Copyright 2018 The TensorFlow Authors.

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

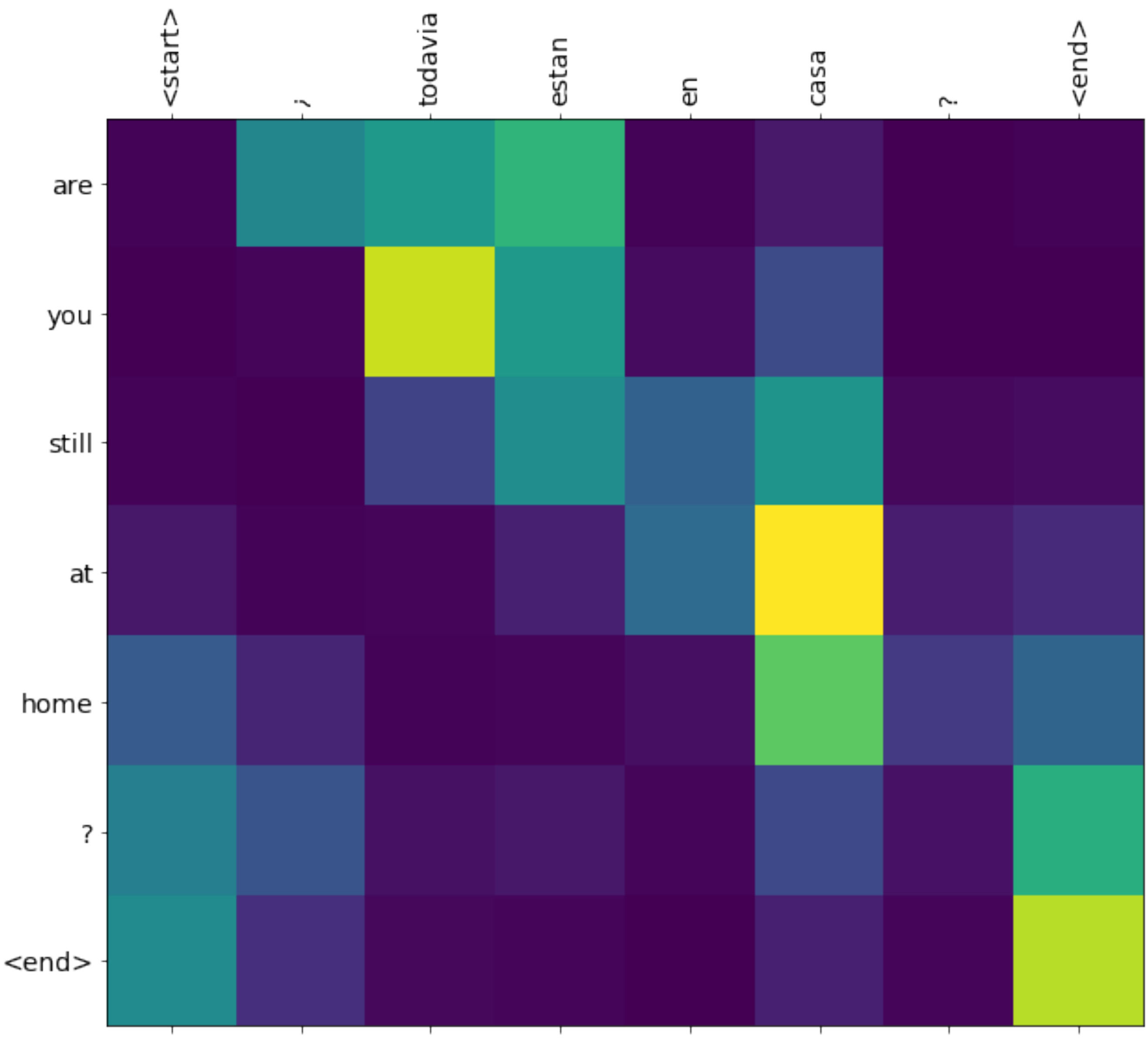
Neural Machine Translation with Attention

[Run in Google Colab]() [View source on GitHub]()

This notebook trains a sequence to sequence (seq2seq) model for Spanish to English translation using [tf.ke]() that assumes some knowledge of sequence to sequence models.

After training the model in this notebook, you will be able to input a Spanish sentence, such as *"¿todavia est *you still at home?"*

The translation quality is reasonable for a toy example, but the generated attention plot is perhaps more inte has the model's attention while translating:



Note: This example takes approximately 10 mintues to run on a single P100 GPU.

```
from __future__ import absolute_import, division, print_function

# Import TensorFlow >= 1.10 and enable eager execution
import tensorflow as tf

tf.enable_eager_execution()

import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

import unicodedata
import re
import numpy as np
import os
import time

print(tf.__version__)
```

```
1.14.0
```

Go to Google Drive to Download French-English Data

After downloading the dataset, here are the steps we'll take to prepare the data:

1. Add a *start* and *end* token to each sentence.
2. Clean the sentences by removing special characters.
3. Create a word index and reverse word index (dictionaries mapping from word → id and id → word).
4. Pad each sentence to a maximum length.

```
path_to_file =  '/content/drive/My Drive/fra.txt'
```

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

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

    # 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-punctuati
    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.rstrip().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
```

