```
from __future__ import print_function

from keras.models import Model
from keras.layers import Input, LSTM, Dense, Flatten, Attention
import numpy as np
import pandas as pd
pd.set_option('display.max_columns', None)
```

```
batch_size = 64  # Batch size for training.
epochs = 50  # Number of epochs to train for.
latent_dim = 256  # Latent dimensionality of the encoding space.
num_samples = 3000  # Number of samples to train on.
# Path to the data txt file on disk.
data_path = 'swe.txt'
```

```
##Question 1
# Vectorize the data.
input_texts = []
target_texts = []
input_characters = set()
target_characters = set()
with open(data_path, 'r', encoding='utf-8') as f:
    lines = f.read().split('\n')
```

```
encoder_input_data
```

```
#now need to create model for character level model
```

```
from __future__ import print_function
import os
from keras.models import Model
from keras.layers import Input, LSTM, Dense
import numpy as np
import pandas as pd
from matplotlib import pyplot
!pip install contractions
pd.set_option('display.max_columns', None)
```

```
Successfully installed anyascii-0.3.0 contractions-0.0.55 pyahocorasick-1.4.2 textsearch
```

```
import numpy as np

import typing
from typing import Any, Tuple
```

```
import tensorflow as tf
from tensorflow.keras.layers.experimental import preprocessing
!pip install tensorflow_text
import tensorflow_text as tf_text

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
```

```
    Installing collected packages: tensorflow-text
    Successfully installed tensorflow-text-2.6.0
```

Shape checker

```
#@title Shape checker
class ShapeChecker():
  def __init__(self):
    # Keep a cache of every axis-name seen
    self.shapes = {}

  def __call__(self, tensor, names, broadcast=False):
    if not tf.executing_eagerly():
      return

    if isinstance(names, str):
      names = (names,)

    shape = tf.shape(tensor)
    rank = tf.rank(tensor)

    if rank != len(names):
      raise ValueError(f'Rank mismatch:\n'
                       f'    found {rank}: {shape.numpy()}\n'
                       f'    expected {len(names)}: {names}\n')

    for i, name in enumerate(names):
      if isinstance(name, int):
        old_dim = name
      else:
        old_dim = self.shapes.get(name, None)
      new_dim = shape[i]

      if (broadcast and new_dim == 1):
        continue

      if old_dim is None:
        # If the axis name is new, add its length to the cache.
        self.shapes[name] = new_dim
        continue

      if new_dim != old_dim:
        raise ValueError(f"Shape mismatch for dimension: '{name}'\n"
```

```
                         f"    found: {new_dim}\n"
```

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

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

```
use_builtins = True
```

```
import contractions
input_texts2 = []
target_texts2 = []
with open('swe.txt', 'r', encoding='utf-8') as f:
 lines = f.read().split('\n')
for line in lines[: min(num_samples, len(lines) - 1)]:
 input_text2, target_text2, useless2 = line.split('\t')
 #target_text2 = '\t' + target_text2 + '\n'
 input_texts2.append(input_text2)
 target_texts2.append(target_text2)
```

```
input_texts3= tf.convert_to_tensor(input_texts2)
target_texts3= tf.convert_to_tensor(target_texts2)
```

```
input_texts3.shape
```

```
     TensorShape([3000])
```

```
BUFFER_SIZE = len(input_texts2)
BATCH_SIZE = 64

dataset = tf.data.Dataset.from_tensor_slices((input_texts3, target_texts3)).shuffle(BUFFER_SI
dataset = dataset.batch(BATCH_SIZE)
```

```
for example_input_batch, example_target_batch in dataset.take(4):
  print(example_input_batch[:5])
  print()
  print(example_target_batch[:5])
  break
```

```
def tf_lower_and_split_punct(text):
  # Split accecented characters.
```

```
  text = tf_text.normalize_utf8(text, 'NFKD')
  text = tf.strings.lower(text)
  # Keep space, a to z, and select punctuation.
  text = tf.strings.regex_replace(text, '[^ a-z.?!,¿]', '')
  # Add spaces around punctuation.
  text = tf.strings.regex_replace(text, '[.?!,¿]', r' \0 ')
  # Strip whitespace.
  text = tf.strings.strip(text)

  text = tf.strings.join(['[START]', text, '[END]'], separator=' ')
  return text
```

```
max_vocab_size = 5000

input_text_processor = preprocessing.TextVectorization(
    standardize=tf_lower_and_split_punct,
    max_tokens=max_vocab_size)
```

```
input_text_processor.adapt(input_texts3)

# Here are the first 10 words from the vocabulary:
input_text_processor.get_vocabulary()[:10]
```

```
['', '[UNK]', '[START]', '[END]', '.', 'i', '?', 'tom', 'it', 'you']
```

```
output_text_processor = preprocessing.TextVectorization(
    standardize=tf_lower_and_split_punct,
    max_tokens=max_vocab_size)

output_text_processor.adapt(target_texts3)
output_text_processor.get_vocabulary()[:10]
```

```
['', '[UNK]', '[START]', '[END]', '.', 'jag', 'ar', 'tom', '?', 'det']
```

```
#dont use this???
input_tokens = input_text_processor(input_texts3)
input_tokens[:3, :10]
```

