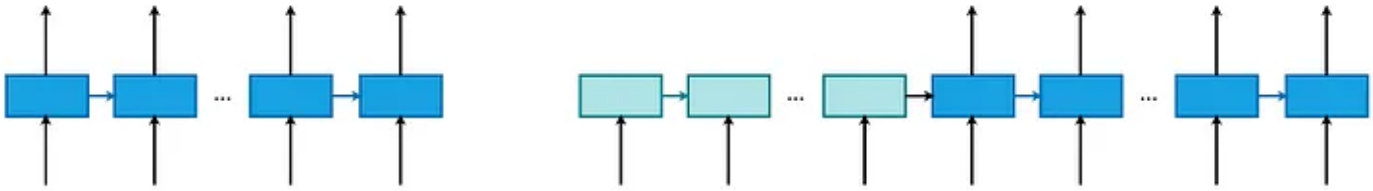
Jonathan Hui Follow

Feb 1, 2021 · 10 min read · Listen

Save

TensorFlow Sequence to Sequence Model Examples

Sequence-to-sequence models are particularly popular in NLP. This article, as part of the TensorFlow series, will cover examples for the sequence to sequence model.



The examples include:

- Shakespear text generation with GRU,
- Seq2seq language translation with attention,
- Image captioning with visual attention and Inception V3 for feature extraction.

Shakespear text generation with GRU

Given a large file with Shakespear's writings,

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.
```

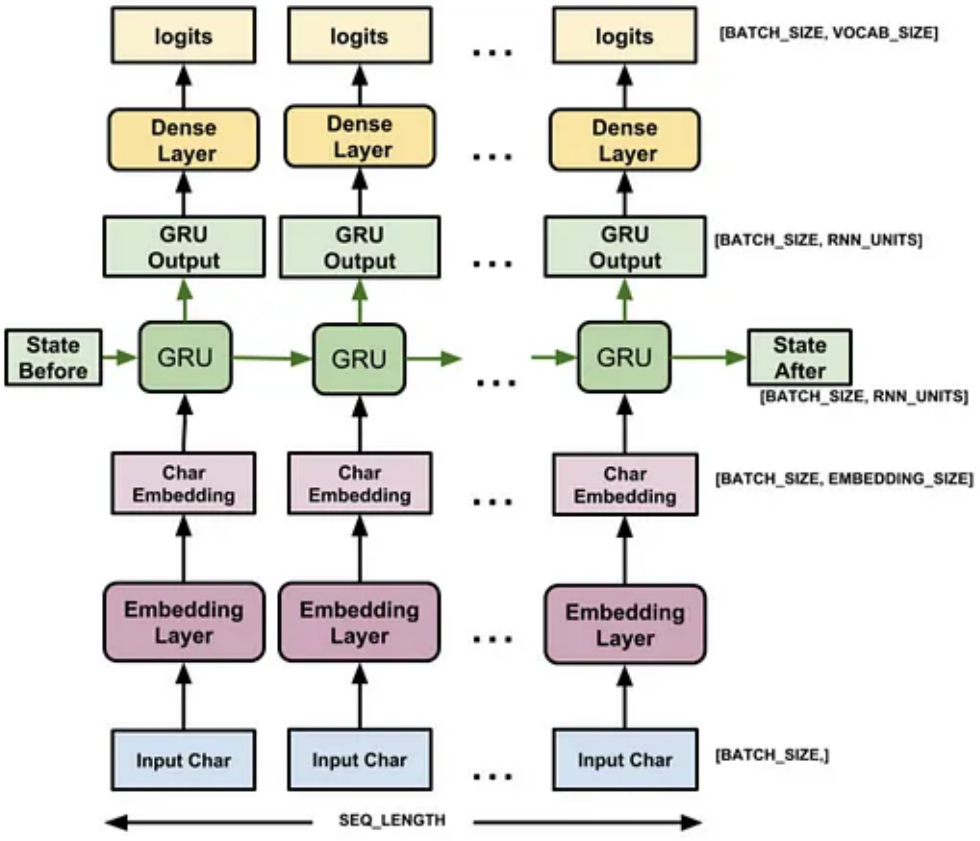
we can use its content as a stream of characters to train a model to write like Shakespeare.

For example, during inference, we start with an initial text, say “ROMEO: ”. The model predicts what is the next character once at a time.

```
print(generate_text(model, start_string=u"ROMEO: "))
```

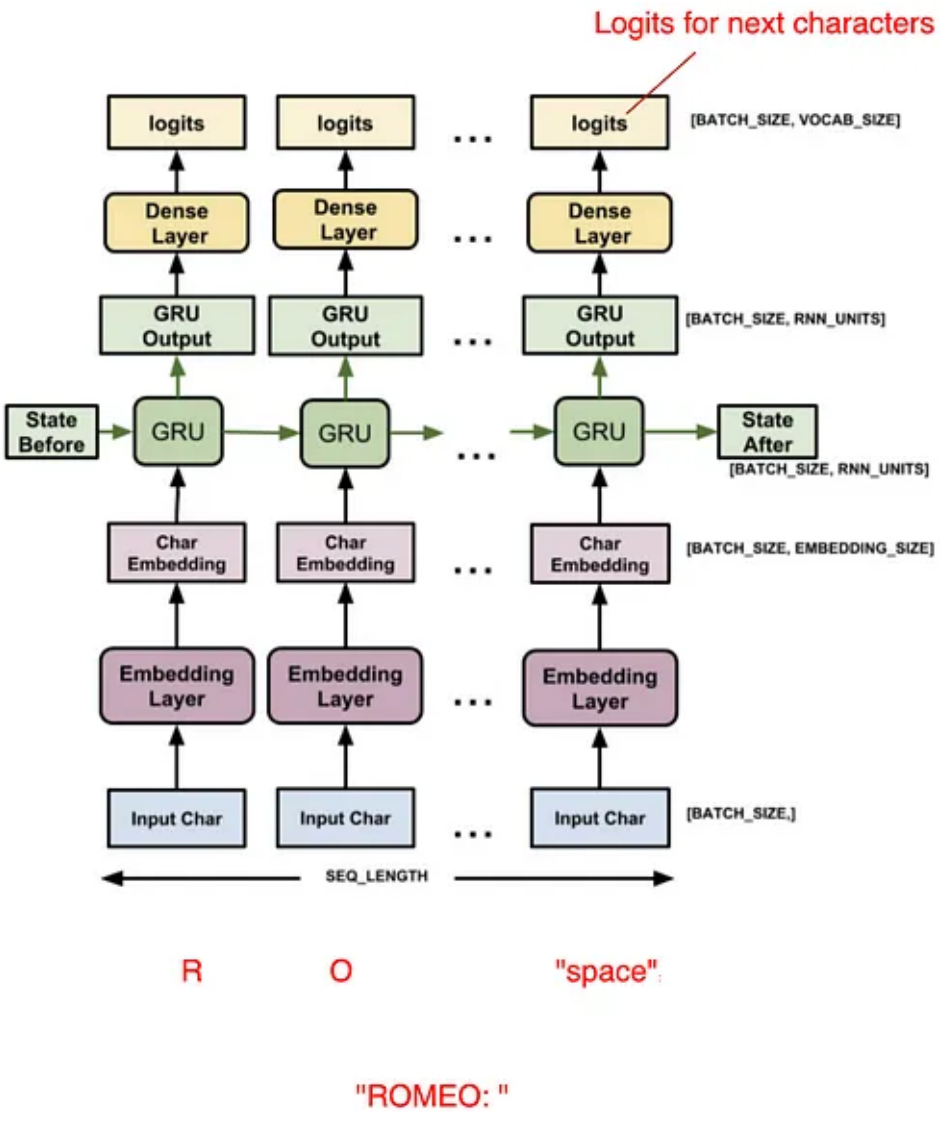
```
ROMEO: ghast I cut go,
Know the normander and the wrong:
To our Morsuis misdress are behiod;
And after as if no other husion.

VALERIS:
Your father and of worms?

LADY GREY:
Your hot can dost.

