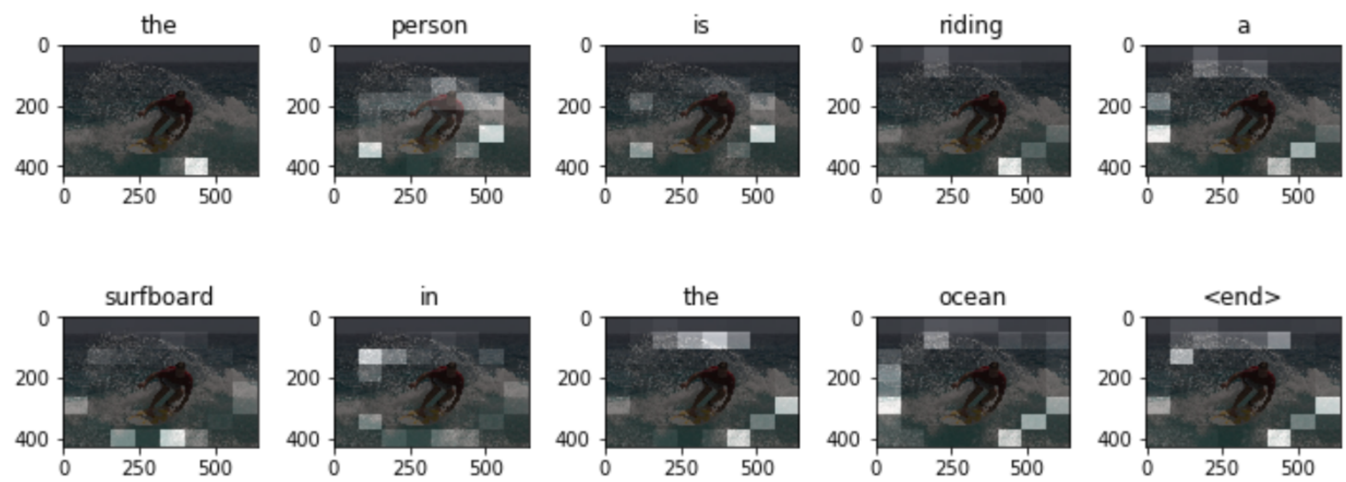
[View on TensorFlow.org] [Run in Google Colab] [View source on GitHub] [Download notebook]

Given an image like the example below, your goal is to generate a caption such as "a surfer riding on a wave".

Image Source; License: Public Domain

To accomplish this, you'll use an attention-based model, which enables us to see what parts of the image the model focuses on as it generates a caption.

```
Prediction Caption: the person is riding a surfboard in the ocean <end>
```



The model architecture is similar to [Show, Attend and Tell: Neural Image Caption Generation with Visual Attention](#).

This notebook is an end-to-end example. When you run the notebook, it downloads the [MS-COCO](#) dataset, preprocesses and caches a subset of images using Inception V3, trains an encoder-decoder model, and generates captions on new images using the trained model.

In this example, you will train a model on a relatively small amount of data—the first 30,000 captions for about 20,000 images (because there are multiple captions per image in the dataset).

```
import tensorflow as tf

# You'll generate plots of attention in order to see which parts of an image
# your model focuses on during captioning
import matplotlib.pyplot as plt

import collections
import random
import numpy as np
import os
import time
import json
from PIL import Image
```

Download and prepare the MS-COCO dataset

You will use the MS-COCO dataset to train your model. The dataset contains over 82,000 images, each of which has at least 5 different caption annotations. The code below downloads and extracts the dataset automatically.