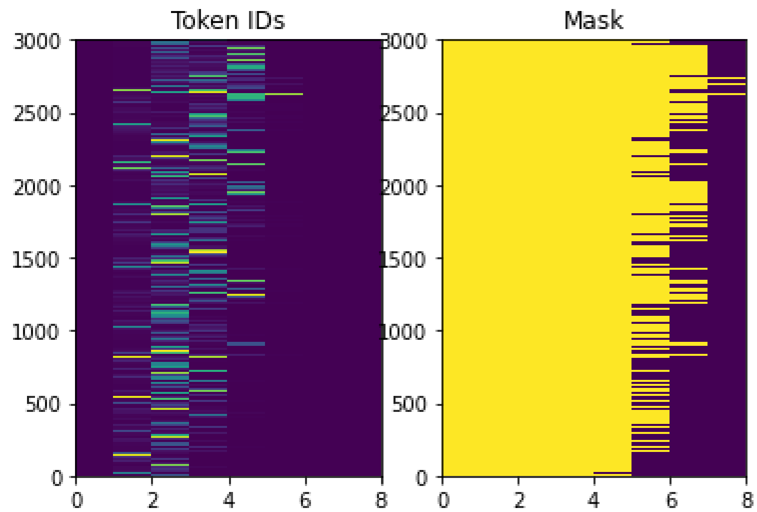
```
<tf.Tensor: shape=(3, 8), dtype=int64, numpy=
array([[  2,  33,   4,   3,   0,   0,   0,   0],
       [  2, 133,  18,   3,   0,   0,   0,   0],
       [  2,  70,   6,   3,   0,   0,   0,   0]])>
```

```
input_vocab = np.array(input_text_processor.get_vocabulary())
tokens = input_vocab[input_tokens[0].numpy()]
' '.join(tokens)
```

```
'[START] go . [END]    '
```

```
plt.subplot(1, 2, 1)
plt.pcolormesh(input_tokens)
plt.title('Token IDs')

plt.subplot(1, 2, 2)
plt.pcolormesh(input_tokens != 0)
plt.title('Mask')
```

```
Text(0.5, 1.0, 'Mask')
```




```
embedding_dim = 256
units = 1024
```

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, input_vocab_size, embedding_dim, enc_units):
    super(Encoder, self).__init__()
    self.enc_units = enc_units
    self.input_vocab_size = input_vocab_size

    # The embedding layer converts tokens to vectors
    self.embedding = tf.keras.layers.Embedding(self.input_vocab_size,
                                               embedding_dim)

    # The GRU RNN layer processes those vectors sequentially.
    self.gru = tf.keras.layers.GRU(self.enc_units,
                                   # Return the sequence and state
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')

  def call(self, tokens, state=None):
    shape_checker = ShapeChecker()
    shape_checker(tokens, ('batch', 's'))
```

```
    # 2. The embedding layer looks up the embedding for each token.
    vectors = self.embedding(tokens)
    shape_checker(vectors, ('batch', 's', 'embed_dim'))

