Zachary Chenausky

Tensorflow Recurrent Neural Network translation with attention

Initial setup requires pip installations and library imports.

```
!pip install "tensorflow-text>=2.10"
!pip install einops
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simp
Collecting tensorflow-text>=2.10
  Downloading tensorflow_text-2.10.0-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
     |████████████████████████████████| 5.9 MB 4.1 MB/s
Collecting tensorflow<2.11,>=2.10.0
  Downloading tensorflow-2.10.0-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (578.
     |████████████████████████████████| 578.0 MB 16 kB/s
Requirement already satisfied: tensorflow-hub>=0.8.0 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.7
Requirement already satisfied: keras-preprocessing>=1.1.1 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-packages (from tensor
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.7/dist-packages (from
Collecting tensorflow-estimator<2.11,>=2.10.0
  Downloading tensorflow_estimator-2.10.0-py2.py3-none-any.whl (438 kB)
     |████████████████████████████████| 438 kB 55.4 MB/s
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.7/dist-packages (from te
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (from tenso
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.7/dist-packages (from tens
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.7/dist-packages (from tens
Collecting flatbuffers>=2.0
  Downloading flatbuffers-22.10.26-py2.py3-none-any.whl (26 kB)
Collecting tensorboard<2.11,>=2.10
  Downloading tensorboard-2.10.1-py3-none-any.whl (5.9 MB)
     |████████████████████████████████| 5.9 MB 41.0 MB/s
Collecting keras<2.11,>=2.10.0
  Downloading keras-2.10.0-py2.py3-none-any.whl (1.7 MB)
     |████████████████████████████████| 1.7 MB 46.9 MB/s
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: protobuf<3.20,>=3.9.2 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.7/dist-packages (from tens
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.7/dist-packages (from t
Requirement already satisfied: gast<=0.4.0,>=0.2.1 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr/local/lib/python3.
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/lib/python3.7/dis
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-packages (from go
```

```
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: importlib-metadata>=4.4 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importl
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requ
```

```
import numpy as np

import typing
from typing import Any, Tuple

import einops
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

import tensorflow as tf
import tensorflow_text as tf_text
```

```
#@title
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

    parsed = einops.parse_shape(tensor, names)

    for name, new_dim in parsed.items():
      old_dim = self.shapes.get(name, None)

      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"
                         f"    expected: {old_dim}\n")
```

The datasets used to train the model were gathered from manyThings.org/anki. The data sets were structered as side by side sentences left side english, right side czech. Every czech sentence was imported into its own text file same as the english sentences. Then the files were stored as arrays in order to create a tensorflow.data dataset for the model.

```
# Download the file
import pathlib

czech_dataset = pathlib.Path('czech.txt')
english_dataset = pathlib.Path('english.txt')



def load_data(czech, english):

  cs = czech.read_text(encoding='utf-8')
  en = english.read_text(encoding='utf-8')
  cs_lines = cs.splitlines()
  en_lines = en.splitlines()

  context = np.array(cs_lines)
  target = np.array(en_lines)

  return target, context


target_raw, context_raw = load_data(czech_dataset, english_dataset)
print(context_raw[-1])
```

```
Leden, únor, březen, duben, květen, červen, červenec, srpen, září, říjen, listopad a prosinec je c
```

```
print(target_raw[-1])
```

```
January, February, March, April, May, June, July, August, September, October, November and Decembe
```

Create a tf.data dataset

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

is_train = np.random.uniform(size=(len(target_raw),)) < 0.8

train_raw = (
    tf.data.Dataset
    .from_tensor_slices((context_raw[is_train], target_raw[is_train]))
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE))
val_raw = (
    tf.data.Dataset
    .from_tensor_slices((context_raw[~is_train], target_raw[~is_train]))
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE))


for example_context_strings, example_target_strings in train_raw.take(1):
  print(example_context_strings[:5])
  print()
```

