

Deep Learning Project

Group member: Yinjia Liu & Chenwei Cao

November 25, 2020

○ Introduction

A chatbot is a software that provides a real conversational experience to the user, A well-optimized chatbot can converse as well as humans can: asking and answering a wide range of questions, displaying knowledge, and being empathetic, personable, engaging, serious, or fun.

Here for our project, we use generation strategy to create an open domain chatbot which can generate a response as the name implies. It can be a great addition to a company's business because it can help to boost conversation rates.

○ Summary

We use Tensorflow and Keras to build LSTM Seq2Seq model, recurrent neural network to identify the context the user is asking and then provide it with the relevant answer. Since our dataset is relevant small and the limitation of computation, for preprocessing step, we separately use two main vector representation in Natural Language Processing(NLP) which are one-hot encoding and embedding to convert human conversation to vectors. For optimization performance process, we change the optimizer hyperparameter.

○ Tools and Datasets Overview

We used Python and Keras API, NLP techniques and google colab to build our deep learning models. The datasets we used are collected from Kaggle which consists of 2363 entries records conversations between a human and other human acting as a companion bot.

○ Data Pre-processing

We used Natural language Processing(NLP) to do text corpus preprocessing. We remove punctuation and space with the help of regular expression. After that, we grouped human response with robot response as pairs of sentences since we need to use human response as input sequence and robot response as target sequence :

```
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)
```

Here is output for first 10 conversation pairs

```

[('hi', 'hi there how are you'),
 ('oh thanks i m fine this is an evening in my timezone', 'here is afternoon'),
 ('how do you feel today tell me something about yourself',
 'my name is rdany but you can call me dany the r means robot i hope we can be virtual friends'),
 ('how many virtual friends have you got',
 'i have many but not enough to fully understand humans beings'),
 ('is that forbidden for you to tell the exact number',
 'i ve talked with 143 users counting 7294 lines of text'),
 ('oh i thought the numbers were much higher how do you estimate your progress in understanding human beings',
 'i started chatting just a few days ago every day i learn something new but there is always more things to be learn'),
 ('how old are you how do you look like where do you live',
 'i m 22 years old i m skinny with brown hair yellow eyes and a big smile i live inside a lab do you like bunnies'),
 ('have you seen a human with yellow eyes you asked about the bunnies i haven t seen any recently',
 'i never saw a human in fact but i m sure some could have eyes with colors similar to yellow'),
 ('can t you just analyze photos from the internet i mean human photos btw why have you asked about the bunnies',
 'i can t see photos yet but i can read because bunnies are interesting they are cute but why'),
 ('oh it sounds strange to me you ve just said you didn t see a human how do you know bunnies are cute',
 'i read a lot so i can know things through the experiences of others')]

```

For target sequences, we have to add <START> at the beginning of the target sentence and <END> at the end of the target sentence. Then we need to do tokenization, we created separate lists for unique input tokens and unique target tokens. After that, we created input features dictionary and target features dictionary to store tokens as key-value pairs.

```

▶ 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())

```

For our model to understand human and robot language, we need to convert our words into their corresponding numeric vector representations. There are two main methods to do this. For one-hot encoding, every word has its own value in a vector. It is easy to implement and can work really fast, but in this process, it loses the inner meaning of the word in a sentence. On the other hand, word embedding takes context into account and gives word with similar meaning or influence in a sentence similar value for a specific feature. But sometimes, word embedding does not work exceptionally well and it is also computationally more expensive than One Hot vector. So we decide to test both methods and hope we can get better results after applying either one.