    # 3. The GRU processes the embedding sequence.
    #    output shape: (batch, s, enc_units)
    #    state shape: (batch, enc_units)
    output, state = self.gru(vectors, initial_state=state)
    shape_checker(output, ('batch', 's', 'enc_units'))
    shape_checker(state, ('batch', 'enc_units'))

    # 4. Returns the new sequence and its state.
    return output, state

# Convert the input text to tokens.
input_tokens2 = input_text_processor(input_texts3)

# Encode the input sequence.
encoder = Encoder(input_text_processor.vocabulary_size(),
                  embedding_dim, units)
example_enc_output, example_enc_state = encoder(input_tokens2)

print(f'Input batch tokens, shape (batch, s): {input_tokens2.shape}')
print(f'Encoder output, shape (batch, s, units): {example_enc_output.shape}')
print(f'Encoder state, shape (batch, units): {example_enc_state.shape}')
```

```
Input batch tokens, shape (batch, s): (3000, 8)
Encoder output, shape (batch, s, units): (3000, 8, 1024)
Encoder state, shape (batch, units): (3000, 1024)
```

```
class BahdanauAttention(tf.keras.layers.Layer):
  def __init__(self, units):
    super().__init__()
    # For Eqn. (4), the  Bahdanau attention
    self.W1 = tf.keras.layers.Dense(units, use_bias=False)
    self.W2 = tf.keras.layers.Dense(units, use_bias=False)

    self.attention = tf.keras.layers.AdditiveAttention()

  def call(self, query, value, mask):
    shape_checker = ShapeChecker()
    shape_checker(query, ('batch', 't', 'query_units'))
    shape_checker(value, ('batch', 's', 'value_units'))
    shape_checker(mask, ('batch', 's'))

    # From Eqn. (4), `W1@ht`.
    w1_query = self.W1(query)
    shape_checker(w1_query, ('batch', 't', 'attn_units'))

    # From Eqn. (4), `W2@hs`.
    w2_key = self.W2(value)
```

```
        shape_checker(w2_key, ('batch', 's', 'attn_units'))

        query_mask = tf.ones(tf.shape(query)[:-1], dtype=bool)
        value_mask = mask

        context_vector, attention_weights = self.attention(
            inputs = [w1_query, value, w2_key],
            mask=[query_mask, value_mask],
            return_attention_scores = True,
        )
        shape_checker(context_vector, ('batch', 't', 'value_units'))
        shape_checker(attention_weights, ('batch', 't', 's'))

        return context_vector, attention_weights
```

```
attention_layer = BahdanauAttention(units)
```

```
(input_tokens != 0).shape
```

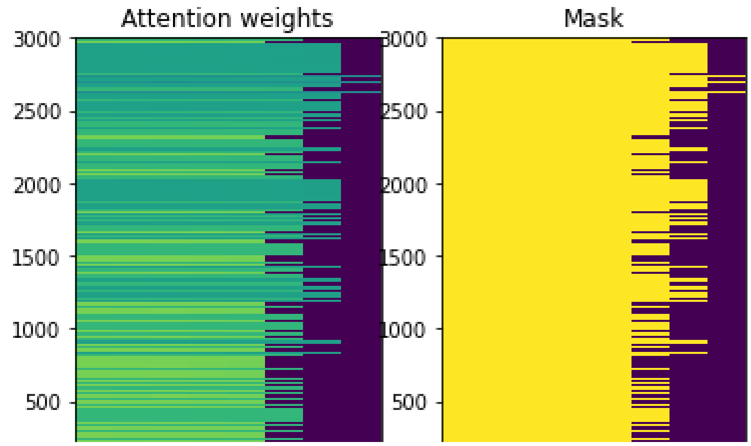
```
TensorShape([3000, 8])
```

```
# Later, the decoder will generate this attention query
example_attention_query = tf.random.normal(shape=[len(input_tokens), 2, 10])