```
  print(example_target_strings[:5])
  break
```

```
tf.Tensor(
[b'Byl bych ud\xc4\x9blal to sam\xc3\xa9, co ty'
 b'D\xc4\x9blej to opatrn\xc4\x9b'
 b'Je d\xc5\xafle\xc5\xbeit\xc3\xa9 zn\xc3\xa1t sv\xc3\xa9 limity'
 b'Tom odeslal sv\xc3\xbdch \xc5\xa1est posledn\xc3\xadch textov\xc3\xbdch zpr\xc3\xa1v pouh\xc3\x
 b'Nem\xc3\xa1me pravidla'], shape=(5,), dtype=string)

tf.Tensor(
[b"I would've done exactly what you did\t" b'Do it carefully\t'
 b"It's important to know one's limits\t"
 b'Tom sent his last text message just three minutes before the crash\t'
 b'We have no rules\t'], shape=(5,), dtype=string)
```

Standardization

The first step of the training implementation is to standardize the text. The tensorflow_text package contains a unicode operation to normalize and replace charchters with their ASCII equivlents.

Unicode normalization will be the first step in the text standardization function:

```
def tf_lower_and_split_punct(text):
  # Split accented 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
```

Text Vectorization

For the text vectorization the tensorflow keras.layers will handle all the vocabulary tokenization

```
max_vocab_size = 5000

context_text_processor = tf.keras.layers.TextVectorization(
    standardize=tf_lower_and_split_punct,
    max_tokens=max_vocab_size,
    ragged=True)
```

The text vectorization method reads in one epoch of the training dataset and initializes each layer to determine the vocabulary

```
context_text_processor.adapt(train_raw.map(lambda context, target: context))

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

```
['', '[UNK]', '[START]', '[END]', ',', 'tom', 'se', 'to', 'je', 'jsem']
```

That's the Spanish `TextVectorization` layer, now build and `.adapt()` the English one:

```
target_text_processor = tf.keras.layers.TextVectorization(
    standardize=tf_lower_and_split_punct,
    max_tokens=max_vocab_size,
    ragged=True)

target_text_processor.adapt(train_raw.map(lambda context, target: target))
target_text_processor.get_vocabulary()[:10]
```

```
['', '[UNK]', '[START]', '[END]', 'tom', 'i', 'to', 'the', 'you', 'a']
```

Now these layers can convert a batch of strings into a batch of token IDs:

```
example_tokens = context_text_processor(example_context_strings)
example_tokens[:3, :]
```

```
<tf.RaggedTensor [[2, 23, 36, 66, 7, 937, 4, 13, 75, 3], [2, 2098, 7, 2299, 3],
 [2, 8, 557, 869, 91, 1, 3]]>
```

The `get_vocabulary` method can be used to convert token IDs back to text:

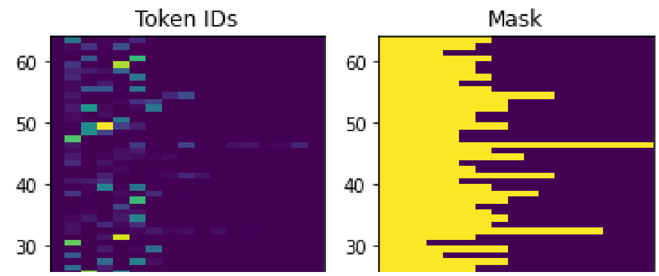
```
context_vocab = np.array(context_text_processor.get_vocabulary())
tokens = context_vocab[example_tokens[0].numpy()]
' '.join(tokens)
```

```
'[START] byl bych udelal to same , co ty [END]'
```

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

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

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



Process the dataset

The tensorflow datasets created previosly are converted into 0-padded tensors of token IDs in the process_text function. In order to use keras modle.fit the input must be the context(english) and the taregt(czech) with target in and target ou t lables.

```
def process_text(context, target):
  context = context_text_processor(context).to_tensor()
  target = target_text_processor(target)
  targ_in = target[:,:-1].to_tensor()
  targ_out = target[:,1:].to_tensor()
  return (context, targ_in), targ_out


train_ds = train_raw.map(process_text, tf.data.AUTOTUNE)
val_ds = val_raw.map(process_text, tf.data.AUTOTUNE)
```