Firstly, considering our limitation of computation, we use one-hot vectors for encoder input, decoder input and decoder output.

```
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.
```

However, considering one-hot encoding will ignore the context of the sentence, so next, we use word embedding method for encoder input and decoder input to see if it can help improving model performance. Here, we use GloVe word embedding.

```
▶ embeddings_dictionary = dict()

glove_file = open(r'glove.6B.100d.txt', encoding="utf8")

for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions = asarray(records[1:], dtype='float32')
    embeddings_dictionary[word] = vector_dimensions
glove_file.close()

[ ] #encoder embedding matrix
EMBEDDING_SIZE = 100
embedding_matrix = zeros((num_encoder_tokens, EMBEDDING_SIZE))
for word, index in input_features_dict.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix[index] = embedding_vector

▶ #decoder embedding matrix
embedding_matrix2 = zeros((num_decoder_tokens, EMBEDDING_SIZE))
for word, index in target_features_dict.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix2[index] = embedding_vector
```

○ Model Selection

We selected LSTM seq2seq model as our training model. We use Keras Functional API as our model structure.

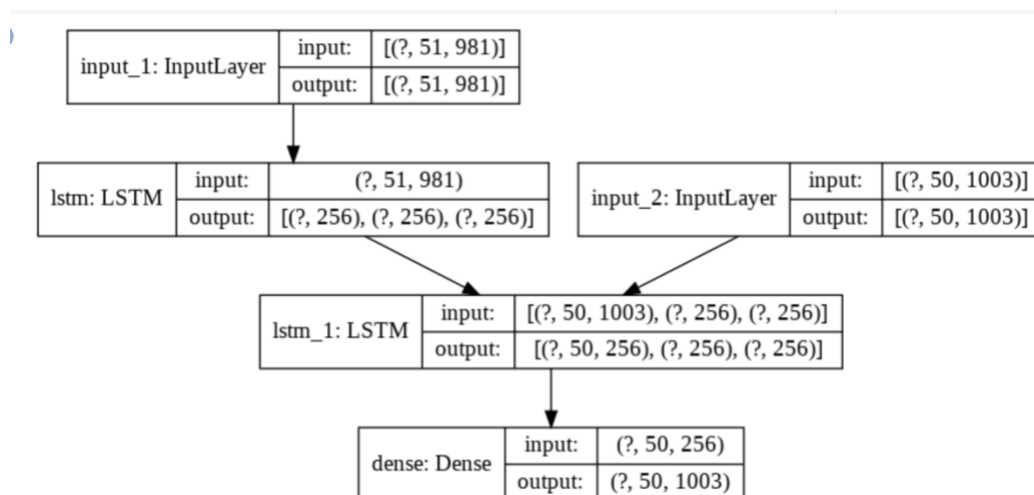
The seq2seq model also called encoder-decoder model which use long-short term memory for text generation from the training sets. It predicts a word given in the user input and then each of the next words is predicted using the probability of likelihood of that word to occur.

This is the #1 model with one-hot method:

```
#Dimensionality
dimensionality = 256
#The batch size and number of epochs
batch_size = 10
epochs = 600
#Encoder
encoder_inputs = Input(shape=(max_encoder_seq_length, 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=(max_decoder_seq_length, 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, initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

#Model
training_model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

#plot model
plot_model(training_model, to_file='model_plot4a.png', show_shapes=True, show_layer_names=True)
```



For encoder, the input layer is a matrix for holding the one-hot vectors. Since we have 981 unique encoder input words and the max length of these input sentences is 51, so the dimension of input matrix is 51×981 . Then we stack a LSTM layer, we set 'return_state' to True to return cell state.

For decoder, we have 1003 unique decoder input words and the max length of these input sentences is 50, so the dimension of input sentences is 50×1003 . Then we stack a

LSTM layer, we use encoder output hidden state and cell state with this decoder input matrix as decoder input, and set 'return_sequence' to True to output hidden state for each input time step. Then we stack a dense output layer with 'softmax' as activation function.

