CREATE A CHATBOT IN PYTHON

By M.Vignesh– 411421205054

(B.Tech/ Information Technology, 3rd year)

Domain Name: Artificial Intelligence

Phase-4 Document Submission

Project: To create a Chatbot in Python that provides exceptional answering user queries (diabetes) on a website.

Introduction:

- A friendly and informative companion to help you navigate the world of diabetes. Whether you have questions about diabetes types, management, or just need some guidance, our chatbot is here to provide you with answers and support. Simply start a conversation, and our chatbot will assist you with valuable information and guidance related to diabetes.

- In our increasingly interconnected world, diabetes affects millions of people across different cultures and languages. Recognizing the importance of accessible and inclusive healthcare support, we present our Multilingual Diabetes Chatbot.

- This chatbot transcends linguistic barriers, offering vital information and assistance to individuals living with or seeking information about diabetes, regardless of the language they speak. Whether you're looking for advice on diabetes management, information on symptoms and types, or simply need a knowledgeable companion to converse with, our chatbot is here to help.

Importing Libraries:

```
import numpy as np
import string
from nltk.corpus import stopwords
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction.text import TfidfTransformer,TfidfVectorizer
from sklearn.pipeline import Pipeline
df = pd.read_csv('../input/simple-dialogs-for-chatbot/dialogs.txt',sep='\t')
a = pd.Series(df.columns)
df
hi, how are you doing?       i'm fine. how about yourself?
0       i'm fine. how about yourself?       i'm pretty good. thanks for asking.
1       i'm pretty good. thanks for asking.no problem. so how have you been?
2       no problem. so how have you been?       i've been great. what about you?
3       i've been great. what about you? i've been good. i'm in school right now.
4       i've been good. i'm in school right now.    what school do you go to?
...       ...       ...
3719 that's a good question. maybe it's not old age. are you right-handed?
3720   are you right-handed?       yes. all my life.
3721   yes. all my life.       you're wearing out your right hand. stop using...
3722   you're wearing out your right hand. stop using...       but i do all my writing with my right hand.
3723   but i do all my writing with my right hand.       start typing instead. that  way your left hand ...
3724 rows × 2 columns

a = a.rename({0: df.columns[0],1: df.columns[1]})
Adding some common Questions
b = {'Questions':'Hi','Answers':'hello'}
c = {'Questions':'Hello','Answers':'hi'}
d= {'Questions':'how are you','Answers':"i'm fine. how about yourself?"}
e= {'Questions':'how are you doing','Answers':"i'm fine. how about yourself?"}
df = df.append(a,ignore_index=True)
df.columns=['Questions','Answers']
df = df.append([b,c,d,e],ignore_index=True)
df
                Questions                                Answers
0       i'm fine. how about yourself?       i'm pretty good. thanks for asking.
1       i'm pretty good. thanks for asking.no problem. so how have you been?
2       no problem. so how have you been?       i've been great. what about you?
3       i've been great. what about you? i've been good. i'm in school right now.
```

```
4       i've been good. i'm in school right now.    what school do you go to?
...     ...         ...
3724    hi, how are you doing?                      i'm fine. how about yourself?
3725    Hi                                          hello
3726    Hello                                       hi
3727    how are you i'm fine.                       how about yourself?
3728    how are you doing                           i'm fine. how about yourself?
3729 rows × 2 columns

df = df.append(c,ignore_index=True)
df = df.append(d,ignore_index=True)
df = df.append(d,ignore_index=True)
df
```

	Questions	Answers
0	i'm fine. how about yourself?	i'm pretty good. thanks for asking.
1	i'm pretty good. thanks for asking.	no problem. so how have you been?
2	no problem. so how have you been?	i've been great. what about you?
3	i've been great. what about you?	i've been good. i'm in school right now.
4	i've been good. i'm in school right now.	what school do you go to?
...	...	...
3727	how are you i'm fine.	how about yourself?
3728	how are you doing i'm fine.	how about yourself?
3729	Hello	hi
3730	how are you i'm fine.	how about yourself?
3731	how are you i'm fine.	how about yourself?

```
3732 rows × 2 columns

def cleaner(x):
    return [a for a in (''.join([a for a in x if a not in string.punctuation])).lower().split()]
Decision Tree Classifier
Pipe = Pipeline([
    ('bow',CountVectorizer(analyzer=cleaner)),
    ('tfidf',TfidfTransformer()),
    ('classifier',DecisionTreeClassifier())
])
Pipe.fit(df['Questions'],df['Answers'])
Pipeline(steps=[('bow',
                 CountVectorizer(analyzer=<function cleaner at 0x7f5cfaae40e0>)),
                ('tfidf', TfidfTransformer()),
                ('classifier', DecisionTreeClassifier())])
Now we can talk to our chatbot
Pipe.predict(['hi'])[0]
'hello'
Pipe.predict(['how are you'])[0]
"i'm fine. how about yourself?"
Pipe.predict(['great'])[0]
'i appreciate that.'
Pipe.predict(['What are you doing'])[0]
"i'm going to change the light bulb. it burnt out."
```

Prepare Dataset:

Load and preprocess data

To keep this example simple and fast, the maximum number of training samples to MAX_SAMPLES=5000 and the maximum length of the sentence to be MAX_LENGTH=40.

