Chapter 16
Workshop#7 Building Chatbot with TensorFlow and Transformer Technology (Hour 13–14)

16.1 Introduction

In previous 6 NLP workshops, we studied NLP implementation tools and techniques ranging from tokenization, N-gram generation to semantic and sentiment analysis with various key NLP Python enabling technologies: NLTK, spaCy, TensorFlow, and contemporary Transformer Technology. This final workshop will explore how to integrate them for the design and implementation of a live domain-based chatbot system on a movie domain.

This workshop will explore:

1. Technical requirements for chatbot system.
2. Knowledge domain—the Cornell Large Movie Conversation Dataset is a well-known conversation dataset with over 200,000 movie dialogs of 10,000+ movie characters (Cornell 2022; Cornell_Movie_Corpus 2022)
3. A step-by-step Movie Chatbot system implementation which involves movie dialog preprocessing, model construction, attention learning, system integration with spaCy, TensorFlow, Keras and Transformer Technology, an important tool in NLP system implementation (Bansal 2021; Devlin et al. 2019; Géron 2019; Rothman 2022; Tunstall et al. 2022; Yıldırım and Asgari-Chenaghlu 2021).
4. Evaluation metrics with live chat examples.

16.2 Technical Requirements

Transformers, Tensorflow, and spaCy and Python modules include numpy and scikit-learn that are to be installed in machine.

R. S. T. Lee, *Natural Language Processing*

Use pip install commands to:

- pip install spacy
- pip install tensorflow (note: version 2.2 or above)
- pip install transformers
- pip install scikit-learn
- pip install numpy

The data files used in this workshop can be found in DATA sub-directory of NLPWorkshop7 directory of JupyterHub Server (NLPWorkshop7 2022).

16.3 AI Chatbot in a Nutshell

16.3.1 What Is a Chatbot?

Conversational artificial intelligence (conversational AI) is a field of machine learning that aims to create technology and enables users to have text or speech-based interactions with machines. Chatbots, virtual assistants, and voice assistants are typical conversational AI products (Batish 2018; Freed 2021; Janarthanam 2017; Raj 2018).

A chatbot is a software application designed to make conversations with humans.

Chatbots are widely used in human resources, marketing and sales, banking, healthcare, and non-commercial areas such as personal conversations. They include:

– Amazon Alexa is a voice-based virtual assistant to perform tasks per user requests or inquiries, i.e. play music, podcasts, set alarms, read audiobooks, provide real-time weather, traffic, and other information. Alexa Home can connect smart home devices to oversee premises and electrical appliances.
– Facebook Messenger and Telegram instant messaging services provide interfaces and API documentations (Facebook 2022; Telegram 2022) for developers to connect bots.
– Google Assistant provides real-time weather, flight, traffic information, send and receive text messages, email services, device information, set alarms and integrate with smart home devices, etc. available on Google Maps, Google Search, and standalone Android and iOS applications.
– IKEA provides customer service chatbot called Anna, AccuWeather, and FAQ chatbots.
– Sephora has virtual make-up artist and customer service chatbots at Facebook messenger.
– Siri integrates with iPhone, iPad, iPod, and macOS to initiate, answer calls, send, receive text messages and WhatsApp messages at iPhone.

Other virtual assistants include AllGenie, Bixby, Celia, Cortana, Duer, and Xiaowei.

16.3.2 What Is a Wake Word in Chatbot?

A wake word is the gateway between user and user's digital assistant/Chatbot. Voice assistants such as Alexa and Siri are powered by AI with word detection abilities to queries response and commands.

Common wake words include Hey, Google, Alexa, and Hey Siri.

Today's wake word performance and speech recognition are operated by machine learning or AI with cloud processing.

Sensory's wake word and phrase recognition engines use deep neural networks to provide an embedded or on-device wake word and phrase recognition engine (Fig. 16.1).

16.3.2.1 Tailor-Made Wake Word

Wake words like Alexa, Siri, and Google are associated with highly valued and technical products experiences, other companies had created tailor-made wake word and uniqueness to their products, i.e. Hi Toyota had opened a doorway to voice user interface to strengthen the relationship between customers and the brand.

16.3.2.2 Why Embedded Word Detection?

Wake word technology has been used in cases beyond mobile applications. Some battery powered devices like Bluetooth headphones, smart watches, cameras, and emergency alert devices.

Fig. 16.1 Wake word to invoke Chatbot (Tuchong 2022)

Chatbot allow users to utter commands naturally. Queries like *what time is it?* or *how many steps have I taken?* are phrases examples that a chatbot can process zero latency with high accuracy.

Wake word technology can integrate with voice recognition applications like touch screen food ordering, voice-control microwaves, or user identification settings at televisions or vehicles.

16.3.3 NLP Components in a Chatbot

A typical chatbot consists of major components:

1. Speech-to-text converts user speech into text. The input is a wav/mp3 file and the output is a text file containing user's utterance.
2. Conversational NLU performs intent recognition and entity extraction on user's utterance text. The output is the user's intent with a list of entities. Resolving references in the current to previous utterances is processed by this component.
3. Dialog manager retains conversation memory to generate a meaningful and coherent chat. This component is regarded as dialog memory in conversational state hitherto entities and intents appeared. Hence, the input is the previous dialog state for current user to parse intent and entities to a new dialog state output.
4. Answer generator gives all inputs from previous stages to generate answers to user's utterance.
5. Text-to-speech generates a speech file (WAV or mp3) from system's answers

Each of these components is trained and evaluated separately, e.g. speech-to-text training is performed by speech files and corresponding transcriptions on an annotated speech corpus.

16.4 Building Movie Chatbot by Using TensorFlow and Transformer Technology

This workshop will integrate the learnt technologies including: TensorFlow (Bansal 2021; Ekman 2021; TensorFlow 2022), Keras (Géron 2019; Keras 2022a), Transformer technology with Attention Learning Scheme (Ekman 2021; Kedia and Rasu 2020; Rothman 2022; Tunstall et al. 2022; Vaswani et al. 2017; Yıldırım and Asgari-Chenaghlu 2021) to build a live domain-based chatbot system. The Cornell Large Movie Dialog Corpus (Cornell 2022) will be used as conversation dataset for system training. The movie dataset can be downloaded either from Cornell databank (2022) or Kaggle's Cornell Movie Corpus archive (2022).

Use pip install command to invoke TensorFlow package and install its dataset:

In[1]

```
import tensorflow as tflow
tflow.random.set_seed(1234)

# !pip install tensorflow-datasets==1.2.0
import tensorflow_datasets as tflowDS

import re
import matplotlib.pyplot as pyplt
```

1. Install and import TensorFlow-datasets in addition to TensorFlow package. Please use pip install command as script if not installed already
2. Use *random.set_seed()* method to set all random seeds required to replicate TensorFlow codes

16.4.1 The Chatbot Dataset

The Cornell Movie Dialogs corpus is used in this project. This dataset, movie_conversations.txt contains lists of conversation IDs and movie_lines.txt associative conversation ID. It has generated 220,579 conversations and 10,292 movie characters amongst movies.

