作业四——自然语言处理

搭建 Transformer 编码器完成文本语配任务

1.数据集构建，包括:利用词表将句子中的每个中文字符转换成 id、对不在词汇表里面的字做出适当处理、在输入中加入句子的分隔符号、在起始位置加入占位符、小批量数据的组装及对齐。构建完成后打印一条 mini-batch 的数据进行验证。

加载词表：

```
# 加载词表
def load_vocab():
    word_dict = {}
    with open('/kaggle/input/afqmc-datset-zip1/afqmc-datset/vocab.txt') as f:
        for idx, item in enumerate(f.readlines()):
            word_dict[item.strip()] = idx

    return word_dict
```

从 afqmc 数据集中加载数据：

```
# 加载数据
def load_dataset(data_path, is_test):
    examples = []
    with open(data_path) as f:
        for line in f.readlines():
            line = json.loads(line)
            text_a = line["sentence1"]
            text_b = line["sentence2"]
            if is_test:
                examples.append((text_a, text_b,))
            else:
                label = line["label"]
                examples.append((text_a, text_b, label))
    return examples

def load_afqmc_data(path):
    train_path = os.path.join(path, 'train.json')
    dev_path = os.path.join(path, 'dev.json')
    test_path = os.path.join(path, 'test.json')

    train_data = load_dataset(train_path, False)
    dev_data = load_dataset(dev_path, False)
    test_data = load_dataset(test_path, True)
    return train_data, dev_data, test_data
```

字符转 id：

```
# 字符转id
def words2id(example, word_dict):
    cls_id = word_dict['[CLS]']
    sep_id = word_dict['[SEP]']

    text_a, text_b, label = example

    # 将中文字符切分成单个字符
    text_a = list(text_a)
    text_b = list(text_b)

    input_a = [word_dict[item] if item in word_dict else word_dict['[UNK]'] for item in text_a]
    input_b = [word_dict[item] if item in word_dict else word_dict['[UNK]'] for item in text_b]
    input_ids = [cls_id] + input_a + [sep_id] + input_b + [sep_id]
    segment_id = [0] * (len(input_a) + 2) + [1] * (len(input_b) + 1)
    return input_ids, segment_id, int(label)
```

加载 vocab.txt：

```
# 使用例子
vocab = load_vocab()
train_data, dev_data, test_data = load_afqmc_data('/kaggle/input/afqmc-datset-zip1/afqmc-datset/AFQMC/')
```

将句子转换成 id：

```
# 将句子转换成id
example = train_data[0]
input_ids, segment_ids, label = words2id(example, vocab)
print("Input IDs:", input_ids)
print("Segment IDs:", segment_ids)
print("Label:", label)
```

构建 mini-batch 并进行对齐：

```
# 构建mini-batch并进行对齐
batch_data = [words2id(example, vocab) for example in train_data[:2]]
batch_input, batch_label = collate_fn(batch_data)
print("Batch Input:", batch_input)
print("Batch Label:", batch_label)
```

打印结果：

```
Input IDs: [1, 3802, 2975, 1051, 4947, 43, 852, 201, 699, 48, 22, 806, 33, 254, 399, 49, 89, 1114, 2, 1051, 4947, 9, 254, 399, 45, 195, 2
2]
Segment IDs: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Label: 0
Batch Input: (tensor([[   1, 3802, 2975, 1051, 4947,   43,  852,  201,  699,   48,   22,  806,
          33,  254,  399,   49,   89, 1114,    2, 1051, 4947,    9,  254,  399,
          45,  195,  201,   89, 1114,    2],
        [   1, 3802, 2975,  283, 4947,  178,   75, 1147,  450,    7,  218,    2,
         3802, 2975,  283, 4947, 1147,  450,   40,   13,   10,  614,  356,    2,
            0,    0,    0,    0,    0,    0]]), tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         0, 0, 0, 0, 0, 0]]))
```