WARWICK:
Then, atient the bade, truckle aid,
```

This can be done with the GRU model below with embedding and dense layers. The next character is the character predicted in the last GRU cell.

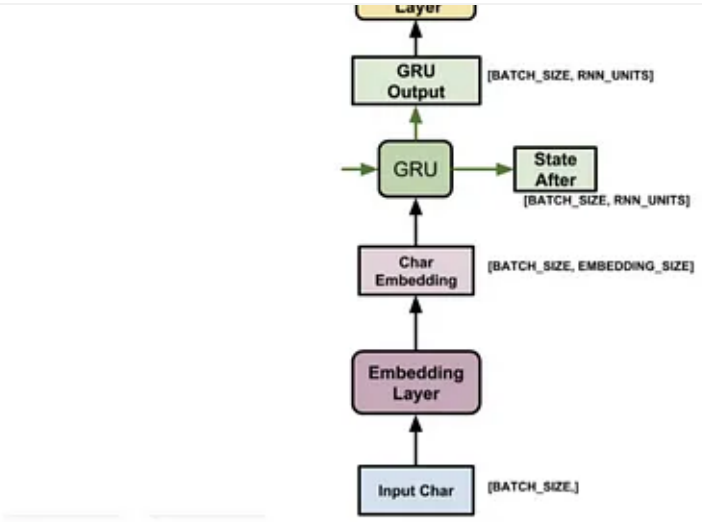


Source (The round rectangles represent operation and the square rectangles represent Tensors.)

Let trace it from the beginning with the initial text. The logits output of the last cell will be used to determine the next character after “ROMEO: “.



We take the generated character as the input to the model again and generates the next character.



As we keep repeating the process, it generates a script similar to the one below. Even though not grammatically sound, it can be improved with more complex designs. For now, we stick with the simple model.

```
ROMEO: ghast I cut go,
Know the normander and the wrong:
To our Morsuis misdress are behiod;
And after as if no other husion.
```

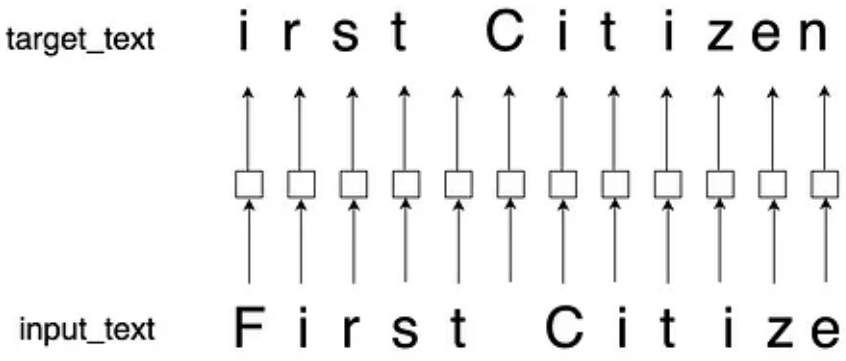
Let's go back to the code. First, we load the Shakespear file and uses its unique characters (65) to form a vocabulary (this vocabulary contains characters instead of words). Then, we create a mapping between a character and an integer index.

```
# Modified from
# https://www.tensorflow.org/tutorials/text/text_generation

import tensorflow as tf
import numpy as np
import os
import time

path_to_file = tf.keras.utils.get_file('shakespeare.txt',
                                       'https://storage.googleapis.com/download.te

# Read, then decode for py2 compat.
# text contains 1115394 characters
text = open(path_to_file, 'rb').read().decode(encoding='utf-8')

# The unique characters in the file
vocab = sorted(set(text))  # 65 unique characters

# Creating a mapping from unique characters to indices
char2idx = {u: i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)
```

And we convert the characters in the downloaded Shakespear text into a long sequence of integer indexes. We slice and batch it to create the dataset "*sequences*". Next, we create a *dataset* with an additional field holding the target text in line 39 as the labels.

```
text_as_int = np.array([char2idx[c] for c in text])

# The maximum length sentence you want for a single input in characters
seq_length = 100
examples_per_epoch = len(text) // (seq_length + 1)

# Create training examples / targets
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

sequences = char_dataset.batch(seq_length + 1, drop_remainder=True)


# Create the target text by left shift of one character
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text


dataset = sequences.map(split_input_target)
```

Given an input character, the corresponding character in the target text (the character we want to predict) should be the next character of the input text. So these labels are simply formed by shifting the input left by one.



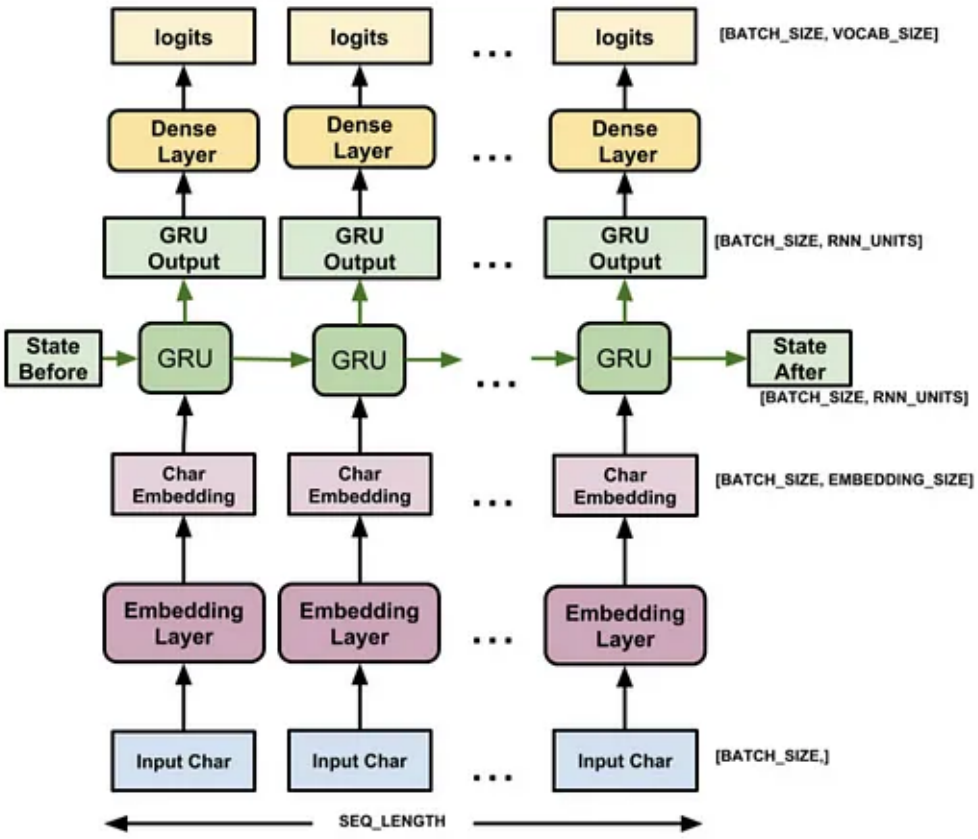
Then, we shuffle and batch the dataset.

```
# Create training batches
# Batch size
BATCH_SIZE = 64

# Buffer size to shuffle the dataset
# (TF data is designed to work with possibly infinite sequences,
# so it doesn't attempt to shuffle the entire sequence in memory. Instead,
# it maintains a buffer in which it shuffles elements).
BUFFER_SIZE = 10000

dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
```

Then we build the model.

