seq2seq

April 24, 2019

1 Home 4: Build a seq2seq model for machine translation.

1.0.1 Name: Yao Xiao

1.0.2 Task: Translate English to Spanish

1.1 1. Data preparation

1. Download data (e.g., "deu-eng.zip") from http://www.manythings.org/anki/
2. Unzip the .ZIP file.
3. Put the .TXT file (e.g., "deu.txt") in the directory "./Data/".

1.1.1 1.1. Load and clean text

In [1]:
```
import re
import string
from unicodedata import normalize
import numpy

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, mode='rt', encoding='utf-8')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text


# split a loaded document into sentences
def to_pairs(doc):
    lines = doc.strip().split('\n')
    pairs = [line.split('\t') for line in  lines]
    return pairs

def clean_data(lines):
    cleaned = list()
```

```
    # prepare regex for char filtering
    re_print = re.compile('[^%s]' % re.escape(string.printable))
    # prepare translation table for removing punctuation
    table = str.maketrans('', '', string.punctuation)
    for pair in lines:
        clean_pair = list()
        for line in pair:
            # normalize unicode characters
            line = normalize('NFD', line).encode('ascii', 'ignore')
            line = line.decode('UTF-8')
            # tokenize on white space
            line = line.split()
            # convert to lowercase
            line = [word.lower() for word in line]
            # remove punctuation from each token
            line = [word.translate(table) for word in line]
            # remove non-printable chars form each token
            line = [re_print.sub('', w) for w in line]
            # remove tokens with numbers in them
            line = [word for word in line if word.isalpha()]
            # store as string
            clean_pair.append(' '.join(line))
        cleaned.append(clean_pair)
    return numpy.array(cleaned)
```

In [2]:
```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

# encode and pad sequences
def text2sequences(max_len, lines):
    tokenizer = Tokenizer(char_level=True, filters='')
    tokenizer.fit_on_texts(lines)
    seqs = tokenizer.texts_to_sequences(lines)
    seqs_pad = pad_sequences(seqs, maxlen=max_len, padding='post')
    return seqs_pad, tokenizer.word_index
```

```
/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the se
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

In [3]:
```
from keras.utils import to_categorical

# one hot encode target sequence
def onehot_encode(sequences, max_len, vocab_size):
    n = len(sequences)
    data = numpy.zeros((n, max_len, vocab_size))
    for i in range(n):
```

```
            data[i, :, :] = to_categorical(sequences[i], num_classes=vocab_size)
        return data
```

Fill the following blanks:

```
In [28]: # e.g., filename = 'Data/deu.txt'
         filename = 'deu.txt'

         # e.g., n_train = 20000
         n_train = 20000
```

```
In [29]: # load dataset
         doc = load_doc(filename)

         # split into Language1-Language2 pairs
         pairs = to_pairs(doc)

         # clean sentences
         clean_pairs = clean_data(pairs)[0:n_train, :]
```

```
In [30]: for i in range(3000, 3010):
             print('[' + clean_pairs[i, 0] + '] => [' + clean_pairs[i, 1] + ']')
```

```
[is it broken] => [ist es kaputt]
[is it for me] => [ist das fur mich]
[is that a no] => [ist das ein nein]
[is that love] => [ist das liebe]
[is that mine] => [ist das meins]
[is that snow] => [ist das schnee]
[is that true] => [ist das wahr]
[is this love] => [ist das liebe]
[is this love] => [ist das hier liebe]
[is this mine] => [ist das meines]
```

```
In [31]: input_texts = clean_pairs[:, 0]
         target_texts = ['\t' + text + '\n' for text in clean_pairs[:, 1]]

         print('Length of input_texts:  ' + str(input_texts.shape))
         print('Length of target_texts: ' + str(input_texts.shape))
```

```
Length of input_texts:  (20000,)
Length of target_texts: (20000,)
```

```
In [32]: max_encoder_seq_length = max(len(line) for line in input_texts)
         max_decoder_seq_length = max(len(line) for line in target_texts)

         print('max length of input  sentences: %d' % (max_encoder_seq_length))
         print('max length of target sentences: %d' % (max_decoder_seq_length))
```

```
max length of input  sentences: 17
max length of target sentences: 73
```

1.2 2. Text processing

1.2.1 2.1. Convert texts to sequences

- Input: A list of n sentences (with max length t).
- It is represented by a $n \times t$ matrix after the tokenization and zero-padding.