2.实现输入编码、分段编码和位置编码，并组装为嵌入层，打印该层的输入输出。

编写组装嵌入层：

```python
class TransformerEmbedding(Layer):
    def __init__(self, vocab_size, emb_size, max_position_embeddings=512):
        super(TransformerEmbedding, self).__init__()

        self.word_embedding = Embedding(vocab_size, emb_size)
        self.position_embedding = Embedding(max_position_embeddings, emb_size)
        self.segment_embedding = Embedding(2, emb_size)  # 0 for sentence A, 1 for sentence B

    def call(self, inputs):
        input_ids, segment_ids = inputs

        # 输入编码
        word_embeddings = self.word_embedding(input_ids)

        # 分段编码
        segment_embeddings = self.segment_embedding(segment_ids)

        # 位置编码
        position_ids = tf.range(tf.shape(input_ids)[1], dtype=tf.int32)
        position_embeddings = self.position_embedding(position_ids)

        # 将各个编码相加得到最终的嵌入表示
        embeddings = word_embeddings + position_embeddings + segment_embeddings

        return embeddings
```

创建 TransformerEmbedding 实例：

```python
# 例子
vocab_size = 10000  # 假设词汇表大小为10000
emb_size = 300  # 假设词向量维度为300

# 创建TransformerEmbedding实例
embedding_layer = TransformerEmbedding(vocab_size, emb_size)
```

构造一个 mini-batch 的输入数据：

```
# 构造一个mini-batch的输入数据
input_ids = tf.constant([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  # 示例输入
segment_ids = tf.constant([[0, 0, 0, 1, 1], [0, 1, 1, 0, 1]])  # 示例分段标记
```

使用嵌入层进行编码并打印输入和输出：

```
# 使用嵌入层进行编码
embeddings = embedding_layer([input_ids, segment_ids])

# 打印输入和输出
print("Input IDs:", input_ids.numpy())
print("Segment IDs:", segment_ids.numpy())
print("Output Embeddings:", embeddings.numpy())
```

打印结果：

```
Input IDs: [[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
Segment IDs: [[0 0 0 1 1]
 [0 1 1 0 1]]
Output Embeddings: [[[ 2.00917050e-02 -1.27615146e-02  6.05499372e-02 ... -1.37562007e-02
   -4.57705930e-04 -3.66860367e-02]
  [ 1.12702504e-01  7.27216825e-02  4.87567782e-02 ... -4.97104637e-02
    8.45743343e-03 -5.25941700e-03]
  [-1.36149004e-02 -2.15323567e-02  3.76145244e-02 ... -9.54090804e-02
    9.79416072e-05 -2.44886056e-02]
  [-3.66672091e-02 -6.85585849e-03  4.82129678e-03 ... -9.55194458e-02
    4.85824496e-02 -2.76071578e-03]
  [ 2.95716189e-02  4.91918921e-02  8.60066563e-02 ...  3.80598418e-02
    4.74755839e-03  9.56726223e-02]]

 [[ 2.21020374e-02 -4.26160581e-02  6.10963479e-02 ... -5.01122251e-02
   -1.22538842e-02 -5.77039234e-02]
  [ 2.64224522e-02  5.58498129e-03  6.20494708e-02 ... -6.67733401e-02
    6.23079911e-02  1.41042084e-01]
  [-7.47188330e-02 -6.19168878e-02  3.66553627e-02 ... -1.39921904e-05
    9.13840905e-02 -5.95682487e-03]
  [-9.16514546e-03  4.01876792e-02  1.01122819e-02 ... -3.59370112e-02
   -1.42647512e-02 -2.62821764e-02]
  [ 2.42441557e-02  2.98968926e-02  4.61011603e-02 ...  5.96805364e-02
    2.09049731e-02  2.24103369e-02]]]
```

3.实现多头自注意力层和 add&norm 层。

编写多头自注意力层：

```
class MultiHeadAttention(Layer):
    def __init__(self, emb_size, num_heads, dropout=0.1):
        super(MultiHeadAttention, self).__init__()

        self.emb_size = emb_size
        self.num_heads = num_heads
        self.head_dim = emb_size // num_heads

        assert self.head_dim * num_heads == emb_size, "Embedding size needs to be divisible by heads"

        self.query_linear = Dense(emb_size)
        self.key_linear = Dense(emb_size)
        self.value_linear = Dense(emb_size)

        self.fc_out = Dense(emb_size)

        self.dropout = Dropout(dropout)
```