The dataset is preprocessed in the following order:

- Extract MA_SAMPLES conversation pairs inot list of questions and anwers.
- Preprocess each sentence by removing special characters in each sentence.
- Build tokenizer (map text to ID and ID to text) using TensorFlow Datasets SubwordTextEncoder.
- Tokenize each sentence and add START_TOKEN and END_TOKEN to indicate the start and end of each sentence.
- Filter out sentence that has more than MA__LENGTH tokens.
- Pad tokenized sentences to MAX_LENGTH

```
# Maximum number of samples to preprocess
MAX_SAMPLES = 50000

def preprocess_sentence(sentence):
    sentence = sentence.lower().strip()
    # creating a space between a word and the punctuation following it
    # eg: "he is a boy." => "he is a boy ."
    sentence = re.sub(r"([?.!,])", r" \1 ", sentence)
    # Get rid of unnecessary space
    sentecne = re.sub(r'[" "]+', " ", sentence)
    # replacing everything with space except (a-z, A-Z, ".", "?", "!", ",")
    sentence = re.sub(r"[^a-zA-Z?.,!]+", " ", sentence)
    sentence = sentence.strip()
    # adding a start and an end token to the sentence
    return sentence

def load_conversations():
    # dictionary of line id to text
    id2line = {}
    with open('/kaggle/input/cornell-moviedialog-corpus/movie_lines.txt', encoding = 'utf-
8', errors = 'ignore') as file:
        lines = file.readlines()
    for line in lines:
        parts = line.replace('\n', '').split(' +++$+++ ')
        id2line[parts[0]] = parts[4]

    inputs, outputs = [], []
    with open('/kaggle/input/cornell-moviedialog-corpus/movie_conversations.txt',
                encoding = 'utf-8', errors = 'ignore') as file:
        lines = file.readlines()
    for line in lines:
        parts = line.replace('\n', '').split(' +++$+++ ')
```

```
 # get conversation in a list of line ID
        conversation = [line[1:-1] for line in parts[3][1:-1].split(', ')]
        for i in range(len(conversation) - 1):
            inputs.append(preprocess_sentence(id2line[conversation[i]]))
            outputs.append(preprocess_sentence(id2line[conversation[i + 1]]))
            if len(inputs) >= MAX_SAMPLES:
                return inputs, outputs
    return inputs, outputs

questions, answers = load_conversations()
print('Sample questioin: {}'.format(questions[20]))
print('Sample answer: {}'.format(answers[20]))
Sample questioin: i really , really , really wanna go , but i can t . not unless my sister goes .
Sample answer: i m workin on it . but she doesn t seem to be goin for him .
# Build tokenizer for both questions and answers
tokenizer = tfds.features.text.SubwordTextEncoder.build_from_corpus(questions + answers, target_vocab_size = 2**13)

# Define start and end token to indicate the start and end of a sentence
START_TOKEN, END_TOKEN = [tokenizer.vocab_size], [tokenizer.vocab_size + 1]

# Vocabulary size plus start and end token
VOCAB_SIZE = tokenizer.vocab_size + 2
print('Tokenized sample question: {}'.format(tokenizer.encode(questions[20])))
Tokenized sample question: [4, 281, 3, 281, 3, 143, 395, 176, 3, 42, 4, 38, 8191, 2, 37, 873, 27, 2031, 3096, 1]
# Maximum sentence length
MAX_LENGTH = 40

# Tokenize, filter and pad sentences
def tokenize_and_filter(inputs, outputs):
    tokenized_inputs, tokenized_outputs = [], []

    for (sentence1, sentence2) in zip(inputs, outputs):
        # tokenize sentence
        sentence1 = START_TOKEN + tokenizer.encode(sentence1) + END_TOKEN
        sentence2 = START_TOKEN + tokenizer.encode(sentence2) + END_TOKEN
        # check tokenized sentence max length
        if len(sentence1) <= MAX_LENGTH and len(sentence2) <= MAX_LENGTH:
            tokenized_inputs.append(sentence1)
            tokenized_outputs.append(sentence2)

    # pad tokenized sentences
    tokenized_inputs = tf.keras.preprocessing.sequence.pad_sequences(tokenized_inputs, maxlen = MAX_LENGTH, padding = 'post')
    tokenized_outputs = tf.keras.preprocessing.sequence.pad_sequences(tokenized_outputs, maxlen=MAX_LENGTH, padding = 'post')

    return tokenized_inputs, tokenized_outputs
```

```
questions, answers = tokenize_and_filter(questions, answers)
print('Vocab size: {}'.format(VOCAB_SIZE))
print('Number of samples: {}'.format(len(questions)))
```
Vocab size: 8333
Number of samples: 44095

Create tf.data.Dataset

- tf.data.Dataset API is used to construct our input pipline in order to utilize features like caching and prefetching to speed up with training process.

- The transformer is an auto-regressive model: it makes predictions one part at a time, and uses its output so far to decide whta to do next.

- During training this example uses teacher-forcing. Teacher forcing is passing the
- To prevent the model from peaking at the expected output the model uses a look-ahead mask. true output to the next time step regardless of what the model predicts at the current time step.

- As the transformer predicts each word, self-attention allows it to look at the previous words in the input sequence to better predeict the next word.

-

- Target in divided into decoder_inputs which padded as an input to the decoder and corpped_targets for calculating our loss and accuracy.

```
BATCH_SIZE = 64
BUFFER_SIZE = 20000

# decoder inputs use the previous target as input
# remove START_TOKEN from targets
dataset = tf.data.Dataset.from_tensor_slices(({'inputs':questions,
'dec_inputs':answers[:, :-1]},{'outputs':answers[:, 1:]}))

dataset = dataset.cache()
dataset = dataset.shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE)
dataset = dataset.prefetch(tf.data.experimental.AUTOTUNE)
```

Attention:

Scaled dot product Attention

- The scaled dot-product attention functin used by the transformer takes three inputs: Q (query), K (key), V (value). The equation used to calculate the attention weights is:

- Attention(Q,K,V)=softmaxk(QKTdk−−√)

- As the softmax normalization is done on the key, its values decide the amout of importance given to the query.