Caution: large download ahead. You'll use the training set, which is a 13GB file.

```
# Download caption annotation files
annotation_folder = '/annotations/'
if not os.path.exists(os.path.abspath('.') + annotation_folder):
   annotation_zip = tf.keras.utils.get_file('captions.zip',
                                            cache_subdir=os.path.abspath('.'),
                                            origin='http://images.cocodataset.org/annotations
                                            extract=True)
   annotation_file = os.path.dirname(annotation_zip)+'/annotations/captions_train2014.json'
   os.remove(annotation_zip)

# Download image files
image_folder = '/train2014/'
if not os.path.exists(os.path.abspath('.') + image_folder):
  image_zip = tf.keras.utils.get_file('train2014.zip',
                                      cache_subdir=os.path.abspath('.'),
                                      origin='http://images.cocodataset.org/zips/train2014.zi
                                      extract=True)
  PATH = os.path.dirname(image_zip) + image_folder
  os.remove(image_zip)
else:
  PATH = os.path.abspath('.') + image_folder
```

```
Downloading data from http://images.cocodataset.org/annotations/annotations_trainval201
252878848/252872794 [==============================] - 5s 0us/step
252887040/252872794 [==============================] - 5s 0us/step
Downloading data from http://images.cocodataset.org/zips/train2014.zip
13510574080/13510573713 [==============================] - 199s 0us/step
13510582272/13510573713 [==============================] - 199s 0us/step
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Optional: limit the size of the training set

To speed up training for this tutorial, you'll use a subset of 30,000 captions and their corresponding images to train your model. Choosing to use more data would result in improved captioning quality.

```
annotation_file = 'annotations/captions_train2014.json'
PATH = "train2014/"
```

```
with open(annotation_file, 'r') as f:
    annotations = json.load(f)


# Group all captions together having the same image ID.
image_path_to_caption = collections.defaultdict(list)
for val in annotations['annotations']:
  caption = f"<start> {val['caption']} <end>"
  image_path = PATH + 'COCO_train2014_' + '%012d.jpg' % (val['image_id'])
  image_path_to_caption[image_path].append(caption)


image_paths = list(image_path_to_caption.keys())
random.shuffle(image_paths)

# Select the first 6000 image_paths from the shuffled set.
# Approximately each image id has 5 captions associated with it, so that will
# lead to 30,000 examples.
train_image_paths = image_paths[:10000]
print(len(train_image_paths))
```

```
10000
```

```
train_captions = []
img_name_vector = []

for image_path in train_image_paths:
  caption_list = image_path_to_caption[image_path]
  train_captions.extend(caption_list)
  img_name_vector.extend([image_path] * len(caption_list))


print(train_captions[5])
Image.open(img_name_vector[5])
```

Preprocess the images using InceptionV3

Next, you will use InceptionV3 (which is pretrained on Imagenet) to classify each image. You will extract features from the last convolutional layer.

First, you will convert the images into InceptionV3's expected format by:

- Resizing the image to 299px by 299px
- Preprocess the images using the preprocess_input method to normalize the image so that it contains pixels in the range of -1 to 1, which matches the format of the images used to train InceptionV3.

```
def load_image(image_path):
    img = tf.io.read_file(image_path)
    img = tf.io.decode_jpeg(img, channels=3)
    img = tf.keras.layers.Resizing(299, 299)(img)
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    return img, image_path
```

Initialize InceptionV3 and load the pretrained Imagenet weights

Now you'll create a tf.keras model where the output layer is the last convolutional layer in the InceptionV3 architecture. The shape of the output of this layer is `8x8x2048`. You use the last convolutional layer because you are using attention in this example. You don't perform this initialization during training because it could become a bottleneck.

- You forward each image through the network and store the resulting vector in a dictionary (image_name --> feature_vector).
- After all the images are passed through the network, you save the dictionary to disk.

```
image_model = tf.keras.applications.InceptionV3(include_top=False,
                                                weights='imagenet')
new_input = image_model.input
hidden_layer = image_model.layers[-1].output

image_features_extract_model = tf.keras.Model(new_input, hidden_layer)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/ince
87916544/87910968 [==============================] - 2s 0us/step
87924736/87910968 [==============================] - 2s 0us/step
```

Caching the features extracted from InceptionV3

You will pre-process each image with InceptionV3 and cache the output to disk. Caching the output in RAM would be faster but also memory intensive, requiring 8 * 8 * 2048 floats per image. At the time of writing, this exceeds the memory limitations of Colab (currently 12GB of memory).

Performance could be improved with a more sophisticated caching strategy (for example, by sharding the images to reduce random access disk I/O), but that would require more code.

The caching will take about 10 minutes to run in Colab with a GPU. If you'd like to see a progress bar, you can:

1. Install tqdm:

   ```
   !pip install tqdm
   ```

2. Import tqdm:

   ```
   from tqdm import tqdm
   ```

3. Change the following line:

   ```
   for img, path in image_dataset:
   ```

   to:

   ```
   for img, path in tqdm(image_dataset):
   ```

