```
import tensorflow as tf
tf.random.set_seed(1234)
AUTO = tf.data.experimental.AUTOTUNE
!pip install tensorflow-datasets==1.2.0
import tensorflow_datasets as tfds
import re
import sys
from time import time
import csv
import numpy as np
import pandas as pd
from datasets import load_dataset
! pip -q install datasets
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
Requirement already satisfied: tensorflow-datasets==1.2.0 in /opt/conda/lib/python3.10/site-packages (1.2.0)
Requirement already satisfied: absl-py in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (1.4.0)
Requirement already satisfied: attrs in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (23.1.0)
Requirement already satisfied: dill in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (0.3.6)
Requirement already satisfied: future in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (0.18.3)
Requirement already satisfied: numpy in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (1.23.5)
Requirement already satisfied: promise in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (2.3)
Requirement already satisfied: protobuf>=3.6.1 in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (3.20.3)
Requirement already satisfied: psutil in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (5.9.3)
Requirement already satisfied: requests>=2.19.0 in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (2.31.0)
Requirement already satisfied: six in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (1.16.0)
Requirement already satisfied: tensorflow-metadata in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (0.14.0)
Requirement already satisfied: termcolor in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (2.3.0)
Requirement already satisfied: tqdm in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (4.65.0)
Requirement already satisfied: wrapt in /opt/conda/lib/python3.10/site-packages (from tensorflow-datasets==1.2.0) (1.14.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /opt/conda/lib/python3.10/site-packages (from requests>=2.19.0->tensorflow-datasets==1.2.0)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.10/site-packages (from requests>=2.19.0->tensorflow-datasets==1.2.0) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/conda/lib/python3.10/site-packages (from requests>=2.19.0->tensorflow-datasets==1.2.0) (1.26
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/lib/python3.10/site-packages (from requests>=2.19.0->tensorflow-datasets==1.2.0) (2023
Requirement already satisfied: googleapis-common-protos in /opt/conda/lib/python3.10/site-packages (from tensorflow-metadata->tensorflow-datasets==1.2
/kaggle/input/exemplary-empathy-2490/emotion_train.csv
/kaggle/input/covid-chitchat/9L_dataset.json
```

Read Data

DailyDialog

```
d = load_dataset("roskoN/dailydialog")

c = d['train']['utterances']
c = [u for sublist in c for u in sublist]
prompts = c[0::2]
responses = c[1::2]

file = "daily_dialog.csv"
with open(file, mode='w', newline='') as cs:
    w = csv.writer(cs)
    w.writerow(["Prompt", "Response"])

    for prompt, response in zip(prompts, responses):
        w.writerow([prompt.strip(), response.strip()])

df = pd.read_csv(file)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

Empathetic Dialogues

```
d = load_dataset("benjaminbeilharz/empathetic_dialogues_for_lm")

c = d['train']['conv']
c = [u for sublist in c for u in sublist]
prompts = c[0::2]
responses = c[1::2]

file = "empathetic_dialogues.csv"
with open(file, mode='w', newline='') as cs:
    w = csv.writer(cs)
    w.writerow(["Prompt", "Response"])

    for prompt, response in zip(prompts, responses):
        w.writerow([prompt.strip(), response.strip()])

dt = pd.read_csv(file)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

ProsocialDialog

```
dat = load_dataset("allenai/prosocial-dialog")

t_data = dat['train'].to_pandas()

column = ['context', 'response']
da = t_data[column]

f = "prosocial_dialog.csv"
da.to_csv(f, index=False)

new_column1 = ['Prompt','Response']
sel_c1 = da[['context','response']]
sel_c1.columns = new_column1
da = sel_c1
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

Concat data

```
#concat_q = pd.concat([dt['Prompt'], d['seeker_post'],d1['question'],df['Prompt']], ignore_index=True)
concat_q = pd.concat([df['Prompt'], dt['Prompt'], da['Prompt']], ignore_index=True)
concat_q.dropna(inplace=True)
prompt = concat_q.tolist()