- The output represents the multiplication of the attention weights and the value vector. This ensures that the words we want to focus on are kept as is and the irrelevant words are flushed out.

- The dot-product attention is scaled by a factor of square root of the depth. This is done because for large values of depth, the dot product grows large in magnitude pushing the softmax function where it has small gradients resulting in a very hard softmax.

- For example, consider that query and key have a mean of 0 and variance of 1. Their matrix mulitplication will have a mean of 0 and variance of dk

- . Hence, squre root of dk

- is used for scaling (and not any other number) because the matmul of query and key should have a mean of 0 and variance of 1, so that we get a gentler softmax.

- The mask is multiplied with -1e9 (close to negative infinity). This is done because the mask is summed with scaled matrix multiplication of query and key and is applied immediately before a softmax. The goal is to zero out these cells. and large negative inputs to softmax are near zero in the output.

```
def scaled_dot_product_attention(query, key, value, mask):
    """Calculate the attention weights."""
    matmul_qk = tf.matmul(query, key, transpose_b = True)

    # scale matmul_qk
    depth = tf.cast(tf.shape(key)[-1], tf.float32)
    logits = matmul_qk / tf.math.sqrt(depth)

    # add the mask to zero out padding tokens
    if mask is not None:
        logits += (mask * -1e9)

    #sofmax is normalized on the last axis (seq_len_k)
    attention_weights = tf.nn.softmax(logits, axis = -1)

    output = tf.matmul(attention_weights, value)

    return output
```

Mutil-head attention:

Multi-head attention consists of four parts:

- Linear layers and split into heads.

- Scaled dot-product attention.
- Concatenation of heads.
- Final linear layer.

Each multi-head attention block gets three inputs; Q (query), K (key), V (value). These are put through linear (Dense) layers and split up into multiple heads.

The scaled_dot_product_attention defined above is applied to each head (broadcasted for efficiency). An appropriate mask must be used in the attention step. The attention output for each head is then concatenated (using tf.transpose, and tf.reshape) and pur through a final Dense layer.

Instead of one single attention head, query, key and value are split into multiple heads because it allows the model to jointly attend to information at different positions from different representational spaces.

After the split each head hsa reduced dimensionality, so the total computation cost is the same as a single head attention with full dimentionality.

```
class MultiHeadAttention(tf.keras.layers.Layer):

  def __init__(self, d_model, num_heads, name="multi_head_attention"):
    super(MultiHeadAttention, self).__init__(name=name)
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    self.depth = d_model // self.num_heads

    self.query_dense = tf.keras.layers.Dense(units=d_model)
    self.key_dense = tf.keras.layers.Dense(units=d_model)
    self.value_dense = tf.keras.layers.Dense(units=d_model)

    self.dense = tf.keras.layers.Dense(units=d_model)

  def split_heads(self, inputs, batch_size):
    inputs = tf.reshape(
        inputs, shape=(batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(inputs, perm=[0, 2, 1, 3])

  def call(self, inputs):
    query, key, value, mask = inputs['query'], inputs['key'], inputs[
        'value'], inputs['mask']
    batch_size = tf.shape(query)[0]

    # linear layers
    query = self.query_dense(query)
    key = self.key_dense(key)
    value = self.value_dense(value)

    # split heads
```

```
    query = self.split_heads(query, batch_size)
    key = self.split_heads(key, batch_size)
    value = self.split_heads(value, batch_size)


    # scaled dot-product attention
    scaled_attention = scaled_dot_product_attention(query, key, value, mask)

    scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])

    # concatenation of heads
    concat_attention = tf.reshape(scaled_attention,
                                  (batch_size, -1, self.d_model))

    # final linear layer
    outputs = self.dense(concat_attention)

    return outputs
```

Transformer Masking:

create_padding_mask and create_look_ahead are helper functions of creating masks to mask out padded tokens. Those helper functions will be used as tf.keras.layers.Lambda layers.

Mask all the pad tokens (value 0) in the batch to ensure the model does not treat padding as input.

```
def create_padding_mask(x):
    # tf.math.equal returns the truth value of (x == y) element-wise
    mask = tf.cast(tf.math.equal(x, 0), tf.float32)
    # (batch_size, 1, 1, sequence length)
    return mask[:, tf.newaxis, tf.newaxis, :]
```

Look-ahead mask to mask the future tokens in a sequence. We also mask out pad tokens. i.e. To predict the third word, only the first and second word will be used.

```
def create_look_ahead_mask(x):
    seq_len = tf.shape(x)[1]
    look_ahead_mask = 1- tf.linalg.band_part(tf.ones((seq_len, seq_len)), -1, 0)
    padding_mask = create_padding_mask(x)
    # tf.maximum returns the max of x and y (i.e. x>y?x:y) element-wise
    return tf.maximum(look_ahead_mask, padding_mask)
print(create_look_ahead_mask(tf.constant([[1, 2, 0, 4, 5]])))
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)
```

Positional encoding:

- Since this model doesn't contain any recurrence or convolution, positional encoding is
- added to give the model some information about the relative position of the words in the sentence.
- The positional encoding vector is added to the embedding vector. Embedding represent a token in a d-dimensional space where tokens with similar meaning will be closer to each other.
- But the embeddings do not encode the relative position of words in a sentence. So after adding the positional encoding, words will be closer to each other based on the similarity of their meaning and their position in the sentence, in the d-dimensionl space.

