Word Model

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", f
```

```
from __future__ import print_function
from keras.models import Model
from keras.layers import Input, LSTM, GRU, Dense, Activation
from keras.activations import softmax as Softmax
from keras.preprocessing.sequence import pad_sequences
from keras.callbacks import ModelCheckpoint
from nltk.translate.bleu_score import corpus_bleu
from keras.utils import to_categorical
from keras import backend as K
import numpy as np
import unicodedata
import re
import tensorflow as tf
```

```
# clean unwanted data
sep = '\tCC'
with open("/content/drive/My Drive/DeepLearning/mar.txt") as file_in:
    lines = []
    for line in file_in:
        res = line.split(sep,1)[0]
        #print(res)
        lines.append(res + "\n")
f = open('/content/drive/My Drive/DeepLearning/mar_1.txt','w')
for line in lines:
  f.write(line)
f.close()
```

```
# modified text file
with open("/content/drive/My Drive/DeepLearning/mar_1.txt", "r") as a:
  print (a.readline())
```

```
Go.	जा.
```

```
BATCH_SIZE = 32  # Batch size for training
NUM_SAMPLES = 10000
EPOCHS = 50
OPTIMIZER = "rmsprop"
EMBED_DIM = 300
HIDDEN_DIM = 50
DATA_PATH = '/content/drive/My Drive/DeepLearning/mar_1.txt'
```

```
def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s)
                   if unicodedata.category(c) != 'Mn')
```

```
def preprocess_sentence(w):
    w = unicode_to_ascii(w.lower().strip())

    # creating a space between words and punctation following it
    # eg: "he is a boy." => "he is a boy ."
    w = re.sub(r"([?.!,¿])", r" \1 ", w)
    w = re.sub(r'[" "]+', " ", w)

    # replacing everything with space except (a-z, A-Z, ".", "?", "!", ",")
```

```
        #w = re.sub(r"[^a-zA-Z?.!,¿]+", " ", w)

        w = w.rstrip().strip()

        # adding a start and an end token to the sequence
        # so that the model know when to start and stop predicting
        w = "\t " + w + " \n"
        return w


    def loadGloveModel(gloveFile):
        print("Loading Glove Model")
        f = open(gloveFile, 'r', encoding = "utf8")
        model = {}
        for line in f:
            splitLine = line.split()
            word = splitLine[0]
            embedding = np.array([float(val) for val in splitLine[1:]])
            model[word] = embedding
        print("Done.", len(model), " words loaded!")
        return model


    def load_embedding(filename):
        print("Loading Glove Model")
        embedding_model = {}
        f = open(filename,'r', encoding="utf8")
        for line in f:
            values = line.split()
            word = ''.join(values[:-300])
            coefs = np.array(values[-300:], dtype='float32')
            embedding_model[word] = coefs
        print("Done.", len(embedding_model), " words loaded!")
        f.close()
        return embedding_model


    mar_embedding = load_embedding('/content/drive/My Drive/DeepLearning/indicnlp.v1.mr.vec')

         Loading Glove Model
         Done. 533456  words loaded!


    eng_embedding= loadGloveModel("/content/drive/My Drive/DeepLearning/glove.6B.300d.txt")

         Loading Glove Model
         Done. 400000  words loaded!


    input_texts = []
    target_texts = []
    target_words = set()

    with open(DATA_PATH, 'r', encoding='utf-8') as f:
        lines = f.read().split("\n")
        for line in lines[:NUM_SAMPLES]:
            input_text, target_text = line.split('\t')
            # print(target_text)
            input_text = preprocess_sentence(input_text)
            target_text = preprocess_sentence(target_text)
            input_texts.append(input_text)
            target_texts.append(target_text)
            #target_words.update(list(target_words))
            target_words.update(target_text.split())

    target_words = sorted(list(target_words))
    #print(target_text)


    max_input_seqlen = max([len(txt.split()) for txt in input_texts])
    #print(max_input_seqlen)
```

```
max_target_seqlen = max([len(txt.split()) for txt in target_texts])
#print(max_target_seqlen)
target_vocab_size = len(target_words) + 1


# get decoder input data
decoder_data = []
for text in target_texts:
    tmp = []
    for word in text.split():
        embed = np.random.randn(EMBED_DIM)# output is an array
        if word in mar_embedding:
            embed = mar_embedding[word]
        tmp.append(embed)
    decoder_data.append(tmp)# list
#print(type(decoder_data))
decoder_data = pad_sequences(decoder_data, max_target_seqlen, padding="post")