16.4.2 Movie Dialog Preprocessing

The maximum numbers of conversations (MAX_CONV) and the maximum length of utterance (MLEN) are set for 50,000 and 40 for system training, respectively.

Preprocessing data procedure (PP) involves the following steps:

1. Obtain 50,000 movie dialog pairs from dataset.
2. PP each utterance by special and control characters removal.
3. Construct tokenizer.
4. Tokenize each utterance.
5. Cap the max utterance length to MLEN.
6. Filter and pad utterances.

In[2]

```
# Set the maximum number of training conversation
MAX_CONV = 50000

# Preprocess all utterances
def pp_utterance(utterance):
    utterance = utterance.lower().strip()
    # Add a space to the following special characters
    utterance = re.sub(r"([?.!,])", r" \1 ", utterance)
    # Delete extrac spaces
    utterance = re.sub(r'[" "]+', " ", utterance)
    # Other than below characters, the other character replace by spaces
    utterance = re.sub(r"[^a-zA-Z?.,!]+", " ", utterance)
    utterance = utterance.strip()
    return utterance

def get_dialogs():
    # Create the dialog object (dlogs)
    id2dlogs = {}
    # Open the movie_lines text file
    with open('data/movie_lines.txt', encoding = 'utf-8', errors = 'ignore') as
f_dlogs:
        dlogs = f_dlogs.readlines()
    for dlog in dlogs:
        sections = dlog.replace('\n', '').split(' +++$+++ ')
        id2dlogs[sections[0]] = sections[4]

    query, ans = [], []
    with open('data/movie_conversations.txt',
                 encoding = 'utf-8', errors = 'ignore') as f_conv:
        convs = f_conv.readlines()
    for conv in convs:
        sections = conv.replace('\n', '').split(' +++$+++ ')
        # Create movie conservation object m_conv as a list
        m_conv = [conv[1:-1] for conv in sections[3][1:-1].split(', ')]
        for i in range(len(m_conv) - 1):
            query.append(pp_utterance(id2dlogs[m_conv[i]]))
            ans.append(pp_utterance(id2dlogs[m_conv[i + 1]]))
            if len(query) >= MAX_CONV:
                return query, ans
    return query, ans

queries, responses = get_dialogs()
```

Select query 13 and verify response:

In[3]
```
print('Query 13: {}'.format(queries[13]))
print('Response 13: {}'.format(responses[13]))
```
Out[3]
```
Query 13: that s because it s such a nice one .
Response 13: forget french .
```

Select query 100 and verify response:

In[4]
```
print('Query 100: {}'.format(queries[100]))
print('Response 100: {}'.format(responses[100]))
```
Out[4]
```
Query 100: you set me up .
Response 100: i just wanted
```

Verify queries (responses) size to see whether it situates within MAX_CONV:

In[5]
```
len(queries)
```
Out[5]
```
50000
```

In[6]
```
Len(responses)
```
Out[6]
```
50000
```

1. After max 50,000 movie conversations had obtained to perform basic preprocessing, it is sufficient for model training
2. Perform tokenization procedure to add START and END tokens using commands below

16.4.3 Tokenization of Movie Conversation

In[7]
```
# Define the Movie Token object
m_token =
tflowDS.deprecated.text.SubwordTextEncoder.build_from_corpus
(queries + responses, target_vocab_size = 2**13)

# Define the Start and End tokens
START_TOKEN, END_TOKEN =
[m_token.vocab_size], [m_token.vocab_size + 1]

# Define the size of Vocab (SVCAB)
SVCAB = m_token.vocab_size + 2
```

Verify movie token lists for conv 13 and 100:

In[8]
```
print('The movie token of conv 13: {}'.format(m_token.encode
(queries[13])))
```
Out[8]
```
The movie token of conv 13: [15, 8, 151, 12, 8, 354, 10, 347, 188, 1]
```

In[9]
```
print('The movie token of conv 100: {}'.format(m_token.encode
(queries[100])))
```
Out[9]
```
The movie token of conv 100: [5, 539, 36, 119, 1]
```

16.4.4 Filtering and Padding Process

Cap utterance max length (MLEN) to 40, perform filtering and padding:

In[10]
```
# Set the maximum length of each utterance MLEN to 40
MLEN = 40

# Performs the filtering and padding of each utterance
def filter_pad (qq, aa):
    m_token_qq, m_token_aa = [], []

    for (utterance1, utterance2) in zip(qq, aa):
        utterance1 = START_TOKEN + m_token.encode(utterance1) +
END_TOKEN
        utterance2 = START_TOKEN + m_token.encode(utterance2) +
END_TOKEN
        if len(utterance1) <= MLEN and len(utterance2) <= MLEN:
            m_token_qq.append(utterance1)
            m_token_aa.append(utterance2)

    # pad tokenized sentences
    m_token_qq = tflow.keras.preprocessing.sequence.pad_sequences
(m_token_qq, maxlen=MLEN, padding = 'post')

    m_token_aa = tflow.keras.preprocessing.sequence.pad_sequences
(m_token_aa, maxlen=MLEN, padding = 'post')

    return m_token_qq, m_token_aa

queries, responses = filter_pad (queries, responses)
```

Review the size of movie vocab (SVCAB) and total number of conversation (conv):

In[11]
```
print('Size of vocab: {}'.format(SVCAB))
print('Total number of conv: {}'.format(len(queries)))
```

Out[11]
```
Size of vocab: 8333
Total number of conv: 44095
```

1. Note that the total number of conversations after filtering and padding process is 44,095 which is less than previous max conv size 50,000 as some conversations are filtered out
2. SVCAB size is around 8000 which makes sense as the total numbers of conversation are around 44,000 lines, the number of vocabulary used is between 5000 and 10,000

16.4.5 Creation of TensorFlow Movie Dataset Object (mDS)

TensorFlow dataset object is created by using Dataset.from_tensor_slices() method of TensorFlow Data class as below:

In[12]
```
tflow.data.Dataset.from_tensor_slices?
```

In[13]
```
# Define the Batch and Buffer size
sBatch = 64
sBuffer = 20000

# Create mDS object from TensorFlow class
mDS = tflow.data.Dataset.from_tensor_slices(({'inNodes':queries,
'decNodes':responses[:, :-1]},{'outNodes':responses[:, 1:]}))

mDS = mDS.cache()
mDS = mDS.shuffle(sBuffer)
mDS = mDS.batch(sBatch)
mDS = mDS.prefetch(tflow.data.experimental.AUTOTUNE)
```

1. Create a TensorFlow Dataset object first to define Batch and Buffer size
2. Define three layers of Transformer Model: a. Input node layer (inNodes) – Queries b. Decoder input node layer (decNodes) – Responses c. Output node layer (outNodes) - Responses
3. Define prefetch scheme—AUTOTUNE in our project

16.4.6 Calculate Attention Learning Weights

The main concept of Transformer Technology is Attention Learning technique, which aimed at network capability to focus *attention* to various parts of training sequence during recurrent network learning. AI chatbot corresponds to *self-attention* learning on movie dialogs, in which the network has attention ability to different positions of dialog token sequences to compute utterances representation. A system architecture of Attention Learning model with Transformer Technology is shown in Fig. 16.2. Implement Attention Equation to calculate the attention weight is given by

$$\text{Attention}(Q,K,V) = soft\max_k\left(\frac{QK^T}{\sqrt{d_k}}\right) \tag{16.1}$$