```
from tqdm import tqdm
```

```
# Get unique images
encode_train = sorted(set(img_name_vector))

# Feel free to change batch_size according to your system configuration
image_dataset = tf.data.Dataset.from_tensor_slices(encode_train)
image_dataset = image_dataset.map(
  load_image, num_parallel_calls=tf.data.AUTOTUNE).batch(16)

for img, path in tqdm(image_dataset):
  batch_features = image_features_extract_model(img)
  batch_features = tf.reshape(batch_features,
                              (batch_features.shape[0], -1, batch_features.shape[3]))

  for bf, p in zip(batch_features, path):
    path_of_feature = p.numpy().decode("utf-8")
    np.save(path_of_feature, bf.numpy())
```

```
100%|██████████| 625/625 [03:22<00:00,  3.09it/s]
```

```
print(path_of_feature)
```

```
train2014/COCO_train2014_000000581921.jpg
```

Preprocess and tokenize the captions

You will transform the text captions into integer sequences using the TextVectorization layer, with the following steps:

- Use adapt to iterate over all captions, split the captions into words, and compute a vocabulary of the top 5,000 words (to save memory).
- Tokenize all captions by mapping each word to it's index in the vocabulary. All output sequences will be padded to length 50.
- Create word-to-index and index-to-word mappings to display results.

```
caption_dataset = tf.data.Dataset.from_tensor_slices(train_captions)

# We will override the default standardization of TextVectorization to preserve
# "<>" characters, so we preserve the tokens for the <start> and <end>.
def standardize(inputs):
  inputs = tf.strings.lower(inputs)
  return tf.strings.regex_replace(inputs,
                                  r"!\"#$%&\(\)\*\+.,-/:;=?@\[\\\]^_`{|}~", "")

# Max word count for a caption.
max_length = 50
# Use the top 5000 words for a vocabulary.
vocabulary_size = 5000
```

```
tokenizer = tf.keras.layers.TextVectorization(
    max_tokens=vocabulary_size,
    standardize=standardize,
    output_sequence_length=max_length)
# Learn the vocabulary from the caption data.
tokenizer.adapt(caption_dataset)

# Create the tokenized vectors
cap_vector = caption_dataset.map(lambda x: tokenizer(x))
```

```
cap_vector
```

```
<MapDataset element_spec=TensorSpec(shape=(None,), dtype=tf.int64, name=None)>
```

```
# Create mappings for words to indices and indicies to words.
word_to_index = tf.keras.layers.StringLookup(
    mask_token="",
    vocabulary=tokenizer.get_vocabulary())
index_to_word = tf.keras.layers.StringLookup(
    mask_token="",
    vocabulary=tokenizer.get_vocabulary(),
    invert=True)
```

Split the data into training and testing

```
img_to_cap_vector = collections.defaultdict(list)
for img, cap in zip(img_name_vector, cap_vector):
  img_to_cap_vector[img].append(cap)

# Create training and validation sets using an 80-20 split randomly.
img_keys = list(img_to_cap_vector.keys())
random.shuffle(img_keys)

slice_index = int(len(img_keys)*0.8)
img_name_train_keys, img_name_val_keys = img_keys[:slice_index], img_keys[slice_index:]

img_name_train = []
cap_train = []
for imgt in img_name_train_keys:
  capt_len = len(img_to_cap_vector[imgt])
  img_name_train.extend([imgt] * capt_len)
  cap_train.extend(img_to_cap_vector[imgt])

img_name_val = []
cap_val = []
for imgv in img_name_val_keys:
  capv_len = len(img_to_cap_vector[imgv]) # number of ref captions per image
```

```
  img_name_val.extend([imgv] * capv_len) # image files * number of ref captions
  cap_val.extend(img_to_cap_vector[imgv]) # captions as vector
```