# get decoder target data
targword2idx = dict([(word, i + 1) for i, word in enumerate(target_words)])
idx2targword = dict((i, word) for word, i in targword2idx.items())
decoder_target_data = []
for text in target_texts:
    tmp = []
    for idx, word in enumerate(text.split()):
        if idx > 0:
            tmp.append(targword2idx[word])
    decoder_target_data.append(tmp)
decoder_target_data = pad_sequences(
    decoder_target_data, max_target_seqlen, padding="post")
decoder_target_data = to_categorical(decoder_target_data)



# get encoder data
encoder_data = []
for text in input_texts:
    tmp = []
    for word in text.split():
        embed = np.random.randn(EMBED_DIM)
        if word in eng_embedding:
            embed = eng_embedding[word]
        tmp.append(embed)
    encoder_data.append(tmp)
encoder_data = pad_sequences(encoder_data, max_input_seqlen, padding="post")


encoder_data.shape
```

```
(10000, 7, 300)
```

```
decoder_data.shape
```

```
(10000, 12, 300)
```

```
decoder_target_data.shape
```

```
(10000, 12, 3475)
```

```
# construct model
encoder_inputs = Input(shape=(max_input_seqlen, EMBED_DIM))
encoder_lstm = LSTM(HIDDEN_DIM, return_state=True, name="encoder_lstm")
_, state_h, state_c = encoder_lstm(encoder_inputs)
encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(None,EMBED_DIM))
#decoder inputs = Input(shape=(max target seqlen, EMBED DIM))
```

```
decoder_lstm = LSTM(HIDDEN_DIM, return_sequences=True,
                    return_state=True, name="decoder_lstm")
decoder_outputs, _, _ = decoder_lstm(
    decoder_inputs, initial_state=encoder_states)
decoder_dense = Dense(
    target_vocab_size, activation="softmax", name="decoder_dense")
decoder_outputs = decoder_dense(decoder_outputs)


#define training model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.compile(optimizer=OPTIMIZER,
              loss='categorical_crossentropy', metrics=["acc"])
print(model.summary())
filename = 'seq2seq_keras.h5'
checkpoint = ModelCheckpoint(
    filename, verbose=1, save_best_only=True, mode='min')
# checkpoint = ModelCheckpoint(filename, verbose=1, mode='min')
model.fit([encoder_data, decoder_data], decoder_target_data,
          batch_size=BATCH_SIZE,
          epochs=EPOCHS,
          validation_split=0.2)
```

```
250/250 [==============================] - 21s 83ms/step - loss: 1.0151 - acc: 0.8293 - val_loss: 1.9046 - va
Epoch 22/50
250/250 [==============================] - 20s 82ms/step - loss: 1.0004 - acc: 0.8301 - val_loss: 1.9139 - va
Epoch 23/50
250/250 [==============================] - 21s 82ms/step - loss: 0.9880 - acc: 0.8330 - val_loss: 1.9519 - va
Epoch 24/50
250/250 [==============================] - 21s 83ms/step - loss: 0.9772 - acc: 0.8343 - val_loss: 1.8931 - va
Epoch 25/50
250/250 [==============================] - 21s 82ms/step - loss: 0.9658 - acc: 0.8368 - val_loss: 1.9069 - va
Epoch 26/50
250/250 [==============================] - 20s 82ms/step - loss: 0.9549 - acc: 0.8387 - val_loss: 1.9313 - va
Epoch 27/50
250/250 [==============================] - 21s 82ms/step - loss: 0.9451 - acc: 0.8403 - val_loss: 1.9169 - va
Epoch 28/50
250/250 [==============================] - 21s 82ms/step - loss: 0.9352 - acc: 0.8420 - val_loss: 1.9415 - va
Epoch 29/50
250/250 [==============================] - 21s 82ms/step - loss: 0.9268 - acc: 0.8438 - val_loss: 1.9661 - va
Epoch 30/50
250/250 [==============================] - 21s 83ms/step - loss: 0.9190 - acc: 0.8450 - val_loss: 1.9726 - va
Epoch 31/50
250/250 [==============================] - 21s 82ms/step - loss: 0.9100 - acc: 0.8465 - val_loss: 1.9672 - va
Epoch 32/50
250/250 [==============================] - 20s 82ms/step - loss: 0.9028 - acc: 0.8483 - val_loss: 1.9977 - va
Epoch 33/50
250/250 [==============================] - 21s 82ms/step - loss: 0.8960 - acc: 0.8496 - val_loss: 2.0014 - va
Epoch 34/50
250/250 [==============================] - 20s 82ms/step - loss: 0.8879 - acc: 0.8514 - val_loss: 2.0290 - va
Epoch 35/50
250/250 [==============================] - 21s 82ms/step - loss: 0.8819 - acc: 0.8528 - val_loss: 2.0272 - va
Epoch 36/50
250/250 [==============================] - 21s 82ms/step - loss: 0.8778 - acc: 0.8543 - val_loss: 2.0295 - va
Epoch 37/50
250/250 [==============================] - 21s 82ms/step - loss: 0.8741 - acc: 0.8552 - val_loss: 2.0388 - va
Epoch 38/50
250/250 [==============================] - 21s 83ms/step - loss: 0.8706 - acc: 0.8564 - val_loss: 2.0653 - va
Epoch 39/50
250/250 [==============================] - 21s 83ms/step - loss: 0.8684 - acc: 0.8575 - val_loss: 2.0391 - va
Epoch 40/50
250/250 [==============================] - 21s 82ms/step - loss: 0.8637 - acc: 0.8591 - val_loss: 2.1044 - va
Epoch 41/50
250/250 [==============================] - 21s 85ms/step - loss: 0.8607 - acc: 0.8602 - val_loss: 2.0504 - va
Epoch 42/50
250/250 [==============================] - 21s 83ms/step - loss: 0.8522 - acc: 0.8619 - val_loss: 2.0706 - va
Epoch 43/50
250/250 [==============================] - 21s 83ms/step - loss: 0.8459 - acc: 0.8625 - val_loss: 2.0587 - va
Epoch 44/50
250/250 [==============================] - 21s 83ms/step - loss: 0.8420 - acc: 0.8637 - val_loss: 2.0510 - va
Epoch 45/50
250/250 [==============================] - 21s 84ms/step - loss: 0.8370 - acc: 0.8653 - val_loss: 2.0799 - va
Epoch 46/50
250/250 [==============================] - 21s 82ms/step - loss: 0.8333 - acc: 0.8661 - val_loss: 2.0863 - va
```

