```
import re
import random
data_path = "human_text.txt"
data_path2 = "robot_text.txt"
# Defining lines as a list of each line
with open(data_path, 'r', encoding='utf-8') as f:
  lines = f.read().split('\n')
with open(data_path2, 'r', encoding='utf-8') as f:
  lines2 = f.read().split('\n')
lines = [re.sub(r"\[\w+\]",'hi',line) for line in lines]
lines = [" ".join(re.findall(r"\w+",line)) for line in lines]
lines2 = [re.sub(r"\[\w+\]",'',line) for line in lines2]
lines2 = [" ".join(re.findall(r"\w+",line)) for line in lines2]
# Grouping lines by response pair
pairs = list(zip(lines,lines2))
#random.shuffle(pairs)


import numpy as np

input_docs = []
target_docs = []
input_tokens = set()
target_tokens = set()
for line in pairs[:400]:
  input_doc, target_doc = line[0], line[1]
  # Appending each input sentence to input_docs
  input_docs.append(input_doc)
  # Splitting words from punctuation
  target_doc = " ".join(re.findall(r"[\w']+|[^\s\w]", target_doc))
  # Redefine target_doc below and append it to target_docs
  target_doc = '<START> ' + target_doc + ' <END>'
  target_docs.append(target_doc)

  # Now we split up each sentence into words and add each unique word to our vocabulary set
  for token in re.findall(r"[\w']+|[^\s\w]", input_doc):
    if token not in input_tokens:
      input_tokens.add(token)
  for token in target_doc.split():
    if token not in target_tokens:
      target_tokens.add(token)
input_tokens = sorted(list(input_tokens))
target_tokens = sorted(list(target_tokens))
num_encoder_tokens = len(input_tokens)
num_decoder_tokens = len(target_tokens)

input_features_dict = dict(
    [(token, i) for i, token in enumerate(input_tokens)])
target_features_dict = dict(
    [(token, i) for i, token in enumerate(target_tokens)])

reverse_input_features_dict = dict(
    (i, token) for token, i in input_features_dict.items())
reverse_target_features_dict = dict(
    (i, token) for token, i in target_features_dict.items())
```

```
max_encoder_seq_length = max([len(re.findall(r"[\w']+|[^\s\w]", input_doc)) for input_doc i
max_decoder_seq_length = max([len(re.findall(r"[\w']+|[^\s\w]", target_doc)) for target_doc

encoder_input_data = np.zeros(
    (len(input_docs), max_encoder_seq_length, num_encoder_tokens),
    dtype='float32')
decoder_input_data = np.zeros(
    (len(input_docs), max_decoder_seq_length, num_decoder_tokens),
    dtype='float32')
decoder_target_data = np.zeros(
    (len(input_docs), max_decoder_seq_length, num_decoder_tokens),
    dtype='float32')

for line, (input_doc, target_doc) in enumerate(zip(input_docs, target_docs)):
    for timestep, token in enumerate(re.findall(r"[\w']+|[^\s\w]", input_doc)):
        #Assign 1. for the current line, timestep, & word in encoder_input_data
        encoder_input_data[line, timestep, input_features_dict[token]] = 1.

    for timestep, token in enumerate(target_doc.split()):
        decoder_input_data[line, timestep, target_features_dict[token]] = 1.
        if timestep > 0:
            decoder_target_data[line, timestep - 1, target_features_dict[token]] = 1.