Here is the first sequence of each, from the first batch:

```
for (ex_context_tok, ex_tar_in), ex_tar_out in train_ds.take(1):
  print(ex_context_tok[0, :10].numpy())
  print()
  print(ex_tar_in[0, :10].numpy())
  print(ex_tar_out[0, :10].numpy())
```

```
    [   2    1    7  182 4454    3    0    0    0    0]

    [   2   20 2010   13  122 1151    0    0    0    0]
    [  20 2010   13  122 1151    3    0    0    0    0]
```

```
UNITS = 256
```

The encoder

The encoder is used to process the context sequence into a sequence of vectors so that the decoder may use that information to predict the output at each timestep. The sequence is constant so the model uses a bidirectional RNN.

A bidirectional RNN

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, text_processor, units):
    super(Encoder, self).__init__()
    self.text_processor = text_processor
    self.vocab_size = text_processor.vocabulary_size()
    self.units = units

    # The embedding layer converts tokens to vectors
    self.embedding = tf.keras.layers.Embedding(self.vocab_size, units,
                                               mask_zero=True)

    # The RNN layer processes those vectors sequentially.
    self.rnn = tf.keras.layers.Bidirectional(
        merge_mode='sum',
        layer=tf.keras.layers.GRU(units,
                            # Return the sequence and state
                            return_sequences=True,
                            recurrent_initializer='glorot_uniform'))

  def call(self, x):
    shape_checker = ShapeChecker()
    shape_checker(x, 'batch s')

    # 2. The embedding layer looks up the embedding vector for each token.
    x = self.embedding(x)
    shape_checker(x, 'batch s units')

    # 3. The GRU processes the sequence of embeddings.
    x = self.rnn(x)
    shape_checker(x, 'batch s units')

    # 4. Returns the new sequence of embeddings.
    return x

  def convert_input(self, texts):
    texts = tf.convert_to_tensor(texts)
    if len(texts.shape) == 0:
      texts = tf.convert_to_tensor(texts)[tf.newaxis]
    context = self.text_processor(texts).to_tensor()
    context = self(context)
    return context
```

```
# Encode the input sequence.
encoder = Encoder(context_text_processor, UNITS)
ex_context = encoder(ex_context_tok)

print(f'Context tokens, shape (batch, s): {ex_context_tok.shape}')
print(f'Encoder output, shape (batch, s, units): {ex_context.shape}')
```

```
Context tokens, shape (batch, s): (64, 13)
Encoder output, shape (batch, s, units): (64, 13, 256)
```

The attention layer

The attention layer computes the vector sequence and adds the outcome to the decoder.

The attention layer

```
class CrossAttention(tf.keras.layers.Layer):
  def __init__(self, units, **kwargs):
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(key_dim=units, num_heads=1, **kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()

  def call(self, x, context):
    shape_checker = ShapeChecker()

    shape_checker(x, 'batch t units')
    shape_checker(context, 'batch s units')

    attn_output, attn_scores = self.mha(
        query=x,
        value=context,
        return_attention_scores=True)

    shape_checker(x, 'batch t units')
    shape_checker(attn_scores, 'batch heads t s')

    # Cache the attention scores for plotting later.
    attn_scores = tf.reduce_mean(attn_scores, axis=1)
    shape_checker(attn_scores, 'batch t s')
    self.last_attention_weights = attn_scores

    x = self.add([x, attn_output])
    x = self.layernorm(x)

    return x


attention_layer = CrossAttention(UNITS)

# Attend to the encoded tokens
embed = tf.keras.layers.Embedding(target_text_processor.vocabulary_size(),
                                  output_dim=UNITS, mask_zero=True)
ex_tar_embed = embed(ex_tar_in)

result = attention_layer(ex_tar_embed, ex_context)

print(f'Context sequence, shape (batch, s, units): {ex_context.shape}')
print(f'Target sequence, shape (batch, t, units): {ex_tar_embed.shape}')
print(f'Attention result, shape (batch, t, units): {result.shape}')
print(f'Attention weights, shape (batch, t, s):    {attention_layer.last_attention_weights.shape}')
```