```
In [33]: from keras.preprocessing.text import Tokenizer
         from keras.preprocessing.sequence import pad_sequences

         # encode and pad sequences
         def text2sequences(max_len, lines):
             tokenizer = Tokenizer(char_level=True, filters='')
             tokenizer.fit_on_texts(lines)
             seqs = tokenizer.texts_to_sequences(lines)
             seqs_pad = pad_sequences(seqs, maxlen=max_len, padding='post')
             return seqs_pad, tokenizer.word_index

         encoder_input_seq, input_token_index = text2sequences(max_encoder_seq_length, input_te
         decoder_input_seq, target_token_index = text2sequences(max_decoder_seq_length, target_

         print('shape of encoder_input_seq: ' + str(encoder_input_seq.shape))
         print('shape of input_token_index: ' + str(len(input_token_index)))
         print('shape of decoder_input_seq: ' + str(decoder_input_seq.shape))
         print('shape of target_token_index: ' + str(len(target_token_index)))
```

```
shape of encoder_input_seq: (20000, 17)
shape of input_token_index: 27
shape of decoder_input_seq: (20000, 73)
shape of target_token_index: 29
```

```
In [34]: num_encoder_tokens = len(input_token_index) + 1
         num_decoder_tokens = len(target_token_index) + 1

         print('num_encoder_tokens: ' + str(num_encoder_tokens))
         print('num_decoder_tokens: ' + str(num_decoder_tokens))
```

```
num_encoder_tokens: 28
num_decoder_tokens: 30
```

Remark: To this end, the input language and target language texts are converted to 2 matrices.

- Their number of rows are both n_train.

- Their number of columns are respective max_encoder_seq_length and max_decoder_seq_length.

The followings print a sentence and its representation as a sequence.

```
In [35]: target_texts[100]
```

```
Out[35]: '\tmach dich fort\n'
```

```
In [36]: decoder_input_seq[100, :]
```

```
Out[36]: array([ 8, 13, 10, 12,  7,  1, 16,  3, 12,  7,  1, 21, 15, 11,  4,  9,  0,
                 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
                 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
                 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
                 0,  0,  0,  0,  0], dtype=int32)
```

1.3 2.2. One-hot encode

- Input: A list of n sentences (with max length t).
- It is represented by a $n \times t$ matrix after the tokenization and zero-padding.
- It is represented by a $n \times t \times v$ tensor (t is the number of unique chars) after the one-hot encoding.

```
In [37]: from keras.utils import to_categorical

         # one hot encode target sequence
         def onehot_encode(sequences, max_len, vocab_size):
             n = len(sequences)
             data = numpy.zeros((n, max_len, vocab_size))
             for i in range(n):
                 data[i, :, :] = to_categorical(sequences[i], num_classes=vocab_size)
             return data

         encoder_input_data = onehot_encode(encoder_input_seq, max_encoder_seq_length, num_enc
         decoder_input_data = onehot_encode(decoder_input_seq, max_decoder_seq_length, num_dec

         decoder_target_seq = numpy.zeros(decoder_input_seq.shape)
         decoder_target_seq[:, 0:-1] = decoder_input_seq[:, 1:]
         decoder_target_data = onehot_encode(decoder_target_seq, max_decoder_seq_length, num_d