```
Epoch 47/50
250/250 [==============================] - 21s 83ms/step - loss: 0.8295 - acc: 0.8671 - val_loss: 2.1336 - va
Epoch 48/50
250/250 [==============================] - 20s 82ms/step - loss: 0.8262 - acc: 0.8682 - val_loss: 2.1273 - va
Epoch 49/50
250/250 [==============================] - 20s 82ms/step - loss: 0.8237 - acc: 0.8688 - val_loss: 2.1303 - va
Epoch 50/50
250/250 [==============================] - 21s 82ms/step - loss: 0.8196 - acc: 0.8703 - val_loss: 2.1233 - va
<tensorflow.python.keras.callbacks.History at 0x7f9a1d72be10>
```

```
# create inference model
encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(HIDDEN_DIM,))
decoder_state_input_c = Input(shape=(HIDDEN_DIM,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
# decoder_outputs = (BATCH_SIZE, seqlen, HIDDEN_DIM)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
# decoder_outputs = (BATCH_SIZE, seqlen, target_token_size)
decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs, [decoder_outputs] + decoder_states)


def decode(input_seq):
    states = encoder_model.predict(input_seq)
    target_seq = np.random.randn(EMBED_DIM)
    target_seq = [[target_seq]]
    stop_condition = False
    prediction = ''

    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states)
        sampled_token_idx = np.argmax(output_tokens[0, -1, :])
        sampled_word = idx2targword[sampled_token_idx]
        prediction += sampled_word + " "

        if (sampled_word == '\n' or len(prediction) > max_target_seqlen):
            stop_condition = True

        if sampled_word in mar_embedding:
            target_seq = mar_embedding[sampled_word]
        else:
            target_seq = np.random.randn(EMBED_DIM)
        target_seq = [[target_seq]]
        states = [h, c]

    return prediction


actual, predicted = list(), list()

for index in [1900, 5534, 7467, 1258, 4500, 1345, 7863, 7688, 6782]: # considered random index
    input_seq = encoder_data[index]
    input_seq = np.expand_dims(input_seq, axis=0)
    actual.append(target_texts[index].split())
    prediction = decode(input_seq)
    predicted.append(prediction.split())
    print('-')
    print("Input sentence: ", input_texts[index])
    print("Translation: ", prediction)
```

```
-
Input sentence:          you will die .

Translation:     मरशील .
```

```
-
Input sentence:          give me my money .

Translation:     मला माझ पस द .

-
Input sentence:          i tried to forget .

Translation:     मी विसरायचा परयतन कला .

-
Input sentence:          what's that ?

Translation:     त काय आह ?

-
Input sentence:          i'm your friend .

Translation:     मी तझी मतरिण आह .

-
Input sentence:          come help me .

Translation:     य माझी मदत कर .

-
Input sentence:          she is her friend .

Translation:     ती तयाची मतरिण आह .

-
Input sentence:          it was quite cold .

Translation:     बरयापकी थड होता .

-
Input sentence:          why didn't i die ?

Translation:     मी का नाही मलो ?
```