neural_machine_translation_attention_mechanism

December 12, 2022

1 Neural Machine Translation with Attention mechanism

[1]:
```
!pip install chart-studio
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting chart-studio
  Downloading chart_studio-1.1.0-py3-none-any.whl (64 kB)
                           64.4/64.4 KB
321.4 kB/s eta 0:00:00a 0:00:01
Requirement already satisfied: six in /usr/lib/python3/dist-packages (from
chart-studio) (1.16.0)
Collecting retrying>=1.3.3
  Downloading retrying-1.3.4-py3-none-any.whl (11 kB)
Collecting plotly
  Downloading plotly-5.11.0-py2.py3-none-any.whl (15.3 MB)
                           15.3/15.3 MB
1.2 MB/s eta 0:00:0000:0100:01m
Requirement already satisfied: requests in /usr/lib/python3/dist-packages
(from chart-studio) (2.25.1)
Collecting tenacity>=6.2.0
  Downloading tenacity-8.1.0-py3-none-any.whl (23 kB)
Installing collected packages: tenacity, retrying, plotly, chart-studio
Successfully installed chart-studio-1.1.0 plotly-5.11.0 retrying-1.3.4
tenacity-8.1.0
```

[2]:
```
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import tensorflow as tf

#tf.enable_eager_execution()

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

import unicodedata
import re
import numpy as np
import time
import string
import chart_studio.plotly
import chart_studio.plotly as py
from plotly.offline import init_notebook_mode, iplot
#%plotly.offline.init_notebook_mode(connected=True)
import plotly.graph_objs as go

# import os
# for dirname, _, filenames in os.walk('/kaggle/input'):
#     for filename in filenames:
#         print(os.path.join(dirname, filename))
```

1.0.1 As in case of any NLP task, after reading the input file, we perform the basic cleaning and preprocessing as follows:

```
[3]: file_path = 'hin.txt' # please set the path according to your system
```

```
[4]: lines = open(file_path, encoding='UTF-8').read().strip().split('\n')
     lines[2000:2010]
```

```
[4]: ['I was able to play piano very well.\t                  \tCC-BY
     2.0 (France) Attribution: tatoeba.org #28988 (CK) & #505130 (minshirui)',
      'I wish to go to Paris to study art.\t                  \tCC-
     BY 2.0 (France) Attribution: tatoeba.org #256599 (CK) & #450412 (minshirui)',
      'I wish to go to Paris to study art.\t                  \tCC-
     BY 2.0 (France) Attribution: tatoeba.org #256599 (CK) & #450413 (minshirui)',
      "I'm anxious for him to return safe.\t
      \tCC-BY 2.0 (France) Attribution: tatoeba.org #284204 (CM) & #505252
     (minshirui)",
      "If I don't do it now, I never will.\t
       \tCC-BY 2.0 (France) Attribution: tatoeba.org #2193 (CK) & #494208
     (minshirui)",
      'If I had wings, I would fly to you.\t
               \tCC-BY 2.0 (France) Attribution: tatoeba.org #30785 (CM) &
     #459446 (minshirui)',
      'It has been fine since last Friday.\t
      \tCC-BY 2.0 (France) Attribution: tatoeba.org #272726 (CK) & #491591
     (minshirui)',
      'It is going to rain this afternoon.\t                \tCC-BY 2.0
     (France) Attribution: tatoeba.org #240067 (Zifre) & #487252 (minshirui)',
      'It seems that everybody likes golf.\t
            \tCC-BY 2.0 (France) Attribution: tatoeba.org #40404 (CK) &
     #509344 (minshirui)',
      "It shouldn't take long to find Tom.\t
        \tCC-BY 2.0 (France) Attribution: tatoeba.org #3540364 (CK) & #3540404
     (nurendra)"]
```

[5]:

```
print("total number of records: ",len(lines))
```

```
total number of records:  2909
```

[6]:

```
exclude = set(string.punctuation) # Set of all special characters
remove_digits = str.maketrans('', '', string.digits) # Set of all digits
```