```
# 1. Remove the accents
# 2. Clean the sentences
# 3. Return word pairs in the format: [ENGLISH, SPANISH]
def create_dataset(path, num_examples):
    lines = 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

    return word_pairs


# This class creates a word -> index mapping (e.g,. "dad" -> 5) and vice-versa
# (e.g., 5 -> "dad") for each language,
class LanguageIndex():
  def __init__(self, lang):
    self.lang = lang
    self.word2idx = {}
    self.idx2word = {}
    self.vocab = set()

    self.create_index()

  def create_index(self):
    for phrase in self.lang:
      self.vocab.update(phrase.split(' '))

    self.vocab = sorted(self.vocab)

    self.word2idx['<pad>'] = 0
    for index, word in enumerate(self.vocab):
      self.word2idx[word] = index + 1

    for word, index in self.word2idx.items():
      self.idx2word[index] = word
```

```
def max_length(tensor):
    return max(len(t) for t in tensor)


def load_dataset(path, num_examples):
    # creating cleaned input, output pairs
    pairs = create_dataset(path, num_examples)

    # index language using the class defined above
    inp_lang = LanguageIndex(sp for en, sp in pairs)
    targ_lang = LanguageIndex(en for en, sp in pairs)

    # Vectorize the input and target languages

    # Spanish sentences
    input_tensor = [[inp_lang.word2idx[s] for s in sp.split(' ')] for en, sp in pairs]

    # English sentences
    target_tensor = [[targ_lang.word2idx[s] for s in en.split(' ')] for en, sp in pair

    # Calculate max_length of input and output tensor
    # Here, we'll set those to the longest sentence in the dataset
    max_length_inp, max_length_tar = max_length(input_tensor), max_length(target_tensc

    # Padding the input and output tensor to the maximum length
    input_tensor = tf.keras.preprocessing.sequence.pad_sequences(input_tensor,
                                                                 maxlen=max_length_inp
                                                                 padding='post')

    target_tensor = tf.keras.preprocessing.sequence.pad_sequences(target_tensor,
                                                                  maxlen=max_length_ta
                                                                  padding='post')

    return input_tensor, target_tensor, inp_lang, targ_lang, max_length_inp, max_lengt
```

Change the size of the dataset to 35000 from 30000

Training on the complete dataset of >100,000 sentences will take a long time. To train faster, we can limit the translation quality degrades with less data):

```
# Try experimenting with the size of that dataset
num_examples = 35000
input_tensor, target_tensor, inp_lang, targ_lang, max_length_inp, max_length_targ = lc
```

Change Train-Test Split Ratio to 85% : 15%

```
# Creating training and validation sets using an 85-15 split
input_tensor_train, input_tensor_val, target_tensor_train, target_tensor_val = train_t

# Show length
len(input_tensor_train), len(target_tensor_train), len(input_tensor_val), len(target_t
```

```
(29750, 29750, 5250, 5250)
```

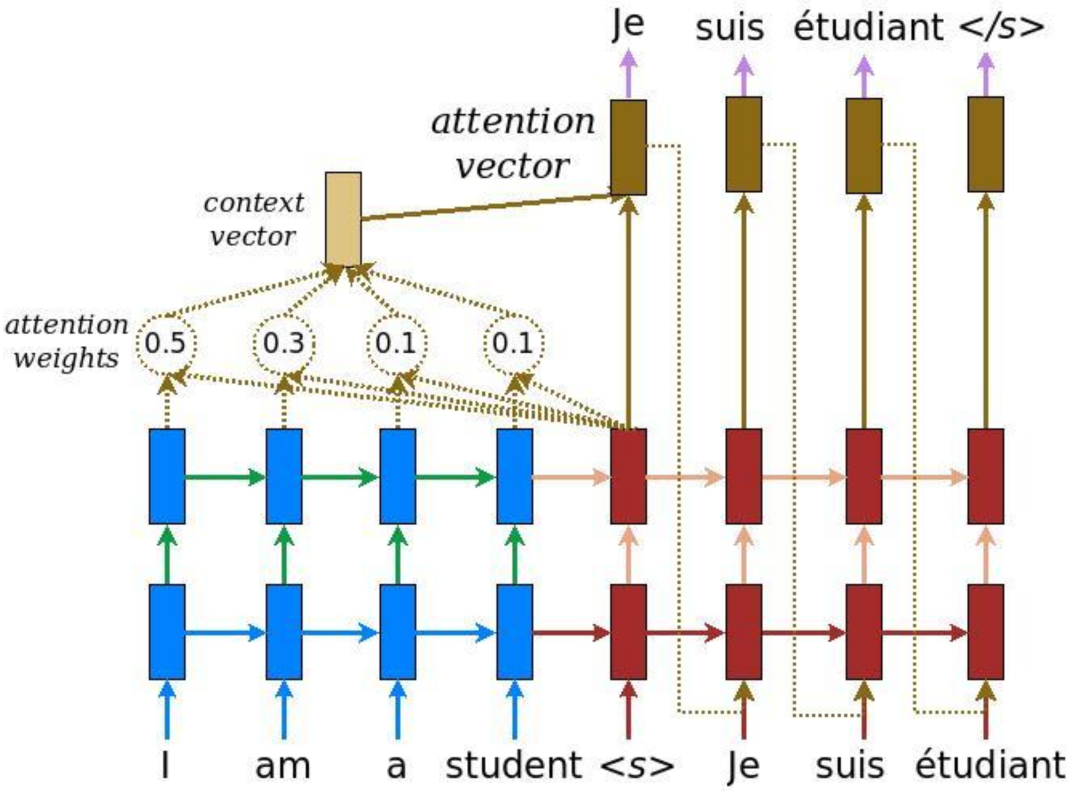
Create a tf.data dataset

```
BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
N_BATCH = BUFFER_SIZE//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(inp_lang.word2idx)
vocab_tar_size = len(targ_lang.word2idx)

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

Write the encoder and decoder model

Here, we'll implement an encoder-decoder model with attention which you can read about in the TensorFlow example uses a more recent set of APIs. This notebook implements the attention equations from the seq2se word is assigned a weight by the attention mechanism which is then used by the decoder to predict the next



The input is put through an encoder model which gives us the encoder output of shape *(batch_size, max_leng* *(batch_size, hidden_size)*.