         print(encoder_input_data.shape)
         print(decoder_input_data.shape)
```

```
(20000, 17, 28)
(20000, 73, 30)
```

1.4 3. Build the networks (for training)

- Build encoder, decoder, and connect the two modules to get "model".
- Fit the model on the bilingual data to train the parameters in the encoder and decoder.

1.4.1 3.1. Encoder network

- Input: one-hot encode of the input language
- Return:

  -- output (all the hidden states $h_1, \cdots, h_t$) are always discarded

  -- the final hidden state h_t

  -- the final conveyor belt c_t

In [38]:
```
from keras.layers import Input, LSTM
from keras.models import Model

latent_dim = 256

# inputs of the encoder network
encoder_inputs = Input(shape=(None, num_encoder_tokens), name='encoder_inputs')

# set the LSTM layer
encoder_lstm = LSTM(latent_dim, return_state=True, dropout=0.5, name='encoder_lstm')_

# build the encoder network model
encoder_model = Model(inputs=encoder_inputs, outputs=[state_h, state_c], name='encode
```

Print a summary and save the encoder network structure to "./encoder.pdf"

In [39]:
```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot, plot_model

SVG(model_to_dot(encoder_model, show_shapes=False).create(prog='dot', format='svg'))

plot_model(
    model=encoder_model, show_shapes=False,
    to_file='encoder.pdf'
)

encoder_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_inputs (InputLayer)  (None, None, 28)          0
_________________________________________________________________
encoder_lstm (LSTM)          [(None, 256), (None, 256) 291840
```

```
=================================================================
Total params: 291,840
Trainable params: 291,840
Non-trainable params: 0
_________________________________________________________________
```

1.4.2 3.2. Decoder network

- Inputs:

  -- one-hot encode of the target language

  -- The initial hidden state h_t

  -- The initial conveyor belt c_t

- Return:

  -- output (all the hidden states) $h_1, \cdots, h_t$

  -- the final hidden state h_t (discarded in the training and used in the prediction)

  -- the final conveyor belt c_t (discarded in the training and used in the prediction)

In [40]:
```python
from keras.layers import Input, LSTM, Dense
from keras.models import Model

# inputs of the decoder network
decoder_input_h = Input(shape=(latent_dim,), name='decoder_input_h')
decoder_input_c = Input(shape=(latent_dim,), name='decoder_input_c')
decoder_input_x = Input(shape=(None, num_decoder_tokens), name='decoder_input_x')

# set the LSTM layer
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True, dropout=0.5
decoder_lstm_outputs, state_h, state_c = decoder_lstm(decoder_input_x, initial_state=

# set the dense layer
decoder_dense = Dense(num_decoder_tokens, activation='softmax', name='decoder_dense')
decoder_outputs = decoder_dense(decoder_lstm_outputs)

# build the decoder network model
decoder_model = Model(inputs=[decoder_input_x, decoder_input_h, decoder_input_c], outp
```

Print a summary and save the encoder network structure to "./decoder.pdf"

In [41]:
```python
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot, plot_model

SVG(model_to_dot(decoder_model, show_shapes=False).create(prog='dot', format='svg'))

plot_model(
    model=decoder_model, show_shapes=False,
```

```
    to_file='decoder.pdf'
)

decoder_model.summary()
```

```
________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
========================================================================================
decoder_input_x (InputLayer)    (None, None, 30)     0
________________________________________________________________________________________
decoder_input_h (InputLayer)    (None, 256)          0
________________________________________________________________________________________
decoder_input_c (InputLayer)    (None, 256)          0
________________________________________________________________________________________
decoder_lstm (LSTM)             [(None, None, 256),  293888      decoder_input_x[0][0]
                                                                 decoder_input_h[0][0]
                                                                 decoder_input_c[0][0]
________________________________________________________________________________________
decoder_dense (Dense)           (None, None, 30)     7710        decoder_lstm[0][0]
========================================================================================
Total params: 301,598
Trainable params: 301,598
Non-trainable params: 0
________________________________________________________________________________________
```

1.4.3 3.3. Connect the encoder and decoder