print(pairs[:5])
print(input_docs[:5])
```

```
[('hi', 'hi there how are you'), ('oh thanks i m fine this is an evening in my timezo
['hi', 'oh thanks i m fine this is an evening in my timezone', 'how do you feel today
```

```
from tensorflow import keras
from keras.layers import Input, LSTM, Dense
from keras.models import Model
#Dimensionality
dimensionality = 256
#The batch size and number of epochs
batch_size = 10
epochs = 600
#Encoder
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder_lstm = LSTM(dimensionality, return_state=True)
encoder_outputs, state_hidden, state_cell = encoder_lstm(encoder_inputs)
encoder_states = [state_hidden, state_cell]
#Decoder
decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(dimensionality, return_sequences=True, return_state=True)
decoder_outputs, decoder_state_hidden, decoder_state_cell = decoder_lstm(decoder_inputs,
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

#Model
training_model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
#Compiling
training_model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['ac
#Training
training_model.fit([encoder_input_data, decoder_input_data], decoder_target_data, batch_s
training_model.save('training_model.h5')
```

```
32/32 [==============================] - 0s 14ms/step - loss: 1.1186 - accuracy: 0
Epoch 406/600
32/32 [==============================] - 0s 14ms/step - loss: 1.1189 - accuracy: 0
Epoch 407/600
32/32 [==============================] - 0s 14ms/step - loss: 1.1176 - accuracy: 0
Epoch 408/600
32/32 [==============================] - 0s 14ms/step - loss: 1.1206 - accuracy: 0
Epoch 409/600
32/32 [==============================] - 0s 13ms/step - loss: 1.1177 - accuracy: 0
Epoch 410/600
32/32 [==============================] - 0s 14ms/step - loss: 1.1175 - accuracy: 0
Epoch 411/600
32/32 [==============================] - 0s 13ms/step - loss: 1.1151 - accuracy: 0
Epoch 412/600
32/32 [==============================] - 0s 13ms/step - loss: 1.1158 - accuracy: 0
Epoch 413/600
32/32 [==============================] - 0s 14ms/step - loss: 1.1152 - accuracy: 0
Epoch 414/600
32/32 [==============================] - 0s 14ms/step - loss: 1.1152 - accuracy: 0
Epoch 415/600
32/32 [==============================] - 0s 13ms/step - loss: 1.1158 - accuracy: 0
Epoch 416/600
32/32 [==============================] - 0s 14ms/step - loss: 1.1168 - accuracy: 0
Epoch 417/600
32/32 [==============================] - 0s 13ms/step - loss: 1.1144 - accuracy: 0
Epoch 418/600
```

```
from keras.models import load_model
training_model = load_model('training_model.h5')
encoder_inputs = training_model.input[0]
encoder_outputs, state_h_enc, state_c_enc = training_model.layers[2].output
encoder_states = [state_h_enc, state_c_enc]
encoder_model = Model(encoder_inputs, encoder_states)

latent_dim = 256
decoder_state_input_hidden = Input(shape=(latent_dim,))
decoder_state_input_cell = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_hidden, decoder_state_input_cell]
decoder_outputs, state_hidden, state_cell = decoder_lstm(decoder_inputs, initial_state=de
decoder_states = [state_hidden, state_cell]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model([decoder_inputs] + decoder_states_inputs, [decoder_outputs] + decod

def decode_response(test_input):
    states_value = encoder_model.predict(test_input)
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    target_seq[0, 0, target_features_dict['<START>']] = 1.

    decoded_sentence = ''
    stop_condition = False  # Define stop_condition here

    while not stop_condition:
        output_tokens, hidden_state, cell_state = decoder_model.predict([target_seq] + st
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_token = reverse_target_features_dict[sampled_token_index]
        decoded_sentence += " " + sampled_token

        if (sampled_token == '<END>' or len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        states_value = [hidden_state, cell_state]

    return decoded_sentence
```

```
import numpy as np
import re

class ChatBot:
    negative_responses = ("no", "nope", "nah", "naw", "not a chance", "sorry")
    positive_responses = ("yes", "yeah", "yup", "sure", "ok", "okay", "yep", "alright")
    exit_commands = ("quit", "pause", "exit", "goodbye", "bye", "later", "stop")

    # Method to start the conversation
    def start_chat(self):
        user_response = input("Hi, I'm a chatbot trained on random dialogs. Would you lik

        if user_response.lower() in self.negative_responses:
            print("Ok, have a great day!")
            return
        elif user_response.lower() in self.positive_responses:
            self.chat(user_response)
        else:
            print("I didn't understand that. Please respond with 'yes' or 'no'.")
            self.start_chat()

    # Method to handle the conversation
    def chat(self, reply):
        while not self.make_exit(reply):
            reply = input(self.generate_response(reply) + "\n")

    # Method to convert user input into a matrix
    def string_to_matrix(self, user_input):
        tokens = re.findall(r"[\w']+|[^\s\w]", user_input)
        user_input_matrix = np.zeros(
            (1, max_encoder_seq_length, num_encoder_tokens),
            dtype='float32')
        for timestep, token in enumerate(tokens):
            if token in input_features_dict:
                user_input_matrix[0, timestep, input_features_dict[token]] = 1.
        return user_input_matrix

    # Method that will create a response using seq2seq model we built
    def generate_response(self, user_input):
        input_matrix = self.string_to_matrix(user_input)
        chatbot_response = decode_response(input_matrix)
        # Remove <START> and <END> tokens from chatbot_response
        chatbot_response = chatbot_response.replace("<START>",'')
        chatbot_response = chatbot_response.replace("<END>",'')
        return chatbot_response


chatbot = ChatBot()
chatbot.start_chat()
```

```
••• Hi, I'm a chatbot trained on random dialogs. Would you like to chat with me?
```