```
Context sequence, shape (batch, s, units): (64, 13, 256)
Target sequence, shape (batch, t, units): (64, 13, 256)
Attention result, shape (batch, t, units): (64, 13, 256)
Attention weights, shape (batch, t, s):    (64, 13, 13)
```

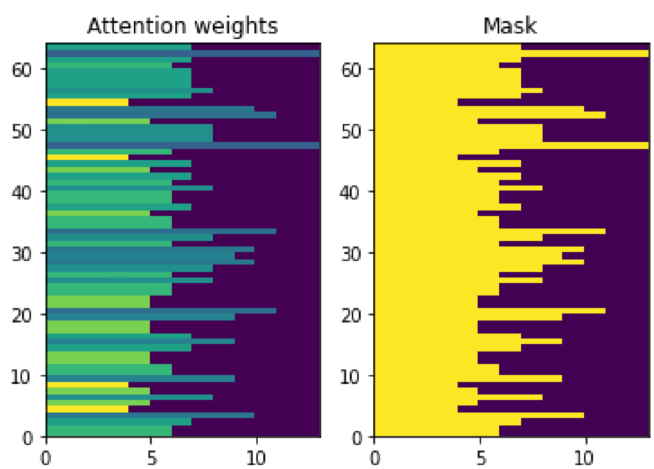
```
attention_layer.last_attention_weights[0].numpy().sum(axis=-1)
```

```
array([0.9999999 , 1.0000001 , 1.        , 1.        , 1.        ,
       1.        , 0.99999994, 0.99999994, 0.99999994, 0.99999994,
       0.99999994, 0.99999994, 0.99999994], dtype=float32)
```

```
attention_weights = attention_layer.last_attention_weights
mask=(ex_context_tok != 0).numpy()

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

plt.subplot(1, 2, 2)
plt.pcolormesh(mask)
plt.title('Mask');
```



The decoder

The decoder will predict tokens at each target sequence. The decoder uses the RNN to process the target sequence and attention to keep track of each generated prediction. The model while training predicts the next word at each location one word at a time.

A unidirectional RNN

```
class Decoder(tf.keras.layers.Layer):
  @classmethod
  def add_method(cls, fun):
    setattr(cls, fun.__name__, fun)
    return fun

  def __init__(self, text_processor, units):
    super(Decoder, self).__init__()
    self.text_processor = text_processor
    self.vocab_size = text_processor.vocabulary_size()
    self.word_to_id = tf.keras.layers.StringLookup(
        vocabulary=text_processor.get_vocabulary(),
        mask_token='', oov_token='[UNK]')
```

```
        mask_token='', oov_token='[UNK]')
    self.id_to_word = tf.keras.layers.StringLookup(
        vocabulary=text_processor.get_vocabulary(),
        mask_token='', oov_token='[UNK]',
        invert=True)
    self.start_token = self.word_to_id('[START]')
    self.end_token = self.word_to_id('[END]')

    self.units = units


    # 1. The embedding layer converts token IDs to vectors
    self.embedding = tf.keras.layers.Embedding(self.vocab_size,
                                               units, mask_zero=True)

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

    # 3. The RNN output will be the query for the attention layer.
    self.attention = CrossAttention(units)

    # 4. This fully connected layer produces the logits for each
    # output token.
    self.output_layer = tf.keras.layers.Dense(self.vocab_size)
```

Training

```
@Decoder.add_method
def call(self,
         context, x,
         state=None,
         return_state=False):
  shape_checker = ShapeChecker()
  shape_checker(x, 'batch t')
  shape_checker(context, 'batch s units')

