Attention Mechanism

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", f
```

```
import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from sklearn.model_selection import train_test_split
import unicodedata
import re
import numpy as np
import os
import io
import time
```

```
path_to_file ='/content/drive/My Drive/DeepLearning/mar_1.txt'
```

```
# Converts the unicode file to ascii
def unicode_to_ascii(s):
  return ''.join(c for c in unicodedata.normalize('NFD', s)
      if unicodedata.category(c) != 'Mn')

def preprocess_sentence(w):
  w = unicode_to_ascii(w.lower().strip())
  w = re.sub(r"([?.!,¿])", r" \1 ", w) # creating a space between a word and the punctuation following it
  w = re.sub(r'[" "]+', " ", w)
  w = w.strip()
  w = '<start> ' + w + ' <end>' # adding a start and an end token to the sentence
  return w


def create_dataset(path, num_examples):
  lines = io.open(path, encoding='UTF-8').read().strip().split('\n')
  word_pairs = [[preprocess_sentence(w) for w in l.split('\t')]  for l in lines[:num_examples]]
  return zip(*word_pairs)


en, mar = create_dataset(path_to_file, None)
print(en[5000])
print(mar[5000])
```

```
<start> tom saw a snake . <end>
<start> टॉमला एक साप दिसला . <end>
```

```
def tokenize(lang):
  lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='')
  lang_tokenizer.fit_on_texts(lang)
  tensor = lang_tokenizer.texts_to_sequences(lang)
  tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor,padding='post')
  return tensor, lang_tokenizer


def load_dataset(path, num_examples=None):
  # creating cleaned input, output pairs
  targ_lang, inp_lang = create_dataset(path, num_examples)
  input_tensor, inp_lang_tokenizer = tokenize(inp_lang)
  target_tensor, targ_lang_tokenizer = tokenize(targ_lang)
  return input_tensor, target_tensor, inp_lang_tokenizer, targ_lang_tokenizer


# Try experimenting with the size of that dataset
```

```
num_examples = 30000
input_tensor, target_tensor, inp_lang, targ_lang = load_dataset(path_to_file, num_examples)

# Calculate max_length of the target tensors
max_length_targ, max_length_inp = target_tensor.shape[1], input_tensor.shape[1]


# Creating training and validation sets using an 80-20 split
input_tensor_train, input_tensor_val, target_tensor_train, target_tensor_val = train_test_split(input_tensor, targe

# Show length
print(len(input_tensor_train), len(target_tensor_train), len(input_tensor_val), len(target_tensor_val))
```

```
24000 24000 6000 6000
```

```
def convert(lang, tensor):
  for t in tensor:
    if t!=0:
      print ("%d ----> %s" % (t, lang.index_word[t]))


print ("Input Language; index to word mapping")
convert(inp_lang, input_tensor_train[0])
print ()
print ("Target Language; index to word mapping")
convert(targ_lang, target_tensor_train[0])
```

```
Input Language; index to word mapping
1 ----> <start>
10 ----> टॉम
48 ----> आता
96 ----> अजन
3770 ----> नाकार
37 ----> शकत
11 ----> नाही
3 ----> .
2 ----> <end>

Target Language; index to word mapping
1 ----> <start>
6 ----> tom
65 ----> can't
2331 ----> deny
21 ----> it
388 ----> anymore
3 ----> .
2 ----> <end>
```

```
#Create a tf.data dataset
BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
steps_per_epoch = len(input_tensor_train)//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(inp_lang.word_index)+1
vocab_tar_size = len(targ_lang.word_index)+1

dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train, target_tensor_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)


example_input_batch, example_target_batch = next(iter(dataset))
example_input_batch.shape, example_target_batch.shape
```

```
(TensorShape([64, 15]), TensorShape([64, 12]))
```

```
class Encoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
    super(Encoder, self).__init__()
    self.batch_sz = batch_sz
```

```
        self.enc_units = enc_units
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = tf.keras.layers.GRU(self.enc_units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_uniform')

      def call(self, x, hidden):
        x = self.embedding(x)
        output, state = self.gru(x, initial_state = hidden)
        return output, state

      def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))


    encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)

    # sample input
    sample_hidden = encoder.initialize_hidden_state()
    sample_output, sample_hidden = encoder(example_input_batch, sample_hidden)
    print ('Encoder output shape: (batch size, sequence length, units) {}'.format(sample_output.shape))
    print ('Encoder Hidden state shape: (batch size, units) {}'.format(sample_hidden.shape))