```
# Build The Model
# Length of the vocabulary in chars
vocab_size = len(vocab)

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024


def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
    model = tf.keras.Sequential([
        tf.keras.layers.Embedding(vocab_size, embedding_dim,
                                  batch_input_shape=[batch_size, None]),
        tf.keras.layers.GRU(rnn_units,
                            return_sequences=True,
                            stateful=True,
                            recurrent_initializer='glorot_uniform'),
        tf.keras.layers.Dense(vocab_size)
    ])
    return model


model = build_model(
    vocab_size=len(vocab),
    embedding_dim=embedding_dim,
    rnn_units=rnn_units,
    batch_size=BATCH_SIZE)
```

It contains an embedding layer, a GRU, and a dense layer.



Source

And here is a summary of the model.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (64, None, 256)           16640
_________________________________________________________________
gru (GRU)                    (64, None, 1024)          3938304
_________________________________________________________________
dense (Dense)                (64, None, 65)            66625
=================================================================
Total params: 4,021,569
Trainable params: 4,021,569
Non-trainable params: 0
_________________________________________________________________
```

Then, we train the model and create checkpoints.

```
# Train the model
def loss(labels, logits):
    return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)


model.compile(optimizer='adam', loss=loss)

# Directory where the checkpoints will be saved
checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_prefix,
    save_weights_only=True)

EPOCHS = 10

history = model.fit(dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])
```

Next, we reload the model from the checkpoint.

```
model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)
model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

model.build(tf.TensorShape([1, None]))
```

With the model restored, we start generating text in the Shapespear style with the initial text “ROMEO: ”

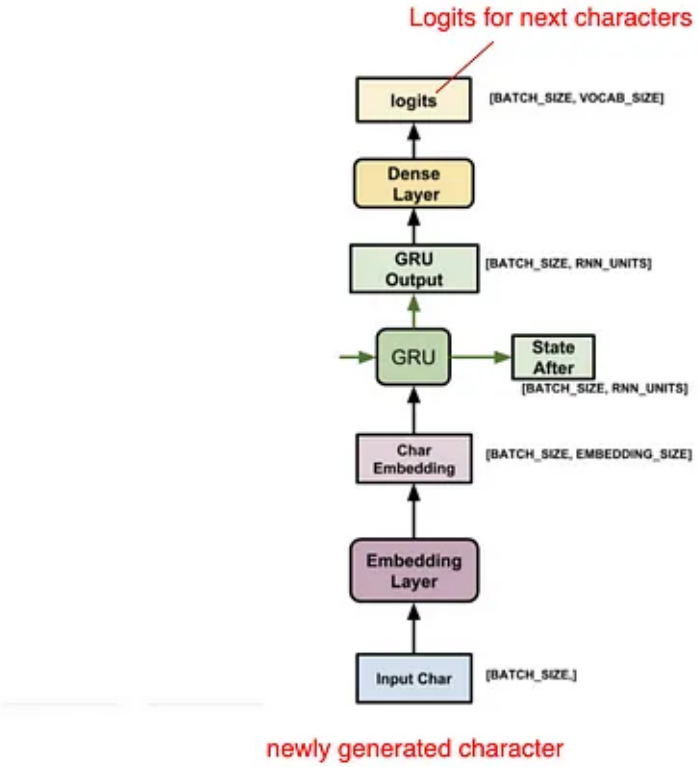
```
113  def generate_text(model, start_string):
114      # Evaluation step (generating text using the learned model)
115
116      # Number of characters to generate
117      num_generate = 1000
118
119      # Converting our start string to numbers (vectorizing)
120      input_eval = [char2idx[s] for s in start_string]
121      input_eval = tf.expand_dims(input_eval, 0)
122
123      # Empty string to store our results
124      text_generated = []
125
126      # Low temperature results in more predictable text.
127      # Higher temperature results in more surprising text.
128      # Experiment to find the best setting.
129      temperature = 1.0
130
131      # Here batch size == 1
132      model.reset_states()
133      for i in range(num_generate):
134          predictions = model(input_eval)
135          # remove the batch dimension
136          predictions = tf.squeeze(predictions, 0)
137
138          # using a categorical distribution to predict the character returned by the model
139          predictions = predictions / temperature
140          predicted_id = tf.random.categorical(predictions, num_samples=1)[-1, 0].numpy()
141
142          # Pass the predicted character as the next input to the model
143          # along with the previous hidden state
144          input_eval = tf.expand_dims([predicted_id], 0)
145
146          text_generated.append(idx2char[predicted_id])
147
148      return (start_string + ''.join(text_generated))
149
150
151  print(generate_text(model, start_string=u"ROMEO: "))
```

We convert this text into the corresponding sequence of integer indexes. Then, we fit it into the model which outputs a Tensor logits with the shape (1, 7, 65). One prediction for each input character in the initial text “ROMEO: ” (batch size, sequence length, vocabulary size).

In line 140, we use *tf.random.categorical* to sample a character prediction in each sequence using the logit values. The output will have the shape of (7, 1). We are only

interested in the last GRU cell (what is next after the last character), so we take the last sampled value [-1, 0] as the next generated character.

For the rest of the "for" loop iterations, we just use the newly generated character as input to the model and predict the next character. We repeat the iterations until 1000 characters are generated.



Seq2seq language translation with attention

This example trains a sequence to sequence (seq2seq) model for Spanish to English translation. First, we download the dataset files. The following is part of the Spanish to English translation file.

```
Go.     Ve.
Go.     Vete.
Go.     Vaya.
Go.     Váyase.
Hi.     Hola.
Run!    ¡Corre!
Run.    Corred.
Who?    ¿Quién?
Fire!   ¡Fuego!
Fire!   ¡Incendio!
Fire!   ¡Disparad!
Help!   ¡Ayuda!
Help!   ¡Socorro! ¡Auxilio!
Help!   ¡Auxilio!
Jump!   ¡Salta!
Jump.   Salte.
Stop!   ¡Parad!
Stop!   ¡Para!
Stop!   ¡Pare!
Wait!   ¡Espera!
Wait.   Esperen.
Go on.  Continúa.
Go on.  Continúe.
Hello!  Hola.
I ran.  Corrí.
I ran.  Corría.
I try.  Lo intento.
I won!  ¡He ganado!
Oh no!  ¡Oh, no!
Relax.  Tomátelo con soda.
Smile.  Sonríe.
Attack! ¡Al ataque!
Attack! ¡Atacad!
Get up. Levanta.
Go now. Ve ahora mismo.
```

```python
# Modified from
# https://www.tensorflow.org/tutorials/text/nmt_with_attention

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

# Download the file
path_to_zip = tf.keras.utils.get_file(
    'spa-eng.zip', origin='http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip',
    extract=True)

path_to_file = os.path.dirname(path_to_zip) + "/spa-eng/spa.txt"
```

Next, we create methods to preprocess a sentence.

```
u"May I borrow this book?"        ——→ <start> may i borrow this book ? <end>
u"¿Puedo tomar prestado este libro?" → b'<start> \xc2\xbf puedo tomar prestado este libro ? <end>'
```

```
# 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())

    # creating a space between a word and the punctuation following it
    # eg: "he is a boy." => "he is a boy ."
    # Reference:- https://stackoverflow.com/questions/3645931/python-padding
    w = re.sub(r"([?.!,¿])", r" \1 ", w)
    w = re.sub(r'[" "]+', " ", w)

    # replacing everything with space except (a-z, A-Z, ".", "?", "!", ",")
    w = re.sub(r"[^a-zA-Z?.!,¿]+", " ", w)

    w = w.strip()

    # adding a start and an end token to the sentence
    # so that the model know when to start and stop predicting.
    w = '<start> ' + w + ' <end>'
    return w
```

Given a translation file, *create_dataset* generates samples for the target text and the original text and *tokenize* converts the text into a sequence of integers.

```
path_to_file = os.path.dirname(path_to_zip)+"/spa-eng/spa.txt"
```

```
en, sp = create_dataset(path_to_file, None)
```

```
# 1. Remove the accents
# 2. Clean the sentences
# 3. Return word pairs in the format: [ENGLISH, SPANISH]
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)


# Convert text into a a sequence of int
# Lang contains 30,000 sentences/phrases
def tokenize(lang):
    lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(
        filters='')
    lang_tokenizer.fit_on_texts(lang)

    tensor = lang_tokenizer.texts_to_sequences(lang)

    # Shape (30000, 16) int32
    tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor,
                                                           padding='post')

    return tensor, lang_tokenizer
```

Now, *load_dataset* put them together in loading source (Spanish) and target (English) text into samples of source and target integer sequences. It also returns the tokenizers used.

```
# Given a translation file containing source and target text,
# return source and target integer sequence and tokenizers
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
```

```
> targ_lang = {tuple: 30000} ('<start> go . <end>', '<start> go . <end>', '<start> go . <end>', '<start> go . <end>',
> targ_lang_tokenizer = {Tokenizer} <keras_preprocessing.text.Tokenizer object at 0x7fb99df8cf70>
> target_tensor = {ndarray: (30000, 11)} [[ 1 36  3 ...  0  0  0], [ 1 36  3 ...  0  0  0], [ 1 36  3 ...  0  0  0],
```

Following is a sample of the integer sequences for the source and the target text.