Here are the equations that are implemented:

$$\alpha_{ts} = \frac{\exp\left(\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \qquad \text{[Attention weights]}$$

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \qquad \text{[Context vector]}$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_{\boldsymbol{c}}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \qquad \text{[Attention vector]}$$

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative st} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_1 \boldsymbol{h}_t + \boldsymbol{W}_2 \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive styl} \end{cases}$$

We're using *Bahdanau attention*. Lets decide on notation before writing the simplified form:

- FC = Fully connected (dense) layer
- EO = Encoder output
- H = hidden state
- X = input to the decoder

And the pseudo-code:

- `score = FC(tanh(FC(EO) + FC(H)))`
- `attention weights = softmax(score, axis = 1)`. Softmax by default is applied on the last the shape of score is *(batch_size, max_length, 1)*. `Max_length` is the length of our input. Since we are be applied on that axis.
- `context vector = sum(attention weights * EO, axis = 1)`. Same reason as above for
- `embedding output` = The input to the decoder X is passed through an embedding layer.
- `merged vector = concat(embedding output, context vector)`
- This merged vector is then given to the GRU

The shapes of all the vectors at each step have been specified in the comments in the code:

```
def gru(units):
  # If you have a GPU, we recommend using CuDNNGRU(provides a 3x speedup than GRU)
  # the code automatically does that.
  if tf.test.is_gpu_available():
    return tf.keras.layers.CuDNNGRU(units,
                                    return_sequences=True,
                                    return_state=True,
                                    recurrent_initializer='glorot_uniform')
  else:
    return tf.keras.layers.GRU(units,
                               return_sequences=True,
                               return_state=True,
                               recurrent_activation='sigmoid',
                               recurrent_initializer='glorot_uniform')
```

```
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 = gru(self.enc_units)

    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))
```

```
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 = gru(self.dec_units)
        self.fc = tf.keras.layers.Dense(vocab_size)

        # used for attention
        self.W1 = tf.keras.layers.Dense(self.dec_units)
        self.W2 = tf.keras.layers.Dense(self.dec_units)
        self.V = tf.keras.layers.Dense(1)

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

        # hidden shape == (batch_size, hidden size)
        # hidden_with_time_axis shape == (batch_size, 1, hidden size)
        # we are doing this to perform addition to calculate the score
        hidden_with_time_axis = tf.expand_dims(hidden, 1)

        # score shape == (batch_size, max_length, 1)
        # we get 1 at the last axis because we are applying tanh(FC(EO) + FC(H)) to se
        score = self.V(tf.nn.tanh(self.W1(enc_output) + self.W2(hidden_with_time_axis)

        # 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 * enc_output
        context_vector = tf.reduce_sum(context_vector, axis=1)

        # 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 * 1, vocab)
        x = self.fc(output)

        return x, state, attention_weights

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


encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)
decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)
```

Define the optimizer and the loss function

```
optimizer = tf.train.AdamOptimizer()


def loss_function(real, pred):
  mask = 1 - np.equal(real, 0)
  loss_ = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=real, logits=pred) * m
  return tf.reduce_mean(loss_)
```

Checkpoints (Object-based saving)

```
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(optimizer=optimizer,
                                 encoder=encoder,
                                 decoder=decoder)
```

Training

1. Pass the *input* through the *encoder* which return *encoder output* and the *encoder hidden state*.
2. The encoder output, encoder hidden state and the decoder input (which is the *start token*) is passed tc
3. The decoder returns the *predictions* and the *decoder hidden state*.
4. The decoder hidden state is then passed back into the model and the predictions are used to calculate
5. Use *teacher forcing* to decide the next input to the decoder.
6. *Teacher forcing* is the technique where the *target word* is passed as the *next input* to the decoder.
7. The final step is to calculate the gradients and apply it to the optimizer and backpropagate.

```
EPOCHS = 10

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

    hidden = encoder.initialize_hidden_state()
    total_loss = 0

    for (batch, (inp, targ)) in enumerate(dataset):
        loss = 0

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

            dec_hidden = enc_hidden

            dec_input = tf.expand_dims([targ_lang.word2idx['<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]))
```

```
            total_loss += batch_loss

            variables = encoder.variables + decoder.variables

            gradients = tape.gradient(loss, variables)

            optimizer.apply_gradients(zip(gradients, variables))

            if batch % 100 == 0:
                print('Epoch {} Batch {} Loss {:.4f}'.format(epoch + 1,
                                                             batch,
                                                             batch_loss.numpy()))
        # saving (checkpoint) the model every 2 epochs
        if (epoch + 1) % 2 == 0:
          checkpoint.save(file_prefix = checkpoint_prefix)

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