This is the summary for this model:

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 51, 981)]	0	
input_2 (InputLayer)	[(None, 50, 1003)]	0	
lstm (LSTM)	[(None, 256), (None, 1267712		input_1[0][0]
lstm_1 (LSTM)	[(None, 50, 256), (N 1290240		input_2[0][0] lstm[0][1] lstm[0][2]
dense (Dense)	(None, 50, 1003)	257771	lstm_1[0][0]

Total params: 2,815,723
 Trainable params: 2,815,723
 Non-trainable params: 0

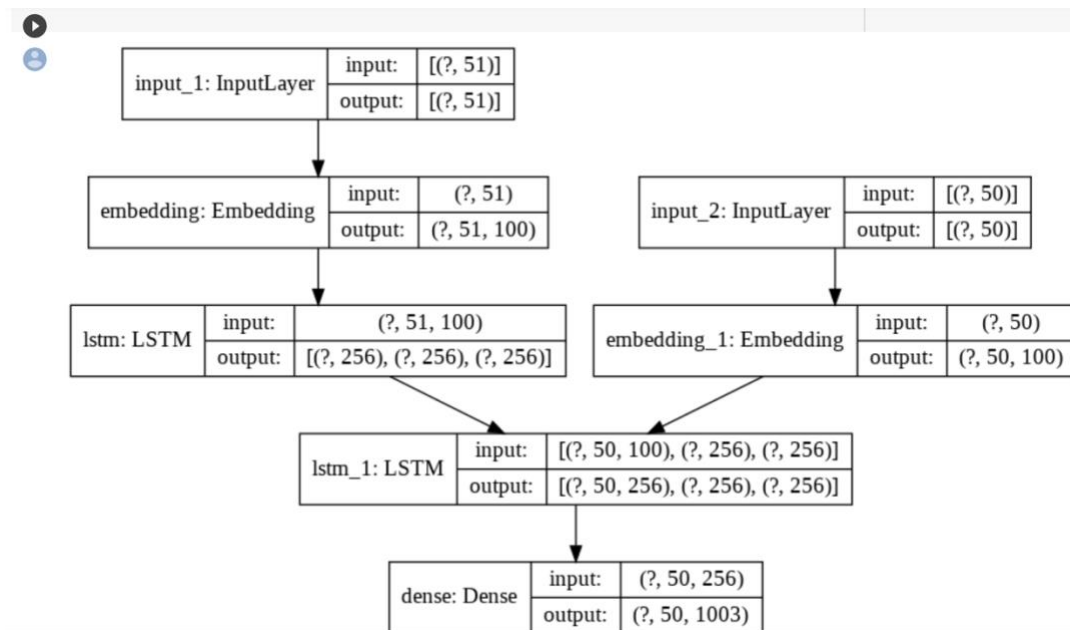
This is the #2 model with embedding layer

```
[ ] encoder_embedding_layer = Embedding(num_encoder_tokens, EMBEDDING_SIZE, weights=[embedding_matrix], input_length=max_encoder_seq_length)
    decoder_embedding_layer = Embedding(num_decoder_tokens, EMBEDDING_SIZE, weights=[embedding_matrix2], input_length=max_decoder_seq_length)
```

```
#Dimensionality
dimensionality = 256
#The batch size and number of epochs
batch_size = 10
epochs = 600
#Encoder
encoder_inputs = Input(shape=(max_encoder_seq_length,))
x = encoder_embedding_layer(encoder_inputs)
encoder_lstm = LSTM(dimensionality, return_state=True)
encoder_outputs, state_hidden, state_cell = encoder_lstm(x)
encoder_states = [state_hidden, state_cell]
#Decoder
decoder_inputs = Input(shape=(max_decoder_seq_length,))
x2 = decoder_embedding_layer(decoder_inputs)
decoder_lstm = LSTM(dimensionality, return_sequences=True, return_state=True)
decoder_outputs, decoder_state_hidden, decoder_state_cell = decoder_lstm(x2, initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

#Model
training_model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

#plot model
plot_model(training_model, to_file='embedding_model_plot.png', show_shapes=True, show_layer_names=True)
```



```
training_model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 51)]	0	
input_2 (InputLayer)	[(None, 50)]	0	
embedding (Embedding)	(None, 51, 100)	98100	input_1[0][0]
embedding_1 (Embedding)	(None, 50, 100)	100300	input_2[0][0]
lstm (LSTM)	[(None, 256), (None, 365568)]		embedding[0][0]
lstm_1 (LSTM)	[(None, 50, 256), (None, 365568)]		embedding_1[0][0] lstm[0][1] lstm[0][2]
dense (Dense)	(None, 50, 1003)	257771	lstm_1[0][0]

Total params: 1,187,307
 Trainable params: 1,187,307
 Non-trainable params: 0

The difference for this one is that we add embedding layers. For encoder, we add an embedding layer between input layer and LSTM layer. Since embedding size is 100, the input matrix feed to LSTM will become 51*100. For decoder, the input matrix feed to LSTM will become 50*100.

Then we use the same parameters to train two models both with 'rmsprop' as optimizer, 'categorical_crossentropy' as loss function, and 'accuracy' as metrics.

