Final Project

Kevin Jairam

CMP 414

Today we'll be replicating the results of an advanced TensorFlow project called Neural machine translation with attention and modifying models.

- Link To Project: https://www.tensorflow.org/tutorials/text/nmt_with_attention#training.
- This project is under the category of *Advanced => Text.*
- **Context of the project:** During this project, we will be building a sequence to sequence model to help train the translations from Spanish to English. The results should be the best possible translation from the language and attention plots that would display the parts of the model's attention.

Basis of a sequence to sequence model:

- This type of model is a way to train differen types of models and converting these sequences from one domain to another. An example of that can be translating between langauges and is very applicable when using machine learning for langauge processing. This also involved a fixed length input and output to help map out the lengths of the phrases and the words.

Inner workings of the Sequence To Sequence Model:

- The following model would consists of 3 parts in order to run its operations properly which is encoder, intermediate and the decoder.

  1) **Encoder**: This would be the input of the process that would consist of the stack of units taking in the single element of the input sequence and collect all of that information to propagate it. The formula that is represented by the encoder layer is the result of the **"ordinary recurrent neural network"** while adding the weights to be applied.

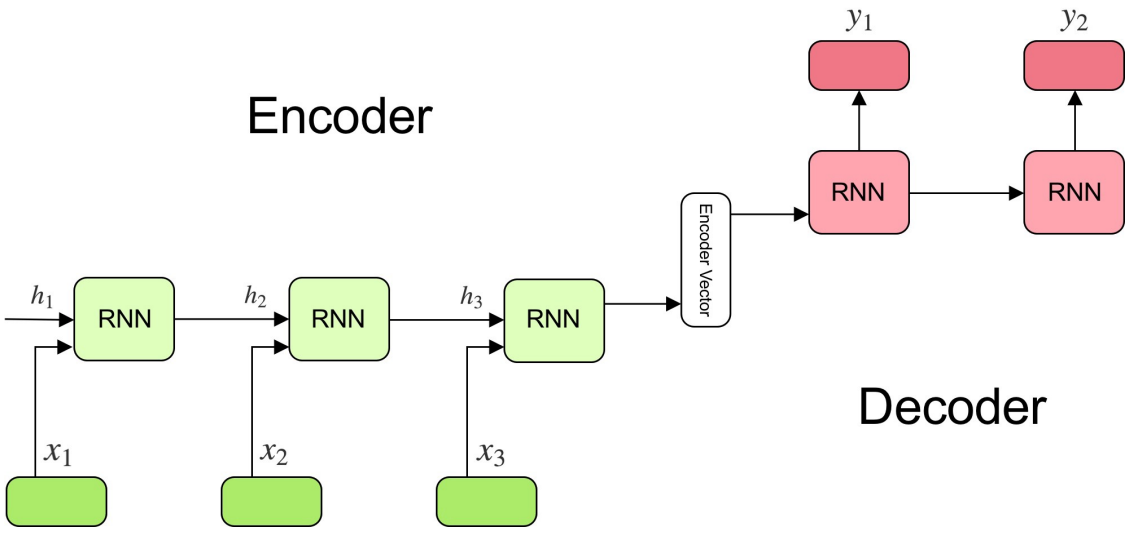
  2) **Intermediate**: This would be also known as the "encoder vector" where it'll represent the final hidden layer of the model. The vector puts up all of the input elements that assists the decoder make accurate predictions.

  3) **Decoder**: This layer would represent the output layer with a stack of units would it would predict the output with a time variable. We would be calculating the outputs using the hidden state with the use of the probability vector.