```
len(img_name_train), len(cap_train), len(img_name_val), len(cap_val)
```

```
(40018, 40018, 10005, 10005)
```

```
import pickle

with open("img_name_train", "wb") as fp:   #Pickling
  pickle.dump(img_name_train, fp)
with open("img_name_val", "wb") as fp:   #Pickling
  pickle.dump(img_name_val, fp)
with open("cap_train", "wb") as fp:   #Pickling
  pickle.dump(cap_train, fp)
with open("cap_val", "wb") as fp:   #Pickling
  pickle.dump(cap_val, fp)
```

Create a tf.data dataset for training

Your images and captions are ready! Next, let's create a `tf.data` dataset to use for training your model.

```
import pickle

with open("img_name_train", "rb") as fp:   # Unpickling
  img_name_train = pickle.load(fp)
with open("img_name_val", "rb") as fp:   # Unpickling
  img_name_val = pickle.load(fp)
with open("cap_train", "rb") as fp:   # Unpickling
  cap_train = pickle.load(fp)
with open("cap_val", "rb") as fp:   # Unpickling
  cap_val = pickle.load(fp)
```

```
# Feel free to change these parameters according to your system's configuration

BATCH_SIZE = 64
BUFFER_SIZE = 1000
embedding_dim = 256
units = 512
num_steps = len(img_name_train) // BATCH_SIZE
# Shape of the vector extracted from InceptionV3 is (64, 2048)
```

```
# These two variables represent that vector shape
features_shape = 2048
attention_features_shape = 64


# Load the numpy files
def map_func(img_name, cap):
  img_tensor = np.load(img_name.decode('utf-8')+'.npy')
  return img_tensor, cap


dataset = tf.data.Dataset.from_tensor_slices((img_name_train, cap_train))

# Use map to load the numpy files in parallel
dataset = dataset.map(lambda item1, item2: tf.numpy_function(
          map_func, [item1, item2], [tf.float32, tf.int64]),
          num_parallel_calls=tf.data.AUTOTUNE)

# Shuffle and batch
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
dataset = dataset.prefetch(buffer_size=tf.data.AUTOTUNE)
```

Model

Fun fact: the decoder below is identical to the one in the example for Neural Machine Translation with Attention.

The model architecture is inspired by the Show, Attend and Tell paper.

- In this example, you extract the features from the lower convolutional layer of InceptionV3 giving us a vector of shape (8, 8, 2048).
- You squash that to a shape of (64, 2048).
- This vector is then passed through the CNN Encoder (which consists of a single Fully connected layer).
- The RNN (here GRU) attends over the image to predict the next word.

```
class BahdanauAttention(tf.keras.Model):
  def __init__(self, units):
    super(BahdanauAttention, self).__init__()
    self.W1 = tf.keras.layers.Dense(units)
    self.W2 = tf.keras.layers.Dense(units)
    self.V = tf.keras.layers.Dense(1)

  def call(self, features, hidden):
    # features(CNN_encoder output) shape == (batch_size, 64, embedding_dim)

    # hidden shape == (batch_size, hidden_size)
```

```
    # hidden_with_time_axis shape == (batch_size, 1, hidden_size)
    hidden_with_time_axis = tf.expand_dims(hidden, 1)

    # attention_hidden_layer shape == (batch_size, 64, units)
    attention_hidden_layer = (tf.nn.tanh(self.W1(features) +
                                         self.W2(hidden_with_time_axis)))

    # score shape == (batch_size, 64, 1)
    # This gives you an unnormalized score for each image feature.
    score = self.V(attention_hidden_layer)

    # attention_weights shape == (batch_size, 64, 1)
    attention_weights = tf.nn.softmax(score, axis=1)

    # context_vector shape after sum == (batch_size, hidden_size)
    context_vector = attention_weights * features
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights

class CNN_Encoder(tf.keras.Model):
    # Since you have already extracted the features and dumped it
    # This encoder passes those features through a Fully connected layer
    def __init__(self, embedding_dim):
        super(CNN_Encoder, self).__init__()
        # shape after fc == (batch_size, 64, embedding_dim)
        self.fc = tf.keras.layers.Dense(embedding_dim)