# Attend to the encoded tokens

context_vector, attention_weights = attention_layer(
    query=example_attention_query,
    value=example_enc_output,
    mask=(input_tokens != 0))

print(f'Attention result shape: (batch_size, query_seq_length, units):           {context_vec
print(f'Attention weights shape: (batch_size, query_seq_length, value_seq_length): {attention
```

```
Attention result shape: (batch_size, query_seq_length, units):           (3000, 2, 1024)
Attention weights shape: (batch_size, query_seq_length, value_seq_length): (3000, 2, 8)
```

```
plt.subplot(1, 2, 1)
plt.pcolormesh(attention_weights[:, 0, :])
plt.title('Attention weights')

plt.subplot(1, 2, 2)
plt.pcolormesh(input_tokens != 0)
plt.title('Mask')
```

```
Text(0.5, 1.0, 'Mask')
```



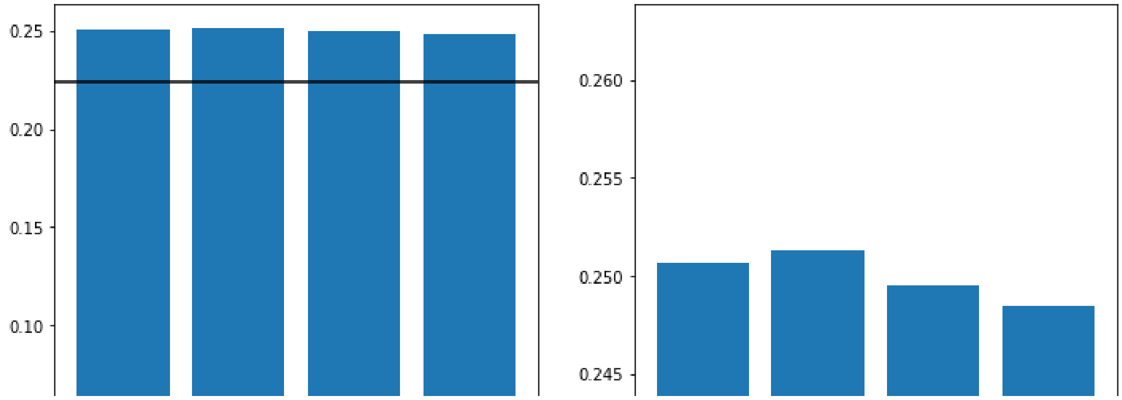

```
attention_weights.shape
```

```
TensorShape([3000, 2, 8])
```

```
attention_slice = attention_weights[0, 0].numpy()
attention_slice = attention_slice[attention_slice != 0]
```

```
#@title
plt.suptitle('Attention weights for one sequence')

plt.figure(figsize=(12, 6))
a1 = plt.subplot(1, 2, 1)
plt.bar(range(len(attention_slice)), attention_slice)
# freeze the xlim
plt.xlim(plt.xlim())
plt.xlabel('Attention weights')

a2 = plt.subplot(1, 2, 2)
plt.bar(range(len(attention_slice)), attention_slice)
plt.xlabel('Attention weights, zoomed')

# zoom in
top = max(a1.get_ylim())
zoom = 0.85*top
a2.set_ylim([0.90*top, top])
a1.plot(a1.get_xlim(), [zoom, zoom], color='k')
```

```
[<matplotlib.lines.Line2D at 0x7f512b39d9d0>]
<Figure size 432x288 with 0 Axes>
```




```
class Decoder(tf.keras.layers.Layer):
  def __init__(self, output_vocab_size, embedding_dim, dec_units):
    super(Decoder, self).__init__()
    self.dec_units = dec_units
    self.output_vocab_size = output_vocab_size
    self.embedding_dim = embedding_dim

    # For Step 1. The embedding layer convets token IDs to vectors
    self.embedding = tf.keras.layers.Embedding(self.output_vocab_size,
                                               embedding_dim)

    # For Step 2. The RNN keeps track of what's been generated so far.
    self.gru = tf.keras.layers.GRU(self.dec_units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')

    # For step 3. The RNN output will be the query for the attention layer.
    self.attention = BahdanauAttention(self.dec_units)