```
Epoch 1 Batch 0 Loss 5.0870
Epoch 1 Batch 100 Loss 2.2675
Epoch 1 Batch 200 Loss 1.8775
Epoch 1 Batch 300 Loss 1.7955
Epoch 1 Batch 400 Loss 1.6053
Epoch 1 Loss 2.0394

Time taken for 1 epoch 105.282461643219 sec

Epoch 2 Batch 0 Loss 1.4785
Epoch 2 Batch 100 Loss 1.3554
Epoch 2 Batch 200 Loss 1.2233
Epoch 2 Batch 300 Loss 1.1859
Epoch 2 Batch 400 Loss 0.9822
Epoch 2 Loss 1.2483

Time taken for 1 epoch 102.64092350006104 sec

Epoch 3 Batch 0 Loss 0.9499
Epoch 3 Batch 100 Loss 0.8785
Epoch 3 Batch 200 Loss 0.7825
Epoch 3 Batch 300 Loss 0.7695
Epoch 3 Batch 400 Loss 0.6613
Epoch 3 Loss 0.7940

Time taken for 1 epoch 104.66619086265564 sec

Epoch 4 Batch 0 Loss 0.5707
Epoch 4 Batch 100 Loss 0.5846
Epoch 4 Batch 200 Loss 0.4641
Epoch 4 Batch 300 Loss 0.4479
Epoch 4 Batch 400 Loss 0.4189
Epoch 4 Loss 0.4987

Time taken for 1 epoch 105.64688444137573 sec

Epoch 5 Batch 0 Loss 0.3801
Epoch 5 Batch 100 Loss 0.4011
Epoch 5 Batch 200 Loss 0.2806
Epoch 5 Batch 300 Loss 0.2701
Epoch 5 Batch 400 Loss 0.2836
```

```
Epoch 5 Batch 400 Loss 0.2836
Epoch 5 Loss 0.3242

Time taken for 1 epoch 104.60964703559875 sec

Epoch 6 Batch 0 Loss 0.2556
Epoch 6 Batch 100 Loss 0.2914
Epoch 6 Batch 200 Loss 0.1921
Epoch 6 Batch 300 Loss 0.1655
Epoch 6 Batch 400 Loss 0.2220
Epoch 6 Loss 0.2215

Time taken for 1 epoch 105.12043786048889 sec

Epoch 7 Batch 0 Loss 0.1867
Epoch 7 Batch 100 Loss 0.2190
Epoch 7 Batch 200 Loss 0.1326
Epoch 7 Batch 300 Loss 0.1149
Epoch 7 Batch 400 Loss 0.1566
Epoch 7 Loss 0.1583

Time taken for 1 epoch 104.96051955223083 sec

Epoch 8 Batch 0 Loss 0.1378
Epoch 8 Batch 100 Loss 0.1477
Epoch 8 Batch 200 Loss 0.1126
Epoch 8 Batch 300 Loss 0.0884
Epoch 8 Batch 400 Loss 0.1249
Epoch 8 Loss 0.1213

Time taken for 1 epoch 105.23345017433167 sec

Epoch 9 Batch 0 Loss 0.0983
Epoch 9 Batch 100 Loss 0.1130
Epoch 9 Batch 200 Loss 0.0732
Epoch 9 Batch 300 Loss 0.0600
Epoch 9 Batch 400 Loss 0.0887
Epoch 9 Loss 0.0952

Time taken for 1 epoch 104.50294089317322 sec

Epoch 10 Batch 0 Loss 0.0865
Epoch 10 Batch 100 Loss 0.0861
Epoch 10 Batch 200 Loss 0.0634
Epoch 10 Batch 300 Loss 0.0492
Epoch 10 Batch 400 Loss 0.0636
Epoch 10 Loss 0.0761

Time taken for 1 epoch 105.40462970733643 sec
```

Translate

- The evaluate function is similar to the training loop, except we don't use *teacher forcing* here. The inpu 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*.

Note: The encoder output is calculated only once for one input.

```
def evaluate(sentence, encoder, decoder, inp_lang, targ_lang, max_length_inp, max_leng
    attention_plot = np.zeros((max_length_targ, max_length_inp))

    sentence = preprocess_sentence(sentence)

    inputs = [inp_lang.word2idx[i] for i in sentence.split(' ')]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs], maxlen=max_length
    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.word2idx['<start>']], 0)

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

        # 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.idx2word[predicted_id] + ' '

        if targ_lang.idx2word[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)

    plt.show()
```

```
def translate(sentence, encoder, decoder, inp_lang, targ_lang, max_length_inp, max_len
    result, sentence, attention_plot = evaluate(sentence, encoder, decoder, inp_lang,

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

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

Add a function to report BLEU score with NLTK

```
import nltk
nltk.download('punkt')
from nltk.translate.bleu_score import corpus_bleu
from nltk.translate.bleu_score import SmoothingFunction

def report_bleu(reference, result):
  ref_tokens = nltk.word_tokenize(reference)
  result_tokens = nltk.word_tokenize(result)
  smoothie = SmoothingFunction().method4
  print('Smoothed BLEU Score: {:.4f}'.format(corpus_bleu([ref_tokens],[result_tokens],
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Restore the latest checkpoint and test and report BLEU score