Attention Equation is a typical scaled-dot-product attention function in transformer object Query (Q), K (Key), and V (Value) Value and Python implementation is given below:

In[14]

```
# Calculate the Attention Weight, Query (q), Key(k), Value(v), Mask(m)
def calc_attention(q, k, v, m):
    qk = tflow.matmul(q, k, transpose_b = True)
    dep = tflow.cast(tflow.shape(k)[-1], tflow.float32)
    mlogs = qk / tflow.math.sqrt(dep)

    # Use the masking for padding
    if m is not None:
        mlogs += (m * -1e9)

    # Apply softmax on the final axis of the utterance sequence
    att_wts = tflow.nn.softmax(mlogs, axis = -1)

    # Apply matmul() operation
    out_wts = tflow.matmul(att_wts, v)

    return out_wts
```

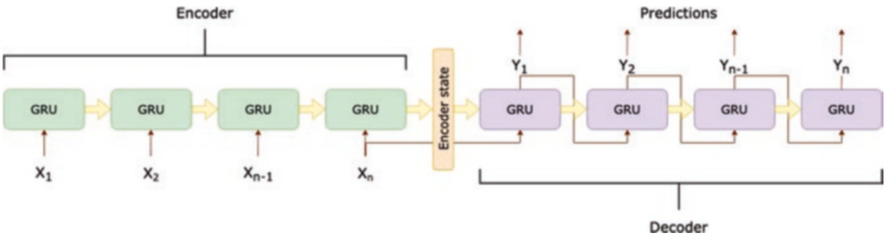


Fig. 16.2 Attention Learning with Transformer Technology

16.4.7 Multi-Head-Attention (MHAttention)

Multi-Head-Attention (MHAttention) consists of the following steps:

1. Construct linear layers.
2. Perform head-splitting.
3. Calculate attention weights.
4. Combine heads.
5. Condense layers

MHAttention is implemented as follows:

In[15]

```
class MHAttention(tflow.keras.layers.Layer):

  def __init__(self, dm, nhd, name="MHAttention"):
    super(MHAttention, self).__init__(name=name)
    self.nhd = nhd
    self.dm  = dm

    assert dm % self.nhd == 0

    self.dep = dm // self.nhd

    self.qdes = tflow.keras.layers.Dense(units=dm)
    self.kdes = tflow.keras.layers.Dense(units=dm)
    self.vdes = tflow.keras.layers.Dense(units=dm)

    self.des = tflow.keras.layers.Dense(units=dm)

  def sheads(self, inNodes, bsize):
    inNodes = tflow.reshape(
        inNodes, shape=(bsize, -1, self.nhd, self.dep))
    return tflow.transpose(inNodes, perm=[0, 2, 1, 3])

  def call(self, inNodes):
    q, k, v, m = inNodes['q'], inNodes['k'], inNodes['v'], inNodes['m']
    bsize = tflow.shape(q)[0]

    # 1. Construct Linear-layers
    q = self.qdes(q)
    k = self.kdes(k)
    v = self.vdes(v)
```

```
    # 2. Perform Head-splitting
    q = self.sheads(q, bsize)
    k = self.sheads(k, bsize)
    v = self.sheads(v, bsize)

    # 3. Calculate Attention Weights
    sattention = calc_attention(q, k, v, m)

    sattention = tflow.transpose(sattention, perm=[0, 2, 1, 3])

    # 4. Head Combining
    cattention = tflow.reshape(sattention,
                    (bsize, -1, self.dm))

    # 5. Layer Condensation
    outNodes = self.des(cattention)

    return outNodes
```

16.4.8 System Implementation

16.4.8.1 Step 1. Implement Masking

Implement (1) Padding Mask and (2) Look_ahead Mask to mask token sequences.

In[16]

```
# Generate Padding Mask (gen_pmask)
def gen_pmask(p):

    pmask = tflow.cast(tflow.math.equal(p, 0), tflow.float32)

    return pmask[:, tflow.newaxis, tflow.newaxis, :]
```

In[17]

```
# Generate Look_Ahead Mask (gen_lamask)
def gen_lamask(x):
    slen = tflow.shape(x)[1]
    lamask  = 1- tflow.linalg.band_part(tflow.ones((slen, slen)), -1, 0)
    pmask   = gen_pmask(x)

    return tflow.maximum(lamask, pmask)
```

Review *lamask* with a sample matrix:

In[18]
```
print(gen_lamask(tflow.constant([[1, 2, 0, 4, 5]])))
```

Out[18]
```
tf.Tensor(
[[[[0. 1. 1. 1. 1.]
   [0. 0. 1. 1. 1.]
   [0. 0. 1. 1. 1.]
   [0. 0. 1. 0. 1.]
   [0. 0. 1. 0. 0.]]]], shape=(1, 1, 5, 5), dtype=float32)
```

16.4.8.2 Step 2. Implement Positional Encoding

The main function of positional encoding is to provide model with information about the relative position of word tokens within utterance for attention learning given by the following formula:

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_{model}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{model}}\right) \end{aligned} \tag{16.2}$$

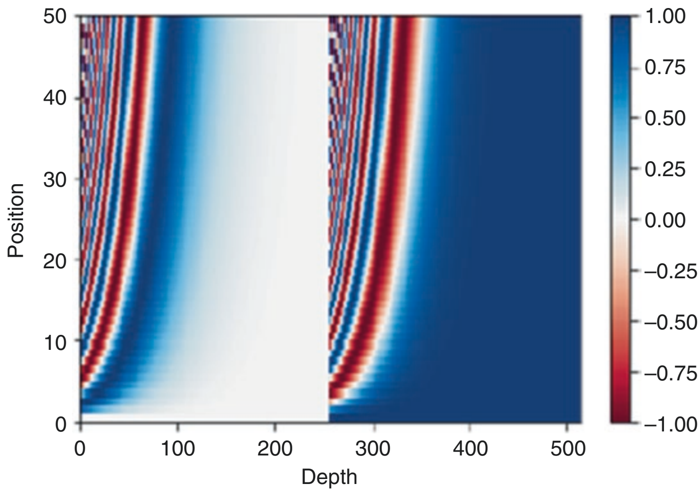
In[19]
```
# Implementation of Positional Encoding Class (PEncoding)
class PEncoding(tflow.keras.layers.Layer):

    def __init__(self, pos, dm):
        super(PEncoding, self).__init__()
        self.pencode = self.pencods(pos, dm)

    def gdeg(self, pos, i, dm):
        deg = 1 / tflow.pow(10000,(2 * (i // 2)) /
 tflow.cast(dm, tflow.float32))
        return pos * deg

    def pencods(self, pos, dm):
        deg_rads = self.gdeg(pos = tflow.range(pos, dtype=tflow.float32)
[:, tflow.newaxis], i=tflow.range(dm, dtype=tflow.float32)
[tflow.newaxis, :], dm = dm)
        m_sin = tflow.math.sin(deg_rads[:, 1::2])
        m_cos = tflow.math.cos(deg_rads[:, 1::2])

        pencode = tflow.concat([m_sin, m_cos], axis = -1)
        pencode = pencode[tflow.newaxis, ...]
        return tflow.cast(pencode, tflow.float32)

    def call(self, inNodes):
        return inNodes + self.pencode[:, :tflow.shape(inNodes)[1], :]
```