In [33]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(url= "https://miro.medium.com/max/3972/1*1JcHGUU7rFgtXC_mydUA_Q.jpeg", width=700, height=530)
```

Out[33]:



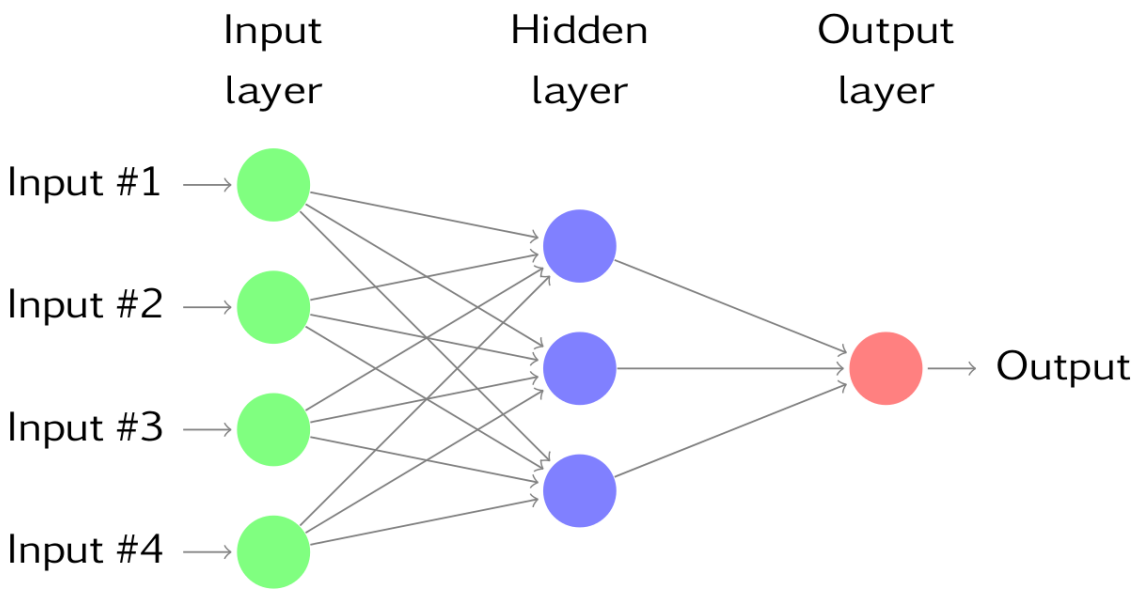
Parallels to Neural Networks:

- The construction of the neural networks are simular in operations to the sequence to sequence with 3 steps to help build out the construction.
- **Input Layer:** Creates a node to read its value.
- **Activation Function/Hidden Layer:** Decides to trigger actions based on weights and computes most of the calculations.
- **Output Layer:** Evaluates the activation function.

In [6]:

```
Image(url= "https://otexts.com/fpp2/nnet2.png", width=700, height=530)
```

Out[6]:



The following Python imports:

- **import tensorflow as tf** : The following is going to import TensorFlow 2 package into our program.
- **import matplotlib.pyplot as plt** : The following will being in the library for data visualization with MATLAB library.
- **import matplotlib.ticker as ticker** : The following module contains complete tick locating and formatting.
- **from sklearn.model_selection import train_test_split** : This will import the module from scikit learn to help split arrays or matrices into random train & test subsets.
- **import unicodedata** : module provides access to the Unicode Character Database which helps defines the character properties.
- **import re** : module for regular expression matching operations.
- **import numpy as np** : module to help initialize numpy arrays.
- **import os** : module to use operating system dependent functionality.
- **import io** : module to deal with various types of I/O.
- **import time** : module to deal with time access and conversions.

In [21]:

```
import tensorflow as tf

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from sklearn.model_selection import train_test_split
import pandas as pd

import unicodedata
import re
import numpy as np
import os
import io
import time
import string
```

Data Preperation

We currently have a variable called **path_to_zip** and this would equate to an operation in TensorFlow (technically in Keras but it's a now a subset of TensorFlow) would download the file from the URL if it's not in the cache. The **spa-eng.zip** file is a two part text document where we'll be the **about** text document and the **spa** text document where it has the many English to Spanish translation pairings. The **origin** is the original URL of the file and the **extract** is True to help extract the files from the tar and/or zip. The **os.path.dirname()** is an O.S. module that helps interact with the operating system and will concat the end of the file we're extracting the information from.

In [39]:

```
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"
```

Decoding and processing the words/sentences:

- **Function of unicode_to_ascii() :** The following function will help return the ascii value from the unicode file and helping transform the text into something more readable for Python.
- **Function of preprocess_sentence() :** The following is a function that helps takes the **unicode_to_ascii** function to help convert them to lowercase and strips away the leading or trailing spaces within the string.
  - Using the regular expression **(re_sub)** helps match the puncuation characters that is surrounded by the spaces and return a space after the desried characters.
  - The last piece helps us add the start and end token to return a more modified sentence that will dictate when to begin and end with the predictions.

In [4]:

```
# 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-punctuation-with-white-
spaces-keeping-punctuation
  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
```

Right now we'll be using a two sentences; one in English and one in Spanish. This will then print out the modified sentence where any trailing spaces or capitalization will be gone from the **preprocess_sentence** function call. The encode will help use the 8-bit value encodings.

In [5]:

```
en_sentence = u"May I borrow this book?"
sp_sentence = u"¿Puedo tomar prestado este libro?"
print(preprocess_sentence(en_sentence))
print(preprocess_sentence(sp_sentence).encode('utf-8'))
```

```
<start> may i borrow this book ? <end>
b'<start> \xc2\xbf puedo tomar prestado este libro ? <end>'
```

The following function **create_dataset** will help remove the accents and clean up the sentences. This will create a new dataset that will help the Python interface with stream handeling and start to pair up the word/sentence pairs from English to Spanish. This will then return the word pairings for our new and revised dataset.

In [6]:

```
# 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)
```

We will now access the new dataset and print out the sentences and word pairing in English and Spanish. It's using index of **-1** which will access the last word pairing and help lower case the characters and create spaces in specific charcaters containing the **start** and **end** token.

In [7]:

```
en, sp = create_dataset(path_to_file, None)
print(en[-1])
print(sp[-1])
```

```
<start> if you want to sound like a native speaker , you must be willing to practice saying the sa
me sentence over and over in the same way that banjo players practice the same phrase over and ove
r until they can play it correctly and at the desired tempo . <end>
<start> si quieres sonar como un hablante nativo , debes estar dispuesto a practicar diciendo la m
isma frase una y otra vez de la misma manera en que un musico de banjo practica el mismo fraseo un
a y otra vez hasta que lo puedan tocar correctamente y en el tiempo esperado . <end>
```

The function **tokenize** will use the text tokenization utility class called **tf.keras.preprocessing.text.Tokenizer** where it'll help us vectorize the text corpus that can convert to a sequence of integers and/or a traslation of binary values to make it easier to read from Python and Keras. This class import will set an argument to update the internal vocabulary from the lists of texts and assuming that each entry of the lists will be the begginning anf ending token to follow through out translated conversions. The **tf.keras.preprocessing.sequence.pad_sequences** will help pad up the sequences to the same length to help us convert without confusing unequal amount of characters with the qrgument of padding that helps us with pads after each sequence. Finally it'll return the tensor and lang_tokenizer which we'll use with our **tokenize** function.

In [8]:

```
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
```

The function **load_dataset** will help cleaned and revised input and output value pairs to help translate. This is all after padding the sentences and created word indexes when we start to access these different phrases with dictionary mapping. This will return the tokenized values of the input_tensor, target_tensor, inp_lang_tokenizer, and targ_lang_tokenizer.

In [9]:

```
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
```

Limiting Dataset:

We'll be finding a way to limit the sample sizes of the dataset to help us train the portion of translation pairs. Since the dataset is a little over 100 thousand sentences, we can reduce to a number like 30,000 pairs to help save time and space. The number can aslways be modified into our new dataset and with the new number of examples. The second portion will help us calculate the max length with the shape of the **target_tensor** and **input_tensor**.

In [10]:

```
# 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]
```

This is where the training gets into play and use validation sets that helps us use the 80/20 train/test split. The **train_test_split** function will help us get the argument of the test_size being 0.2. This will indicate the percentage of the data that will be held over the testing and in this case, it'll be in the range of 80/20. The length will now print out the length of the input_tensor_train, input_tensor_val, target_tensor_train, and the target_tensor_val values to see how much we'll be testing/training for our dataset.

In [11]:

```
# 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(inp
ut_tensor, target_tensor, test_size=0.2)

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

```
24000 24000 6000 6000
```

The function called **convert** will have a for loop for each time it'll iterate through the dataset to access the index of the word and the key pair value that is correlating to it.

In [12]:

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

We are now calling the function **convert** to test out if it does work. From the following output, it looks to be working fine by accessing each set of characters from the index that it was called from and first displaying the index to word mapping with the Spanish version first and then the Engish version. It's good to point out that the index of **1** maps to the **start** token and the index of **2** maps to the **end** token for future reference.

In [13]:

```
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>
18 ----> lo
5366 ----> logramos
27 ----> !
```

```
2 ----> <end>

Target Language; index to word mapping
1 ----> <start>
16 ----> we
2463 ----> succeeded
37 ----> !
2 ----> <end>
```

We're now going to embed our data within TensorFlow. We start to set up the **BUFFER_SIZE** and the **steps_per_epoch** which will be the length of the **input_tensor_train**. The **BACTH_SIZE** will be 64, **embedding_dim** will be 265 and the **units** will be 1024. The **vocab_inp_size** and the **vocab_tar_size** will be the size of their respected length of the index plus one.The **dataset** will now be initialized to the embedded TensroFlow version of the data and set up the size of our set.

In [14]:

```
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(BU
FFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)
```

The following will run the shape of the current state of the dataset with TensorFlow embedded with it.

In [15]:

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

Out[15]:

```
(TensorShape([64, 16]), TensorShape([64, 11]))
```

Writing The Encoder Model:

The Encoder Model will help retrieve any information that is fed as input into the multi-layered RNN's that may have weights. This is just represented as the input for the model so far.