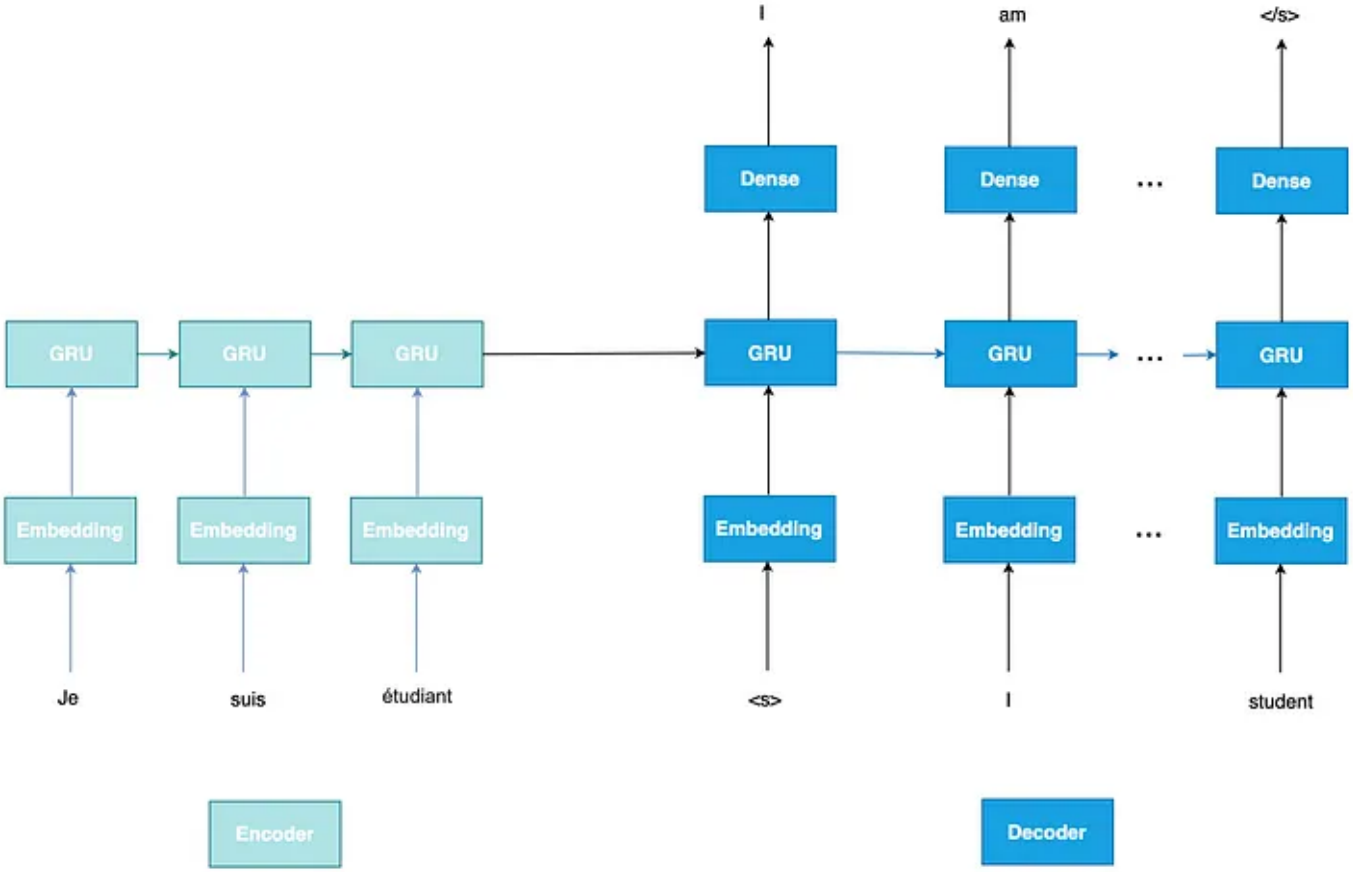
```
Input Language; index to word mapping
1 ----> <start>
6379 ----> dese
395 ----> vuelta
32 ----> ,
22 ----> por
50 ----> favor
3 ----> .
2 ----> <end>
```

```
Target Language; index to word mapping
1 ----> <start>
56 ----> please
205 ----> turn
197 ----> over
3 ----> .
2 ----> <end>
```

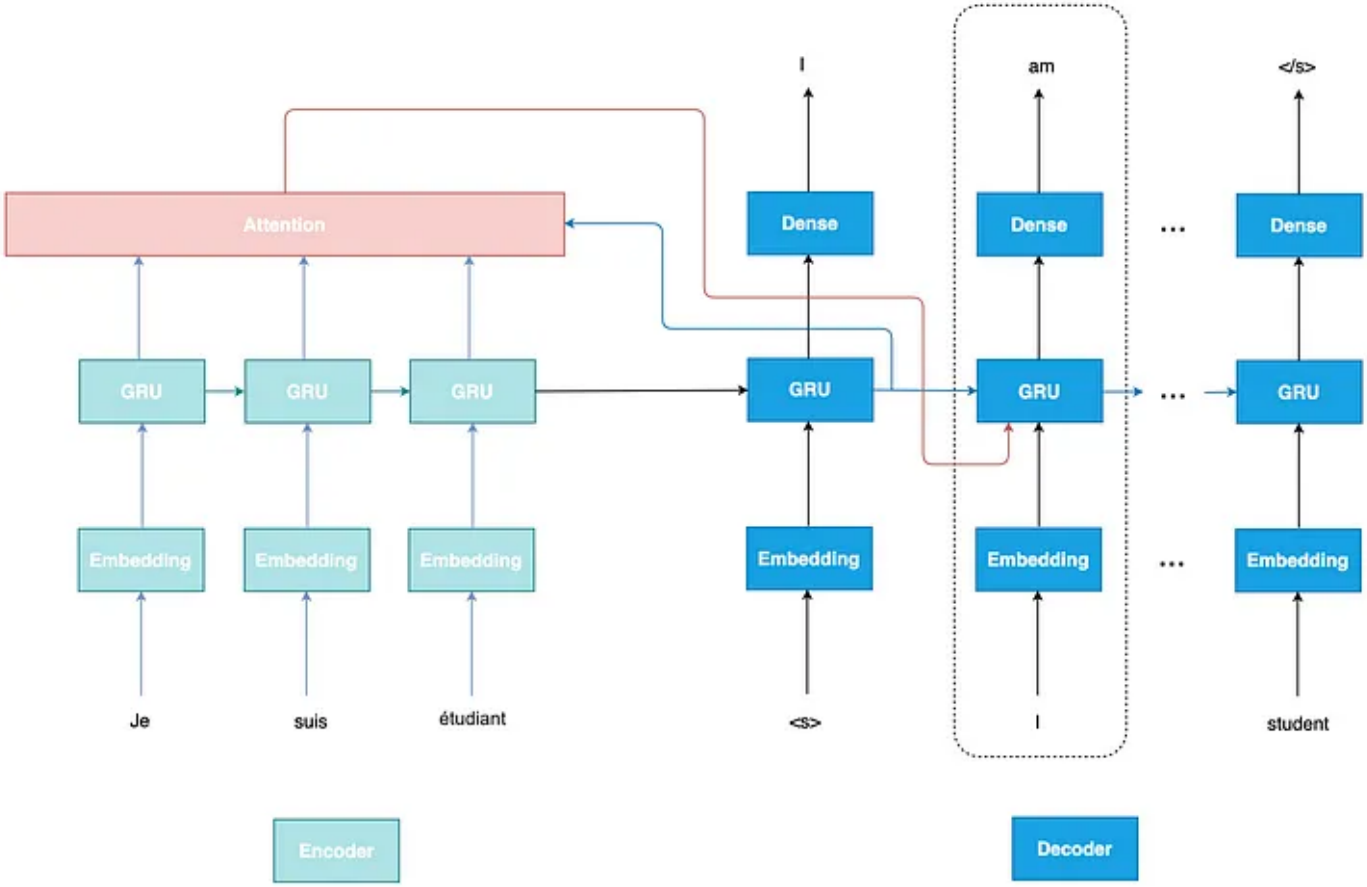
Here is the boilerplate code for creating a smaller dataset for training and validation.

```
# Smaller sample size for faster training
# 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
# 24000 samples for training 6000 for validation
input_tensor_train, input_tensor_val, target_tensor_train, target_tensor_val = \
    train_test_split(input_tensor, target_tensor, test_size=0.2)

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)
```