    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x


class RNN_Decoder(tf.keras.Model):
  def __init__(self, embedding_dim, units, vocab_size):
    super(RNN_Decoder, self).__init__()
    self.units = units

    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')
    self.fc1 = tf.keras.layers.Dense(self.units)
    self.fc2 = tf.keras.layers.Dense(vocab_size)

    self.attention = BahdanauAttention(self.units)

  def call(self, x, features, hidden):
    # defining attention as a separate model
```

```
    context_vector, attention_weights = self.attention(features, hidden)

    # x shape after passing through embedding == (batch_size, 1, embedding_dim)
    x = self.embedding(x)

    # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

    # passing the concatenated vector to the GRU
    output, state = self.gru(x)

    # shape == (batch_size, max_length, hidden_size)
    x = self.fc1(output)

    # x shape == (batch_size * max_length, hidden_size)
    x = tf.reshape(x, (-1, x.shape[2]))

    # output shape == (batch_size * max_length, vocab)
    x = self.fc2(x)

    return x, state, attention_weights

  def reset_state(self, batch_size):
    return tf.zeros((batch_size, self.units))

encoder = CNN_Encoder(embedding_dim)
decoder = RNN_Decoder(embedding_dim, units, tokenizer.vocabulary_size())


optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')


def loss_function(real, pred):
  mask = tf.math.logical_not(tf.math.equal(real, 0))
  loss_ = loss_object(real, pred)

  mask = tf.cast(mask, dtype=loss_.dtype)
  loss_ *= mask

  return tf.reduce_mean(loss_)
```

Checkpoint

```
checkpoint_path = "./checkpoints_new/train"
ckpt = tf.train.Checkpoint(encoder=encoder,
                           decoder=decoder,
```

```
                           optimizer=optimizer)
ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)

start_epoch = 0
if ckpt_manager.latest_checkpoint:
  start_epoch = int(ckpt_manager.latest_checkpoint.split('-')[-1])
  # restoring the latest checkpoint in checkpoint_path
  ckpt.restore(ckpt_manager.latest_checkpoint)
```

```
ckpt.restore(ckpt_manager.latest_checkpoint)
```

```
<tensorflow.python.training.tracking.util.InitializationOnlyStatus at 0x7f17b1ab3490>
```

Training

- You extract the features stored in the respective `.npy` files and then pass those features through the encoder.
- The encoder output, hidden state(initialized to 0) and the decoder input (which is the start token) is passed to the decoder.
- The decoder returns the predictions and the decoder hidden state.
- The decoder hidden state is then passed back into the model and the predictions are used to calculate the loss.
- Use teacher forcing to decide the next input to the decoder.
- Teacher forcing is the technique where the target word is passed as the next input to the decoder.
- The final step is to calculate the gradients and apply it to the optimizer and backpropagate.

```
# adding this in a separate cell because if you run the training cell
# many times, the loss_plot array will be reset
loss_plot = []
```

```
@tf.function
def train_step(img_tensor, target):
  loss = 0

  # initializing the hidden state for each batch
  # because the captions are not related from image to image
  hidden = decoder.reset_state(batch_size=target.shape[0])

  dec_input = tf.expand_dims([word_to_index('<start>')] * target.shape[0], 1)

  with tf.GradientTape() as tape:
      features = encoder(img_tensor)

      for i in range(1, target.shape[1]):
```

```
          # passing the features through the decoder
          predictions, hidden, _ = decoder(dec_input, features, hidden)

          loss += loss_function(target[:, i], predictions)