    # For step 4. Eqn. (3): converting `ct` to `at`
    self.Wc = tf.keras.layers.Dense(dec_units, activation=tf.math.tanh,
                                    use_bias=False)

    # For step 5. This fully connected layer produces the logits for each
    # output token.
    self.fc = tf.keras.layers.Dense(self.output_vocab_size)


class DecoderInput(typing.NamedTuple):
  new_tokens: Any
  enc_output: Any
  mask: Any

class DecoderOutput(typing.NamedTuple):
  logits: Any
  attention_weights: Any
```

```
def call(self,
         inputs: DecoderInput,
         state=None) -> Tuple[DecoderOutput, tf.Tensor]:
  shape_checker = ShapeChecker()
  shape_checker(inputs.new_tokens, ('batch', 't'))
  shape_checker(inputs.enc_output, ('batch', 's', 'enc_units'))
  shape_checker(inputs.mask, ('batch', 's'))

  if state is not None:
    shape_checker(state, ('batch', 'dec_units'))

  # Step 1. Lookup the embeddings
  vectors = self.embedding(inputs.new_tokens)
  shape_checker(vectors, ('batch', 't', 'embedding_dim'))

  # Step 2. Process one step with the RNN
  rnn_output, state = self.gru(vectors, initial_state=state)

  shape_checker(rnn_output, ('batch', 't', 'dec_units'))
  shape_checker(state, ('batch', 'dec_units'))

  # Step 3. Use the RNN output as the query for the attention over the
  # encoder output.
  context_vector, attention_weights = self.attention(
      query=rnn_output, value=inputs.enc_output, mask=inputs.mask)
  shape_checker(context_vector, ('batch', 't', 'dec_units'))
  shape_checker(attention_weights, ('batch', 't', 's'))

  # Step 4. Eqn. (3): Join the context_vector and rnn_output
  #     [ct; ht] shape: (batch t, value_units + query_units)
  context_and_rnn_output = tf.concat([context_vector, rnn_output], axis=-1)

  # Step 4. Eqn. (3): `at = tanh(Wc@[ct; ht])`
  attention_vector = self.Wc(context_and_rnn_output)
  shape_checker(attention_vector, ('batch', 't', 'dec_units'))

  # Step 5. Generate logit predictions:
  logits = self.fc(attention_vector)
  shape_checker(logits, ('batch', 't', 'output_vocab_size'))

  return DecoderOutput(logits, attention_weights), state


Decoder.call = call


decoder = Decoder(output_text_processor.vocabulary_size(),
                  embedding_dim, units)


# Convert the target sequence, and collect the "[START]" tokens
example_output_tokens = output_text_processor(target_texts3)
```

```
start_index = output_text_processor.get_vocabulary().index('[START]')
first_token = tf.constant([[start_index]] * example_output_tokens.shape[0])


# Run the decoder
dec_result, dec_state = decoder(
    inputs = DecoderInput(new_tokens=first_token,
                          enc_output=example_enc_output,
                          mask=(input_tokens != 0)),
    state = example_enc_state
)

print(f'logits shape: (batch_size, t, output_vocab_size) {dec_result.logits.shape}')
print(f'state shape: (batch_size, dec_units) {dec_state.shape}')
```

```
logits shape: (batch_size, t, output_vocab_size) (3000, 1, 1665)
state shape: (batch_size, dec_units) (3000, 1024)
```

```
sampled_token = tf.random.categorical(dec_result.logits[:, 0, :], num_samples=1)


vocab = np.array(output_text_processor.get_vocabulary())
first_word = vocab[sampled_token.numpy()]
first_word[:5]


dec_result, dec_state = decoder(
    DecoderInput(sampled_token,
                 example_enc_output,
                 mask=(input_tokens != 0)),
    state=dec_state)


sampled_token = tf.random.categorical(dec_result.logits[:, 0, :], num_samples=1)
first_word = vocab[sampled_token.numpy()]
first_word[:5]


class MaskedLoss(tf.keras.losses.Loss):
  def __init__(self):
    self.name = 'masked_loss'
    self.loss = tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True, reduction='none')

  def __call__(self, y_true, y_pred):
    shape_checker = ShapeChecker()
    shape_checker(y_true, ('batch', 't'))
    shape_checker(y_pred, ('batch', 't', 'logits'))