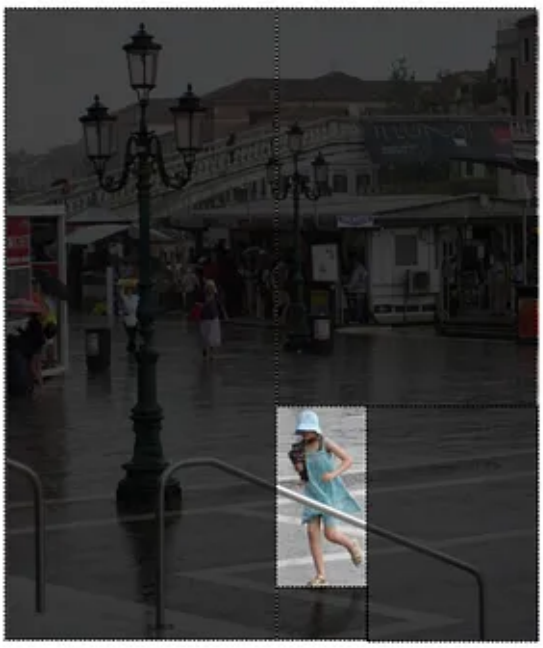
The diagram below contains an encoder to extract the **input context** for “Je suis étudiant” and a decoder to translate this context into “I am a student”.



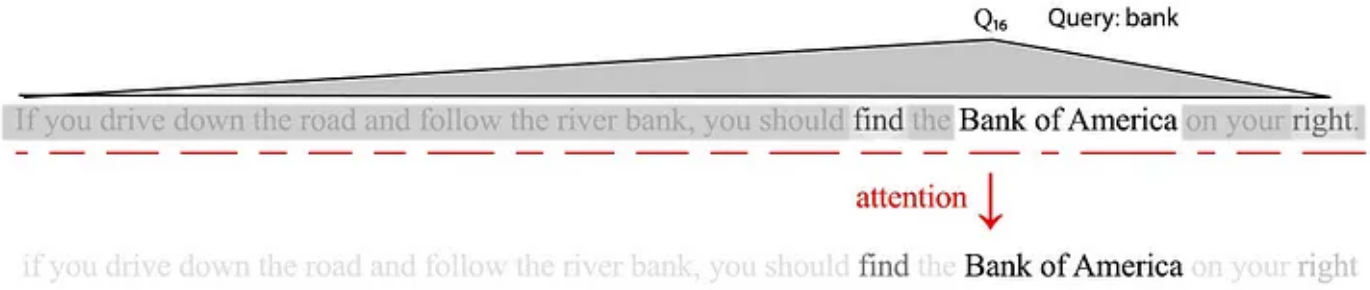
But our example has an additional attention module. Without attention, the decoder predicts the next word based on the predicted word and the GRU hidden states in the last step.



With attention, we also use the input context. But not the complete input context, we use the last GRU hidden states to select the part of the input context that we should pay attention to. In computer vision, we use attention to mask out information that is not important at the current stage.



Here is the conceptual visualization of applying attention in NLP. If we use the 16th word (Bank) as the query, it will pay more attention to the phrase “Bank of America” rather than the “river bank”.

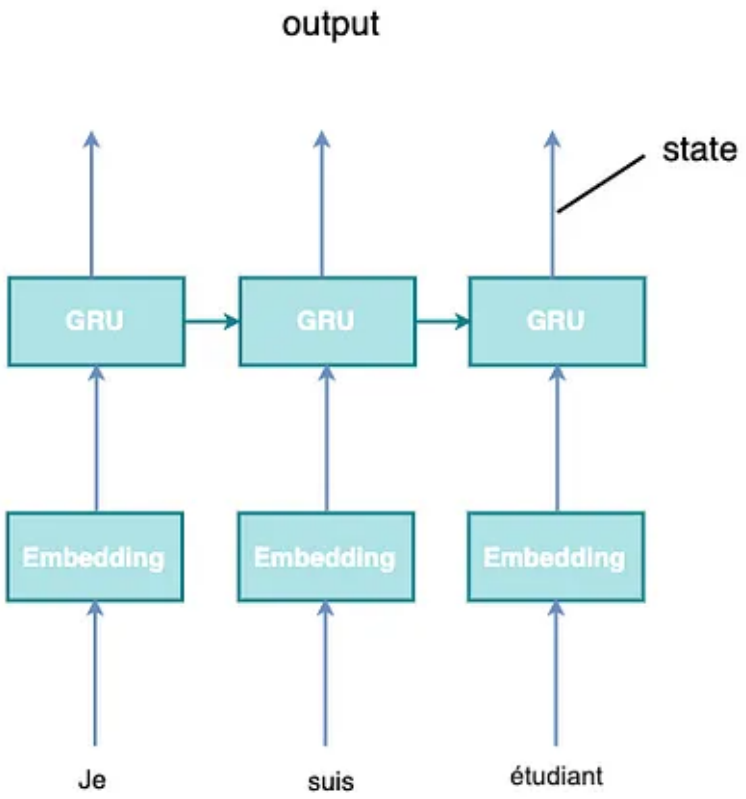


Encoder

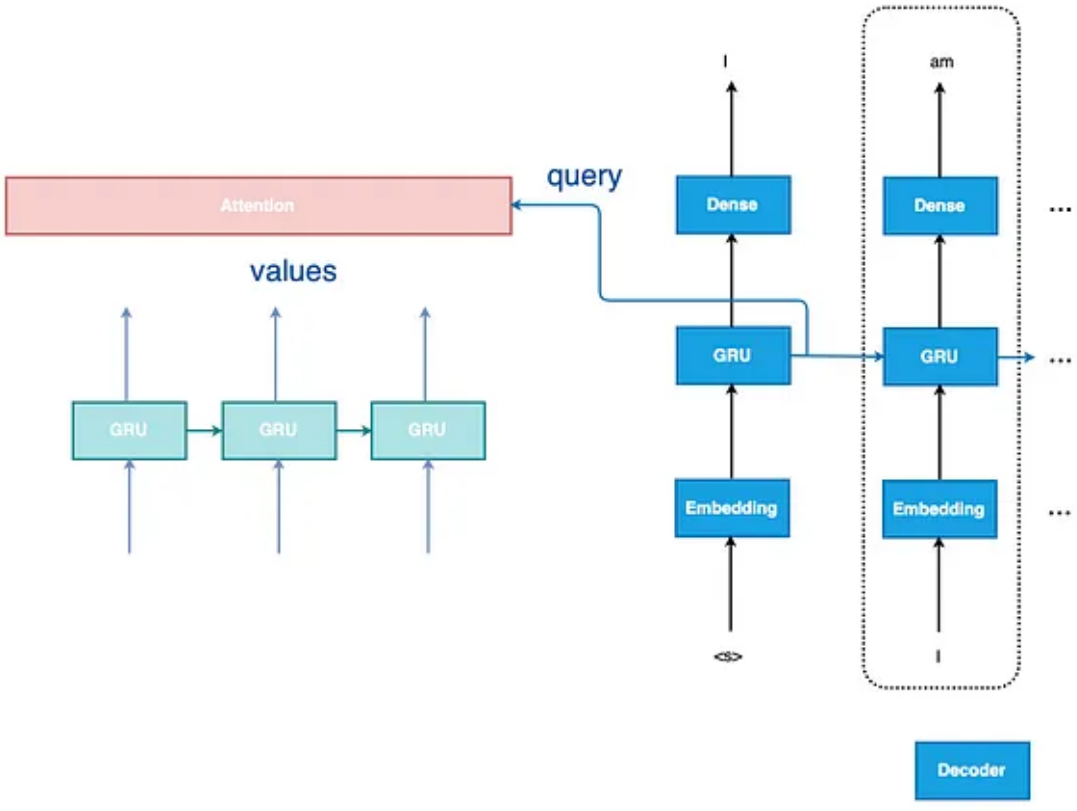
Here is the encoder. It contains an embedding layer and a GRU.

```
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)
```