  # 1. Lookup the embeddings
  x = self.embedding(x)
  shape_checker(x, 'batch t units')

  # 2. Process the target sequence.
  x, state = self.rnn(x, initial_state=state)
  shape_checker(x, 'batch t units')

  # 3. Use the RNN output as the query for the attention over the context.
  x = self.attention(x, context)
  self.last_attention_weights = self.attention.last_attention_weights
  shape_checker(x, 'batch t units')
  shape_checker(self.last_attention_weights, 'batch t s')

  # Step 4. Generate logit predictions for the next token.
  logits = self.output_layer(x)
```

```
    shape_checker(logits, 'batch t target_vocab_size')

    if return_state:
      return logits, state
    else:
      return logits
```

```
decoder = Decoder(target_text_processor, UNITS)
```

Given the context and target tokens, for each taget token it predicts the next target token.

```
logits = decoder(ex_context, ex_tar_in)

print(f'encoder output shape: (batch, s, units) {ex_context.shape}')
print(f'input target tokens shape: (batch, t) {ex_tar_in.shape}')
print(f'logits shape shape: (batch, target_vocabulary_size) {logits.shape}')
```

```
encoder output shape: (batch, s, units) (64, 13, 256)
input target tokens shape: (batch, t) (64, 13)
logits shape shape: (batch, target_vocabulary_size) (64, 13, 5000)
```

```
@Decoder.add_method
def get_initial_state(self, context):
  batch_size = tf.shape(context)[0]
  start_tokens = tf.fill([batch_size, 1], self.start_token)
  done = tf.zeros([batch_size, 1], dtype=tf.bool)
  embedded = self.embedding(start_tokens)
  return start_tokens, done, self.rnn.get_initial_state(embedded)[0]
```

```
@Decoder.add_method
def tokens_to_text(self, tokens):
  words = self.id_to_word(tokens)
  result = tf.strings.reduce_join(words, axis=-1, separator=' ')
  result = tf.strings.regex_replace(result, '^ *\[START\] *', '')
  result = tf.strings.regex_replace(result, ' *\[END\] *$', '')
  return result
```

```
@Decoder.add_method
def get_next_token(self, context, next_token, done, state, temperature = 0.0):
  logits, state = self(
    context, next_token,
    state = state,
    return_state=True)

  if temperature == 0.0:
    next_token = tf.argmax(logits, axis=-1)
  else:
    logits = logits[:, -1, :]/temperature
    next_token = tf.random.categorical(logits, num_samples=1)

  # If a sequence produces an `end_token`, set it `done`
  done = done | (next_token == self.end_token)
  # Once a sequence is done it only produces 0-padding.
```

```
    next_token = tf.where(done, tf.constant(0, dtype=tf.int64), next_token)

    return next_token, done, state

  # Setup the loop variables.
  next_token, done, state = decoder.get_initial_state(ex_context)
  tokens = []

  for n in range(10):
    # Run one step.
    next_token, done, state = decoder.get_next_token(
        ex_context, next_token, done, state, temperature=1.0)
    # Add the token to the output.
    tokens.append(next_token)

  # Stack all the tokens together.
  tokens = tf.concat(tokens, axis=-1) # (batch, t)

  # Convert the tokens back to a a string
  result = decoder.tokens_to_text(tokens)
  result[:3].numpy()
```

```
array([b'smells wont neighbor addictive prune dishes polyglot disagree herself here',
       b'excited trusts factory numbers five regularly retiring tests shadow venice',
       b'sidney sneeze proud repeated fortune eat inviting expense game forever'],
      dtype=object)
```

The model

The model componants include the RNN, attention set up, encoder and decoder functionalities. After each section is implemented the model can be trained on the datasets.