构造线性变换、多头分组、形状重组、掩码处理，以及 QKV 注意力计算、重组恢复与多头融合：

```
    def call(self, inputs):
        query, key, value, mask = inputs
        batch_size = tf.shape(query)[0]
        seq_len = tf.shape(query)[1]

        # 线性变换
        Q = self.query_linear(query)
        K = self.key_linear(key)
        V = self.value_linear(value)

        # 多头分组
        Q = tf.transpose(tf.reshape(Q, (batch_size, seq_len, self.num_heads, self.head_dim)), perm=[0, 2, 1, 3])
        K = tf.transpose(tf.reshape(K, (batch_size, seq_len, self.num_heads, self.head_dim)), perm=[0, 2, 1, 3])
        V = tf.transpose(tf.reshape(V, (batch_size, seq_len, self.num_heads, self.head_dim)), perm=[0, 2, 1, 3])

        # 形状重组
        energy = tf.matmul(Q, tf.transpose(K, perm=[0, 1, 3, 2])) / tf.math.sqrt(tf.cast(self.head_dim, dtype=tf.float32))

        # 掩码处理
        if mask is not None:
            mask = tf.expand_dims(tf.expand_dims(mask, axis=1), axis=2)  # Broadcasting the mask
            energy = tf.where(mask == 0, tf.constant(float('-1e20'), dtype=tf.float32), energy)

        # QKV注意力计算
        attention = tf.nn.softmax(energy, axis=-1)
        x = tf.matmul(self.dropout(attention), V)

        # 重组恢复
        x = tf.transpose(x, perm=[0, 2, 1, 3])
        x = tf.reshape(x, (batch_size, seq_len, self.emb_size))

        # 多头融合
        x = self.fc_out(x)

        return x
```

构造 AddNorm 层：

```
class AddNorm(Layer):
    def __init__(self, emb_size, mlp_units, dropout=0.1):
        super(AddNorm, self).__init__()

        self.norm = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.dropout_layer = Dropout(dropout)

        # 添加线性层，将 MLP 输出维度转换为 emb_size
        self.mlp_linear = Dense(emb_size)

    def call(self, inputs, training=None):
        x, sublayer, mask = inputs

        # 残差连接
        residual = x

        # 确保 sublayer 是一个可以调用的层对象
        sublayer_output = sublayer([x, x, x, mask]) if callable(sublayer) else sublayer

        # 更新这行代码，确保在训练时使用 dropout
        mlp_output = self.mlp_linear(sublayer_output)
        x = residual + self.dropout_layer(mlp_output, training=training)

        # 层规范化
        x = self.norm(x)
        return x
```

创建 MultiHeadAttention 实例：

```
# 示例
emb_size = 300
num_heads = 6
dropout = 0.1

# 创建MultiHeadAttention实例
attention_layer = MultiHeadAttention(emb_size, num_heads, dropout)
```

创建示例输入：

```
# 创建示例输入
query = tf.random.normal((2, 10, emb_size))
key = tf.random.normal((2, 10, emb_size))
value = tf.random.normal((2, 10, emb_size))
mask = tf.ones((2, 10))
```

打印输入和输出：

```
# 多头自注意力层
output = attention_layer([query, key, value, mask])

# 打印输入和输出
print("Query shape:", query.shape)
print("Output shape:", output.shape)
```