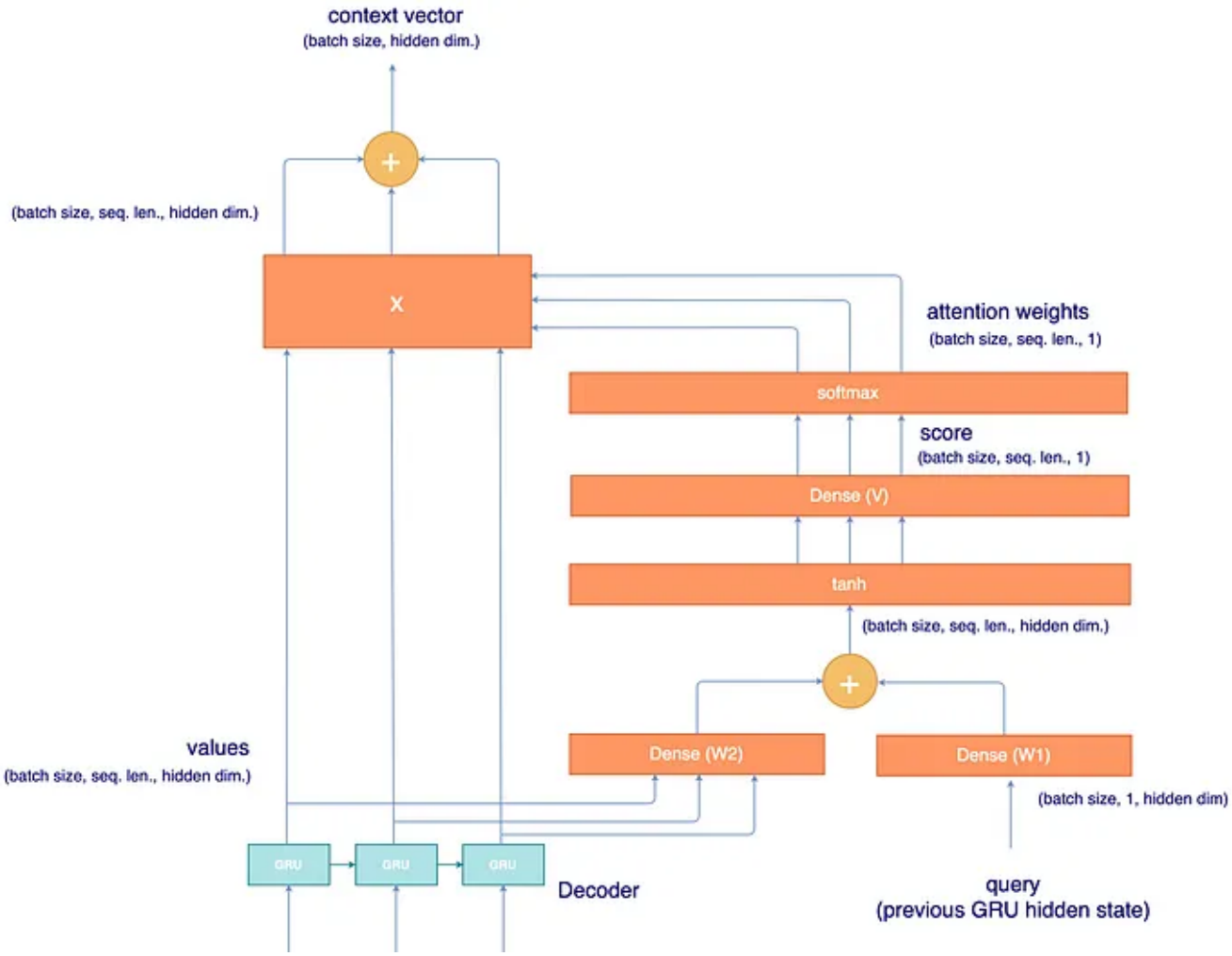
Encoder produces “*output*”, the hidden states of the complete sequence, and “*state*”, the last hidden state.



Let's elaborate further on how attention works. The query is the previous hidden GRU state of the decoder and values are the output of the encoder. The attention keeps the input context that is relevant to the query.



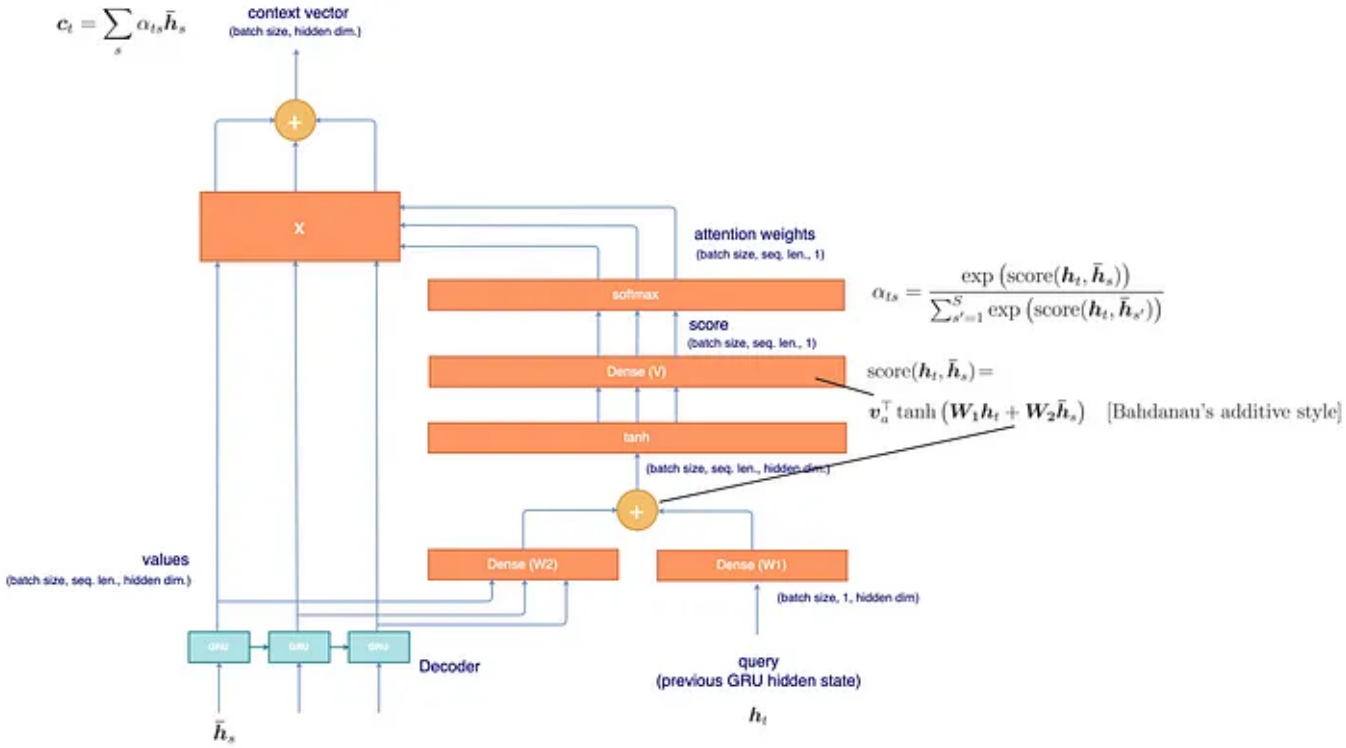
This is the detailed model diagram for the attention module.