          # using teacher forcing
          dec_input = tf.expand_dims(target[:, i], 1)

  total_loss = (loss / int(target.shape[1]))

  trainable_variables = encoder.trainable_variables + decoder.trainable_variables

  gradients = tape.gradient(loss, trainable_variables)

  optimizer.apply_gradients(zip(gradients, trainable_variables))

  return loss, total_loss
```

```
EPOCHS = 30

for epoch in range(start_epoch, EPOCHS):
    start = time.time()
    total_loss = 0

    for (batch, (img_tensor, target)) in enumerate(dataset):
        batch_loss, t_loss = train_step(img_tensor, target)
        total_loss += t_loss

        if batch % 100 == 0:
            average_batch_loss = batch_loss.numpy()/int(target.shape[1])
            print(f'Epoch {epoch+1} Batch {batch} Loss {average_batch_loss:.4f}')
    # storing the epoch end loss value to plot later
    loss_plot.append(total_loss / num_steps)

    if epoch % 5 == 0:
      ckpt_manager.save()

    print(f'Epoch {epoch+1} Loss {total_loss/num_steps:.6f}')
    print(f'Time taken for 1 epoch {time.time()-start:.2f} sec\n')
```

```
Epoch 22 Batch 400 Loss 0.2667
Epoch 22 Batch 500 Loss 0.2881
Epoch 22 Batch 600 Loss 0.2589
Epoch 22 Loss 0.279838
Time taken for 1 epoch 292.97 sec

Epoch 23 Batch 0 Loss 0.2937
Epoch 23 Batch 100 Loss 0.2759
Epoch 23 Batch 200 Loss 0.2823
Epoch 23 Batch 300 Loss 0.2564
Epoch 23 Batch 400 Loss 0.2769
Epoch 23 Batch 500 Loss 0.2487
```

```
Epoch 23 Batch 600 Loss 0.2834
Epoch 23 Loss 0.270517
Time taken for 1 epoch 292.81 sec

Epoch 24 Batch 0 Loss 0.2565
Epoch 24 Batch 100 Loss 0.2968
Epoch 24 Batch 200 Loss 0.2729
Epoch 24 Batch 300 Loss 0.2629
Epoch 24 Batch 400 Loss 0.2446
Epoch 24 Batch 500 Loss 0.2734
Epoch 24 Batch 600 Loss 0.2632
Epoch 24 Loss 0.262851
Time taken for 1 epoch 292.86 sec

Epoch 25 Batch 0 Loss 0.2415
Epoch 25 Batch 100 Loss 0.2329
Epoch 25 Batch 200 Loss 0.2501
Epoch 25 Batch 300 Loss 0.2344
Epoch 25 Batch 400 Loss 0.2553
Epoch 25 Batch 500 Loss 0.2269
Epoch 25 Batch 600 Loss 0.2697
Epoch 25 Loss 0.256080
Time taken for 1 epoch 292.95 sec

Epoch 26 Batch 0 Loss 0.2684
Epoch 26 Batch 100 Loss 0.2568
Epoch 26 Batch 200 Loss 0.2243
Epoch 26 Batch 300 Loss 0.2421
Epoch 26 Batch 400 Loss 0.2600
Epoch 26 Batch 500 Loss 0.2413
Epoch 26 Batch 600 Loss 0.2423
Epoch 26 Loss 0.249071
Time taken for 1 epoch 293.15 sec

Epoch 27 Batch 0 Loss 0.2530
Epoch 27 Batch 100 Loss 0.2870
Epoch 27 Batch 200 Loss 0.2656
Epoch 27 Batch 300 Loss 0.2458
Epoch 27 Batch 400 Loss 0.2418
Epoch 27 Batch 500 Loss 0.2601
Epoch 27 Batch 600 Loss 0.2456
Epoch 27 Loss 0.242620
Time taken for 1 epoch 292.72 sec

Epoch 28 Batch 0 Loss 0.2427
Epoch 28 Batch 100 Loss 0.2235
```

```
encoder.save('saved_model/encoder')
decoder.save('saved_model/decoder')
```

```
INFO:tensorflow:Assets written to: saved_model/encoder/assets
WARNING:absl:Found untraced functions such as gru_cell_layer_call_fn, gru_cell_layer_cal
INFO:tensorflow:Assets written to: saved_model/decoder/assets
INFO:tensorflow:Assets written to: saved_model/decoder/assets
WARNING:absl:<keras.layers.recurrent.GRUCell object at 0x7f17b0a0dc90> has the same name
```

```
plt.plot(loss_plot)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Plot')
plt.show()
```

Caption!

- The evaluate function is similar to the training loop, except you don't use teacher forcing here. The input to the decoder at each time step is its previous predictions along with the hidden state and the encoder output.
- Stop predicting when the model predicts the end token.
- And store the attention weights for every time step.