打印结果：

```
Query shape: (2, 10, 300)
Output shape: (2, 10, 300)
```

4.搭建一个 transformer 编码器，利用嵌入层、transformer 编码器和合适的分类器构建完成语义匹配模型，并说明模型组成。

构建 Transformer 模型，包括输入层、输出层、Transformer 编码器、池化层与分类器：

```
# 构建Transformer模型
def build_transformer_model(vocab_size, emb_size, num_heads, num_transformer_blocks, mlp_units, dropout, max_position_embeddings=512):
    # 输入层
    input_ids = tf.keras.layers.Input(shape=(None,), dtype=tf.int32)
    segment_ids = tf.keras.layers.Input(shape=(None,), dtype=tf.int32)

    # 嵌入层
    embeddings = TransformerEmbedding(vocab_size, emb_size, max_position_embeddings)([input_ids, segment_ids])

    # Transformer 编码器
    for _ in range(num_transformer_blocks):
        attention_output = MultiHeadAttention(emb_size, num_heads, dropout)([embeddings, embeddings, embeddings, None])
        # 加和归一化层
        attention_output = AddNorm(emb_size, dropout)([embeddings, attention_output, None])
        # Feed Forward
        mlp_output = Dense(mlp_units, activation="relu")(attention_output)
        # 加和归一化层
        embeddings = AddNorm(emb_size, dropout)([attention_output, mlp_output, None])

    # 池化层
    pooled = GlobalAveragePooling1D()(embeddings)

    # 分类器
    outputs = Dense(1, activation="sigmoid")(pooled)

    # 构建模型
    model = tf.keras.Model(inputs=[input_ids, segment_ids], outputs=outputs)

    return model
```

构建语义匹配模型并打印模型组成：

```
# 构建语义匹配模型
semantic_matching_model = build_transformer_model(vocab_size, emb_size, num_heads=6, num_transformer_blocks=4, mlp_units=512, dropout=0.1)

# 打印模型组成
semantic_matching_model.summary()
```

打印结果：

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, None)]	0	[]
input_2 (InputLayer)	[(None, None)]	0	[]
transformer_embedding_1 (TransformerEmbedding)	(None, None, 300)	3154200	['input_1[0][0]', 'input_2[0][0]']
multi_head_attention_1 (MultiHeadAttention)	(None, None, 300)	361200	['transformer_embedding_1[0][0]', 'transformer_embedding_1[0][0]', 'transformer_embedding_1[0][0]']
add_norm (AddNorm)	(None, None, 300)	90900	['transformer_embedding_1[0][0]', 'multi_head_attention_1[0][0]']
dense_9 (Dense)	(None, None, 512)	154112	['add_norm[0][0]']
add_norm_1 (AddNorm)	(None, None, 300)	154500	['add_norm[0][0]', 'dense_9[0][0]']
multi_head_attention_2 (MultiHeadAttention)	(None, None, 300)	361200	['add_norm_1[0][0]', 'add_norm_1[0][0]', 'add_norm_1[0][0]']
add_norm_2 (AddNorm)	(None, None, 300)	90900	['add_norm_1[0][0]', 'multi_head_attention_2[0][0]']
dense_16 (Dense)	(None, None, 512)	154112	['add_norm_2[0][0]']
add_norm_3 (AddNorm)	(None, None, 300)	154500	['add_norm_2[0][0]', 'dense_16[0][0]']
multi_head_attention_3 (MultiHeadAttention)	(None, None, 300)	361200	['add_norm_3[0][0]', 'add_norm_3[0][0]', 'add_norm_3[0][0]']
add_norm_4 (AddNorm)	(None, None, 300)	90900	['add_norm_3[0][0]', 'multi_head_attention_3[0][0]']
dense_23 (Dense)	(None, None, 512)	154112	['add_norm_4[0][0]']
add_norm_5 (AddNorm)	(None, None, 300)	154500	['add_norm_4[0][0]', 'dense_23[0][0]']
multi_head_attention_4 (MultiHeadAttention)	(None, None, 300)	361200	['add_norm_5[0][0]', 'add_norm_5[0][0]', 'add_norm_5[0][0]']
add_norm_6 (AddNorm)	(None, None, 300)	90900	['add_norm_5[0][0]', 'multi_head_attention_4[0][0]']
dense_30 (Dense)	(None, None, 512)	154112	['add_norm_6[0][0]']
add_norm_7 (AddNorm)	(None, None, 300)	154500	['add_norm_6[0][0]', 'dense_30[0][0]']
global_average_pooling1d (GlobalAveragePooling1D)	(None, 300)	0	['add_norm_7[0][0]']
dense_32 (Dense)	(None, 1)	301	['global_average_pooling1d[0][0]']