These are the equation implementations:

In [10]:

```
Image(url= "https://www.tensorflow.org/images/seq2seq/attention_equation_0.jpg", width=700, height
=730)
```

Out[10]:

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

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \quad [\text{Context vector}] \quad (2)$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_{\boldsymbol{c}}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad [\text{Attention vector}] \quad (3)$$

In [11]:

```
Image(url= "https://www.tensorflow.org/images/seq2seq/attention_equation_1.jpg", width=700, height
=730)
```

Out[11]:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases} \quad (4)$$

- The **attention weights** will be applied with the Softmax activiation funtion to help gain the attention weights from the input sequence. When using this specific equation, we will have higher attention weight from the input which would correlate to a higher influence in predicting the target word from the translations.
- The **context vector** will help compute the output from the decoder model. This vector is comprised of the weighted sum of the attention weights and the encoder hidden states will map us to the input sentence.
- The **attention vector** is the conversion of the attention weights to use as a vector to help us preciting our target word pairing.
- **Luong multiplicative style** otherwise known as the "Luong attention mechanism" which will help us reduce the encoder states with the decoder state going into the attention scores using matrix multiplication. Under this umbrella, there will be two types of attention which is global and local attention.
  - Global Attention: A model that considers all the hidden states of the encoder when calculating the context vector.
  - Local Attention: A focus on a small subset of source positions per target words unlike the entire source sequence as in global attention.
- **Bahdanau additive style** is also known as the "Bahdanau attention mechanism" where it'll be a mechanism that will learn to translate with the additive attention to perform different linear combination of encoder and decoder states.

Implementation of Encoder Model

Since we have out TensorFlow data ready and to be trained with the sequence to sequence model, we are going to make our **Encoder** class using the **tf.keras.Model**. This will then create a constructor with the **vocab_size**, **units** and **batch_size**. The **super** class will help access the properties of the base class with its inheritance. The **tf.keras.layers.GRU** is from the Gated Recurrent Unit and has multiple arguments with the **tf.keras.layers.Embedding** helping us turn the positive ints into more dense vectors. Then under this class, we made the **call** and **initialize_hidden_state** functions. The encoder will now take into the input and and the encoder model and will spew out the shape with the batch size, hidden state and the length.

In [16]:

```
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))
```

We are now using a sample input where we'll be using the Encoder class for the vocabulary, units and the batch size. We are going to ensure the size and shape of the encoder output shape so we can determining how much we'll be processing and the next level of the hidden layers to gain that shape.

In [17]:

```
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, 16, 1024)
Encoder Hidden state shape: (batch size, units) (64, 1024)
```

For the following model, we'll be using the **Bahdanau additive style** with the **tf.keras.layers.Layer** where it'll contain a constructor for the class named **BahdanauAttention**. We are accessing the queries for the hidden state shape and time axis shape with the shape of the values. We must query this type of information in **call** in order to add along the time axis to help calculate our predictive score. Until we get 1 from the last axis and we will then apply that score into the **self.V** and the shape of the tensor will apply to that vector. We will have the attention weights to get the predictive score and get the context vector when we get the shape and providing ourselves witht the sum.

In [18]:

```
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
```

We are now getting the results from the **BahdanauAttention** class and additive style. From the computations, we will be able to format the attention result and weights shape to format which is much different from the sample inputs of the Encoder model.

In [19]:

```
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.shap
e))
```

```
Attention result shape: (batch size, units) (64, 1024)
Attention weights shape: (batch_size, sequence_length, 1) (64, 16, 1)
```

Building out Decoder class:

Now since we're in the decoder model for the whole model, we will now start getting the output. For the attention portion for calucting our output, we'll be using the attention layer that we helped calculate in the **BahdanauAttention** class. We will now get the **enc_output** shape and get the **x** shape to pass the embedding. And after concatenation of the **x** shape, we will now pass the concatenated vector through the GRU and produce the output shape from it.

In [20]:

```
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
```

The output from the Decoder model is now set and ready for the new **batch_size** and **vocab size**. Using **tf.random.uniform** to output the random values from an uniform distribution, we will now output the decoder output shape is its new format.

In [22]:

```
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, 4935)
```

Defining the optimizer/loss function:

The following optimizer compromises with the **tf.keras.optimizers.Adam** that inherits from the Optimizer that helps implements the Adam algorithm. For the loss funtion, we'll be using **tf.keras.losses.SparseCategoricalCrossentropy** that helps us compute the crossentropy loss between the labels and predictions. The **loss_function** function will help us compute the TensorFlow reduced mean from the loss.

In [23]:

```
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_)
```

The following will provide us with checkpoints within our dataset by accessing the operating system path and will use the

tf.train.Checkpoint that helps us group the trackable objects, saving and restoring them. This will include the optimizer, decoder and encoder.