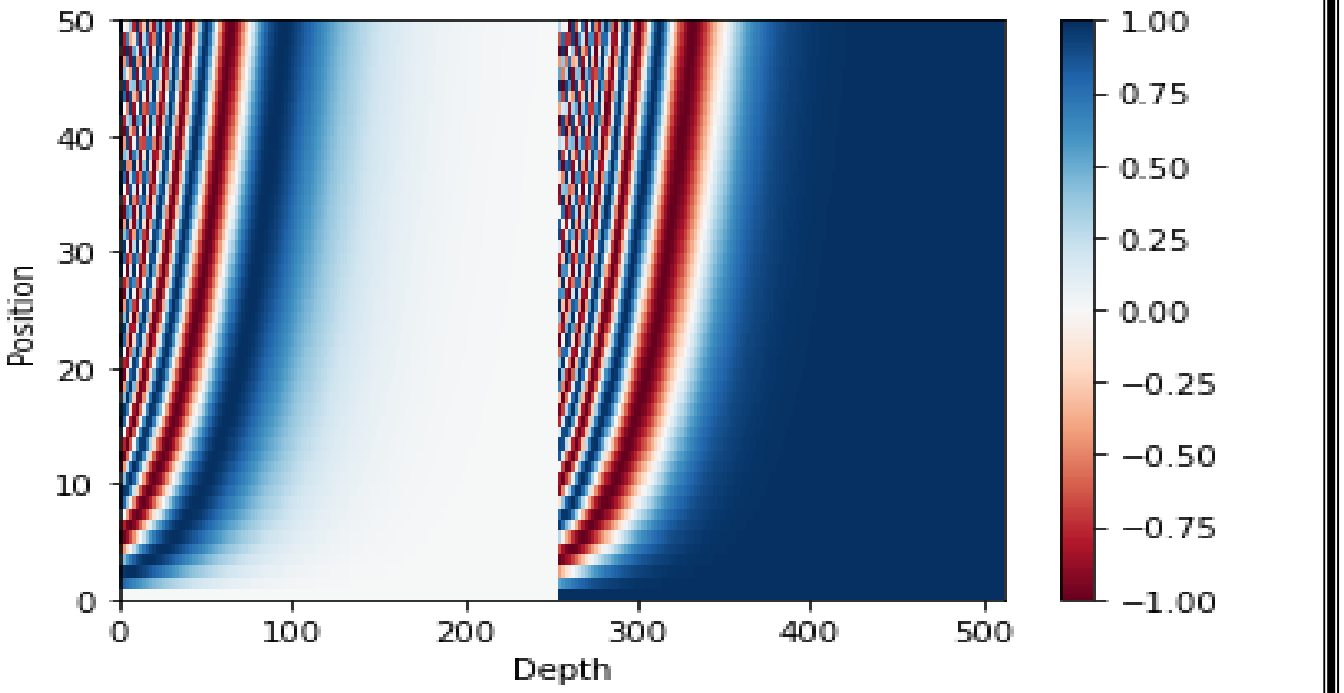
The formula for calculating the positional encoding is as follows:

PE(pos,2i)=sin(pos/100002i/dmodel)

PE(pos,2i+1)=cos(pos/100002i/dmodel)

```
class PositionalEncoding(tf.keras.layers.Layer):

    def___init_(self, position, d_model):
        super(PositionalEncoding, self).__init__()
        self.pos_encoding = self.positional_encoding(position, d_model)

    def get_angles(self, position, i, d_model):
        angles = 1 / tf.pow(10000,(2 * (i // 2)) / tf.cast(d_model, tf.float32))
        return position * angles

    def positional_encoding(self, position, d_model):
        angle_rads = self.get_angles(position = tf.range(position, dtype=tf.float32)[:, tf.newaxis], i=tf.range(d_model, dtype=tf.float32)[tf.newaxis, :], d_model = d_model)
        # apply sin to even index in the array
        sines = tf.math.sin(angle_rads[:, 1::2])
        # apply cos to odd index in the array
        cosines = tf.math.cos(angle_rads[:, 1::2])

        # negative axis refers to axis + rank(values)-th dimension.
        pos_encoding = tf.concat([sines, cosines], axis = -1)
        pos_encoding = pos_encoding[tf.newaxis, ...]
        return tf.cast(pos_encoding, tf.float32)

    def call(self, inputs):
        return inputs + self.pos_encoding[:, :tf.shape(inputs)[1], :]
sample_pos_encoding = PositionalEncoding(50, 512)
```

```
# RdBu means Red-Blue, cmap --> colormap
plt.pcolormesh(sample_pos_encoding.pos_encoding.numpy()[0], cmap = 'RdBu')
plt.xlabel('Depth')
plt.xlim((0, 512))
plt.ylabel('Position')
plt.colorbar()
plt.show()
```



Encoder Layer

Each encoder layer consists of sublayers:

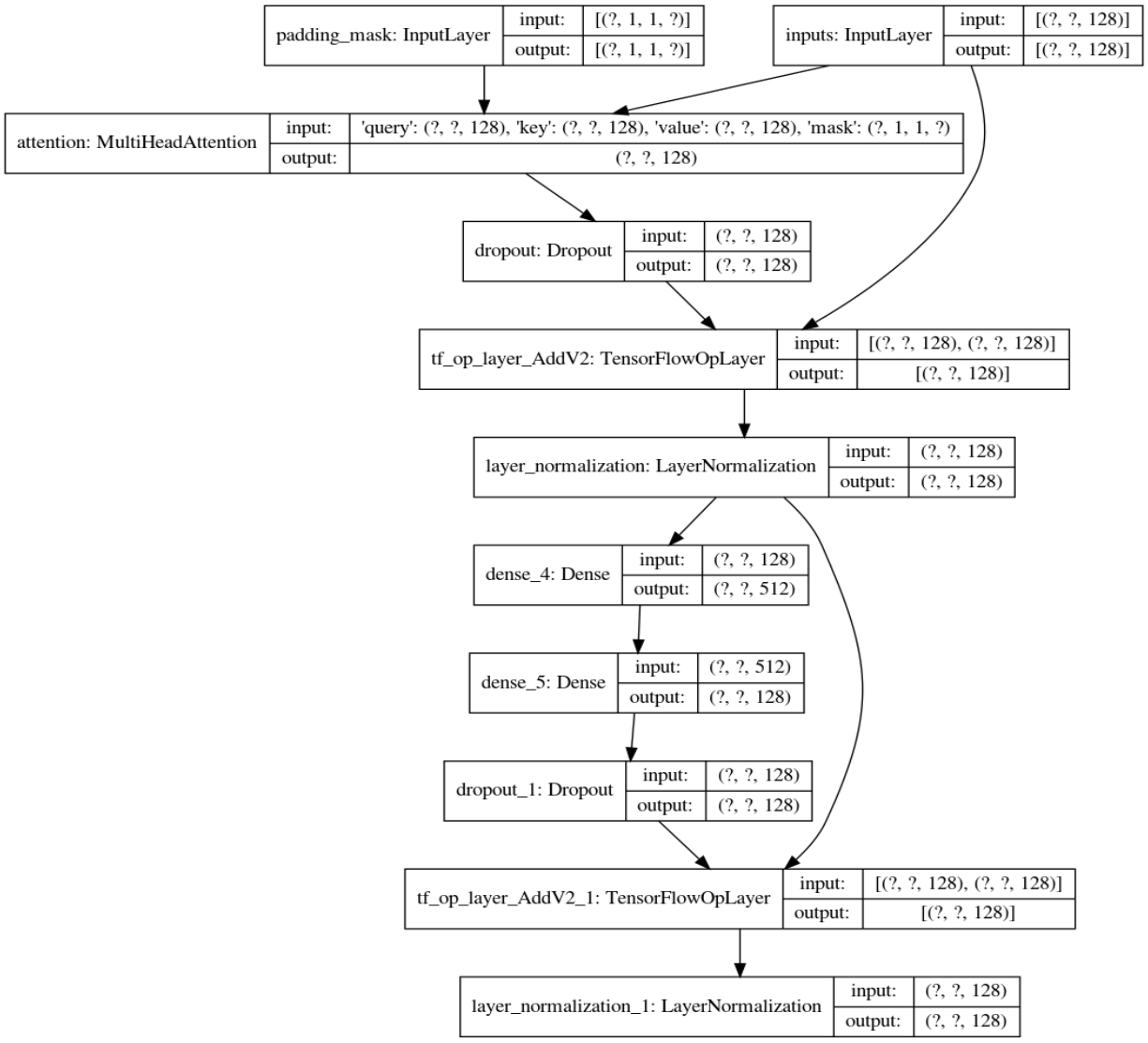
1. Multi-head attention (with padding mask)
2. 2 dense layers followed by dropout

Each of these sublayers ahs a residual connection around it followed by a layer normalization. Residual connections help in avoiding the vanishing gradient problem in deep networks.

The output of each sublayer is LayerNorm(x + Sublayer(x)). The normalization is done on the d_model (last) axis.

```
def encoder_layer(units, d_model, num_heads, dropout, name="encoder_layer"):
  inputs = tf.keras.Input(shape=(None, d_model), name="inputs")
  padding_mask = tf.keras.Input(shape=(1, 1, None), name="padding_mask")

  attention = MultiHeadAttention(
      d_model, num_heads, name="attention")({
          'query': inputs,
          'key': inputs,
          'value': inputs,
          'mask': padding_mask
      })
```

```
attention = tf.keras.layers.Dropout(rate=dropout)(attention)
  attention = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(inputs + attention)
 outputs = tf.keras.layers.Dense(units=units, activation='relu')(attention)
  outputs = tf.keras.layers.Dense(units=d_model)(outputs)
  outputs = tf.keras.layers.Dropout(rate=dropout)(outputs)
  outputs = tf.keras.layers.LayerNormalization(


      epsilon=1e-6)(attention + outputs)
  return tf.keras.Model(
      inputs=[inputs, padding_mask], outputs=outputs, name=name)
sample_encoder_layer = encoder_layer(
                              units = 512,
                              d_model = 128,
                              num_heads = 4,
                              dropout = 0.3,
                              name = "encoder_layer")
tf.keras.utils.plot_model(sample_encoder_layer,      to_file      =      'encoder_layer.png',
show_shapes = True)
```