```
def evaluate(image):
    attention_plot = np.zeros((max_length, attention_features_shape))

    hidden = decoder.reset_state(batch_size=1)

    temp_input = tf.expand_dims(load_image(image)[0], 0)
    img_tensor_val = image_features_extract_model(temp_input)
    img_tensor_val = tf.reshape(img_tensor_val, (img_tensor_val.shape[0],
                                                 -1,
                                                 img_tensor_val.shape[3]))

    features = encoder(img_tensor_val)

    dec_input = tf.expand_dims([word_to_index('<start>')], 0)
    result = []
```

```
    for i in range(max_length):
        predictions, hidden, attention_weights = decoder(dec_input,
                                                         features,
                                                         hidden)

        attention_plot[i] = tf.reshape(attention_weights, (-1, )).numpy()

        predicted_id = tf.random.categorical(predictions, 1)[0][0].numpy()
        predicted_word = tf.compat.as_text(index_to_word(predicted_id).numpy())
        result.append(predicted_word)

        if predicted_word == '<end>':
            return result, attention_plot

        dec_input = tf.expand_dims([predicted_id], 0)

    attention_plot = attention_plot[:len(result), :]
    return result, attention_plot


def plot_attention(image, result, attention_plot):
    temp_image = np.array(Image.open(image))

    fig = plt.figure(figsize=(10, 10))

    len_result = len(result)
    for i in range(len_result):
        temp_att = np.resize(attention_plot[i], (8, 8))
        grid_size = max(int(np.ceil(len_result/2)), 2)
        ax = fig.add_subplot(grid_size, grid_size, i+1)
        ax.set_title(result[i])
        img = ax.imshow(temp_image)
        ax.imshow(temp_att, cmap='gray', alpha=0.6, extent=img.get_extent())

    plt.tight_layout()
    plt.show()


# captions on the validation set
rid = np.random.randint(0, len(img_name_val)) #index
image = img_name_val[rid] # image
real_caption = ' '.join([tf.compat.as_text(index_to_word(i).numpy())
                         for i in cap_val[rid] if i not in [0]])
predicion_caption = ' '.join(result)
result, attention_plot = evaluate(image)

print('Real Caption:', real_caption)
print('Prediction Caption:', predicion_caption)
plot_attention(image, result, attention_plot)
```

```
print(loss_function(real_caption, predicion_caption))
```

Try it on your own images

For fun, below you're provided a method you can use to caption your own images with the model you've just trained. Keep in mind, it was trained on a relatively small amount of data, and your images may be different from the training data (so be prepared for weird results!)

```
image_url = 'https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcThYFdfVHUMNCWe30Yr0rW67O1L
image_extension = image_url[-4:]
image_path = tf.keras.utils.get_file('image'+image_extension, origin=image_url)
result, attention_plot = evaluate(image_path)
print('Prediction Caption:', ' '.join(result))
```

```
plot_attention(image_path, result, attention_plot)
# opening the image
Image.open(image_path)
```

Bleu Score

```
from nltk.translate.bleu_score import sentence_bleu
#from nltk.translate.meteor_score import meteor_score
```

```
from nltk.translate.bleu_score import SmoothingFunction
smoothie = SmoothingFunction().method4
```

```
import nltk
nltk.download('omw-1.4')
```

```
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Unzipping corpora/omw-1.4.zip.
True
```

```
ref  = ["this is an image of three sheep in a field".split(' ')]
pred = "two sheep and a baby sheep standing on lush green field".split(' ')
score1 = sentence_bleu(ref, pred, weights=(1, 0, 0, 0))
score4 = sentence_bleu(ref, pred, weights=(0.25, 0.25, 0.25, 0.25)) # default

print(score1,score4)
```

```
0.2727272727272727 0.7226568811456053
/usr/local/lib/python3.7/dist-packages/nltk/translate/bleu_score.py:490: UserWarning:
Corpus/Sentence contains 0 counts of 2-gram overlaps.
BLEU scores might be undesirable; use SmoothingFunction().
  warnings.warn(_msg)
```