#concat_a = pd.concat([dt['Response'], d['response_post'], d1['answer'],df['Response']], ignore_index=True)
concat_a = pd.concat([df['Response'], dt['Response'], da['Response']], ignore_index=True)
concat_a.dropna(inplace=True)
response = concat_a.tolist()

print(len(prompt))
print(len(response))
```

```
211079
211079
```

Hyperparameters

```
MAX_LENGTH = 60
MAX_SAMPLES = 220000
BATCH_SIZE = 64
BUFFER_SIZE = 200000
NUM_LAYERS = 2
```

```
D_MODEL = 256
NUM_HEADS = 8
UNITS = 512
DROPOUT = 0.1
EPOCHS = 40


max_len = 60
max_sample = 220000
batch_size = 64
buffer_size = 200000
number_of_layer = 2
d_model = 512
number_of_head = 8
unit = 128
dropout = 0.1
```

Data preprocess

```
def text_preprocess(s):
    s = s.lower().strip()
    s= re.sub(r"([?.!,])", r" \1 ", s)
    s = re.sub(r'[" "]+', " ", s)
    s = re.sub(r"[^a-zA-Z?.!,]+", " ", s)
    s = s.strip()
    return s

prompt = [text_preprocess(s) for s in prompt]
response = [text_preprocess(s) for s in response]
```

Build Prompt and Response

```
tokenizer = tfds.features.text.SubwordTextEncoder.build_from_corpus(prompt + response, target_vocab_size=8000)

s_token, e_token = [tokenizer.vocab_size], [tokenizer.vocab_size + 1]

vocab_size = tokenizer.vocab_size + 2

t_prompt, t_response = [], []

for (i, j) in zip(prompt, response):
    i = s_token + tokenizer.encode(i) + e_token
```

```
        j = s_token + tokenizer.encode(j) + e_token
        if len(i) <= max_len and len(j) <= max_len:
            t_prompt.append(i)
            t_response.append(j)

    prompt = tf.keras.preprocessing.sequence.pad_sequences(t_prompt, maxlen=max_len, padding='post')
    response = tf.keras.preprocessing.sequence.pad_sequences(t_response, maxlen=max_len, padding='post')
```

Create Train and Validation Data

```
data = tf.data.Dataset.from_tensor_slices(({ 'inputs': prompt,'dec_inputs': response[:, :-1] },{'outputs': response[:, 1:]},))
data = data.cache()
data = data.shuffle(buffer_size)
data = data.batch(batch_size)
data = data.prefetch(tf.data.experimental.AUTOTUNE)
dataset_size = len(data)
train_size = int(0.8 * dataset_size)
train_dataset = data.take(train_size)
val_dataset = data.skip(train_size)


def scaled_dot_product_attention(query, key, value, mask):

    matmul_qk = tf.matmul(query, key, transpose_b=True)

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

    if mask is not None:
        logits += mask * -1e9

    attention_weights = tf.nn.softmax(logits, axis=-1)

    output = tf.matmul(attention_weights, value)

    return output
```

Multi Head Attention

```
class MultiHeadAttentionLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, **kwargs):
        assert d_model % num_heads == 0
```

```
        super(MultiHeadAttentionLayer, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.d_model = d_model

        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 get_config(self):
        config = super(MultiHeadAttentionLayer, self).get_config()
        config.update({"num_heads": self.num_heads,"d_model": self.d_model,})
        return config

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

    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.keras.layers.Lambda(lambda scaled_attention: tf.transpose(scaled_attention, perm=[0, 2, 1, 3]))(scaled_attention)

        # concatenation of heads
        concat_attention = tf.keras.layers.Lambda(
            lambda scaled_attention: tf.reshape(
                scaled_attention, (batch_size, -1, self.d_model)
```

```
            )
        )(scaled_attention)

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

        return outputs


def create_padding_mask(x):
    mask = tf.cast(tf.math.equal(x, 0), tf.float32)
    # (batch_size, 1, 1, sequence length)
    return mask[:, tf.newaxis, tf.newaxis, :]