1.0.2 Function to preprocess English sentence

[7]:

```
def preprocess_eng_sentence(sent):
    '''Function to preprocess English sentence'''
    sent = sent.lower() # lower casing
    sent = re.sub("'", '', sent) # remove the quotation marks if any
    sent = ''.join(ch for ch in sent if ch not in exclude)
    sent = sent.translate(remove_digits) # remove the digits
    sent = sent.strip()
    sent = re.sub(" +", " ", sent) # remove extra spaces
    sent = '<start> ' + sent + ' <end>' # add <start> and <end> tokens
    return sent
```

1.0.3 Function to preprocess Hindi sentence

[8]:

```
def preprocess_port_sentence(sent):
    '''Function to preprocess Hindi sentence'''
    sent = re.sub("'", '', sent) # remove the quotation marks if any
    sent = ''.join(ch for ch in sent if ch not in exclude)
    #sent = re.sub("[      ]", "", sent) # remove the digits
    sent = sent.strip()
    sent = re.sub(" +", " ", sent) # remove extra spaces
    sent = '<start> ' + sent + ' <end>' # add <start> and <end> tokens
    return sent
```

1.0.4 Generate pairs of cleaned English and Hindi sentences with start and end tokens added.

[10]:

```
# Generate pairs of cleaned English and Hindi sentences
sent_pairs = []
for line in lines:
    sent_pair = []
    eng = line.rstrip().split('\t')[0]
    port = line.rstrip().split('\t')[1]
    eng = preprocess_eng_sentence(eng)
    sent_pair.append(eng)
    port = preprocess_port_sentence(port)
    sent_pair.append(port)
    sent_pairs.append(sent_pair)
```

```
sent_pairs[2000:2010]
```

[10]:

```
[['<start> i was able to play piano very well <end>',
  '<start>                    <end>'],
 ['<start> i wish to go to paris to study art <end>',
  '<start>                    <end>'],
 ['<start> i wish to go to paris to study art <end>',
  '<start>                    <end>'],
 ['<start> im anxious for him to return safe <end>',
  '<start>                     <end>'],
 ['<start> if i dont do it now i never will <end>',
  '<start>                        <end>'],
 ['<start> if i had wings i would fly to you <end>',
  '<start>                                <end>'],
 ['<start> it has been fine since last friday <end>',
  '<start>                     <end>'],
 ['<start> it is going to rain this afternoon <end>',
  '<start>                  <end>'],
 ['<start> it seems that everybody likes golf <end>',
  '<start>                                <end>'],
 ['<start> it shouldnt take long to find tom <end>',
  '<start>                      <end>']]
```

1.0.5 Create a class to map every word to an index and vice-versa for any given vocabulary.

[ ]:

```
class LanguageIndex():
    def __init__(self, lang):
        self.lang = lang
        self.word2idx = {}
        self.idx2word = {}
        self.vocab = set()

        self.create_index()

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

        self.vocab = sorted(self.vocab)

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

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

[14]:

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

1.0.6 Tokenization and Padding

[15]:

```
def load_dataset(pairs, num_examples):
    # pairs => already created cleaned input, output pairs

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

    # Vectorize the input and target languages

    # English sentences
    input_tensor = [[inp_lang.word2idx[s] for s in en.split(' ')] for en, ma in␣
 ↪pairs]

    # Hindi sentences
    target_tensor = [[targ_lang.word2idx[s] for s in ma.split(' ')] for en, ma␣
 ↪in pairs]

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

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

    target_tensor = tf.keras.preprocessing.sequence.pad_sequences(target_tensor,
                                                                ␣
 ↪maxlen=max_length_tar,
                                                                ␣
 ↪padding='post')

    return input_tensor, target_tensor, inp_lang, targ_lang, max_length_inp,␣
 ↪max_length_tar
```

[16]:

```
input_tensor, target_tensor, inp_lang, targ_lang, max_length_inp,␣
 ↪max_length_targ = load_dataset(sent_pairs, len(lines))
```

1.0.7 Creating training and validation sets using an 80-20 split

[17]:
```
# 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(input_tensor, target_tensor, test_size=0.1, random_state =␣
 ↪101)