Try to plot *PositionalEncoding* diagram:

In[20]

```
# Create PositionalEncoding Sample
pencoding_sample = PEncoding(50, 512)

pyplt.pcolormesh(pencoding_sample.pencode.numpy()[0], cmap = 'RdBu')
pyplt.xlabel('Depth')
pyplt.xlim((0, 512))
pyplt.ylabel('Position')
pyplt.colorbar()
pyplt.show()
```

Out[20]



16.4.8.3 Step 3. Implement Encoder Layer

Encoder Layer (enclayer) implementation involves:

1. Create MHAttention object.
2. Two dense layers.

Details as shown below:

In[21]

```
# Implementation of Encoder Layer (enclayer)
def enclayer(i, dm, nhd, drop, name="enclayer"):
  inNodes = tflow.keras.Input(shape=(None, dm), name="inNodes")
  pmask = tflow.keras.Input(shape=(1, 1, None), name="pmask")

  att = MHAttention(
      dm, nhd, name="att")({
          'q': inNodes,
          'k': inNodes,
          'v': inNodes,
          'm': pmask
      })
  att = tflow.keras.layers.Dropout(rate=drop)(att)
  att = tflow.keras.layers.LayerNormalization(
      epsilon=1e-6)(inNodes + att)

  outNodes = tflow.keras.layers.Dense(units=i, activation='relu')(att)
  outNodes = tflow.keras.layers.Dense(units=dm)(outNodes)
  outNodes = tflow.keras.layers.Dropout(rate=drop)(outNodes)
  outNodes = tflow.keras.layers.LayerNormalization(
      epsilon=1e-6)(att + outNodes)

  return tflow.keras.Model(
      inputs=[inNodes, pmask], outputs=outNodes, name=name)
```

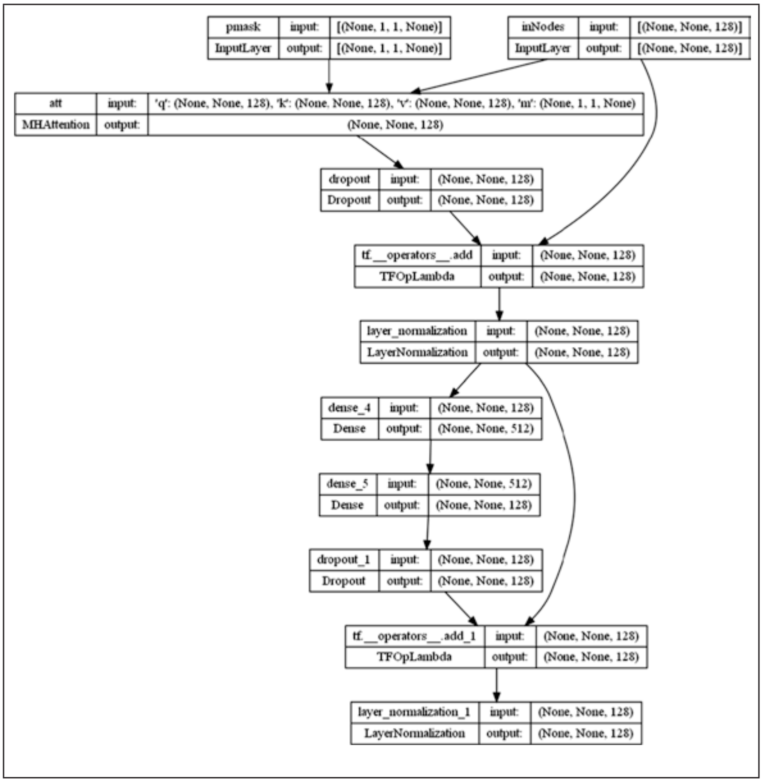
1. An Attention Learning object is defined and used at Encoder Layer implementation class
2. *relu* function is used as default setting for Encoder Layer Activation Function. Current research includes the modification (or change) of activation function for system enhancement

Try to display a sample Encoder Layer using Keras plot model():

In[22]

```
# Create a sample Encoder Layer and display object  diagram
enclayer_sample = enclayer(i = 512, dm = 128, nhd = 4, drop = 0.3, name =
"enclayer_sample")
tflow.keras.utils.plot_model(enclayer_sample, to_file = 'enclayer.png',
show_shapes = True)
```

Out[22]



16.4.8.4 Step 4. Implement Encoder

Encoder implementation involves the following processes:

1. Embed inputs.
2. Perform positional encoding scheme.
3. Encode Num Layers

In[23]

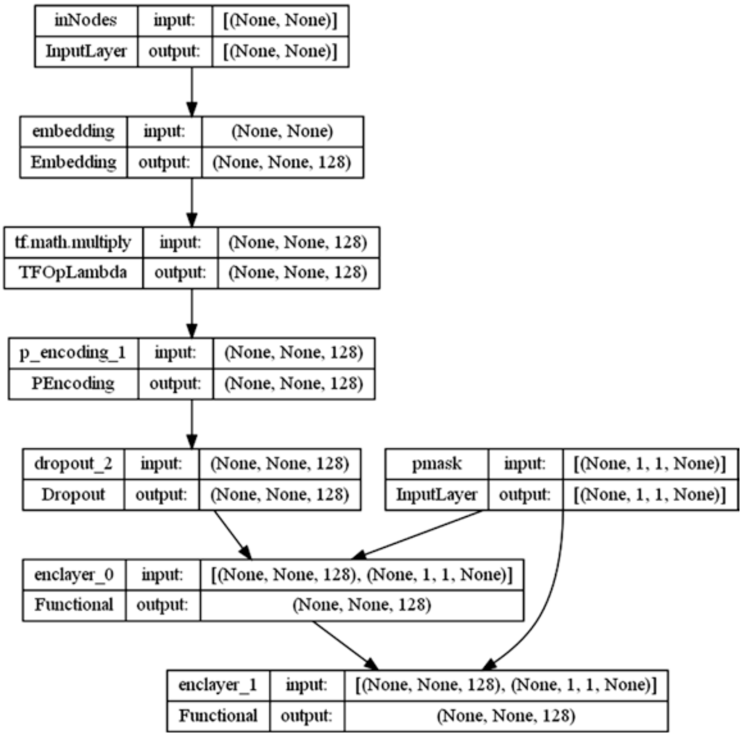
```
# Implementation of Encoder Class (encoder)
def encoder(svcab,
        nlayers,
        x,
        dm,
        nhd,
        drop,
        name="encoder"):
 inNodes = tflow.keras.Input(shape=(None,), name="inNodes")
 pmask = tflow.keras.Input(shape=(1, 1, None), name="pmask")

 embeddings = tflow.keras.layers.Embedding(svcab, dm)(inNodes)
 embeddings *= tflow.math.sqrt(tflow.cast(dm, tflow.float32))
 embeddings = PEncoding(svcab, dm)(embeddings)

 outNodes = tflow.keras.layers.Dropout(rate=drop)(embeddings)

 for i in range(nlayers):
  outNodes = enclayer(
    i=x,
    dm=dm,
    nhd=nhd,
    drop=drop,
    name="enclayer_{}".format(i),
  )([outNodes, pmask])


 return tflow.keras.Model(
   inputs=[inNodes, pmask], outputs=outNodes, name=name)
```