"*values*" in our example contains a sequence of 3 integer vectors representing the input "Je suis étudiant". Each vector has "hidden dim." components. We train dense layers (*W1* and *W2*) to suppress the vectors that are not relevant to the query. Its output passes through a *tanh* function and further trained to score each vector with a dense layer *V*. Then, the scores pass through a softmax function. Each vector in the sequence is multiplied with the corresponding softmax result. The model will be trained to zero out irrelevant vectors in the input sequence according to the current query. Finally, we sum up all vectors. The attention output, the context vector, will have the shape (batch size, hidden dimension). Here is the model code for the attention module.

```python
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 hidden state shape == (batch_size, hidden size)
        # query_with_time_axis shape == (batch_size, 1, hidden size)
        # values shape == (batch_size, max_len, hidden size)
        # we are doing this to broadcast addition along the time axis to calculate the score
        query_with_time_axis = tf.expand_dims(query, 1)

        # score shape == (batch_size, max_length, 1)
        # we get 1 at the last axis because we are applying score to self.V
        # the shape of the tensor before applying self.V is (batch_size, max_length, units)
        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)
```

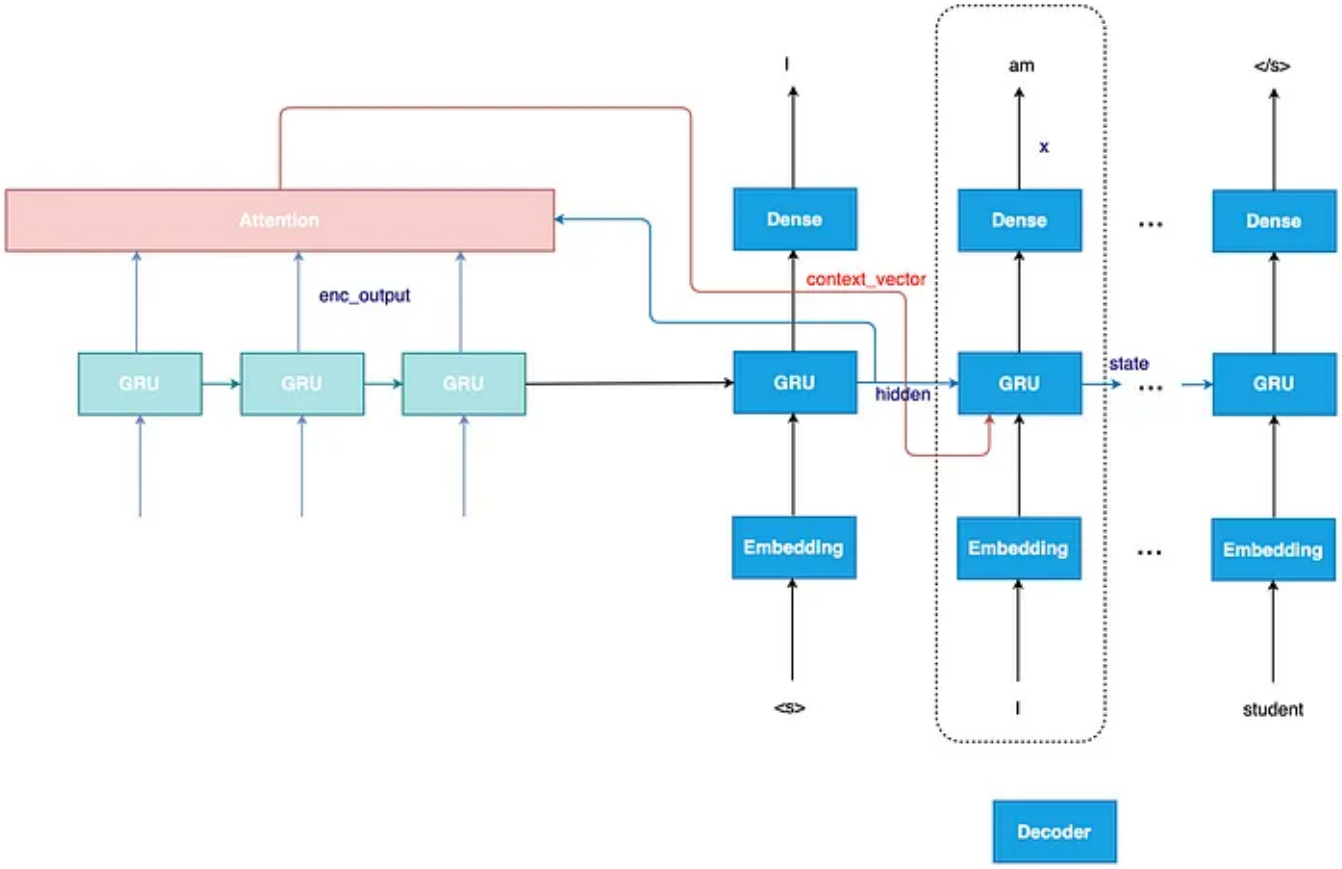
And the equations that we apply.



Finally, this is the decoder.

```python
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)
```

At the second time step in the decoder, here are the notation used in the code above.



Here are the Adam optimizer and the loss function defined.

```
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_)
```

Here is the training step:

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

    with tf.GradientTape() as tape:
        # enc_output: the hidden stats for the whole sequence
        # shape: (batch size, seq. len, hidden dim.)
        # enc_hidden: the last sequence in enc_output
        # shape: (batch size, hidden dim.)
        enc_output, enc_hidden = encoder(inp, enc_hidden)

        # Use the last encoder hidden state to initialize decoder
        dec_hidden = enc_hidden

        # Feed the start token as the first input sequence to the decoder
        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
            # use the target label as the next input
            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
```

And the training itself:

```
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()))

    print('Epoch {} Loss {:.4f}'.format(epoch + 1,
                                        total_loss / steps_per_epoch))
    print('Time taken for 1 epoch {} sec\n'.format(time.time() - start))
```

After the model is trained, we can evaluate the model. The evaluation step is similar to the training, except it uses the previous prediction as input, rather than the target word. When we predict the <end> token, the output is complete for that sample.

```python
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
```

Finally, we plot the weights applied in the attention module when we translate “hace mucho frio aqui”.

```
# 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(' '))
    plt.show()