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

[17]:
```
(2618, 2618, 291, 291)
```

[18]:
```
BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
N_BATCH = BUFFER_SIZE//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(inp_lang.word2idx)
vocab_tar_size = len(targ_lang.word2idx)

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

We'll be using GRUs instead of LSTMs as we only have to create one state and implementation would be easier.

1.0.8 Create GRU units

[19]:
```
def gru(units):

    return tf.keras.layers.GRU(units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_activation='sigmoid',
                                   recurrent_initializer='glorot_uniform')
```

1.0.9 The next step is to define the encoder and decoder network.

The input to the encoder will be the sentence in English and the output will be the hidden state and cell state of the GRU.

[20]:
```
class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
        super(Encoder, self).__init__()
        self.batch_sz = batch_sz
        self.enc_units = enc_units
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = gru(self.enc_units)

    def call(self, x, hidden):
        x = self.embedding(x)
        output, state = self.gru(x, initial_state = hidden)
        return output, state

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

The next step is to define the decoder. The decoder will have two inputs: the hidden state and cell state from the encoder and the input sentence, which actually will be the output sentence with a token appended at the beginning.

[21]:

```
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
        super(Decoder, self).__init__()
        self.batch_sz = batch_sz
        self.dec_units = dec_units
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = gru(self.dec_units)
        self.fc = tf.keras.layers.Dense(vocab_size)

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

    def call(self, x, hidden, enc_output):

        hidden_with_time_axis = tf.expand_dims(hidden, 1)

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

        # attention_weights shape == (batch_size, max_length, 1)
        attention_weights = tf.nn.softmax(score, axis=1)

        # context_vector shape after sum == (batch_size, hidden_size)
        context_vector = attention_weights * enc_output
        context_vector = tf.reduce_sum(context_vector, axis=1)
        # x shape after passing through embedding == (batch_size, 1,␣
 ↪embedding_dim)
        x = self.embedding(x)

        # x shape after concatenation == (batch_size, 1, embedding_dim +␣
 ↪hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

        # passing the concatenated vector to the GRU
        output, state = self.gru(x)

        # output shape == (batch_size * 1, hidden_size)
        output = tf.reshape(output, (-1, output.shape[2]))

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

        return x, state, attention_weights

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

Create encoder and decoder objects from their respective classes.

[22]:

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

1.0.10 Define the optimizer and the loss function.

[23]:

```
optimizer = tf.optimizers.Adam()

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

[27]:

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

1.0.11 Training the Model

[28]:

```
EPOCHS = 10

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

    hidden = encoder.initialize_hidden_state()
    total_loss = 0

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

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

            dec_hidden = enc_hidden

            dec_input = tf.expand_dims([targ_lang.word2idx['<start>']] *␣
 ↪BATCH_SIZE, 1)

            # Teacher forcing - feeding the target as the next input
            for t in range(1, targ.shape[1]):
                # passing enc_output to the decoder
                predictions, dec_hidden, _ = decoder(dec_input, dec_hidden,␣
 ↪enc_output)

                loss += loss_function(targ[:, t], predictions)

                # using teacher forcing
                dec_input = tf.expand_dims(targ[:, t], 1)

        batch_loss = (loss / int(targ.shape[1]))

        total_loss += batch_loss

        variables = encoder.variables + decoder.variables

        gradients = tape.gradient(loss, variables)

        optimizer.apply_gradients(zip(gradients, variables))

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

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

```
Epoch 1 Batch 0 Loss 1.6608
Epoch 1 Loss 1.7606
Time taken for 1 epoch 307.0648567676544 sec

Epoch 2 Batch 0 Loss 1.6687
Epoch 2 Loss 1.5896
Time taken for 1 epoch 307.0440068244934 sec

Epoch 3 Batch 0 Loss 1.6300
Epoch 3 Loss 1.4876
Time taken for 1 epoch 305.1629283428192 sec

Epoch 4 Batch 0 Loss 1.4389
Epoch 4 Loss 1.4083
Time taken for 1 epoch 307.97819995880127 sec

Epoch 5 Batch 0 Loss 1.4190
Epoch 5 Loss 1.3271
Time taken for 1 epoch 307.28461718559265 sec

Epoch 6 Batch 0 Loss 1.2548
Epoch 6 Loss 1.2491
Time taken for 1 epoch 302.8826787471771 sec