    # Calculate the loss for each item in the batch.
    loss = self.loss(y_true, y_pred)
```

```
    shape_checker(loss, ('batch', 't'))

    # Mask off the losses on padding.
    mask = tf.cast(y_true != 0, tf.float32)
    shape_checker(mask, ('batch', 't'))
    loss *= mask

    # Return the total.
    return tf.reduce_sum(loss)


class TrainTranslator(tf.keras.Model):
  def __init__(self, embedding_dim, units,
               input_text_processor,
               output_text_processor,
               use_tf_function=True):
    super().__init__()
    # Build the encoder and decoder
    encoder = Encoder(input_text_processor.vocabulary_size(),
                      embedding_dim, units)
    decoder = Decoder(output_text_processor.vocabulary_size(),
                      embedding_dim, units)

    self.encoder = encoder
    self.decoder = decoder
    self.input_text_processor = input_text_processor
    self.output_text_processor = output_text_processor
    self.use_tf_function = use_tf_function
    self.shape_checker = ShapeChecker()

  def train_step(self, inputs):
    self.shape_checker = ShapeChecker()
    if self.use_tf_function:
      return self._tf_train_step(inputs)
    else:
      return self._train_step(inputs)


def _preprocess(self, input_text, target_text):
  self.shape_checker(input_text, ('batch',))
  self.shape_checker(target_text, ('batch',))

  # Convert the text to token IDs
  input_tokens = self.input_text_processor(input_text)
  target_tokens = self.output_text_processor(target_text)
  self.shape_checker(input_tokens, ('batch', 's'))
  self.shape_checker(target_tokens, ('batch', 't'))

  # Convert IDs to masks.
  input_mask = input_tokens != 0
  self.shape_checker(input_mask, ('batch', 's'))
```

```
    target_mask = target_tokens != 0
    self.shape_checker(target_mask, ('batch', 't'))

    return input_tokens, input_mask, target_tokens, target_mask

  TrainTranslator._preprocess = _preprocess


  def _train_step(self, inputs):
    input_text, target_text = inputs

    (input_tokens, input_mask,
     target_tokens, target_mask) = self._preprocess(input_text, target_text)

    max_target_length = tf.shape(target_tokens)[1]

    with tf.GradientTape() as tape:
      # Encode the input
      enc_output, enc_state = self.encoder(input_tokens)
      self.shape_checker(enc_output, ('batch', 's', 'enc_units'))
      self.shape_checker(enc_state, ('batch', 'enc_units'))

      # Initialize the decoder's state to the encoder's final state.
      # This only works if the encoder and decoder have the same number of
      # units.
      dec_state = enc_state
      loss = tf.constant(0.0)

      for t in tf.range(max_target_length-1):
        # Pass in two tokens from the target sequence:
        # 1. The current input to the decoder.
        # 2. The target for the decoder's next prediction.
        new_tokens = target_tokens[:, t:t+2]
        step_loss, dec_state = self._loop_step(new_tokens, input_mask,
                                               enc_output, dec_state)
        loss = loss + step_loss

      # Average the loss over all non padding tokens.
      average_loss = loss / tf.reduce_sum(tf.cast(target_mask, tf.float32))

    # Apply an optimization step
    variables = self.trainable_variables
    gradients = tape.gradient(average_loss, variables)
    self.optimizer.apply_gradients(zip(gradients, variables))

    # Return a dict mapping metric names to current value
    return {'batch_loss': average_loss}


  TrainTranslator._train_step = _train_step
```

```
def _loop_step(self, new_tokens, input_mask, enc_output, dec_state):
  input_token, target_token = new_tokens[:, 0:1], new_tokens[:, 1:2]