Display a sample Encoder using Keras Plot model:

In[24]

```
# Create a sample Encoder Sample and  display object  diagram
encoder_sample = encoder(svcab = 8192,
                         nlayers = 2,
                         x = 512,
                         dm = 128,
                         nhd = 4,
                         drop = 0.3,
                         name = "encoder_sample")

tflow.keras.utils.plot_model
(encoder_sample, to_file='encoder_sample.png', show_shapes = True)
```

Out[24]



16.4.8.5 Step 5. Implement Decoder Layer

Decoder Layer implementation involves the following steps:

1. MHAttention.
2. 2 Dense Decoder Layers with dropout.

In[25]

```
# Implementation of Decoder Layer (declayer)
def declayer(i, dm, nhd, drop, name = "declayer"):
    inNodes = tflow.keras.Input(shape=(None, dm), name ="inNodes")
    encouts = tflow.keras.Input(shape=(None, dm), name="encouts")
    lamask = tflow.keras.Input(shape=(1, None, None), name = "lamask")
    pmask = tflow.keras.Input(shape=(1, 1, None), name = "pmask")

    att1 = MHAttention(dm, nhd, name="att1")(inNodes={'q':inNodes,
                                              'k':inNodes,
                                              'v':inNodes,
                                              'm':lamask})

    att1 = tflow.keras.layers.LayerNormalization(epsilon=1e-6)
(att1 + inNodes)

    att2 = MHAttention(dm,nhd, name = "att2")(inNodes={'q':att1,
                                              'k':encouts,
                                              'v':encouts,
                                              'm':pmask})
    att2 = tflow.keras.layers.Dropout(rate=drop)(att2)
    att2 = tflow.keras.layers.LayerNormalization(epsilon = 1e-6)(att2 + att1)

    outNodes = tflow.keras.layers.Dense(units=i, activation='relu')(att2)
    outNodes = tflow.keras.layers.Dense(units=dm)(outNodes)
    outNodes = tflow.keras.layers.Dropout(rate=drop)(outNodes)
    outNodes = tflow.keras.layers.LayerNormalization(epsilon=1e-6)(out-
Nodes + att2)

    return tflow.keras.Model(inputs=[inNodes, encouts, lamask, pmask],
                    outputs = outNodes,
                    name = name)
```

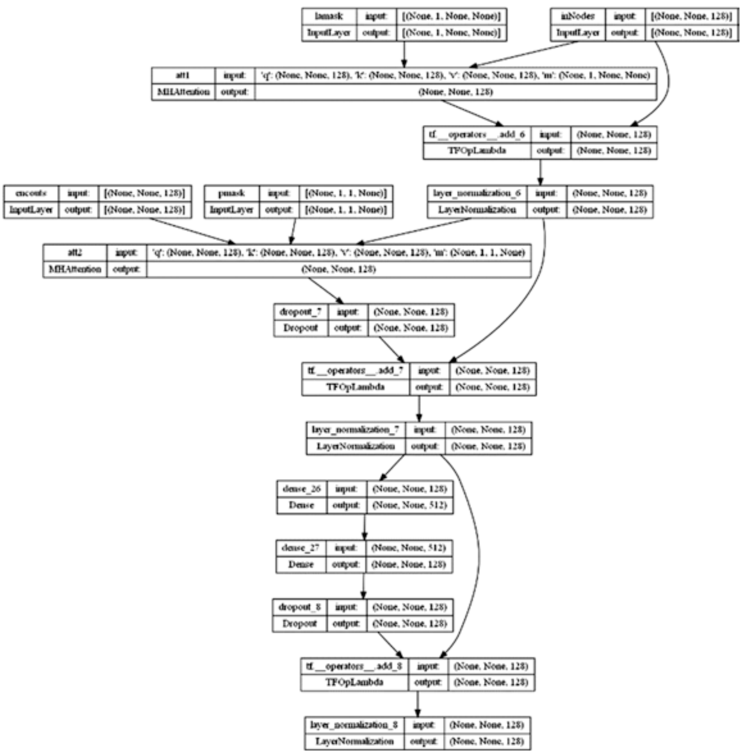
1. Encoder Layer implements single Attention Learning object, and Decoder Layer implements two Attention Learning objects att1 and att2 according to Transformer Learning model
2. Again, *relu* function is used as Activation Function. It can modify or adopt different Activation Function to improve network performance as studied in Sect. 16.1

Display sample Decoder Layer using Keras *plot_model()*:

In[26]

```
# Create a decoder layer sample and show object association diagram
declayer_sample = declayer(i = 512, dm = 128, nhd = 4, drop = 0.3,
name = "declayer_sample")

tflow.keras.utils.plot_model
(declayer_sample, to_file='declayer_sample.png', show_shapes=True)
```

Out[26]



16.4.8.6 Step 6. Implement Decoder

Decoder implementation involves the following processes:

1. Embed network outputs.
2. Look ahead and pad masking.
3. Positional encoding scheme.
4. Perform N-decoder layers.

In[27]

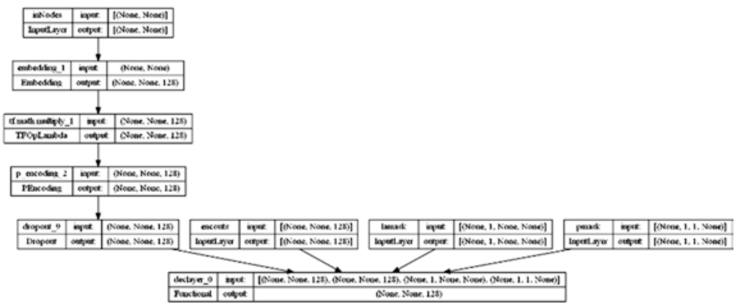
```
# Implementation of Decoder class (decoder)
def decoder(svcab,
        nlayers,
        x,
        dm,
        nhd,
        drop,
        name='decoder'):
    inNodes  = tflow.keras.Input(shape=(None,), name="inNodes")
    encouts = tflow.keras.Input(shape=(None, dm), name="encouts")
    lamask  = tflow.keras.Input(shape=(1, None, None), name="lamask")
    pmask   = tflow.keras.Input(shape=(1, 1, None), name="pmask")

    embeddings = tflow.keras.layers.Embedding(svcab, dm)(inNodes)
    embeddings *= tflow.math.sqrt(tflow.cast(dm, tflow.float32))
    embeddings = PEncoding(svcab, dm)(embeddings)

    outNodes = tflow.keras.layers.Dropout(rate=drop)(embeddings)

    for i in range(nlayers):
      outNodes = declayer(i = x,
                  dm=dm,
                  nhd=nhd,
                  drop=drop,
                  name = 'declayer_{}'.format(i),)(inputs=[outNodes, en-
couts, lamask, pmask])

      return tflow.keras.Model(inputs=[inNodes, encouts, lamask, pmask],
                   outputs = outNodes,
                   name = name)
```

Display sample Decoder using Keras Plot_model:

In[28]

```
# Create a decoder sample and show object association diagram
decoder_sample = decoder(svcab=8192,
                         nlayers=2,
                         x = 512,
                         dm = 128,
                         nhd = 4,
                         drop = 0.3,
                         name = "decoder_sample")
tflow.keras.utils.plot_model(decoder_sample,
to_file='decoder_sample.png', show_shapes = True)
```

Out[28]



16.4.8.7 Step 7. Implement Transformer

Transformer involves implementing Encoder, Decoder, and the final Linear Layer.