Total params: 6197349 (23.64 MB)
Trainable params: 6197349 (23.64 MB)
Non-trainable params: 0 (0.00 Byte)

5.训练模型，在验证集上计算准确率，并保存在验证集上准确率最高的模型使用 tensorboard 等可视化插件，展示训练过程中的精度变化和损失变化。

定义 TensorBoard 回调并编译模型：

```
# 定义TensorBoard回调
tensorboard_callback = TensorBoard(log_dir='./logs', histogram_freq=1)

# 定义模型
semantic_matching_model = build_transformer_model(vocab_size, emb_size, num_heads=6, num_transformer_blocks=4, mlp_units=512, dropout=0.1)

# 编译模型
semantic_matching_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

加载和处理训练集数据，并将其处理成模型输入：

```
# 加载和处理训练集数据
train_data, dev_data, test_data = load_afqmc_data('/kaggle/input/afqmc-datset-zip1/afqmc-datset/AFQMC/')

# 将训练集数据处理成模型输入
train_inputs, train_labels = collate_fn([words2id(example, vocab) for example in train_data])
```

将输入数据转换为 numpy 数组：

```
# 转换为numpy数组
train_inputs = (np.array(train_inputs[0]), np.array(train_inputs[1]))
train_labels = np.array(train_labels)
```

准备验证数据并将其转换为 numpy 数组：

```
# 准备验证数据
dev_inputs, dev_labels = collate_fn([words2id(example, vocab) for example in dev_data])

# 转换为numpy数组
dev_inputs = (np.array(dev_inputs[0]), np.array(dev_inputs[1]))
dev_labels = np.array(dev_labels)
```

定义模型保存数据：

```python
# 定义模型保存路径
model_checkpoint = ModelCheckpoint(
    './logs/best_model.h5',  # 模型保存路径
    save_best_only=True,
    save_weights_only=True,
    monitor='val_accuracy',
    mode='max',
    verbose=1
)
```

训练模型:

```python
# 训练模型
history = semantic_matching_model.fit(
    train_inputs,
    train_labels,
    epochs=10,
    batch_size=32,
    validation_data=(dev_inputs, dev_labels),
    callbacks=[tensorboard_callback, model_checkpoint, early_stopping]
)
```

评估模型:

```python
# 评估模型
eval_loss, eval_accuracy = semantic_matching_model.evaluate(dev_inputs, dev_labels)
print(f"Evaluation Loss: {eval_loss}, Evaluation Accuracy: {eval_accuracy}")
```

训练结果:

```
Epoch 1/10
1073/1073 [==============================] - ETA: 0s - loss: 0.6408 - accuracy: 0.6871
Epoch 1: val_accuracy improved from -inf to 0.68999, saving model to ./logs/best_model.h5
1073/1073 [==============================] - 97s 74ms/step - loss: 0.6408 - accuracy: 0.6871 - val_loss: 0.6313 - val_accuracy: 0.6900
Epoch 2/10
1073/1073 [==============================] - ETA: 0s - loss: 0.6219 - accuracy: 0.6921
Epoch 2: val_accuracy did not improve from 0.68999
1073/1073 [==============================] - 64s 60ms/step - loss: 0.6219 - accuracy: 0.6921 - val_loss: 0.6200 - val_accuracy: 0.6900
Epoch 3/10
1073/1073 [==============================] - ETA: 0s - loss: 0.6205 - accuracy: 0.6921
Epoch 3: val_accuracy did not improve from 0.68999
1073/1073 [==============================] - 64s 60ms/step - loss: 0.6205 - accuracy: 0.6921 - val_loss: 0.6244 - val_accuracy: 0.6900
Epoch 4/10
1073/1073 [==============================] - ETA: 0s - loss: 0.6192 - accuracy: 0.6921
Epoch 4: val_accuracy did not improve from 0.68999
1073/1073 [==============================] - 63s 59ms/step - loss: 0.6192 - accuracy: 0.6921 - val_loss: 0.6269 - val_accuracy: 0.6900
Epoch 4: early stopping
135/135 [==============================] - 3s 20ms/step - loss: 0.6269 - accuracy: 0.6900
Evaluation Loss: 0.6269139051437378, Evaluation Accuracy: 0.689990758895874
```

准确率为 69%

6.加载保存模型，在测试集上随机选取 50 条数据进行语义匹配测试，展示模型的预测结果。

随机选取 50 条样本并获取特殊标记的 id（设置随机种子以保持可重复性）：