In [24]:

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

Training our data with TensorFlow:

- With training out data with TensorFlow, we first have to start off where we'll show the dataset we'll be using a compiler for a callable TensorFlow graph.
- A function called **train_step** will help pass the input to the encoder from which we use the models we have built out. We then will return the hidden state from the encoder
- The **tf.GradientTape() as tape** will help record the different operations for the automatic differenetiation.
- The decoder will help return the predictions and the hidden state of the decoder when these layers get passed through them.
- It then passes the decoder hidden state into the model and the predictions will help calculate the loss from it.
- From the comment of the code, **Teacher forcing** is a way of getting the target word to pass as the next input into the decoder.
- We are allocating all of the times that loss has been calculating and storing it to use for the batch loss and gradients.
- Returning the **batch_loss** will be applied to the optimizer.

In [25]:

```
@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
```

When feeding Python the information of passing the dataset, usually we would need a good amount of times for the dataset to get used to. For example, Epochs helps gain the entire dataset from forwards to backwards; going through the neural network with dependent variable of how many times it is going through it. We would need an optimum amount of Epochs to determind it not being either overfit and/or underfit from the data being given. Due to the strong CPU power that these computations can take, it'll be limited.

In [37]:

```
EPOCHS = 1

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()))
  # saving (checkpoint) the model every 2 epochs
  if (epoch + 1) % 2 == 0:
    checkpoint.save(file_prefix = checkpoint_prefix)

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

```
Epoch 1 Batch 0 Loss 0.4741
Epoch 1 Batch 100 Loss 0.5441
Epoch 1 Batch 200 Loss 0.4735
Epoch 1 Batch 300 Loss 0.3984
Epoch 1 Loss 0.4734
Time taken for 1 epoch 749.309564113617 sec
```

Translating the data from TensorFlow:

- The function called **evaluate** where we'll take a sentence and put it in our **preprocess_sentence** where it'll pad up our sentences for proper usage. From the for loop, we'll be able to store the attention weights to the attention plots and put the data that predictive models will fit into our plottings. This will make sure the predictive model will end as soon as it reached the **end** token and will store the attention weights every step.

In [27]:

```
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
```

The function called **plot_attention** will help use the attention, sentence and predicted sentence will plot the attention weights. The figure and x/y axis plots will be plotted through this function and map out the predictions in our code when translating our language.

In [29]:

```
# 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()
```

The function called **translate** will take the sentence and start to translate and will trigger to configure the attention plots with what the input is and the output of the translation.

In [30]:

```
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(' '))
```

We are in the last step and this peice will help restore the latest checkpoint and we can finally start testing.

In [31]:

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

Out[31]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x24d80e7b388>
```

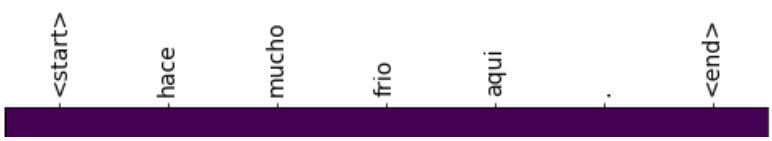
Testing out our translations:

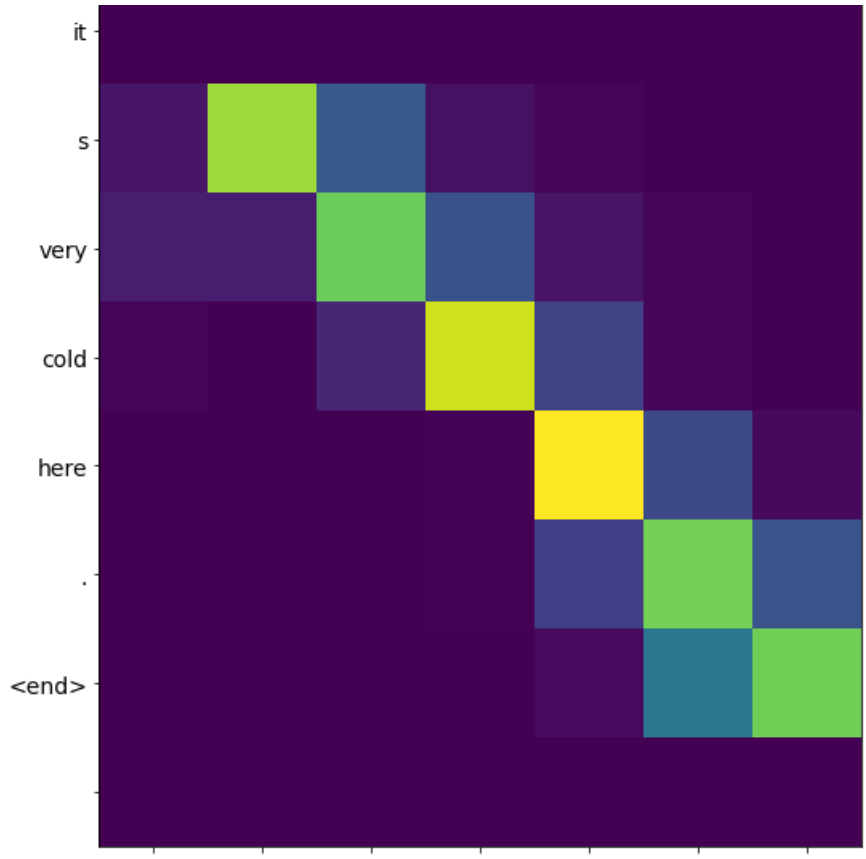
- Using the **translate** function, we can now finally input our Spanish phrases to get the English output and the model readings on how accurate the predictions are going to be. Based on the significant color change, we can see what English words would correlate towards the Spanish words and how closely it would predict to other words and phrases.