Transformer Decoder output is input to Linear Layer as a Recurrent Neural Network (RNN) and output model is returned.

In[29]

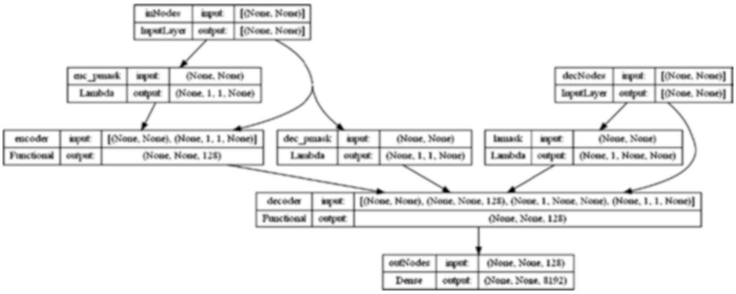
```
# Implementation of Transformer Class
def transformer(svcab, nlayers,  x,  dm, nhd, drop, name="transformer"):
    queries = tflow.keras.Input(shape=(None,), name="inNodes")
    dec_queries = tflow.keras.Input(shape=(None,), name="decNodes")

    enc_pmask = tflow.keras.layers.Lambda(
    gen_pmask, output_shape=(1, 1, None),
    name="enc_pmask")(queries)

    # Perform Look Ahead Masking for Decoder Input for the Att1
    lamask = tflow.keras.layers.Lambda(gen_lamask,
                                       output_shape=(1, None, None),
                                       name = "lamask")(dec_queries)

    # Perform Padding Masking for Encoder Output for the Att2
    dec_pmask = tflow.keras.layers.Lambda(gen_pmask,
                                          output_shape=(1, 1, None),
                                          name="dec_pmask")(queries)

    encouts = encoder(svcab=svcab,
                      nlayers = nlayers,
                      x = x,
                      dm = dm,
                      nhd = nhd,
                      drop = drop,)(inputs = [queries, enc_pmask])

    decouts = decoder(svcab=svcab,
                      nlayers = nlayers,
                      x = x,
                      dm = dm,
                      nhd = nhd,
                      drop=drop,)(inputs=[dec_queries, encouts, lamask, dec_
pmask])

    responses =
tflow.keras.layers.Dense(units=svcab, name="outNodes")(decouts)

    return tflow.keras.Model(inputs=[queries, dec_queries],
outputs=responses, name=name)
```

Display sample transformer object using Keras Plot_model:

In[30]

```
# Create a transformer sample and display object diagram
transformer_sample = transformer(svcab=8192,  nlayers=4, x=512,
      dm=128,  nhd = 4, drop=0.3, name="transformer_sample")

tflow.keras.utils.plot_model(transformer_sample,
to_file="transformer_sample.png", show_shapes=True)
```

Out[30]



16.4.8.8 Step 8. Model Training

Parameters for nLayers, dm, and units (x) had reduced to speed up training process.

In[31]

```
# Create Transformer Model
tflow.keras.backend.clear_session()

model = transformer(svcab = SVCAB,
                    nlayers=2,
                    x=512,
                    dm=256,
                    nhd=8,
                    drop=0.1)
```

1. A Movie Chatbot Transformer Model consists of two layers with 512 units, data-model size 256, head number 8 and dropout rate 0.1 according to Transformer Model as in Fig. 16.2
2. It is recommended to modify these parameter settings to improve network performance as discussed in Sect. 16.1

16.4.8.9 Step 9. Implement Model Evaluation Function

A loss function is implemented for system evaluation. It is important to apply a padding mask when calculating the loss since target sequences are padded.

In[32]
```
# Implementation of Evaluation Function (Loss Function)
def Eval_function(xtrue, xpred):
    xtrue = tflow.reshape(xtrue, shape=(-1, MLEN - 1))

    loss_val = tflow.keras.losses.SparseCategoricalCrossentropy(
    from_logits= True, reduction='none')(xtrue, xpred)

    mask_val = tflow.cast(tflow.not_equal(xtrue, 0), tflow.float32)
    loss_val = tflow.multiply(loss_val, mask_val)

    return tflow.reduce_mean(loss_val)
```

16.4.8.10 Step 10. Implement Customized Learning Rate

Adam_Optimizer with customized learning rate is used with formula below:

$$I_{\text{rate}} = d_{\text{model}}^{-5} * \min\left(step\_num^{-0.5}, step\_num * warmup\_steps^{-1.5}\right) \tag{16.3}$$

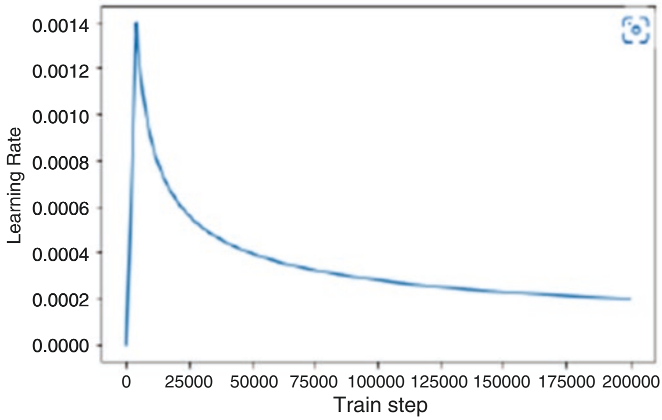
In[33]
```
# Implementation of Customized Learning Rate
class CLearning(tflow.keras.optimizers.schedules.LearningRateSchedule):

    def __init__(self, dm, warmup_steps=4000):
        super(CLearning, self).__init__()
        self.dm = dm
        self.dm = tflow.cast(self.dm, tflow.float32)

        self.warmup_steps = warmup_steps

    def __call__(self, step):
        # arg1 = tflow.math.rsqrt(step)
        arg1 = tflow.math.rsqrt(tflow.cast(step, tflow.float32))
        arg2 = tflow.cast(step, tflow.float32) * (tflow.cast(self.warmup_steps,
tflow.float32)**-1.5)

        return tflow.math.rsqrt(self.dm) * tflow.math.minimum(arg1, arg2)
```