Epoch 7 Batch 0 Loss 1.1702
Epoch 7 Loss 1.1843
Time taken for 1 epoch 306.29307293891907 sec

Epoch 8 Batch 0 Loss 1.0927
Epoch 8 Loss 1.1126
Time taken for 1 epoch 303.90669417381287 sec

Epoch 9 Batch 0 Loss 0.9682
Epoch 9 Loss 1.0431
Time taken for 1 epoch 303.37123346328735 sec

Epoch 10 Batch 0 Loss 0.9995
Epoch 10 Loss 0.9771
Time taken for 1 epoch 306.5512282848358 sec
```

1.0.12 Restoring the latest checkpoint

[ ]:

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

[ ]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at
0x7f6798ba34d0>
```

1.0.13 Inference setup and testing:

[29]:

```
def evaluate(inputs, encoder, decoder, inp_lang, targ_lang, max_length_inp,␣
 ↪max_length_targ):

    attention_plot = np.zeros((max_length_targ, max_length_inp))
    sentence = ''
    for i in inputs[0]:
        if i == 0:
            break
        sentence = sentence + inp_lang.idx2word[i] + ' '
    sentence = sentence[:-1]

    inputs = tf.convert_to_tensor(inputs)

    result = ''

    hidden = [tf.zeros((1, units))]
    enc_out, enc_hidden = encoder(inputs, hidden)

    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([targ_lang.word2idx['<start>']], 0)

    for t in range(max_length_targ):
        predictions, dec_hidden, attention_weights = decoder(dec_input,␣
 ↪dec_hidden, 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.idx2word[predicted_id] + ' '

        if targ_lang.idx2word[predicted_id] == '<end>':
            return result, sentence, attention_plot

        # the predicted ID is fed back into the model
        dec_input = tf.expand_dims([predicted_id], 0)

    return result, sentence, attention_plot
```

1.0.14 Function to predict (translate) a randomly selected test point

[30]:

```
def predict_random_val_sentence():
    actual_sent = ''
    k = np.random.randint(len(input_tensor_val))
    random_input = input_tensor_val[k]
    random_output = target_tensor_val[k]
    random_input = np.expand_dims(random_input,0)
    result, sentence, attention_plot = evaluate(random_input, encoder, decoder,␣
 ↪inp_lang, targ_lang, max_length_inp, max_length_targ)
    print('Input: {}'.format(sentence[8:-6]))
    print('Predicted translation: {}'.format(result[:-6]))
    for i in random_output:
        if i == 0:
            break
        actual_sent = actual_sent + targ_lang.idx2word[i] + ' '
    actual_sent = actual_sent[8:-7]
    print('Actual translation: {}'.format(actual_sent))
    attention_plot = attention_plot[:len(result.split(' '))-2, 1:len(sentence.
 ↪split(' '))-1]
    sentence, result = sentence.split(' '), result.split(' ')
    sentence = sentence[1:-1]
    result = result[:-2]

    # use plotly to generate the heat map
    trace = go.Heatmap(z = attention_plot, x = sentence, y = result,␣
 ↪colorscale='greens')
    data=[trace]
    iplot(data)
```

[31]:

```
predict_random_val_sentence()
```

```
Input: that town is two miles away
Predicted translation:
Actual translation:
```

[32]:

```
predict_random_val_sentence()
```

```
Input: turn your face this way
Predicted translation:
Actual translation:
```

[33]:

```
predict_random_val_sentence()
```

```
Input: he came to see you yesterday
Predicted translation:
Actual translation:
```

```
[34]: predict_random_val_sentence()
```

```
Input: shall i get you a chair
Predicted translation:
Actual translation:
```

```
[35]: predict_random_val_sentence()
```

```
Input: weve decided to leave this town tomorrow morning
Predicted translation:
Actual translation:
```

```
[36]: predict_random_val_sentence()
```

```
Input: ill give you an answer in a day or two
Predicted translation:
Actual translation:
```

```
[37]: predict_random_val_sentence()
```

```
Input: he jumped into water without removing his clothes
Predicted translation:

Actual translation:
```

```
[38]: predict_random_val_sentence()
```

```
Input: we can hear the ocean from here
Predicted translation:
Actual translation:
```

```
[ ]:
```