In [32]:

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

```
Input: <start> hace mucho frio aqui . <end>
Predicted translation: it s very cold here . <end>
```

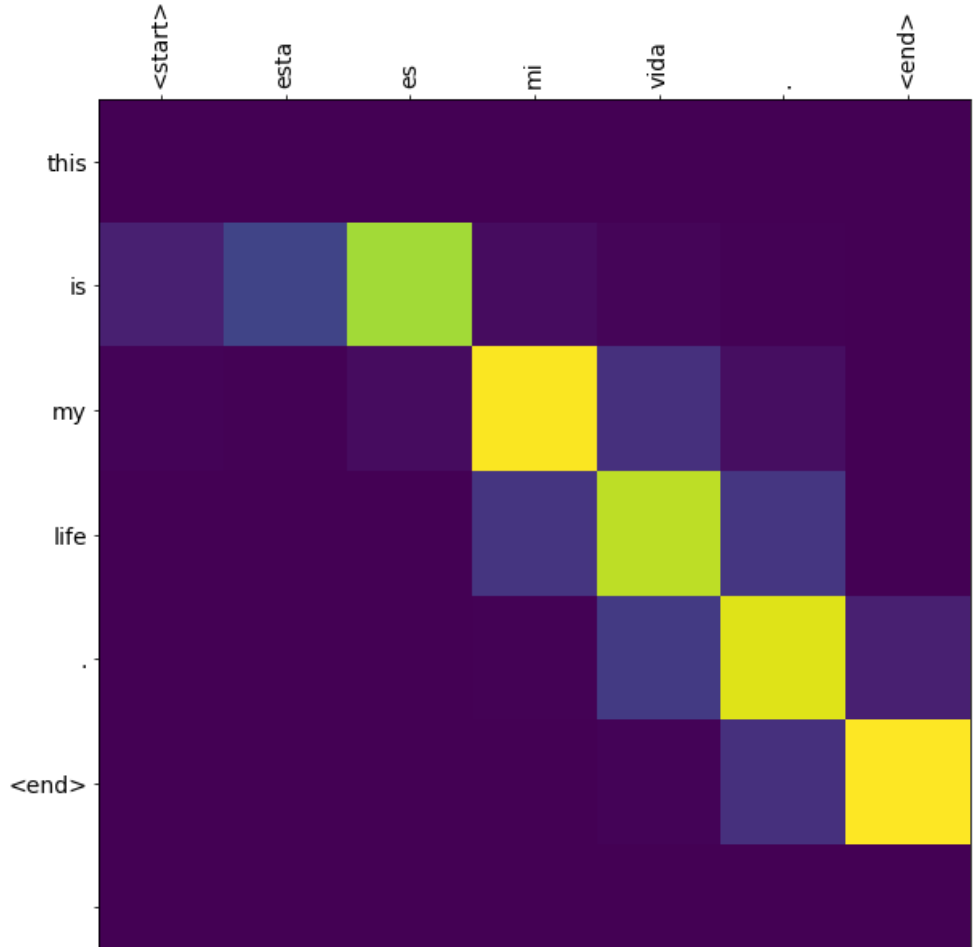




In [33]:

```
translate(u'esta es mi vida.')
```

```
Input: <start> esta es mi vida . <end>
Predicted translation: this is my life . <end>
```



Modifying the Model

We will now change up our model by changing a few things up and displaying something different with our translation models.

From this model, we will be using the Dutch zip file to see some changes that go on with our language processing model and see any parallels or differences in changing up the whole file in itself. This will experiment how the model as a whole would perform under the circumstances of another language.

Link: https://github.com/JaiTheGuy/CMP414-765_Spring2020/blob/master/Final_Project_(Dutch_Translation_Modification).ipynb

IMPORTANT NOTE: Due to the specific zipped files not being able to load within Jupyter Notebook, I've made a path with Google Colab to make it more easy to load the information and data for training and testing. This means the modified models will be linked below this markdown comment for review.