```
In [42]: # input layers
         encoder_input_x = Input(shape=(None, num_encoder_tokens), name='encoder_input_x')
         decoder_input_x = Input(shape=(None, num_decoder_tokens), name='decoder_input_x')

         # connect encoder to decoder
         encoder_final_states = encoder_model([encoder_input_x])
         decoder_lstm_output, _, _ = decoder_lstm(decoder_input_x, initial_state=encoder_final_
         decoder_pred = decoder_dense(decoder_lstm_output)

         model = Model(inputs=[encoder_input_x, decoder_input_x], outputs=decoder_pred, name='m
```

```
In [43]: print(state_h)
         print(decoder_input_h)
```

```
Tensor("decoder_lstm_2/while/Exit_2:0", shape=(?, 256), dtype=float32)
Tensor("decoder_input_h_1:0", shape=(?, 256), dtype=float32)
```

```
In [44]: from IPython.display import SVG
         from keras.utils.vis_utils import model_to_dot, plot_model
```

```
SVG(model_to_dot(model, show_shapes=False).create(prog='dot', format='svg'))

plot_model(
    model=model, show_shapes=False,
    to_file='model_training.pdf'
)

model.summary()
```

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
encoder_input_x (InputLayer)    (None, None, 28)     0

__________________________________________________________________________________________________
decoder_input_x (InputLayer)    (None, None, 30)     0

__________________________________________________________________________________________________
encoder (Model)                 [(None, 256), (None, 291840      encoder_input_x[0][0]

__________________________________________________________________________________________________
decoder_lstm (LSTM)             [(None, None, 256),  293888      decoder_input_x[0][0]
                                                                 encoder[1][0]
                                                                 encoder[1][1]
__________________________________________________________________________________________________
decoder_dense (Dense)           (None, None, 30)     7710        decoder_lstm[1][0]
==================================================================================================
Total params: 593,438
Trainable params: 593,438
Non-trainable params: 0
__________________________________________________________________________________________________
```