         Encoder output shape: (batch size, sequence length, units) (64, 15, 1024)
         Encoder Hidden state shape: (batch size, units) (64, 1024)


    class BahdanauAttention(tf.keras.layers.Layer):
      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, query, values):
        query_with_time_axis = tf.expand_dims(query, 1)
        score = self.V(tf.nn.tanh(
            self.W1(query_with_time_axis) + self.W2(values)))

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

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

        return context_vector, attention_weights


    attention_layer = BahdanauAttention(10)
    attention_result, attention_weights = attention_layer(sample_hidden, sample_output)

    print("Attention result shape: (batch size, units) {}".format(attention_result.shape))
    print("Attention weights shape: (batch_size, sequence_length, 1) {}".format(attention_weights.shape))

         Attention result shape: (batch size, units) (64, 1024)
         Attention weights shape: (batch_size, sequence_length, 1) (64, 15, 1)


    class Decoder(tf.keras.Model):
      def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
        super(Decoder, self).__init__()
        self.batch_sz = batch_sz
        self.dec_units = dec_units
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = tf.keras.layers.GRU(self.dec_units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_uniform')
        self.fc = tf.keras.layers.Dense(vocab size)
```

```
    # used for attention
    self.attention = BahdanauAttention(self.dec_units)

  def call(self, x, hidden, enc_output):
    # enc_output shape == (batch_size, max_length, hidden_size)
    context_vector, attention_weights = self.attention(hidden, enc_output)

    # 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)

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

    # output shape == (batch_size, vocab)
    x = self.fc(output)

    return x, state, attention_weights


decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)

sample_decoder_output, _, _ = decoder(tf.random.uniform((BATCH_SIZE, 1)),
                                      sample_hidden, sample_output)

print ('Decoder output shape: (batch_size, vocab size) {}'.format(sample_decoder_output.shape))
```

```
Decoder output shape: (batch_size, vocab size) (64, 4058)
```

```
# Define the optimizer and the loss function
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_dir = '/content/drive/My Drive/DeepLearning/training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(optimizer=optimizer,
                                 encoder=encoder,
                                 decoder=decoder)


# Training
@tf.function
def train_step(inp, targ, enc_hidden):
  loss = 0

  with tf.GradientTape() as tape:
    enc_output, enc_hidden = encoder(inp, enc_hidden)

    dec_hidden = enc_hidden

    dec_input = tf.expand_dims([targ_lang.word_index['<start>']] * BATCH_SIZE, 1)
```

```
      # Teacher forcing - feeding the target as the next input
      for t in range(1, targ.shape[1]):
        # passing enc_output to the decoder
        predictions, dec_hidden, _ = decoder(dec_input, dec_hidden, enc_output)

        loss += loss_function(targ[:, t], predictions)

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

    batch_loss = (loss / int(targ.shape[1]))

    variables = encoder.trainable_variables + decoder.trainable_variables

    gradients = tape.gradient(loss, variables)

    optimizer.apply_gradients(zip(gradients, variables))

    return batch_loss


EPOCHS = 10

for epoch in range(EPOCHS):
  start = time.time()

  enc_hidden = encoder.initialize_hidden_state()
  total_loss = 0

  for (batch, (inp, targ)) in enumerate(dataset.take(steps_per_epoch)):
    batch_loss = train_step(inp, targ, enc_hidden)
    total_loss += batch_loss

    if batch % 100 == 0:
      print('Epoch {} Batch {} Loss {:.4f}'.format(epoch + 1,
                                                   batch,
                                                   batch_loss.numpy()))
```

```
Epoch 1 Batch 0 Loss 4.4462
Epoch 1 Batch 100 Loss 2.3154
Epoch 1 Batch 200 Loss 1.8573
Epoch 1 Batch 300 Loss 1.7049
Epoch 2 Batch 0 Loss 1.5004
Epoch 2 Batch 100 Loss 1.4974
Epoch 2 Batch 200 Loss 1.4323
Epoch 2 Batch 300 Loss 1.3992
Epoch 3 Batch 0 Loss 1.1650
Epoch 3 Batch 100 Loss 1.0781
Epoch 3 Batch 200 Loss 1.0671
Epoch 3 Batch 300 Loss 0.9788
Epoch 4 Batch 0 Loss 0.7884
Epoch 4 Batch 100 Loss 0.7493
Epoch 4 Batch 200 Loss 0.7839
Epoch 4 Batch 300 Loss 0.6417
Epoch 5 Batch 0 Loss 0.4474
Epoch 5 Batch 100 Loss 0.4609
Epoch 5 Batch 200 Loss 0.5280
Epoch 5 Batch 300 Loss 0.3831
Epoch 6 Batch 0 Loss 0.2571
Epoch 6 Batch 100 Loss 0.2528
Epoch 6 Batch 200 Loss 0.2732
Epoch 6 Batch 300 Loss 0.3432
Epoch 7 Batch 0 Loss 0.1837
Epoch 7 Batch 100 Loss 0.1538
Epoch 7 Batch 200 Loss 0.1562
Epoch 7 Batch 300 Loss 0.1552
Epoch 8 Batch 0 Loss 0.0950
Epoch 8 Batch 100 Loss 0.1286
Epoch 8 Batch 200 Loss 0.0942
Epoch 8 Batch 300 Loss 0.1239
Epoch 9 Batch 0 Loss 0.0706
Epoch 9 Batch 100 Loss 0.0906
Epoch 9 Batch 200 Loss 0.0744
```