```
# restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

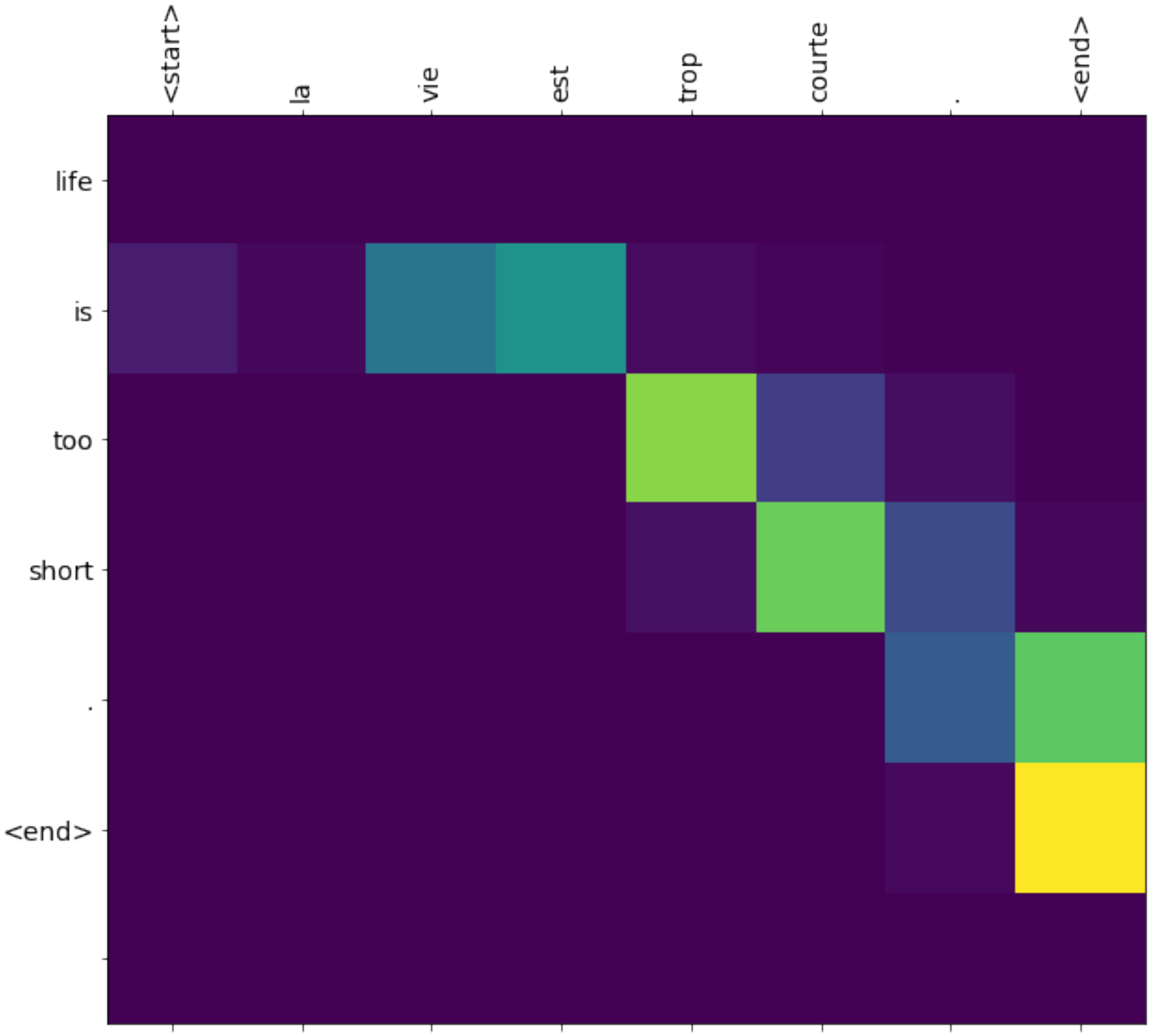
```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7fd9e7c
```

In-Sample Sentences

French Sentence of Length 5

```
result = translate(u'La vie est trop courte.', encoder, decoder, inp_lang, targ_lang,
```

```
Input: <start> la vie est trop courte . <end>
Predicted translation: life is too short . <end>
```




```
reference = 'Life is too short.'
report_bleu(reference, result[:-6])
```