translate(u'hace mucho frio aqui.')
```

The plot indicates the context where the model focuses on at different stages. At the time when it should predict the word “cold”, the input context at that stage focuses on the word “frio” — cold in Spanish.

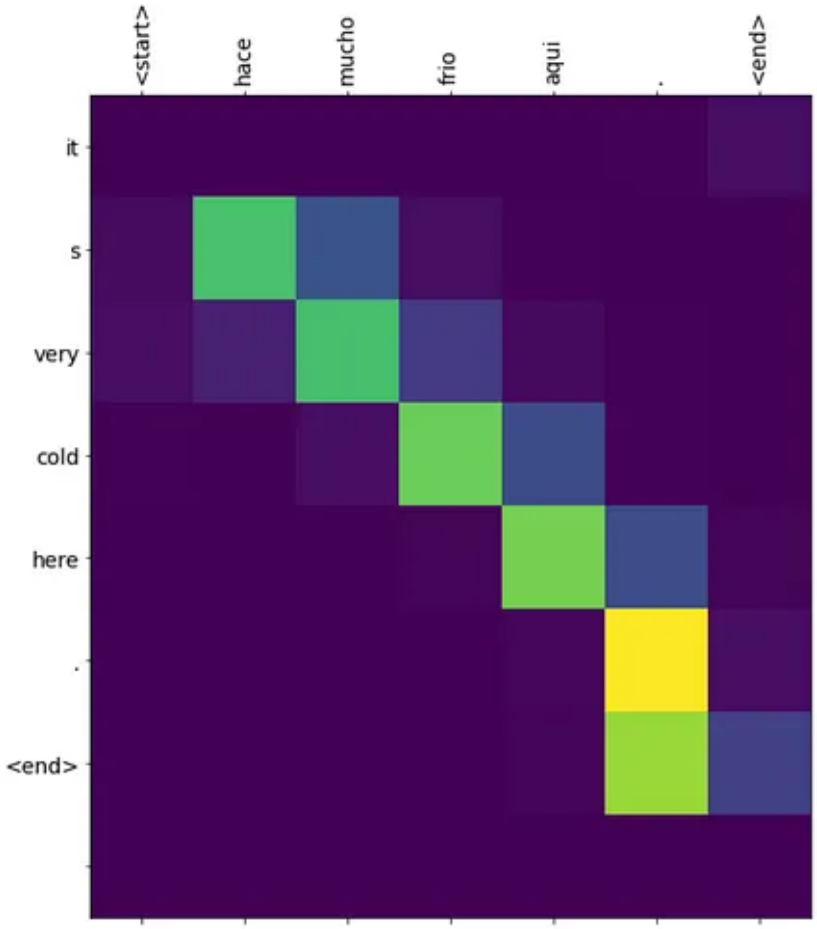
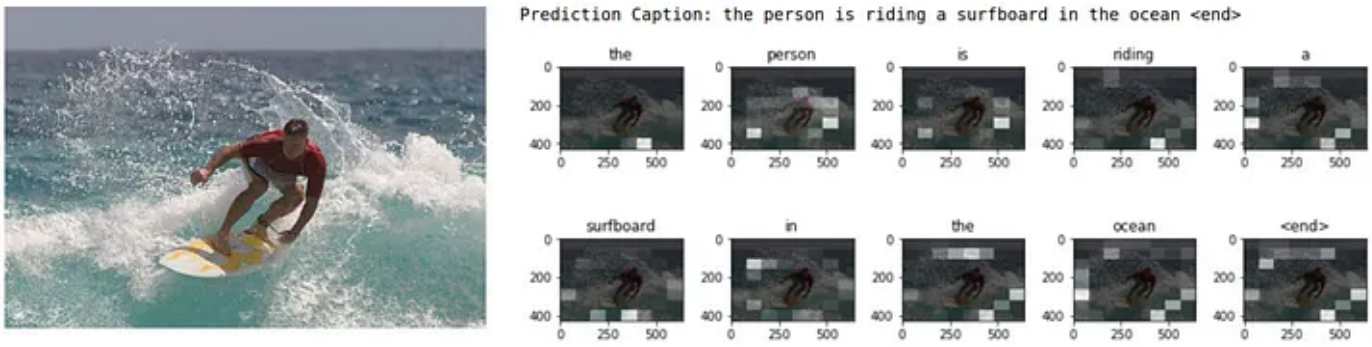


Image captioning with visual attention and Inception V3 for feature extraction

In this example, we generate image caption directly from an image.



Again, we will apply the attention mechanism to pay attention to smaller areas of images in generating the next word. The attention and the decoder module will be similar to the translation example. The major difference is on preprocessing the data. We will use a Inception v3 model to extract image features and save them to files. The

input to the encoder will be these extracted image features. We will go through the code quickly since most parts are similar to the previous translation example.

First, we download the annotation files containing the captions and about 82,000 MSCOCO images (13GB).


```
sky."},{"image_id": 71691,"id": 137454,"caption": "A woman sitting on a bench outside of a building."},{"image_id": 182349,"id": 137465,"caption": "A traffic light in front of a fence next to a forest."},{"image_id": 85657,"id": 137478,"caption": "A herd of
```


```python
# Modified fom
# https://www.tensorflow.org/tutorials/text/image_captioning
import tensorflow as tf

# You'll generate plots of attention in order to see which parts of an image
# our 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 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/anno
                                             extract=True)
    annotation_file = os.path.dirname(annotation_zip) + '/annotations/captions_train2014.json'
    os.remove(annotation_zip)
else:
    annotation_file = os.path.abspath('.') + '/annotations/captions_train2014.json'

# 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.zip',
                                        extract=True)
    PATH = os.path.dirname(image_zip) + image_folder
    os.remove(image_zip)
else:
    PATH = os.path.abspath('.') + image_folder

```

But for faster training, we take 6,000 images only. And prepare a list of about 30,011 captions and a list of 30,011 corresponding file names. Each image can have multiple captions.

```
# Optional: limit the size of the training set
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[:6000]

# Train_captions hold a list of captions.
# img_name_vector holds a list of the corresponding file.
# An image can have multiple captions.
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))
```

We will resize the image and use a pre-built Inception v3 to extract features. Then the features are stored into a file.

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


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)

# 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.experimental.AUTOTUNE).batch(16)

for img, path in image_dataset:
    batch_features = image_features_extract_model(img)
    # flatten: (16, 8, 8, 2048) -> (16, 64, 2048)
    batch_features = tf.reshape(batch_features,
                                (batch_features.shape[0], -1, batch_features.shape[3]))

    # Save the features into a file
    for bf, p in zip(batch_features, path):
        path_of_feature = p.numpy().decode("utf-8")
        np.save(path_of_feature, bf.numpy())
```

Next, we preprocess and tokenize the captions. We will prepare a vocabulary for the top 5,000 words and create the mapping between a word and the word integer index. We also pad all sequences to the same length as the longest one.

```
# Preprocess and tokenize the captions #



# Find the maximum length of any caption in our dataset
def calc_max_length(tensor):
    return max(len(t) for t in tensor)


# Choose the top 5000 words from the vocabulary
top_k = 5000
tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=top_k,
                                                  oov_token="<unk>",
                                                  filters='!"#$%&()*+.,-/:;=?@[\]^_`{|}~ ')
tokenizer.fit_on_texts(train_captions)
train_seqs = tokenizer.texts_to_sequences(train_captions)

tokenizer.word_index['<pad>'] = 0
tokenizer.index_word[0] = '<pad>'

# Create the tokenized vectors
train_seqs = tokenizer.texts_to_sequences(train_captions)

# Pad each vector to the max_length of the captions
# If you do not provide a max_length value, pad_sequences calculates it automatically
cap_vector = tf.keras.preprocessing.sequence.pad_sequences(train_seqs, padding='post')

# Calculates the max_length, which is used to store the attention weights
max_length = calc_max_length(train_seqs)
```

Next, we will split the samples between training and validation.

```
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])
    img_name_val.extend([imgv] * capv_len)
    cap_val.extend(img_to_cap_vector[imgv])
```

And we will create the datasets.

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

BATCH_SIZE = 64
BUFFER_SIZE = 1000
embedding_dim = 256
units = 512
vocab_size = top_k + 1
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.int32]),
                      num_parallel_calls=tf.data.experimental.AUTOTUNE)

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

```

The attention model will be the same as what we already discussed.

```python
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
```

Because our inputs are extracted features already, the encoder is just a simple dense layer followed by ReLU.

```python
class CNN_Encoder(tf.keras.Model):
    # Since you have already extracted the features and dumped it using pickle
    # 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
```

The decoder is similar to the language translation example, with one more dense layer and a couple ReLU layers. It uses an attention layer also.

```python
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))
```

We define the model, an Adam optimizer, the loss function and the CheckpointManager.

```
encoder = CNN_Encoder(embedding_dim)
decoder = RNN_Decoder(embedding_dim, units, vocab_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_path = "./checkpoints/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)

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

In a training step, we encode the extracted image features. Then we use the decoder with the attention to predict one word at a time.

```python
@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([tokenizer.word_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
```

Here is the training loop,

```
EPOCHS = 20

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

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

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

and the evaluation.

```python
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([tokenizer.word_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()
        result.append(tokenizer.index_word[predicted_id])

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

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

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

Credits and References

All the source code is originated or modified from the TensorFlow tutorial.

Machine Learning Artificial Intelligence Deep Learning Software Development

Data Science