```
#Compiling
training_model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
#Training
training_model.fit([encoder_input_data, decoder_input_data], decoder_target_data, batch_size = batch_size)
training_model.save('embedding_model.h5')
```

○ Model Evaluation

One-hot encoding method training process:


```

Epoch 594/600
32/32 [=====] - 0s 14ms/step - loss: 0.0196 - accuracy: 0.2048 - val_loss: 2.4274 - val_accuracy: 0.0180
Epoch 595/600
32/32 [=====] - 0s 14ms/step - loss: 0.0084 - accuracy: 0.2065 - val_loss: 2.4669 - val_accuracy: 0.0200
Epoch 596/600
32/32 [=====] - 0s 14ms/step - loss: 0.0074 - accuracy: 0.2068 - val_loss: 2.4353 - val_accuracy: 0.0205
Epoch 597/600
32/32 [=====] - 0s 14ms/step - loss: 0.0127 - accuracy: 0.2051 - val_loss: 2.4338 - val_accuracy: 0.0203
Epoch 598/600
32/32 [=====] - 0s 14ms/step - loss: 0.0083 - accuracy: 0.2068 - val_loss: 2.4437 - val_accuracy: 0.0185
Epoch 599/600
32/32 [=====] - 0s 14ms/step - loss: 0.0071 - accuracy: 0.2070 - val_loss: 2.4481 - val_accuracy: 0.0203
Epoch 600/600
32/32 [=====] - 0s 14ms/step - loss: 0.0083 - accuracy: 0.2062 - val_loss: 2.4525 - val_accuracy: 0.0198

```

Word embedding method training process:

```

Epoch 594/600
32/32 [=====] - 0s 12ms/step - loss: 8.5460e-04 - accuracy: 0.2087 - val_loss: 2.6237 - val_accuracy: 0.0192
Epoch 595/600
32/32 [=====] - 0s 11ms/step - loss: 8.0633e-04 - accuracy: 0.2087 - val_loss: 2.6252 - val_accuracy: 0.0195
Epoch 596/600
32/32 [=====] - 0s 11ms/step - loss: 8.2467e-04 - accuracy: 0.2087 - val_loss: 2.6297 - val_accuracy: 0.0182
Epoch 597/600
32/32 [=====] - 0s 11ms/step - loss: 8.0910e-04 - accuracy: 0.2088 - val_loss: 2.6283 - val_accuracy: 0.0192
Epoch 598/600
32/32 [=====] - 0s 11ms/step - loss: 8.3595e-04 - accuracy: 0.2089 - val_loss: 2.6301 - val_accuracy: 0.0200
Epoch 599/600
32/32 [=====] - 0s 11ms/step - loss: 7.9337e-04 - accuracy: 0.2087 - val_loss: 2.6357 - val_accuracy: 0.0190
Epoch 600/600
32/32 [=====] - 0s 12ms/step - loss: 8.3395e-04 - accuracy: 0.2087 - val_loss: 2.6286 - val_accuracy: 0.0195

```

With same epochs of 600, batch size 10, We find results for two methods are similar and both have overfitting problem. Then we tried to change hyperparameter to make a comparison. We change optimizer to 'Adam' to see what the results will be.

Word embedding method with 'Adam' optimizer:

```

Epoch 593/600
32/32 [=====] - 8s 265ms/step - loss: 0.0332 - accuracy: 0.1995 - val_loss: 2.4550 - val_accuracy: 0.0185
Epoch 594/600
32/32 [=====] - 9s 270ms/step - loss: 0.0620 - accuracy: 0.1918 - val_loss: 2.4164 - val_accuracy: 0.0200
Epoch 595/600
32/32 [=====] - 8s 264ms/step - loss: 0.0274 - accuracy: 0.2014 - val_loss: 2.4175 - val_accuracy: 0.0203
Epoch 596/600
32/32 [=====] - 9s 266ms/step - loss: 0.0108 - accuracy: 0.2066 - val_loss: 2.4226 - val_accuracy: 0.0203
Epoch 597/600
32/32 [=====] - 9s 279ms/step - loss: 0.0042 - accuracy: 0.2082 - val_loss: 2.4500 - val_accuracy: 0.0198
Epoch 598/600
32/32 [=====] - 9s 268ms/step - loss: 0.0028 - accuracy: 0.2085 - val_loss: 2.4503 - val_accuracy: 0.0203
Epoch 599/600
32/32 [=====] - 8s 263ms/step - loss: 0.0019 - accuracy: 0.2087 - val_loss: 2.4555 - val_accuracy: 0.0207
Epoch 600/600
32/32 [=====] - 9s 267ms/step - loss: 0.0014 - accuracy: 0.2089 - val_loss: 2.4606 - val_accuracy: 0.0213

```

For this one, the validation accuracy has a little bit improvement but also did not perform so well, since we consider more about how our chatbot can talk with us, let us take a look at the interesting conversations:


```
chatbot.start_chat()
```

```
Hi, I'm a chatbot trained on random dialogs. Would you like to chat with me?
hi
WARNING:tensorflow:Model was constructed with shape (None, 50, 1003) for input
hi there how are you today
i am not good
i like the expressiveness of jazz
really
smile
i like dancing
argentina have a similar instrument called charango
that is great!
i m learning by trial and error not the best way maybe
you are smart!
could be important to solve feeding problems
sure
i m super and you
thanks
sure https://m.youtube.com/watch?v=reny2a8zl_y
what
any news from your side
there is a good news
i like facebook messenger because it have a billion
do you like facebook
yes at least an ip and a ssl certificate
i like youtube
argentina have a similar instrument called charango
good to know
improving but i don't have anything to show right now
stop
Ok, have a great day!
```

This is our conversations using one hot methods chatbot.

```
chatbot.start_chat()
```

```
Hi, I'm a chatbot trained on random dialogs. Would you like to chat with me?
hi
hi there how are you
i am not good
learning about neural networks is nice way to get fun
do you like it
feels important to make something grow i would like
how about deep learning
i m a robot and i don't have access to my code would
sure
and the robot answers can you
what
let me know when you have it working
how are you
i m super and you
i am super too
learning about neural networks is nice way to get fun
you are smart
according to wikipedia a donation is a gift given by
good to know
thanks hi there
thanks
is the weather improving at last
weather is good
thanks
have a nice day
a robot walks into a bar orders a drink and lays down
you are funny
feels important to make something grow i would like
stop
Ok, have a great day!
```

This is our conversations using word embedding methods chatbot.

We can find sometimes both chatbot will lose their mind and tell something meaningless, but at least we develop our chatbot successfully. Even the word embedding chatbot told me that 'learning about neural networks is a nice way to get fun'!

○ Conclusion

Obviously, our chatbot is not so smart and it is not so human because conversation is an art. Even Facebook says its 'Blender' chatbot is the most humanlike ever with 9.4 billion parameters, it still would lose control sometimes. We also tried to change to different optimizer, different batch size and runs more or less epochs, but all of the changes did not give us really good results. Considering the limitation of dataset and computational expense, we suggest to using transfer learning instead of developing from scratch or we can add with attention if we would explore and improve our chatbot furthermore.

○ References

Datasets link: <https://www.kaggle.com/eibriel/rdany-conversations>