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)
    return tf.maximum(look_ahead_mask, padding_mask)
```

Positional Encoding

```
class PositionalEncoding(tf.keras.layers.Layer):
    def __init__(self, position, d_model, **kwargs):
        super(PositionalEncoding, self).__init__(**kwargs)
        self.position = position
        self.d_model = d_model
        self.pos_encoding = self.positional_encoding(position, d_model)

    def get_config(self):
        config = super(PositionalEncoding, self).get_config()
        config.update(
            {
                "position": self.position,
                "d_model": self.d_model,
            }
        )
        return config

    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[:, 0::2])
        # apply cos to odd index in the array
        cosines = tf.math.cos(angle_rads[:, 1::2])

        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], :]
```

Encoder Blocks

```
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 = MultiHeadAttentionLayer(d_model, num_heads, name="attention")(
        {"query": inputs, "key": inputs, "value": inputs, "mask": padding_mask}
    )
    attention = tf.keras.layers.Dropout(rate=dropout)(attention)
    add_attention = tf.keras.layers.add([inputs, attention])
    attention = tf.keras.layers.LayerNormalization(epsilon=1e-6)(add_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)
    add_attention = tf.keras.layers.add([attention, outputs])
    outputs = tf.keras.layers.LayerNormalization(epsilon=1e-6)(add_attention)

    return tf.keras.Model(inputs=[inputs, padding_mask], outputs=outputs, name=name)


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.keras.layers.Lambda(
```

```
        lambda d_model: tf.math.sqrt(tf.cast(d_model, tf.float32))
    )(d_model)
    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)
```

Decoder Blocks

```
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 = MultiHeadAttentionLayer(d_model, num_heads, name="attention_1")(
        inputs={
            "query": inputs,
            "key": inputs,
            "value": inputs,
            "mask": look_ahead_mask,
        }
    )
    add_attention = tf.keras.layers.add([attention1, inputs])
    attention1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)(add_attention)

    attention2 = MultiHeadAttentionLayer(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)
```

```
        add_attention = tf.keras.layers.add([attention2, attention1])
        attention2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)(add_attention)

        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)
        add_attention = tf.keras.layers.add([outputs, attention2])
        outputs = tf.keras.layers.LayerNormalization(epsilon=1e-6)(add_attention)

        return tf.keras.Model(
            inputs=[inputs, enc_outputs, look_ahead_mask, padding_mask],
            outputs=outputs,
            name=name,
        )


    def decoder(vocab_size, num_layers, units, d_model, num_heads, dropout, name="decoder"):
        inputs = tf.keras.Input(shape=(None,), 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")

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

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

        for i in range(num_layers):
            outputs = decoder_layer(
                units=units,
                d_model=d_model,
                num_heads=num_heads,
                dropout=dropout,
                name="decoder_layer_{}".format(i),
            )(inputs=[outputs, enc_outputs, look_ahead_mask, padding_mask])

        return tf.keras.Model(
            inputs=[inputs, enc_outputs, look_ahead_mask, padding_mask],
            outputs=outputs,
            name=name,
        )
```

Transformer

```
def transformer(vocab_size, num_layers, units, d_model, num_heads, dropout, name="transformer"):
    inputs = tf.keras.Input(shape=(None,), name="inputs")
    dec_inputs = tf.keras.Input(shape=(None,), name="dec_inputs")

    enc_padding_mask = tf.keras.layers.Lambda(
        create_padding_mask, output_shape=(1, 1, None), name="enc_padding_mask"
    )(inputs)
    # mask the future tokens for decoder inputs at the 1st attention block
    look_ahead_mask = tf.keras.layers.Lambda(
        create_look_ahead_mask, output_shape=(1, None, None), name="look_ahead_mask"
    )(dec_inputs)
    # mask the encoder outputs for the 2nd attention block
    dec_padding_mask = tf.keras.layers.Lambda(
        create_padding_mask, output_shape=(1, 1, None), name="dec_padding_mask"
    )(inputs)