Plot customized Learning Rate:

In[34]

```
# Create customized learning rate object and display performance
CLearning_sample = CLearning(dm=128)

pyplt.plot(CLearning_sample(tflow.range(200000, dtype=tflow.float32)))
pyplt.ylabel("Learning Rate")
pyplt.xlabel("Train Step")
```

Out[34]



16.4.8.11 Step 11. Compile Chatbot Model

In[35]

```
# Compile Movie Chatbot Model
# Set the Customized Learning Rate
cLRate = CLearning(256)

# Set Adam Optimizers
optimizer = tflow.keras.optimizers.Adam
(learning_rate=cLRate, beta_1=0.9, beta_2=0.98, epsilon=1e-9)

# Implement Accuracy Evaluation Scheme
def accuracy(xtrue, xpred):
    xtrue = tflow.reshape(xtrue, shape=(-1, MLEN - 1))
    return tflow.keras.metrics.sparse_categorical_accuracy(xtrue, xpred)

# Compile Chatbot Model
model.compile(optimizer=optimizer, loss=Eval_function,
metrics=[accuracy])
```

16.4.8.12 Step 12. System Training (Model Fitting)

Train Chatbot transformer model by calling *model.fit()* for 20 epochs to save time.

In[36]
```
EPOCHS = 20

model.fit(mDS, epochs = EPOCHS)
```

Out[36]
```
Epoch 1/20
689/689 [=====] - 404s 578ms/step - loss: 2.1267 - accuracy: 0.0412
Epoch 2/20
689/689 [=====] - 406s 589ms/step - loss: 1.5020 - accuracy: 0.0785
Epoch 3/20
689/689 [=====] - 402s 584ms/step - loss: 1.3893 - accuracy: 0.0857
Epoch 4/20
689/689 [=====] - 416s 604ms/step - loss: 1.3230 - accuracy: 0.0903
Epoch 5/20
689/689 [=====] - 419s 608ms/step - loss: 1.2658 - accuracy: 0.0948
Epoch 6/20
689/689 [=====] - 414s 601ms/step - loss: 1.2175 - accuracy: 0.0982
Epoch 7/20
689/689 [=====] - 413s 599ms/step - loss: 1.1641 - accuracy: 0.1022
Epoch 8/20
689/689 [=====] - 392s 569ms/step - loss: 1.1055 - accuracy: 0.1079
Epoch 9/20
689/689 [=====] - 407s 591ms/step - loss: 1.0525 - accuracy: 0.1134
Epoch 10/20
689/689 [=====] - 389s 565ms/step - loss: 1.0030 - accuracy: 0.1189
Epoch 11/20
689/689 [=====] - 392s 569ms/step - loss: 0.9590 - accuracy: 0.1248
Epoch 12/20
689/689 [=====] - 391s 567ms/step - loss: 0.9185 - accuracy: 0.1297
Epoch 13/20
689/689 [=====] - 390s 567ms/step - loss: 0.8817 - accuracy: 0.1350
Epoch 14/20
689/689 [=====] - 391s 567ms/step - loss: 0.8483 - accuracy: 0.1398
Epoch 15/20
689/689 [=====] - 390s 567ms/step - loss: 0.8176 - accuracy: 0.1445
Epoch 16/20
689/689 [=====] - 391s 567ms/step - loss: 0.7901 - accuracy: 0.1483
Epoch 17/20
689/689 [=====] - 391s 568ms/step - loss: 0.7633 - accuracy: 0.1532
Epoch 18/20
689/689 [=====] - 397s 576ms/step - loss: 0.7395 - accuracy: 0.1569
Epoch 19/20
689/689 [=====] - 395s 574ms/step - loss: 0.7178 - accuracy: 0.1604
Epoch 20/20
689/689 [=====] - 399s 579ms/step - loss: 0.6961 - accuracy: 0.1642

<keras.callbacks.History at 0x23ed65e5e80>
```

16.4.8.13 Step 13. System Evaluation and Live Chatting

System evaluation and live chatting implementation involve following steps:

1. Create Mining() method by performing data preprocessing of all utterances.
2. Perform tokenization of utterances and padded with START and END tokens.
3. Perform LookAhead and Padding Masks.
4. Construct Transformer model with Attention Learning.
5. Implement chatting() method by decoder scheme.
6. Combine chatted word sequences to decoder input.
7. Use Transformer Model for system to predict responses based on previous training epochs.

In[37]

```
# Implementation of Movie Chatting class - mchat
def mchat(utterance):
    # Utterance Preprocessing and add START AND END TOKENS
    utterance = pp_utterance(utterance)
    utterance = tflow.expand_dims(START_TOKEN +
m_token.encode(utterance) + END_TOKEN, axis = 0)

    # Create response object
    response = tflow.expand_dims(START_TOKEN, 0)

    for i in range(MLEN):
        chatting = model(inputs = [utterance, response], training = False)

        # Choose last_word from token sequence
        chatting = chatting[:, -1:, :]
        chatted_id = tflow.cast(tflow.argmax(chatting, axis=-1), tflow.int32)

        # Return with chattedID with ENDTOKEN
        if tflow.equal(chatted_id, END_TOKEN[0]):
            break

        # Combine CHATTEDID with utterance response
        response = tflow.concat([response, chatted_id], axis=-1)

    return tflow.squeeze(response, axis = 0)
```

```
# Implementation of main class for Movie Chatting - mchatting
def mchatting(utterance):
    mchatting = mchat(utterance)

    chatted_utterance =
m_token.decode([i for i in mchatting if i < m_token.vocab_size])

    print('Query: {}'.format(utterance))
    print('Response: {}'.format(chatted_utterance))

    return chatted_utterance
```

Try some movie conversations to see whether it works:

In[38]

```
output = mchatting("Where have you been?")
```

Out[38]

```
Query: Where have you been?
Response: i m going to get my father .
```

In[39]

```
output = mchatting("It's a trap")
```

Out[39]

```
Query: It's a trap
Response: i don t know what to do . it s just that way .
```

In[40]

```
output = mchatting("Do you need help?")
```

Out[40]

```
Query: Do you need help?
Response: no .
```

In[41]

```
output = mchatting("What do you think?")
```

Out[41]

```
Query: What do you think?
Response: i don t know . i don t know . i m not sure . i just had to see what
i m saying .
```

In[42]

```
output = mchatting("Are you happy?")
```

Out[42]

```
Query: Are you happy?
Response: no . but you re not . you re not sure ?
```

1. Training showed that epochs 1–20 are rather slow but increased in accuracy and decreased in loss rate
2. Two chatbots experiments with one used 2 epochs and the other used 20 epochs. Results showed that performance on 20 epochs has satisfactory performance than the one with 2 epochs
3. Increase epochs, say up to 50 epochs to review whether accuracy has continuous improvement. It is natural to require more time unless there are sufficient GPUs

WORKSHOP

Workshop 7.1 Fine-tune Chatbot Model

TensorFlow and Transformer Technology are used to develop a domain-based Chatbot system