1.4.4 3.5. Fit the model on the bilingual dataset

- encoder_input_data: one-hot encode of the input language
- decoder_input_data: one-hot encode of the input language
- decoder_target_data: labels (left shift of decoder_input_data)
- tune the hyper-parameters
- stop when the validation loss stop decreasing.

```
In [45]: print('shape of encoder_input_data' + str(encoder_input_data.shape))
         print('shape of decoder_input_data' + str(decoder_input_data.shape))
         print('shape of decoder_target_data' + str(decoder_target_data.shape))
```

```
shape of encoder_input_data(20000, 17, 28)
shape of decoder_input_data(20000, 73, 30)
shape of decoder_target_data(20000, 73, 30)
```

```
In [20]: model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

         model.fit([encoder_input_data, decoder_input_data],  # training data
                   decoder_target_data,                        # labels (left shift of the targ
                   batch_size=64, epochs=50, validation_split=0.2)

         model.save('seq2seq.h5')
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/50
16000/16000 [==============================] - 50s 3ms/step - loss: 0.8515 - val_loss: 0.7213
Epoch 2/50
16000/16000 [==============================] - 49s 3ms/step - loss: 0.6080 - val_loss: 0.5911
Epoch 3/50
16000/16000 [==============================] - 49s 3ms/step - loss: 0.5356 - val_loss: 0.5439
Epoch 4/50
16000/16000 [==============================] - 51s 3ms/step - loss: 0.4936 - val_loss: 0.5130
Epoch 5/50
16000/16000 [==============================] - 50s 3ms/step - loss: 0.4650 - val_loss: 0.4882
Epoch 6/50
16000/16000 [==============================] - 48s 3ms/step - loss: 0.4436 - val_loss: 0.4702
Epoch 7/50
16000/16000 [==============================] - 50s 3ms/step - loss: 0.4262 - val_loss: 0.4541
Epoch 8/50
16000/16000 [==============================] - 48s 3ms/step - loss: 0.4127 - val_loss: 0.4408
Epoch 9/50
16000/16000 [==============================] - 47s 3ms/step - loss: 0.3998 - val_loss: 0.4310
Epoch 10/50
16000/16000 [==============================] - 47s 3ms/step - loss: 0.3883 - val_loss: 0.4210
Epoch 11/50
16000/16000 [==============================] - 52s 3ms/step - loss: 0.3793 - val_loss: 0.4119
Epoch 12/50
16000/16000 [==============================] - 43s 3ms/step - loss: 0.3714 - val_loss: 0.4031
Epoch 13/50
16000/16000 [==============================] - 44s 3ms/step - loss: 0.3625 - val_loss: 0.3954
Epoch 14/50
16000/16000 [==============================] - 45s 3ms/step - loss: 0.3556 - val_loss: 0.3902
Epoch 15/50
16000/16000 [==============================] - 46s 3ms/step - loss: 0.3493 - val_loss: 0.3842
Epoch 16/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3426 - val_loss: 0.3773
Epoch 17/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3366 - val_loss: 0.3725
Epoch 18/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3313 - val_loss: 0.3673
Epoch 19/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3263 - val_loss: 0.3615
Epoch 20/50
```

```
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3213 - val_loss: 0.3601
Epoch 21/50
16000/16000 [==============================] - 43s 3ms/step - loss: 0.3163 - val_loss: 0.3582
Epoch 22/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3131 - val_loss: 0.3505
Epoch 23/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3075 - val_loss: 0.3489
Epoch 24/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3041 - val_loss: 0.3442
Epoch 25/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.3008 - val_loss: 0.3423
Epoch 26/50
16000/16000 [==============================] - 45s 3ms/step - loss: 0.2975 - val_loss: 0.3392
Epoch 27/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.2943 - val_loss: 0.3367
Epoch 28/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.2904 - val_loss: 0.3347
Epoch 29/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.2867 - val_loss: 0.3319
Epoch 30/50
16000/16000 [==============================] - 45s 3ms/step - loss: 0.2842 - val_loss: 0.3313
Epoch 31/50
16000/16000 [==============================] - 46s 3ms/step - loss: 0.2832 - val_loss: 0.3312
Epoch 32/50
16000/16000 [==============================] - 47s 3ms/step - loss: 0.2798 - val_loss: 0.3284
Epoch 33/50
16000/16000 [==============================] - 47s 3ms/step - loss: 0.2762 - val_loss: 0.3272
Epoch 34/50
16000/16000 [==============================] - 47s 3ms/step - loss: 0.2747 - val_loss: 0.3221
Epoch 35/50
16000/16000 [==============================] - 45s 3ms/step - loss: 0.2715 - val_loss: 0.3231
Epoch 36/50
16000/16000 [==============================] - 47s 3ms/step - loss: 0.2688 - val_loss: 0.3211
Epoch 37/50
16000/16000 [==============================] - 45s 3ms/step - loss: 0.2668 - val_loss: 0.3210
Epoch 38/50
16000/16000 [==============================] - 43s 3ms/step - loss: 0.2640 - val_loss: 0.3180
Epoch 39/50
16000/16000 [==============================] - 41s 3ms/step - loss: 0.2624 - val_loss: 0.3207
Epoch 40/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.2601 - val_loss: 0.3181
Epoch 41/50
16000/16000 [==============================] - 43s 3ms/step - loss: 0.2587 - val_loss: 0.3142
Epoch 42/50
16000/16000 [==============================] - 42s 3ms/step - loss: 0.2561 - val_loss: 0.3135
Epoch 43/50
16000/16000 [==============================] - 43s 3ms/step - loss: 0.2545 - val_loss: 0.3146
Epoch 44/50
```