```
import tensorflow as tf  # Deep learning framework
from tensorflow.keras.layers import Embedding, Dense, LayerNormalization, MultiHeadAttention, Dropout # Transformer layers
from tensorflow.keras.models import Model # Base class for model definition
import numpy as np # For numerical operations


# Load a small sample of Shakespeare text (first 20,000 characters)
text = tf.keras.utils.get_file('shakespeare.txt', 'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')
text = open(text, 'rb').read().decode('utf-8')[:20000]  # Slightly more text for better context
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt
1115394/1115394 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
# Tokenizer to convert text to sequences of integers
tokenizer = tf.keras.preprocessing.text.Tokenizer(oov_token="<OOV>") # Handles out-of-vocab tokens
tokenizer.fit_on_texts([text])  # Learn word index from text
seq = tokenizer.texts_to_sequences([text])[0]   # Convert full text to sequence of word indices


# Create input-output pairs (sequence to next word)
seq_len = 10 # Sequence length
input_seqs = [seq[i:i+seq_len] for i in range(len(seq)-seq_len)] # Input sequences
targets = [seq[i+seq_len] for i in range(len(seq)-seq_len)] # Next word for each sequence


# Sample subset to avoid RAM crash
input_seqs, targets = input_seqs[:4000], targets[:4000]
X = tf.convert_to_tensor(input_seqs)
y = tf.convert_to_tensor(targets)
dataset = tf.data.Dataset.from_tensor_slices((X, y)).shuffle(4000).batch(32)


# Positional encoding (standard Transformer)
def positional_encoding(length, depth):
    pos = np.arange(length)[:, None] # Position indices
    i = np.arange(depth)[None, :] # Dimension indices
    angle = pos / np.power(10000, (2 * (i//2)) / depth) # Angle formula
    return tf.cast(np.concatenate([np.sin(angle[:, 0::2]), np.cos(angle[:, 1::2])], axis=-1), tf.float32)


# Transformer block
class TransformerBlock(tf.keras.layers.Layer):
    def __init__(self, dim, heads, ff_dim, drop=0.1):
        super().__init__()
        self.att = MultiHeadAttention(num_heads=heads, key_dim=dim) # Self-attention
        self.ff = tf.keras.Sequential([Dense(ff_dim, activation="relu"), Dense(dim)])  # Feed-forward network
        self.ln1, self.ln2 = LayerNormalization(), LayerNormalization()  # Layer norms
        self.d1, self.d2 = Dropout(drop), Dropout(drop) # Dropout layers


    def call(self, x, training):
        x1 = self.ln1(x + self.d1(self.att(x, x), training=training))  # Residual + norm after attention
        return self.ln2(x1 + self.d2(self.ff(x1), training=training))   # Residual + norm after feedforward



# GPT-like model
class MiniGPT(Model):
    def __init__(self, vocab, maxlen, dim, heads, ff):
        super().__init__()
        self.emb = Embedding(vocab, dim)  # Token embedding
        self.pos = tf.expand_dims(positional_encoding(maxlen, dim), 0)  # Positional embedding
        self.block = TransformerBlock(dim, heads, ff)  # Transformer block
        self.out = Dense(vocab)  # Final classification layer

    def call(self, x, training=False):
        x = self.emb(x) + self.pos[:, :tf.shape(x)[1], :]  # Add embeddings
        x = self.block(x, training=training)  # Transformer processing
        return self.out(x)[:, -1, :]  # Output only last token prediction


# Build and train model
vocab = len(tokenizer.word_index) + 1 # Vocabulary size
model = MiniGPT(vocab, seq_len, 128, 4, 256) # Create model instance
model.compile(optimizer="adam", loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))  # Compile
model.fit(dataset, epochs=10) # Train
```

```
Epoch 1/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 18s 99ms/step - loss: 6.5934
Epoch 2/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 22s 112ms/step - loss: 6.1307
Epoch 3/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 10s 90ms/step - loss: 6.1036
Epoch 4/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 10s 86ms/step - loss: 6.0056
Epoch 5/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 10s 86ms/step - loss: 5.8760
Epoch 6/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 8s 74ms/step - loss: 5.6708
Epoch 7/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 10s 93ms/step - loss: 5.6507
Epoch 8/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 19s 82ms/step - loss: 5.5148
Epoch 9/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 8s 73ms/step - loss: 5.3981
Epoch 10/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 11s 79ms/step - loss: 5.1791
<keras.src.callbacks.history.History at 0x7d6e0985cc50>
```

```
# Function to generate text from a prompt
def generate_text(seed, steps=20, temperature=1.0):
    result = seed  # Start with seed
    for _ in range(steps):
        tokens = tokenizer.texts_to_sequences([result])[0][-seq_len:]  # Get last tokens
        pad = tf.keras.preprocessing.sequence.pad_sequences([tokens], maxlen=seq_len)   # Pad input
        logits = model(pad, training=False)[0] / temperature  # Predict logits, adjust with temperature
        probs = tf.nn.softmax(logits).numpy()  # Convert logits to probabilities
        next_id = np.random.choice(len(probs), p=probs)  # Sample from probabilities
```

```
        word = tokenizer.index_word.get(next_id, '')    # Convert ID to word
        result += ' ' + word  # Append word to result
    return result


# Example output
print(generate_text("To be or not", 20))
```

```
To be or not the thing any quarry and daughter famously the taunts to my helps she my bed nay inkling takes belly my
```