There are rooms to fine-tune model performance like any AI model. It can be conducted by:

1. Dataset Level
   - Enhance preprocessing process
   - Improve data record selection scheme, e.g. sample size, utterance max length, etc.
2. Network Model Level
   - Fine-tune system parameters, e.g. Learning Rate and Method, etc.
   - Fine-tune Transformer Model by modifying Attention Function etc. Compare performances (MUST) and analysis (bonus)

Fine-tune Movie Chatbot model and compare with original version

WORKSHOP

Workshop 7.2 Mini Project - Build a Semantic-Level AI Chatbot System

Extend Character-level and Word-level NLU to a Semantic-Level NLU

1. Modify codes of AI Chatbot learnt in this section to implement a Semantic-level AI Chatbot system
2. Compare system performance of this revised system with previous Character-level and Word-level AI Chatbot system

16.5 Related Works

This workshop had integrated all NLP related implementation techniques including TensorFlow and Keras with Transformer Technology to design an AI-based NLP application chatbot system. It is a step-by-step implementation consisting of data preprocessing, model construction, system training, testing evaluation process; and Attention Learning and Transformer Technology with TensorFlow and Keras implementation platform easily applied to other chatbot domain and interactive QA systems using Cornell Large Movie dataset with over 200,000 movie conversations with 10,000+ movie characters.

Nevertheless, it is only the dawn of journey. There are regular new R&D prevalence and usage in NLP applications. Below are lists of renowned domains and resources related to chatbot systems for reference.

Datasets for Chatbot Systems:

- Taskmaster from Google Research (GoogleResearch 2022a).
- Simulated Dialogue dataset from Google Research (GoogleResearch 2022b).
- Dialog Challenge dataset from Microsoft (MicrosoftDialog 2022).
- Dialog State Tracking Challenge dataset (DSTC 2022).

Keras Modules and Optimizer:

- Keras layers (Keras 2022a).
- Keras optimizers (Keras 2022b).
- An overview of optimizers (Ruder 2022).
- Adam optimizer (Adam 2022).

Famous Chatbot System:

- Amazon Alexa developer blog (Alexa 2022).
- Apple Siri Developer (AppleSiri 2022).
- Duer from Baidu (Duer 2022).
- Google Assistant (GoogleAssistant 2022).
- Microsoft Cortana Developer (MicrosoftCortana 2022).
- Samsung Bixby Developer (SamsungBixby 2022).
- Xiaowei from Tencent (Xiaowei 2022).

References

Adam (2022) Adam optimizer: https://arxiv.org/abs/1412.6980. Accessed 29 June 2022.

Alexa (2022) Amazon Alexa developer blog: https://developer.amazon.com/blogs/home/tag/Alexa. Accessed 29 June 2022.

AppleSiri (2022) Apple Siri Developer: https://developer.apple.com/siri/. Accessed 29 June 2022.

Bansal, A. (2021) Advanced Natural Language Processing with TensorFlow 2: Build effective real-world NLP applications using NER, RNNs, seq2seq models, Transformers, and more. Packt Publishing.

Batish, R. (2018) Voicebot and Chatbot Design: Flexible conversational interfaces with Amazon Alexa, Google Home, and Facebook Messenger. Packt Publishing.

Cornell (2022) https://www.cs.cornell.edu/~cristian/Chameleons_in_imagined_conversations.html. Accessed 29 June 2022.

Cornell_Movie_Corpus (2022) Cornell Movie Corpus archive. https://www.kaggle.com/datasets/Cornell-University/movie-dialog-corpus. Accessed 29 June 2022.

Devlin, J., Chang, M. W., Lee, K. and Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Archive: https://arxiv.org/pdf/1810.04805.pdf.

Duer (2022) Duer Baidu AI Chatbot. http://duer.baidu.com/en/index.html. Accessed 29 June 2022.

DSTC (2022) Dialog State Tracking Challenge dataset: https://github.com/matthen/dstc. Accessed 29 June 2022.

Ekman, M. (2021) Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow. Addison-Wesley Professional.

Facebook (2022) Facebook Messenger API documentation. https://developers.facebook.com/docs/messenger-platform/getting-started/quick-start/. Accessed 29 June 2022.
Freed, A. (2021) Conversational AI: Chatbots that work. Manning.
Géron, A. (2019) Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media.
GoogleAssistant (2022) Google Assistant: https://assistant.google.com/. Accessed 29 June 2022.
GoogleResearch (2022a) Taskmaster from Google Research. https://github.com/google-research-datasets/Taskmaster/tree/master/TM-1-2019. Accessed 29 June 2022.
GoogleResearch (2022b) Simulated Dialogue dataset from Google Research. https://github.com/google-research-datasets/simulated-dialogue. Accessed 29 June 2022.
Janarthanam, S. (2017) Hands-On Chatbots and Conversational UI Development: Build chatbots and voice user interfaces with Chatfuel, Dialogflow, Microsoft Bot Framework, Twilio, and Alexa Skills. Packt Publishing.
Kedia, A. and Rasu, M. (2020) Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications. Packt Publishing.
Keras (2022a) Keras official sites: https://keras.io. Accessed 29 June 2022.
Keras (2022b) Keras optimizers: https://keras.io/api/optimizers/. Accessed 29 June 2022.
MicrosoftCortana (2022) Microsoft Cortana Developer: https://www.microsoft.com/en-us/cortana/. Accessed 29 June 2022.
MicrosoftDialog (2022) https://github.com/xiul-msr/e2e_dialog_challenge. Accessed 29 June 2022.
NLPWorkshop7 (2022) NLP Workshop 6 GitHub archive. https://github.com/raymondshtlee/NLP/tree/main/NLPWorkshop7. Accessed 29 June 2022.
Raj, S. (2018) Building Chatbots with Python: Using Natural Language Processing and Machine Learning. Apress.
Rothman, D. (2022) Transformers for Natural Language Processing: Build, train, and fine-tune deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, and GPT-3. Packt Publishing.
Ruder (2022) An overview of optimizers: https://ruder.io/optimizing-gradient-descent/. Accessed 29 June 2022.
SamsungBixby (2022) Samsung Bixby Developer: https://developer.samsung.com/bixby. Accessed 29 June 2022.
Telegram (2022) Telegram bot API documentation: (https://core.telegram.org/bots. Accessed 29 June 2022.
TensorFlow (2022) TensorFlow official site> https://tensorflow.org /. Accessed 29 June 2022.
Tuchong (2022) Wake word to invoke your Chatbot. https://stock.tuchong.com/image/detail?imageId=918495180260638796. Accessed 29 June 2022.
Tunstall, L, Werra, L. and Wolf, T. (2022) Natural Language Processing with Transformers: Building Language Applications with Hugging Face. O'Reilly Media.
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30. https://arxiv.org/abs/1706.03762.
Xiaowei (2022) Xiaowei chatbot system from Tencent. https://xiaowei.tencent.com/. Accessed 29 June 2022.
Yıldırım, S, Asgari-Chenaghlu, M. (2021) Mastering Transformers: Build state-of-the-art models from scratch with advanced natural language processing techniques. Packt Publishing.