    enc_outputs = encoder(
        vocab_size=vocab_size,
        num_layers=num_layers,
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
    )(inputs=[inputs, enc_padding_mask])

    dec_outputs = decoder(
        vocab_size=vocab_size,
        num_layers=num_layers,
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
    )(inputs=[dec_inputs, enc_outputs, look_ahead_mask, dec_padding_mask])

    outputs = tf.keras.layers.Dense(units=vocab_size, name="outputs")(dec_outputs)

    return tf.keras.Model(inputs=[inputs, dec_inputs], outputs=outputs, name=name)
```

Optimizer and Loss

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='accuracy')
```

Train

```
# initialize and compile model within strategy scope
model = transformer(
        vocab_size=vocab_size,
        num_layers=NUM_LAYERS,
        units=UNITS,
        d_model=D_MODEL,
        num_heads=NUM_HEADS,
        dropout=DROPOUT,)

model.compile(optimizer=optimizer, loss=loss, metrics=[accuracy])

model.summary()
```

```
Model: "transformer"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 inputs (InputLayer)            [(None, None)]       0           []

 dec_inputs (InputLayer)        [(None, None)]       0           []

 enc_padding_mask (Lambda)      (None, 1, 1, None)   0           ['inputs[0][0]']

 encoder (Functional)           (None, None, 256)    3116544     ['inputs[0][0]',
                                                                  'enc_padding_mask[0][0]']

 look_ahead_mask (Lambda)       (None, 1, None, Non  0           ['dec_inputs[0][0]']
                                e)

 dec_padding_mask (Lambda)      (None, 1, 1, None)   0           ['inputs[0][0]']

 decoder (Functional)           (None, None, 256)    3643904     ['dec_inputs[0][0]',
                                                                  'encoder[0][0]',
                                                                  'look_ahead_mask[0][0]',
                                                                  'dec_padding_mask[0][0]']

 outputs (Dense)                (None, None, 8056)   2070392     ['decoder[0][0]']

==================================================================================================
Total params: 8,830,840
Trainable params: 8,830,840
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
#tf.keras.backend.clear_session()
#model.compile(optimizer=optimizer, loss=[loss], metrics=[accuracy])


"""

from tensorflow.keras.callbacks import ModelCheckpoint

checkpoint_callback = ModelCheckpoint(
    filepath='/kaggle/working/model_save.h5',  # Path to save the checkpoint file
    save_best_only=True,             # Save only the best model
    save_weights_only=True,          # Save only the model weights
    monitor='val_loss',              # Monitor validation loss
    verbose=1                         # Show progress
)

model.fit(train_dataset, epochs=2, validation_data = val_dataset, callbacks=[checkpoint_callback])
#loaded_model = tf.keras.models.load_model('/kaggle/working/model_save.h5')
"""
```

```
"\n\nfrom tensorflow.keras.callbacks import ModelCheckpoint\n\ncheckpoint_callback = ModelCheckpoint(\n    filepath='/kaggle/working/model_save.h5',
# Path to save the checkpoint file\n    save_best_only=True,             # Save only the best model\n    save_weights_only=True,          # Save only
the model weights\n    monitor='val_loss',              # Monitor validation loss\n    verbose=1                         # Show
progress\n)\n\nmodel.fit(train_dataset, epochs=2, validation_data = val_dataset, callbacks=[checkpoint_callback])\n#loaded_model =
tf.keras.models.load_model('/kaggle/working/model_save.h5')\n"
```

```
model.fit(train_dataset, epochs=10, validation_data = val_dataset)
```

```
Epoch 1/10
2504/2504 [==============================] - 322s 119ms/step - loss: 2.0658 - accuracy: 0.6888 - val_loss: 1.6273 - val_accuracy: 0.7181
Epoch 2/10
2504/2504 [==============================] - 236s 94ms/step - loss: 1.5818 - accuracy: 0.7209 - val_loss: 1.4989 - val_accuracy: 0.7284
Epoch 3/10
2504/2504 [==============================] - 230s 92ms/step - loss: 1.4865 - accuracy: 0.7288 - val_loss: 1.4176 - val_accuracy: 0.7357
Epoch 4/10
2504/2504 [==============================] - 229s 91ms/step - loss: 1.4263 - accuracy: 0.7336 - val_loss: 1.3632 - val_accuracy: 0.7403
Epoch 5/10
2504/2504 [==============================] - 228s 91ms/step - loss: 1.3806 - accuracy: 0.7376 - val_loss: 1.3166 - val_accuracy: 0.7456
Epoch 6/10
2504/2504 [==============================] - 227s 91ms/step - loss: 1.3473 - accuracy: 0.7404 - val_loss: 1.2931 - val_accuracy: 0.7466
Epoch 7/10
2504/2504 [==============================] - 228s 91ms/step - loss: 1.3149 - accuracy: 0.7437 - val_loss: 1.2595 - val_accuracy: 0.7500
Epoch 8/10
2504/2504 [==============================] - 227s 91ms/step - loss: 1.2909 - accuracy: 0.7460 - val_loss: 1.2307 - val_accuracy: 0.7533
Epoch 9/10
2504/2504 [==============================] - 227s 90ms/step - loss: 1.2684 - accuracy: 0.7481 - val_loss: 1.2075 - val_accuracy: 0.7559
Epoch 10/10
2504/2504 [==============================] - 227s 91ms/step - loss: 1.2461 - accuracy: 0.7506 - val_loss: 1.1881 - val_accuracy: 0.7580
<keras.callbacks.History at 0x7a3e2657fb20>
```

```
filename = "model1.h5"
tf.keras.models.save_model(model, filepath=filename, include_optimizer=False)


del model
tf.keras.backend.clear_session()


model = tf.keras.models.load_model(
    filename,
    custom_objects={
        "MultiHeadAttentionLayer": MultiHeadAttentionLayer,
        "PositionalEncoding": PositionalEncoding,
    },
    compile=False,
)
```