```
class Translator(tf.keras.Model):
  @classmethod
  def add_method(cls, fun):
    setattr(cls, fun.__name__, fun)
    return fun

  def __init__(self, units,
               context_text_processor,
               target_text_processor):
    super().__init__()
    # Build the encoder and decoder
    encoder = Encoder(context_text_processor, units)
    decoder = Decoder(target_text_processor, units)

    self.encoder = encoder
    self.decoder = decoder

  def call(self, inputs):
    context, x = inputs
    context = self.encoder(context)
    logits = self.decoder(context, x)
```

```
    #TODO(b/250038731): remove this
    try:
      # Delete the keras mask, so keras doesn't scale the loss+accuracy.
      del logits._keras_mask
    except AttributeError:
      pass

    return logits


model = Translator(UNITS, context_text_processor, target_text_processor)

logits = model((ex_context_tok, ex_tar_in))

print(f'Context tokens, shape: (batch, s, units) {ex_context_tok.shape}')
print(f'Target tokens, shape: (batch, t) {ex_tar_in.shape}')
print(f'logits, shape: (batch, t, target_vocabulary_size) {logits.shape}')
```

```
Context tokens, shape: (batch, s, units) (64, 13)
Target tokens, shape: (batch, t) (64, 13)
logits, shape: (batch, t, target_vocabulary_size) (64, 13, 5000)
```

Train

```
def masked_loss(y_true, y_pred):
    # Calculate the loss for each item in the batch.
    loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True, reduction='none')
    loss = loss_fn(y_true, y_pred)

    # Mask off the losses on padding.
    mask = tf.cast(y_true != 0, loss.dtype)
    loss *= mask

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


def masked_acc(y_true, y_pred):
    # Calculate the loss for each item in the batch.
    y_pred = tf.argmax(y_pred, axis=-1)
    y_pred = tf.cast(y_pred, y_true.dtype)

    match = tf.cast(y_true == y_pred, tf.float32)
    mask = tf.cast(y_true != 0, tf.float32)

    return tf.reduce_sum(match)/tf.reduce_sum(mask)
```

Configure the model for training:

```
model.compile(optimizer='adam',
              loss=masked_loss,
              metrics=[masked_acc, masked_loss])
```

```
vocab_size = 1.0 * target_text_processor.vocabulary_size()

{"expected_loss": tf.math.log(vocab_size).numpy(),
 "expected_acc": 1/vocab_size}
```

```
{'expected_loss': 8.517193, 'expected_acc': 0.0002}
```

```
model.evaluate(val_ds, steps=20, return_dict=True)
```

```
20/20 [==============================] - 15s 283ms/step - loss: 8.5321 - masked_acc: 1.0460e-04 -
{'loss': 8.53210735321045,
 'masked_acc': 0.00010460250632604584,
 'masked_loss': 8.53210735321045}
```

```
history = model.fit(
    train_ds.repeat(),
    epochs=100,
    steps_per_epoch = 100,
    validation_data=val_ds,
    validation_steps = 20,
    callbacks=[
        tf.keras.callbacks.EarlyStopping(patience=3)])
```

```
Epoch 1/100
100/100 [==============================] - 77s 634ms/step - loss: 5.6003 - masked_acc: 0.1874 - ma
Epoch 2/100
100/100 [==============================] - 63s 633ms/step - loss: 4.3991 - masked_acc: 0.2978 - ma
Epoch 3/100
100/100 [==============================] - 66s 660ms/step - loss: 3.7196 - masked_acc: 0.3716 - ma
Epoch 4/100
100/100 [==============================] - 68s 677ms/step - loss: 3.3129 - masked_acc: 0.4266 - ma
Epoch 5/100
100/100 [==============================] - 65s 652ms/step - loss: 2.7268 - masked_acc: 0.4989 - ma
Epoch 6/100
100/100 [==============================] - 64s 638ms/step - loss: 2.4573 - masked_acc: 0.5412 - ma
Epoch 7/100
100/100 [==============================] - 63s 636ms/step - loss: 2.2954 - masked_acc: 0.5647 - ma
Epoch 8/100
100/100 [==============================] - 68s 678ms/step - loss: 2.1574 - masked_acc: 0.5876 - ma
Epoch 9/100
100/100 [==============================] - 65s 648ms/step - loss: 1.7686 - masked_acc: 0.6396 - ma
Epoch 10/100
100/100 [==============================] - 64s 645ms/step - loss: 1.5720 - masked_acc: 0.6681 - ma
Epoch 11/100
100/100 [==============================] - 63s 631ms/step - loss: 1.5638 - masked_acc: 0.6686 - ma
Epoch 12/100
100/100 [==============================] - 68s 681ms/step - loss: 1.5325 - masked_acc: 0.6743 - ma
Epoch 13/100
100/100 [==============================] - 64s 640ms/step - loss: 1.3271 - masked_acc: 0.7063 - ma
Epoch 14/100
100/100 [==============================] - 66s 659ms/step - loss: 1.1187 - masked_acc: 0.7390 - ma
Epoch 15/100
100/100 [==============================] - 63s 626ms/step - loss: 1.1475 - masked_acc: 0.7295 - ma
Epoch 16/100
100/100 [==============================] - 64s 643ms/step - loss: 1.1915 - masked_acc: 0.7250 - ma
Epoch 17/100
100/100 [==============================] - 64s 637ms/step - loss: 1.0806 - masked_acc: 0.7485 - ma
```