```
# 随机选取50条样本
random.seed(42)  # 设置随机种子以保持可重复性
selected_samples = random.sample(test_data, 50)

# 获取特殊标记的id
cls_id = vocab['[CLS]']
sep_id = vocab['[SEP]']
```

加载模型：

```
# 加载模型
model_path = '/kaggle/working/logs/best_model.h5'  # 模型保存路径
loaded_model = build_transformer_model(vocab_size, emb_size, num_heads=6, num_transformer_blocks=4, mlp_units=512, dropout=0.1)
loaded_model.load_weights(model_path)
```

对每条样本进行测试：

```
# 对每条样本进行测试
for example in selected_samples:
    text_a = example[0]
    text_b = example[1]

    # 转换成id的形式
    input_ids_a = [vocab.get(item, vocab['[UNK]']) for item in list(text_a)]
    input_ids_b = [vocab.get(item, vocab['[UNK]']) for item in list(text_b)]

    input_ids = [cls_id] + input_ids_a + [sep_id] + input_ids_b + [sep_id]
    segment_ids = [0] * (len(input_ids_a) + 2) + [1] * (len(input_ids_b) + 1)
```

转换成 Tensor 张量并进行模型预测：

```python
# 转换成Tensor张量
input_ids = tf.convert_to_tensor([input_ids])
segment_ids = tf.convert_to_tensor([segment_ids])
inputs = [input_ids, segment_ids]

# 模型预测
logits = loaded_model.predict(inputs)

# 取概率值最大的索引
label_id = np.argmax(logits, axis=1)[0]
```

打印预测结果：

```python
# 打印预测结果
print(f"文本A: {text_a}")
print(f"文本B: {text_b}")
print(f"预测的label标签: {label_id}")
print("=" * 50)
```

打印结果：

```
1/1 [==============================] - 1s 1s/step
文本A: 花呗叫绑定银行卡是怎么回事
文本B: 蚂蚁花呗老是提示绑定银行卡是什么原因
预测的label标签: 0
==================================================
1/1 [==============================] - 1s 1s/step
文本A: 花呗如何关
文本B: 如何确定我是否关闭花呗成功
预测的label标签: 0
==================================================
1/1 [==============================] - 0s 29ms/step
文本A: 借呗额度怎么越来越少
文本B: 我借呗额度怎么突然降低了？什么情况
预测的label标签: 0
==================================================
1/1 [==============================] - 0s 29ms/step
文本A: 有蚂蚁借呗都开通不了
文本B: 我蚂蚁借呗为何不通过
预测的label标签: 0
==================================================
1/1 [==============================] - 0s 28ms/step
文本A: 为什么花呗我有记录
文本B: 我的花呗记录有疑问
预测的label标签: 0
==================================================
1/1 [==============================] - 0s 28ms/step
文本A: 蚂蚁借呗怎么认证不了
文本B: 我已实名认证但我还不能蚂蚁借呗
预测的label标签: 0
==================================================
1/1 [==============================] - 0s 28ms/step
文本A: 申请的商家收款码。怎么开通花呗信用卡
文本B: 我的店铺怎么开通花呗付款
预测的label标签: 0
==================================================
1/1 [==============================] - 0s 28ms/step
文本A: 蚂蚁借呗分期还款可以更改吗
文本B: 可以查看朋友的借呗还款日吗
预测的label标签: 0
==================================================
1/1 [==============================] - 0s 29ms/step
文本A: 借呗借***个月，利息是多少
文本B: 蚂蚁借呗的一个月的利率
预测的label标签: 0
==================================================
1/1 [==============================] - 0s 28ms/step
文本A: 经开通了花呗功能
文本B: 我能申请开通蚂蚁花呗吗
预测的label标签: 0
```

8. 改变 transformer 的层数再次实验，输出测试集准确率结果，并与之前的结果对比。

将 Transformer 的层数增加为 6 层，并进行训练：