  # Run the decoder one step.
  decoder_input = DecoderInput(new_tokens=input_token,
                               enc_output=enc_output,
                               mask=input_mask)

  dec_result, dec_state = self.decoder(decoder_input, state=dec_state)
  self.shape_checker(dec_result.logits, ('batch', 't1', 'logits'))
  self.shape_checker(dec_result.attention_weights, ('batch', 't1', 's'))
  self.shape_checker(dec_state, ('batch', 'dec_units'))

  # `self.loss` returns the total for non-padded tokens
  y = target_token
  y_pred = dec_result.logits
  step_loss = self.loss(y, y_pred)

  return step_loss, dec_state



TrainTranslator._loop_step = _loop_step


translator = TrainTranslator(
    embedding_dim, units,
    input_text_processor=input_text_processor,
    output_text_processor=output_text_processor,
    use_tf_function=False)

# Configure the loss and optimizer
translator.compile(
    optimizer=tf.optimizers.Adam(),
    loss=MaskedLoss(),
)
```

```
np.log(output_text_processor.vocabulary_size())
```

```
7.417580402414544
```

```
%%time
for n in range(10):
  print(translator.train_step([input_texts3, target_texts3]))
print()
```

```
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=6.247205>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=6.1790805>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=6.041966>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=5.6836166>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=4.7087393>}
```

```
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=4.7425594>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=3.9036467>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=3.8094451>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=3.8800523>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=3.7265575>}

CPU times: user 21min 1s, sys: 18.1 s, total: 21min 19s
Wall time: 11min 6s
```

```
@tf.function(input_signature=[[tf.TensorSpec(dtype=tf.string, shape=[None]),
                               tf.TensorSpec(dtype=tf.string, shape=[None])]])
def _tf_train_step(self, inputs):
  return self._train_step(inputs)
```

```
TrainTranslator._tf_train_step = _tf_train_step
```

```
translator.use_tf_function = True
```

```
translator.train_step([example_input_batch, example_target_batch])
```

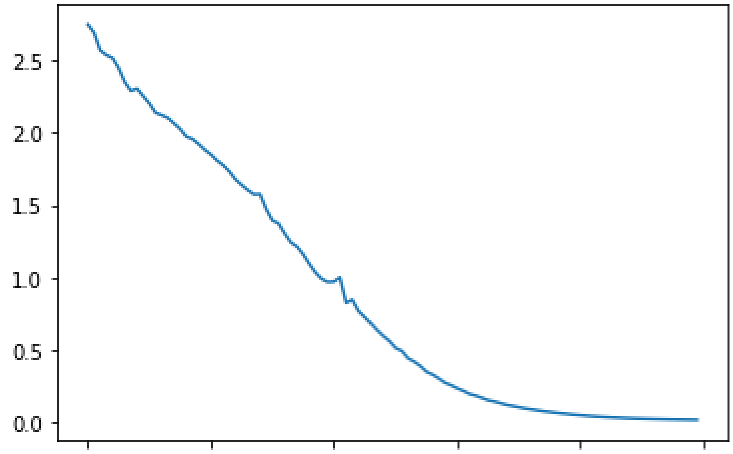
```
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=6.270193>}
```

```
%%time
for n in range(10):
  print(translator.train_step([example_input_batch, example_target_batch]))
print()
```

```
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=6.202393>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=6.0730004>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=5.7399745>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=4.8307624>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=4.1018925>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=3.4566236>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=3.3494666>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=3.2292607>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=2.9973493>}
{'batch_loss': <tf.Tensor: shape=(), dtype=float32, numpy=2.7920635>}

CPU times: user 31.1 s, sys: 536 ms, total: 31.6 s
Wall time: 16.4 s
```

```
losses = []
for n in range(100):
  print('.', end='')
  logs = translator.train_step([example_input_batch, example_target_batch])
  losses.append(logs['batch_loss'].numpy())