```
Epoch 9 Batch 300 Loss 0.0844
Epoch 10 Batch 0 Loss 0.0528
Epoch 10 Batch 100 Loss 0.0628
Epoch 10 Batch 200 Loss 0.0612
Epoch 10 Batch 300 Loss 0.0670
```

```
def evaluate(sentence):
  attention_plot = np.zeros((max_length_targ, max_length_inp))

  sentence = preprocess_sentence(sentence)

  inputs = [inp_lang.word_index[i] for i in sentence.split(' ')]
  inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs],
                                                         maxlen=max_length_inp,
                                                         padding='post')
  inputs = tf.convert_to_tensor(inputs)

  result = ''

  hidden = [tf.zeros((1, units))]
  enc_out, enc_hidden = encoder(inputs, hidden)

  dec_hidden = enc_hidden
  dec_input = tf.expand_dims([targ_lang.word_index['<start>']], 0)

  for t in range(max_length_targ):
    predictions, dec_hidden, attention_weights = decoder(dec_input,
                                                         dec_hidden,
                                                         enc_out)

    # storing the attention weights to plot later on
    attention_weights = tf.reshape(attention_weights, (-1, ))
    attention_plot[t] = attention_weights.numpy()

    predicted_id = tf.argmax(predictions[0]).numpy()

    result += targ_lang.index_word[predicted_id] + ' '

    if targ_lang.index_word[predicted_id] == '<end>':
      return result, sentence, attention_plot

    # the predicted ID is fed back into the model
    dec_input = tf.expand_dims([predicted_id], 0)

  return result, sentence, attention_plot


# function for plotting the attention weights
def plot_attention(attention, sentence, predicted_sentence):
  fig = plt.figure(figsize=(10,10))
  ax = fig.add_subplot(1, 1, 1)
  ax.matshow(attention, cmap='viridis')

  fontdict = {'fontsize': 14}

  ax.set_xticklabels(['']  sentence, fontdict=fontdict, rotation=90)
  ax.set_yticklabels(['']  predicted_sentence, fontdict=fontdict)

  ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
  ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

  plt.show()


def translate(sentence):
  result, sentence, attention_plot = evaluate(sentence)

  print('Input: %s' % (sentence))
  print('Predicted translation: {}'.format(result))
```

```
  attention_plot = attention_plot[:len(result.split(' ')), :len(sentence.split(' '))]
  #plot_attention(attention_plot, sentence.split(' '), result.split(' '))


checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

```
<tensorflow.python.training.tracking.util.InitializationOnlyStatus at 0x7f8490017cf8>
```

```
translate("पळ !")
print("\n")
translate("मी आजारी आहे.")
print("\n")
translate("तयानी वाट बघितली.")
print("\n")
translate("मी इथ मदत करायला आलो .")
print("\n")
translate("टॉमला एक साप दिसला .")
```

```
Input: <start> पळ ! <end>
Predicted translation: run ! <end>


Input: <start> मी आजारी आह . <end>
Predicted translation: i'm ill . <end>


Input: <start> तयानी वाट बघितली . <end>
Predicted translation: they waited . <end>


Input: <start> मी इथ मदत करायला आलो . <end>
Predicted translation: i came here to help . <end>


Input: <start> टॉमला एक साप दिसला . <end>
Predicted translation: tom saw a snake . <end>
```

```
# wrong translation


translate("बारबकय करत अस तर मला तरी काहीही अडचण नाहीय , पण मला तो धर माझया डोळयात गलला आवडत नाही .")
```

```
Input: <start> बारबकय करत अस तर मला तरी काहीही अडचण नाहीय , पण मला तो धर माझया डोळयात गलला आवडत नाही . <end>
Predicted translation: so aren't you can't say so so so so so so so
```