```
Epoch 18/100
100/100 [==============================] - 64s 638ms/step - loss: 0.8230 - masked_acc: 0.7928 - ma
Epoch 19/100
100/100 [==============================] - 65s 645ms/step - loss: 0.8963 - masked_acc: 0.7757 - ma
Epoch 20/100
100/100 [==============================] - 63s 630ms/step - loss: 0.9166 - masked_acc: 0.7712 - ma
Epoch 21/100
100/100 [==============================] - 67s 672ms/step - loss: 0.9380 - masked_acc: 0.7674 - ma
```

Translate

The last step is to implement the translation function.

```
#@title
@Translator.add_method
def translate(self,
              texts, *,
              max_length=50,
              temperature=0.0):
  # Process the input texts
  context = self.encoder.convert_input(texts)
  batch_size = tf.shape(texts)[0]

  # Setup the loop inputs
  tokens = []
  attention_weights = []
  next_token, done, state = self.decoder.get_initial_state(context)

  for _ in range(max_length):
    # Generate the next token
    next_token, done, state = self.decoder.get_next_token(
        context, next_token, done,  state, temperature)

    # Collect the generated tokens
    tokens.append(next_token)
    attention_weights.append(self.decoder.last_attention_weights)

    if tf.executing_eagerly() and tf.reduce_all(done):
      break

  # Stack the lists of tokens and attention weights.
  tokens = tf.concat(tokens, axis=-1)   # t*[(batch 1)] -> (batch, t)
  self.last_attention_weights = tf.concat(attention_weights, axis=1)  # t*[(batch 1 s)] -> (batch, t s)

  result = self.decoder.tokens_to_text(tokens)
  return result
```

```
result = model.translate(['Jsi ještě doma?']) # Are you still home
result[0].numpy().decode()
```

```
'are you still home '
```

The model works fine on short sentences, but when the input increases model loses focus and stops providing acceptable predictions.

One way to improve this model would be to implement re learning the predicted outputs. The model as it is uses teacher-forcing where each correct token is learned by the model regardless of if the predictions are correct or not.

The raw data is sorted by length, so try translating the longest sequence:

```
inputs = [
    'Je tu opravdu zima.', # "It's really cold here."
    'Tohle je můj život.', # "This is my life."
    'Jeho pokoj je nepořádek.' # "His room is a mess"
]
```

```
%%time
for t in inputs:
  print(model.translate([t])[0].numpy().decode())

print()
```

```
its really cold
this is my life doesnt belong to
his room is overconfident

CPU times: user 871 ms, sys: 9.74 ms, total: 881 ms
Wall time: 882 ms
```

File input and outputs for summarization steps.

```
file1 = open("czech_story.txt", "r")
inputs = file1.readlines()
file2 = open("english_story.txt", "w")
for t in inputs:
  file2.writelines(model.translate([t])[0].numpy().decode())

file2.close()
print()
```

Citations

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems,

2015. Software available from tensorflow.org.