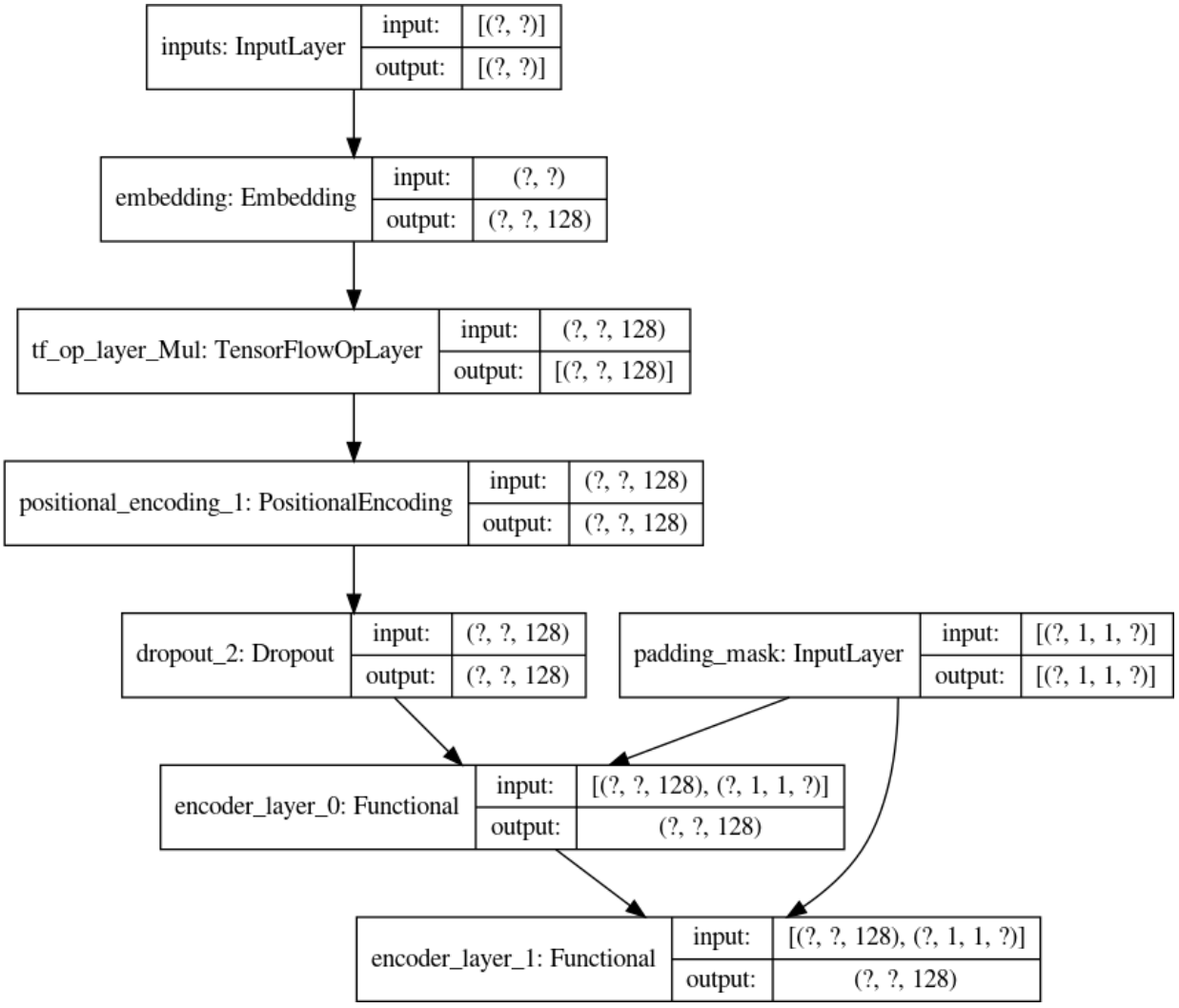
Encoder

The encoder consists of:

Input Embedding
Positional Encoding

num_layers encoder layers
The input is put through an embedding which is summed with the positional encoding. The output of this summation is the input to the encoder layers. The output of the encoder is the input to the decoder.

```
def encoder(vocab_size,
            num_layers,
            units,
            d_model,



            num_heads,
            dropout,
            name="encoder"):
  inputs = tf.keras.Input(shape=(None,), name="inputs")
  padding_mask = tf.keras.Input(shape=(1, 1, None), name="padding_mask")

  embeddings = tf.keras.layers.Embedding(vocab_size, d_model)(inputs)
  embeddings *= tf.math.sqrt(tf.cast(d_model, tf.float32))
  embeddings = PositionalEncoding(vocab_size, d_model)(embeddings)

  outputs = tf.keras.layers.Dropout(rate=dropout)(embeddings)

  for i in range(num_layers):
    outputs = encoder_layer(
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
        name="encoder_layer_{}".format(i),
    )([outputs, padding_mask])


  return tf.keras.Model(
      inputs=[inputs, padding_mask], outputs=outputs, name=name)
sample_encoder = encoder(vocab_size = 8192,
                         num_layers = 2,
                         units = 512,
                         d_model = 128,
                         num_heads = 4,
                         dropout = 0.3,
                         name = "sample_encounter")

tf.keras.utils.plot_model(sample_encoder, to_file='encoder.png', show_shapes = True)
```