```
score_m = nltk.translate.meteor_score.meteor_score(ref,pred)
print(score_m)
```

Validation Loss

```
encoder_new = tf.keras.models.load_model('saved_model/encoder', compile=False)
encoder_new.summary()
```

```
Model: "cnn__encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               multiple                  524544

=================================================================
Total params: 524,544
Trainable params: 524,544
Non-trainable params: 0
_________________________________________________________________
```

```
decoder_new = tf.keras.models.load_model('saved_model/decoder', compile=False)
decoder_new.summary()
```

```
Model: "rnn__decoder"
_________________________________________________________________
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       multiple                  1280000

 gru (GRU)                   multiple                  1575936

 dense_1 (Dense)             multiple                  262656

 dense_2 (Dense)             multiple                  2565000

 bahdanau_attention (Bahdana  multiple                 394753
 uAttention)

=================================================================
Total params: 6,078,345
Trainable params: 6,078,345
Non-trainable params: 0
_________________________________________________________________
```

Evaluating on Validation set

```
img_name_val[0]
Image.open(img_name_val[0])
```

```
# img_name_val = []
# cap_val = []
# for imgv in img_name_val_keys:
#   capv_len = len(img_to_cap_vector[imgv]) # number of ref captions per image
```

```
#   img_name_val.extend([imgv] * capv_len) # image files * number of ref captions
#   cap_val.extend(img_to_cap_vector[imgv]) # captions as vector
```

```
i = 0
bleu_1 = []
bleu_4 = []
while i < len(img_name_val):
  image = img_name_val[i]
  cap_len = len(img_to_cap_vector[img_name_val[i]])
  ref = [tf.compat.as_text(index_to_word(word).numpy())
                       for word in cap_val[i] if word not in [0]]
  if '<start>' in ref:
    ref.remove('<start>')
  if '<end>' in ref:
    ref.remove('<end>')
  ref = [ref]

  pred, attention_plot = evaluate(image)
  if '<start>' in pred:
    pred.remove('<start>')
  if '<end>' in pred:
    pred.remove('<end>')

  score1 = sentence_bleu(ref, pred, weights=(1, 0, 0, 0))
  score4 = sentence_bleu(ref, pred, weights=(0.25, 0.25, 0.25, 0.25))
  bleu_1.append(score1)
  bleu_4.append(score4)
  print(i)
  i += cap_len
```

```
9052
9057
9062
9067
9072
9077
9082
9087
9092
9097
9102
9107
9112
9117
9122
9127
9132
9137
9142
9147
9152
9157
```

```
9162
9167
9172
9177
9182
9187
9192
9197
9202
9207
9212
9217
9222
9227
9232
9237
9242
9247
9252
9257
9262
9267
9272
9277
9282
9287
9292
9297
9302
9307
9312
9317
9322
9327
9332
9337
```

```
reference = [['this', 'is', 'test']]
candidate = ['this', 'is', 'test']
score = sentence_bleu(reference, candidate, weights=(1, 0, 0, 0))
print(score)
```

```
1.0
/usr/local/lib/python3.7/dist-packages/nltk/translate/bleu_score.py:552: UserWarning:
The hypothesis contains 0 counts of 4-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
  warnings.warn(_msg)
```

```
with open("bleu_1", "wb") as fp:   #Pickling
  pickle.dump(bleu_1, fp)
with open("bleu_4", "wb") as fp:   #Pickling
```

```
  pickle.dump(bleu_4, fp)


avg_bleu_1 = sum(bleu_1)/len(bleu_1)
avg_bleu_4 = sum(bleu_4)/len(bleu_4)


print(avg_bleu_1, avg_bleu_4)
```

```
0.21285533929271902 0.005327906965149687
```

```
plt.plot(bleu_1)
```

```
plt.plot(bleu_4)
```

Next steps

Congrats! You've just trained an image captioning model with attention. Next, take a look at this example Neural Machine Translation with Attention. It uses a similar architecture to translate between Spanish and English sentences. You can also experiment with training the code in this notebook on a different dataset.

✓ 4s completed at 3:22 PM