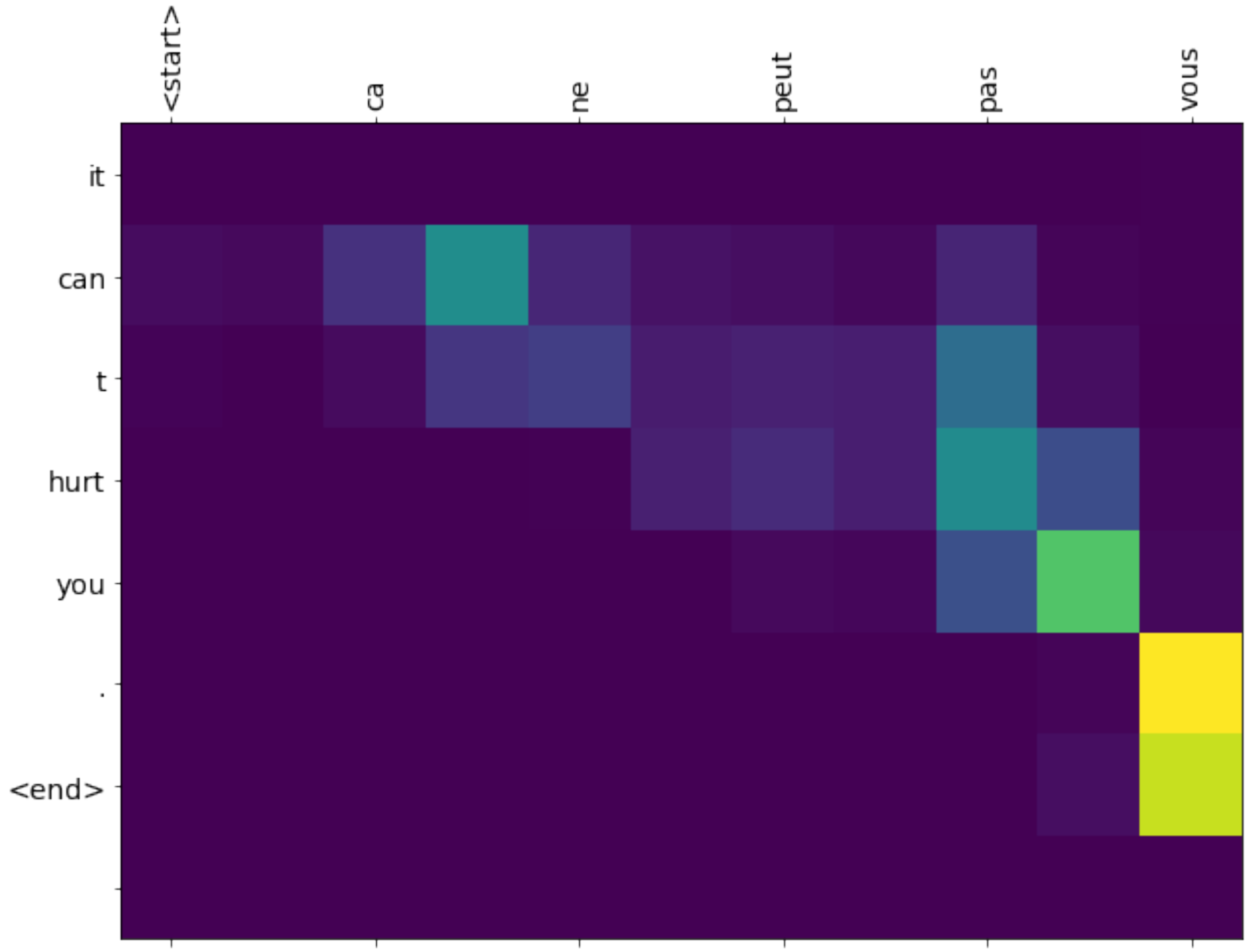
```
Smoothed BLEU Score: 0.1988
```

The heatmap shows a good correlation between verb and verb, adverb and adverb, adjective and adjective in French and Eng sentence is relatively short.

French Sentence of Length 8

```
result = translate(u'Ça ne peut pas vous faire de mal.', encoder, decoder, inp_lang, t
```

```
Input: <start> ca ne peut pas vous faire de mal . <end>
Predicted translation: it can t hurt you . <end>
```




```
reference = "It can't hurt you."
report_bleu(reference, result[:-6])
```