print()
plt.plot(losses)
```

```
..........................................................................
[<matplotlib.lines.Line2D at 0x7f5128ab7510>]
```



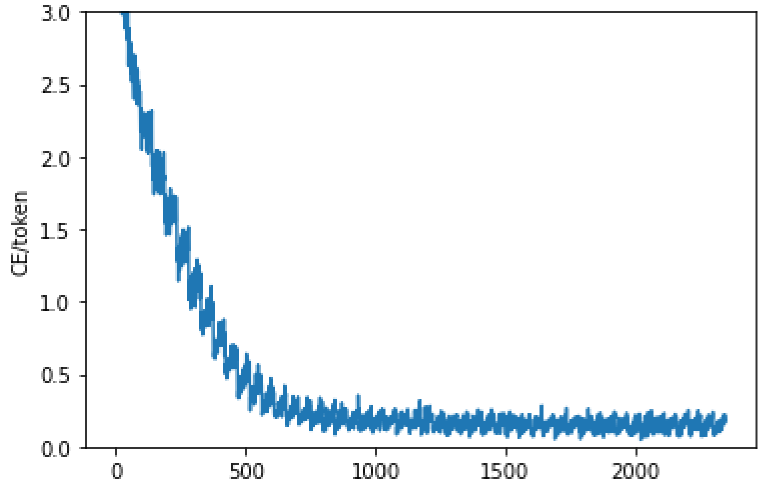

```
train_translator = TrainTranslator(
    embedding_dim, units,
    input_text_processor=input_text_processor,
    output_text_processor=output_text_processor)

# Configure the loss and optimizer
train_translator.compile(
    optimizer=tf.optimizers.Adam(),
    loss=MaskedLoss(),
    metrics=["accuracy"]
)


class BatchLogs(tf.keras.callbacks.Callback):
  def __init__(self, key):
    self.key = key
    self.logs = []

  def on_train_batch_end(self, n, logs):
    self.logs.append(logs[self.key])

batch_loss = BatchLogs('batch_loss')


train_translator.fit(dataset, epochs=50,
                     callbacks=[batch_loss])
```

```
Epoch 1/50
47/47 [==============================] - 86s 2s/step - batch_loss: 3.5051
Epoch 2/50
47/47 [==============================] - 80s 2s/step - batch_loss: 2.5841
Epoch 3/50
47/47 [==============================] - 79s 2s/step - batch_loss: 2.1906
Epoch 4/50
47/47 [==============================] - 80s 2s/step - batch_loss: 1.9036
Epoch 5/50
```

```
47/47 [==============================] - 79s 2s/step - batch_loss: 1.6102
Epoch 6/50
47/47 [==============================] - 80s 2s/step - batch_loss: 1.3565
Epoch 7/50
47/47 [==============================] - 81s 2s/step - batch_loss: 1.1152
Epoch 8/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.9087
Epoch 9/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.7269
Epoch 10/50
47/47 [==============================] - 82s 2s/step - batch_loss: 0.5924
Epoch 11/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.4673
Epoch 12/50
47/47 [==============================] - 81s 2s/step - batch_loss: 0.3870
Epoch 13/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.3124
Epoch 14/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.2761
Epoch 15/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.2409
Epoch 16/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.2234
Epoch 17/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.2124
Epoch 18/50
47/47 [==============================] - 79s 2s/step - batch_loss: 0.2003
Epoch 19/50
47/47 [==============================] - 79s 2s/step - batch_loss: 0.1925
Epoch 20/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.1865
Epoch 21/50
47/47 [==============================] - 79s 2s/step - batch_loss: 0.1803
Epoch 22/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.1763
Epoch 23/50
47/47 [==============================] - 79s 2s/step - batch_loss: 0.1773
Epoch 24/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.1701
Epoch 25/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.1742
Epoch 26/50
47/47 [==============================] - 79s 2s/step - batch_loss: 0.1670
Epoch 27/50
47/47 [==============================] - 79s 2s/step - batch_loss: 0.1666
Epoch 28/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.1615
Epoch 29/50
47/47 [==============================] - 80s 2s/step - batch_loss: 0.1602
Epoch 30/50
```

```
plt.plot(batch_loss.logs)
plt.ylim([0, 3])
plt.xlabel('Batch #')
plt.ylabel('CE/token')
```

```
Text(0, 0.5, 'CE/token')
```



The loss decrease very fast but then does not stabilize. Would have to cancel the noise.However this is a goog model with the lowest lose at .1413. The lowest out of the three models.