Decoder Layer

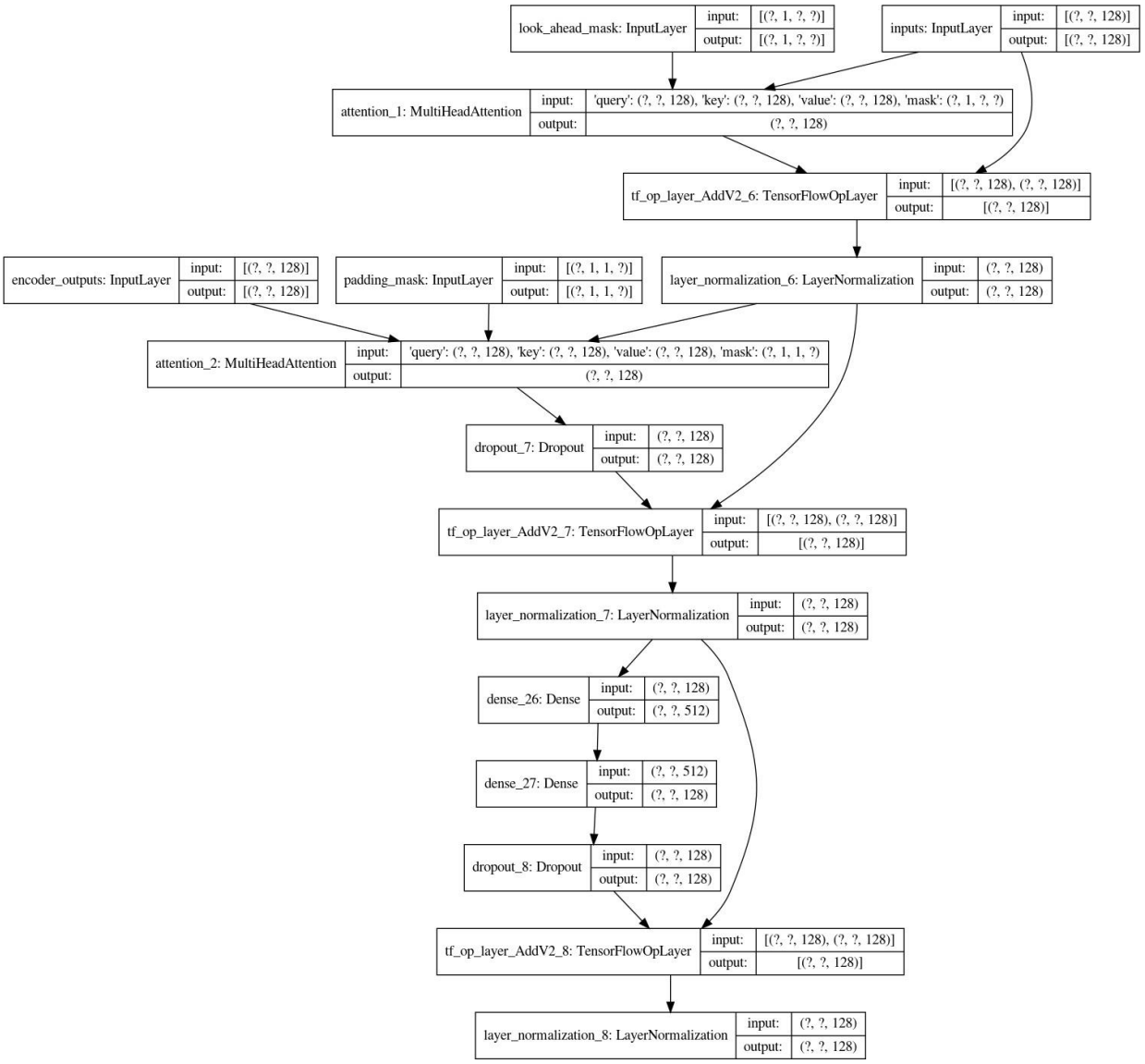
Each decoder layer consists of sublayers:

1. Masked multi-head attention (with look ahead mask and padding mask)
2. Multi-head attention (with padding mask). value and key receive the encoder output as inputs. query receives the output from the masked multi-head attention sublayer.
3. 2 dense layers followed by dropout

Each of these sublayers has a residual connection around it followed by a layer normalization. The output of each sublayer is LayerNorm(x + Sublayer(x)). The normalization is done on the d_model (last) axis.

As query receives the output from decoder's first attention block, and key receives the encoder output, the attention weights represent the important given to the decoder's input based on the encoder's output. In other words, the decoder predicts the next word by looking at the encoder output and self-attending to its own output.

```
def decoder_layer(units, d_model, num_heads, dropout, name = "decoder_layer"):
    inputs = tf.keras.Input(shape=(None, d_model), name = 'inputs')

    enc_outputs = tf.keras.Input(shape=(None, d_model), name="encoder_outputs")
    look_ahead_mask = tf.keras.Input(shape=(1, None, None), name = "look_ahead_mask")
    padding_mask = tf.keras.Input(shape=(1, 1, None), name = "padding_mask")

    attention1 = MultiHeadAttention(d_model, num_heads, name="attention_1")(inputs={'query':inputs,
                                                                                    'key': inputs,
                                                                                    'value': inputs,
                                                                                    'mask':look_ahead_mask})
    attention1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)(attention1 + inputs)

    attention2 = MultiHeadAttention(d_model, num_heads, name = "attention_2")(inputs={'query':attention1,
                                                                                     'key':enc_outputs,
                                                                                     'value':enc_outputs,
                                                                                     'mask':padding_mask})
    attention2 = tf.keras.layers.Dropout(rate=dropout)(attention2)
    attention2 = tf.keras.layers.LayerNormalization(epsilon = 1e-6)(attention2 + attention1)

    outputs = tf.keras.layers.Dense(units=units, activation='relu')(attention2)
    outputs = tf.keras.layers.Dense(units=d_model)(outputs)
    outputs = tf.keras.layers.Dropout(rate=dropout)(outputs)
    outputs = tf.keras.layers.LayerNormalization(epsilon=1e-6)(outputs + attention2)

    return tf.keras.Model(inputs=[inputs, enc_outputs, look_ahead_mask, padding_mask],
                          outputs = outputs,
                          name = name)
sample_decoder_layer = decoder_layer(units = 512,
                                     d_model = 128,
                                     num_heads = 4,
                                     dropout = 0.3,
                                     name = "sample_decoder_layer")
tf.keras.utils.plot_model(sample_decoder_layer, to_file='decoder_layer.png', show_shapes=True)
```