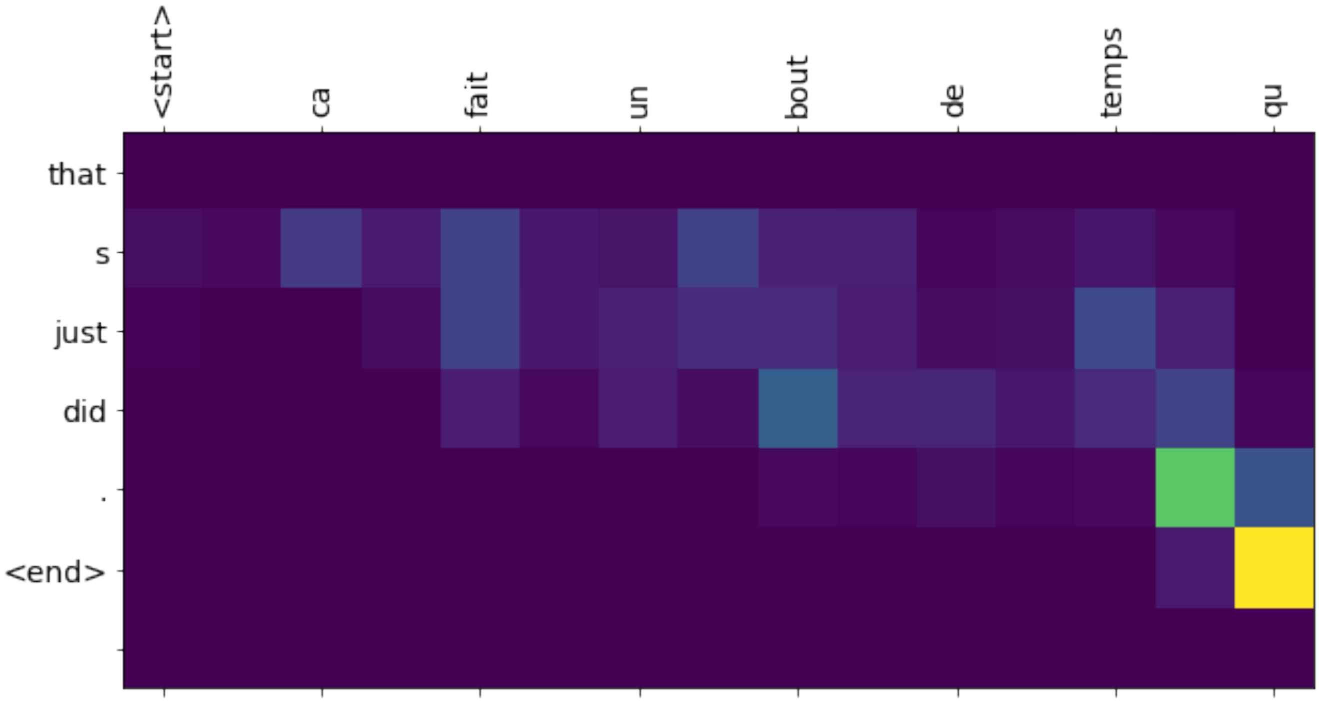
```
Smoothed BLEU Score: 0.2373
```

The negative sentence is hard to translate due to different structure of French and English. While it seems that the attention "it" to "ça", the result is essentially the same as the reference sentence, and the BLEU score captures this fact very well.

French Sentence of Length 10

```
result = translate(u"Ça fait un bout de temps qu'on ne s'est vus.", encoder, decoder,
```

```
Input: <start> ca fait un bout de temps qu on ne s est vus . <end>
Predicted translation: that s just did . <end>
```




```
reference = 'Long time, no see.'
report_bleu(reference, result[:-6])
```

```
Smoothed BLEU Score: 0.2364
```

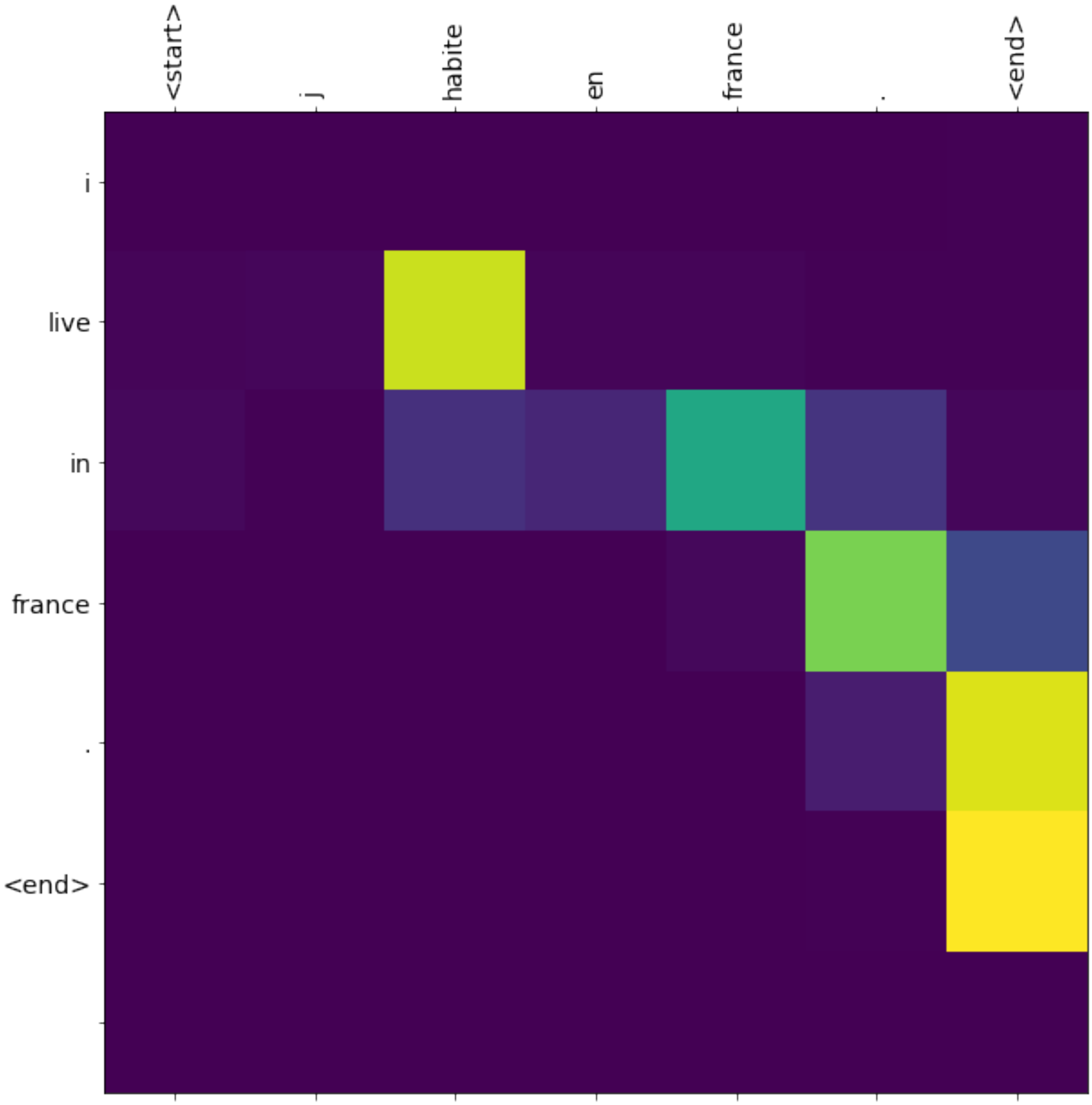
This is a long sentence with complicated structure. The heatmap doesn't tell us much as it becomes really hard to find a one- example, "long time" actually corresponds to "un bout de temps". The BLEU score indicates that the translation is good.