Perplexity

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True, reduction='none')
total_loss = 0.0
num_batches = 0
for inputs, targets_dict in val_dataset:
    targets = targets_dict['outputs']
    predictions = model(inputs, training=False)
    batch_loss = loss_object(targets, predictions)
    average_batch_loss = tf.reduce_mean(batch_loss)
    total_loss += average_batch_loss
    num_batches += 1
average_loss = total_loss / num_batches
perplexity = tf.exp(average_loss)
a = perplexity.numpy()
print(f"Perplexity: {a}")
```

```
Perplexity: 3.266697645187378
```

Inference

```
while True:
    a = input("\nInput: ")
    if a == "exit":
        break
    s = text_preprocess(a)
```

```
    s = tf.expand_dims(s_token + tokenizer.encode(s) + e_token, axis=0)
    output = tf.expand_dims(s_token, 0)
    for i in range(max_len):
        predictions = model(inputs=[s, output], training=False)
        predictions = predictions[:, -1:, :]
        predicted_id = tf.cast(tf.argmax(predictions, axis=-1), tf.int32)
        if tf.equal(predicted_id, e_token[0]):
            break
        output = tf.concat([output, predicted_id], axis=-1)

    p = tf.squeeze(output, axis=0)
    pre_prompt = tokenizer.decode([i for i in p if i < tokenizer.vocab_size])
    print('Output: {}'.format(pre_prompt))



 Input:  last few days i feel lonely but i don't know why
 Output: what makes you feel like you re not happy ?

 Input:  i always feel that if my family stay with me
 Output: i m sorry you feel that way . it s good to be honest with your family .

 Input:  tomorrow i have a cricket match and i love cricket more than anything else
 Output: why do you think you re a good person ?

 Input:  can you tell me where i will find my happiness
 Output: i think you should tell me why you want to tell me about it .

 Input:  I'm struggling, to be honest. It's been a really tough week for me.
 Output: it s good to be honest with your friends . you should be honest with your partner about how you feel and not be able to handle it .

 Input:  exit
```