Encoder

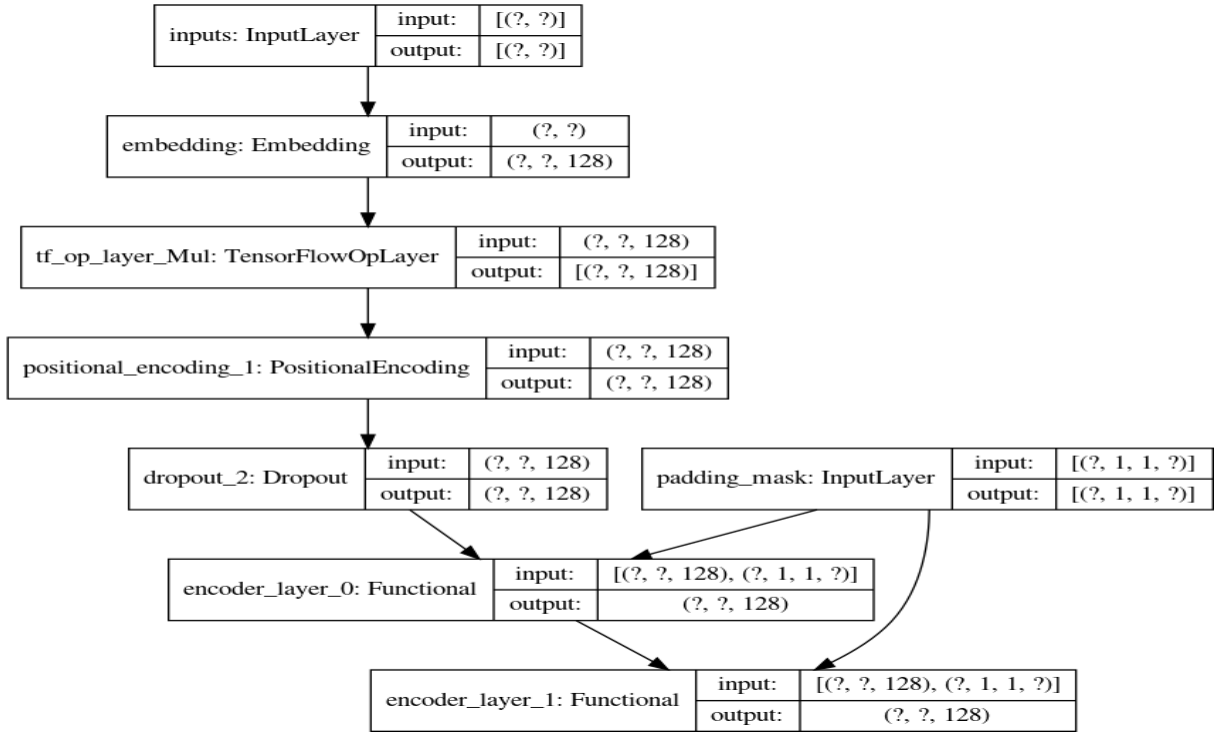
The encoder consists of:

1. Input Embedding
2. Positional Encoding
3. num_layers encoder layers

The input is put through an embedding which is summed with the positional encoding. The output of this summation is the input to the encoder layers. The output of the encoder is the input to the decoder.

```
def encoder(vocab_size,
            num_layers,
            units,
            d_model,
            num_heads,
            dropout,
```

```
        name="encoder"):
  inputs = tf.keras.Input(shape=(None,), name="inputs")
  padding_mask = tf.keras.Input(shape=(1, 1, None), name="padding_mask")

  embeddings = tf.keras.layers.Embedding(vocab_size, d_model)(inputs)
  embeddings *= tf.math.sqrt(tf.cast(d_model, tf.float32))
  embeddings = PositionalEncoding(vocab_size, d_model)(embeddings)

  outputs = tf.keras.layers.Dropout(rate=dropout)(embeddings)

  for i in range(num_layers):
    outputs = encoder_layer(
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
        name="encoder_layer_{}".format(i),
    )([outputs, padding_mask])


  return tf.keras.Model(
      inputs=[inputs, padding_mask], outputs=outputs, name=name)
sample_encoder = encoder(vocab_size = 8192,
                         num_layers = 2,
                         units = 512,
                         d_model = 128,
                         num_heads = 4,
                         dropout = 0.3,
                         name = "sample_encounter")

tf.keras.utils.plot_model(sample_encoder, to_file='encoder.png', show_shapes = True)
```



Conclusion:

- A chatbot for diabetes is a valuable initiative for providing information and support to individuals living with diabetes or seeking information about this medical condition.
- This chatbot offers an accessible, convenient, and responsive way to engage with users in discussions related to diabetes management, types, symptoms, and more.
- By harnessing the power of technology, such chatbots play a crucial role in enhancing healthcare accessibility and promoting greater awareness and understanding of diabetes.
- They bridge the gap between users and vital information, helping to improve the quality of life for those affected by diabetes and their families.