Out-of-Sample Sentences

Good translation

```
result = translate(u"J'habite en France.", encoder, decoder, inp_lang, targ_lang, max_
```

```
Input: <start> j habite en france . <end>
Predicted translation: i live in france . <end>
```




```
reference = 'I live in France.'
report_bleu(reference, result[:-6])
```

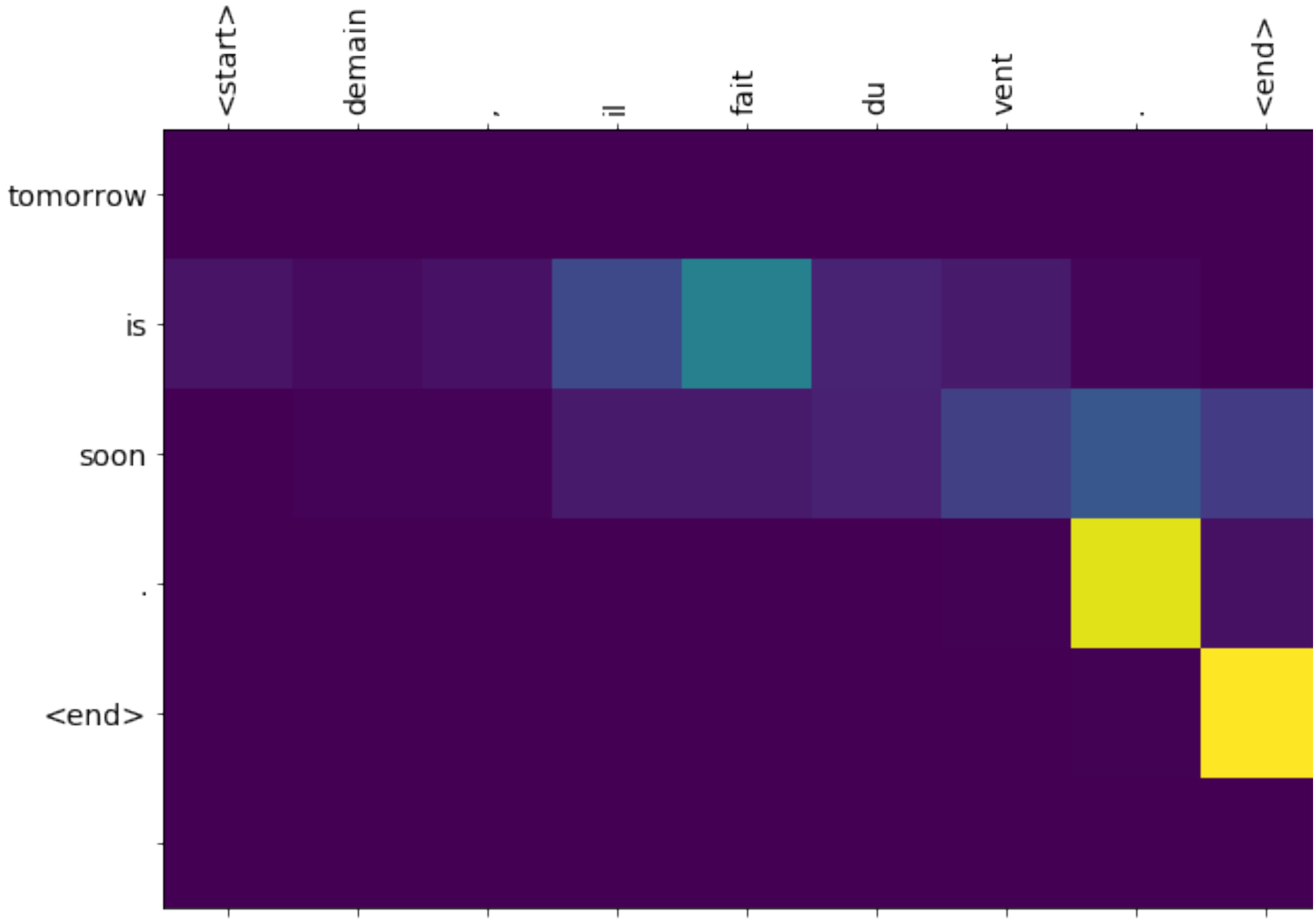
```
Smoothed BLEU Score: 0.2364
```

The most similar sentence to the input out-of-sample sentence in the dataset is "J'habite en ville" , which is good enough for translation. This is an easy one and the model doesn't fail us.

Bad translation

```
result = translate(u"Demain, il fait du vent.", encoder, decoder, inp_lang, targ_lang,
```

```
Input: <start> demain , il fait du vent . <end>
Predicted translation: tomorrow is soon . <end>
```




```
reference = "It'll be windy tomorrow"
report_bleu(reference, result[:-6])
```

```
Smoothed BLEU Score: 0.0000
```

The input sentence is out-of-sample and the model fails us. The original input sentence and the model-translated one doesn other "payday". Plus, the heatmap shows that the model doesn't capture the mapping from "demain" to "tomorrow", which sho quality of the translation and becomes 0 for such a bad translation.

Summary

- While the model performs quite good on in-sample sentences despite their length or complexity, it's nc
- BLEU score is indeed a good measure of the quality of translation.
- The attention heatmap could provide interesting information of attention mechanism when the transla would become increasingly hard to interpret the heatmap when the sentence becomes complex or the