Since the modified model is very simular to the original code made in this notebook, I will state all of the differences in this notebook rather than writing out the same notes put into this one.

This would be the way of loading the different dataset of the different langauge of the code. This will use the operating system to help reach to the sipped file included from **Tab-delimited Bilingual Sentence Pairs** section of the manythings.org website. This modified version will be from Spanish to Dutch.

In [27]:

```
path_to_zip = tf.keras.utils.get_file(
    'nld-eng.zip', origin='http://storage.googleapis.com/download.tensorflow.org/data/nld-eng.zip'
,
    extract=True)

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

To test out this code, we would have to again take into account any symbols or characteristics that are invilved with the Dutch language compared to the English language. For example, we wouldn't need ¡ since this is Dutch and only Spanish involves that character in their langauge.

In [25]:

```
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)
  w = re.sub(r'[" "]+', " ", w)
  w = re.sub(r"[^a-zA-Z?.!,]+", " ", w)
  w = w.strip()
  w = '<start> ' + w + ' <end>'
  return w
```

The following is a test if the start and end token of the translations to ensure the function is able to seek the begginning and end of the translated sentences. So from English and Dutch sentences, they both faces the process of lower casing the characters and putting a space after each and every punctuation mark.

In [26]:

```
en_sentence = u"They fell asleep after the movie."
dh_sentence = u"Ze zijn na de film in slaap gevallen."
print(preprocess_sentence(en_sentence))
print(preprocess_sentence(dh_sentence).encode('utf-8'))
```

```
<start> they fell asleep after the movie . <end>
b'<start> ze zijn na de film in slaap gevallen . <end>'
```

The following is the same function used to help remove any of the accents and cleaning of the Dutch and English statements with its corresponding word pairings. From this point forward, the rest of the code will be written in Google Colab. This was to start off the code and how much can be written without Jupyter notebook getting an error which is the reason for the move to that platform.

In [6]:

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

This is a way of formatting the txt file into a csv file to be compared with the word pairings. This Dutch csv will show the three 3 columns that involves the English, Dutch and the Attribution that the translation came from.

In [13]:

```
data = pd.read_csv("dutch.csv",encoding='utf-8')
data.head()
```

Out[13]:

	Column1	Column2	Column3
0	Hi.	Hoi.	CC-BY 2.0 (France) Attribution: tatoeba.org #5...
1	Hi.	HÃ©!	CC-BY 2.0 (France) Attribution: tatoeba.org #5...
2	Hi.	Hai!	CC-BY 2.0 (France) Attribution: tatoeba.org #5...
3	Run!	Ren!	CC-BY 2.0 (France) Attribution: tatoeba.org #9...
4	Run.	Ren!	CC-BY 2.0 (France) Attribution: tatoeba.org #4...

I dropped the third column because it wouldn't be necessary for our comparisons

In [4]:

```
dth = data.drop(['Column3'], axis=1)
dth.head()
```

Out[4]:

	Column1	Column2
0	Hi.	Hoi.
1	Hi.	HÃ©!
2	Hi.	Hai!
3	Run!	Ren!
4	Run.	Ren!

I am naming the columns to its designated name such as English and Dutch

In [28]:

```
dutch = dth.rename(columns={"Column1": "English", "Column2": "Dutch"})
dutch.head()
```

Out[28]:

	English	Dutch
0	Hi.	Hoi.
1	Hi.	HÃ©!
2	Hi.	Hai!
3	Run!	Ren!
4	Run.	Ren!

This function would replace the special characters in the dataset

In [29]:

```
printable = set(string.printable)

def remove_spec_chars(in_str):
    return ''.join([c for c in in_str if c in printable])

dutch.apply(remove_spec_chars)
```

Out[29]:

```
English
Dutch
dtype: object
```

The dutch data would remove any of the ASCII and unnessary characters that may have been in the mix while converting the characters throughout the process. The purpose of this conversion of the txt file is to display the translations in a lighter manner than can be used in various data visualization models that help us seek more into commonalities between the two languages.

In [32]:

```
dutch.replace({r'[^\x00-\x7F]+':''}, regex=True, inplace=True)
dutch
```

Out[32]:

	English	Dutch
0	Hi.	Hoi.
1	Hi.	H!
2	Hi.	Hai!
3	Run!	Ren!
4	Run.	Ren!
...	...	...
48955	The world's first Ferris wheel was built in Ch...	Het eerste reuzenrad ter wereld werd gebouwd i...
48956	Jingle Bells, a popular song around Christmas ...	Jingle Bells, het bekende kerstlied, heeft in ...
48957	Mary thought to herself that she ought to buy ...	Mary dacht bij zichzelf dat ze een nieuwe bede...
48958	Always use distilled water in steam irons beca...	Gebruik altijd gedistilleerd water in stoomstr...
48959	If someone who doesn't know your background sa...	Als iemand die je achtergrond niet kent zegt d...

48960 rows × 2 columns

Key Modifications from Google Colab (Dutch Training Models)

- Language change: The new language being tested is Dutch instead of the original Spanish
- Path change: The data being extracted being used is the path from Google Drive which is another alternative in getting the data extraction
- Number of num_examples: The change of this was to test out how much we can reduce the dataset from the last one with was 35,00 down to 20,000 instead of using the whole dataset which consisted of 100 thousand plus datasets that may not be tested in the long run. From this change, it changed up the length of the input/target tensor value and the TensorShape in general.
- Number of EPOCHS: The number of epochs used in this training set is 10 which is better than the 1 used for the English-Spanish translations for better testing and gain a better handle to the Dutch language dataset to make it learn more effectively. After each epoch cycle, the number gets closer and closer to 0 to show that it's gaining of the knowledge it can extract from the dataset.
- Results brought up different model attention graphs.

The Encoder, Decoder, Bahdanau Attention, optimizer and the hidden layer does not change its operations.

The New Changes of Translation Attention Maps

From the two pictures alone, you can tell by the attention models that we were able to train the Dutch dataset with our sequence to sequence model that can use the colors to correctly identify the correct translation given in the translate functions. For example, the words like soup, cold and pay and very distinct colors that made it pop out from the translations.

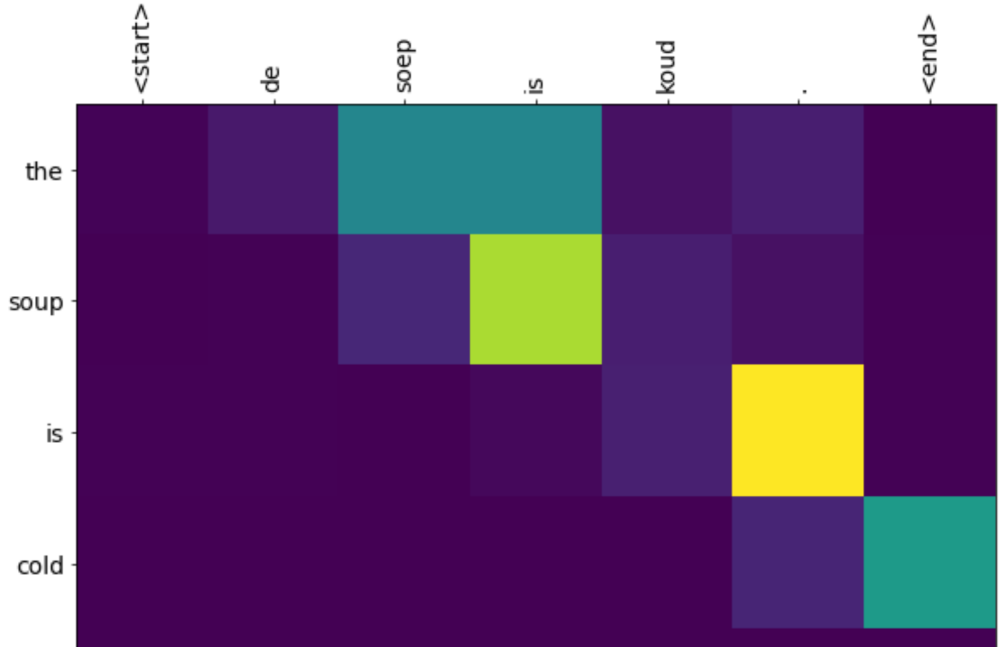
In [44]:

```
Image(filename= "Pictures/dutch1.png", width=700, height=530)
```

Out[44]:

```
translate("De soep is koud.")
```

```
Input: <start> de soep is koud . <end>
Predicted translation: the soup is cold . <end>
```



In [45]:

```
Image(filename= "Pictures/dutch2.png", width=700, height=530)
```

Out[45]:

```
[35] translate(u'Tom zal je betalen.')
```

```
Input: <start> tom zal je betalen . <end>
Predicted translation: tom will pay you . <end>
```

Conclusion Of Project:

I've learned that with sequence 2 sequence models, we can do a lot of language learning with Tensorflow and multiple ways of training the different models such as the encoder and encoder to ensure visual models that can be used in real-life applications. These attention mechanisms help use the architecture with the cases such as abstractive text summerization and meural machine trasnlation that provides the basic idea of attention mechanism in mapping source to target sequences. Smaller scaled projects such as this can bring out the basic idea of how much we can do and use with languaging sequencing and conversion.

- Kevin Jairam