```
# 定义模型
semantic_matching_model = build_transformer_model(vocab_size, emb_size, num_heads=6, num_transformer_blocks=6, mlp_units=512, dropout=0.1)
```

训练结果：

```
Epoch 1/10
1073/1073 [==============================] - ETA: 0s - loss: 0.6381 - accuracy: 0.6871
Epoch 1: val_accuracy improved from -inf to 0.68999, saving model to ./logs/best_model.h5
1073/1073 [==============================] - 132s 101ms/step - loss: 0.6381 - accuracy: 0.6871 - val_loss: 0.6201 - val_accuracy: 0.6900
Epoch 2/10
1073/1073 [==============================] - ETA: 0s - loss: 0.6219 - accuracy: 0.6921
Epoch 2: val_accuracy did not improve from 0.68999
1073/1073 [==============================] - 92s 85ms/step - loss: 0.6219 - accuracy: 0.6921 - val_loss: 0.6194 - val_accuracy: 0.6900
Epoch 3/10
1073/1073 [==============================] - ETA: 0s - loss: 0.6201 - accuracy: 0.6921
Epoch 3: val_accuracy did not improve from 0.68999
1073/1073 [==============================] - 91s 85ms/step - loss: 0.6201 - accuracy: 0.6921 - val_loss: 0.6200 - val_accuracy: 0.6900
Epoch 4/10
1073/1073 [==============================] - ETA: 0s - loss: 0.6192 - accuracy: 0.6921
Epoch 4: val_accuracy did not improve from 0.68999
1073/1073 [==============================] - 91s 85ms/step - loss: 0.6192 - accuracy: 0.6921 - val_loss: 0.6204 - val_accuracy: 0.6900
Epoch 4: early stopping
135/135 [==============================] - 4s 30ms/step - loss: 0.6204 - accuracy: 0.6900
Evaluation Loss: 0.6203815937042236, Evaluation Accuracy: 0.689990758895874
```

准确率为 69%，与之前的结果相差不大，有些许的提升。

9. 寻找方法提升模型精度。

通过：

（1）试验不同的超参数，如学习率、嵌入维度、Transformer 块的数量等。

（2）数据增强，对训练数据进行扩充，并引入一些随机变化。

（3）增加训练时间。

进行精度提升，但提升效果不明显，准确率依然在 69%上下。

10.层规范化的位置有两种 prenorm 和 postnorm，查询资料了解二者区别并说明自己的模型中层规范化操作的位置是 prenorm 还

是 postnorm ，然后尝试另一种层规范化操作，对比二者在具体训练中的区别并分析原因。

Pre-norm (前规范化)：

在每个子层的输入之前应用规范化。

输入经过规范化后，再经过子层（如多头自注意力、前馈神经网络等）。

这样，规范化操作会调整输入的分布，使其具有零均值和单位方差，然后输入子层进行处理。

Post-norm (后规范化)：

在每个子层的输出之后应用规范化。

子层的输出经过规范化，然后再传递到下一个子层。

这样，规范化操作会调整子层的输出分布。

区别：

稳定性： Pre-norm 更容易训练，因为它在输入到子层之前规范化，有助于维持较稳定的分布。这对于较深的网络可能更有帮助。

表达能力： Post-norm 允许子层看到原始的未经规范化的输入，可能使得网络更具表达能力，但也更容易受到梯度消失或梯度爆炸等问题的影响。

本次实验模型中层规范化操作的位置是 Post-norm。

换成 Pre-norm：

```python
class TransformerEncoder(tf.keras.layers.Layer):
    def __init__(self, vocab_size, embed_size, max_position_embeddings, num_heads, num_layers, mlp_units, dropout_rate):
        super(TransformerEncoder, self).__init__()

        self.embedding_layer = TransformerEmbedding(vocab_size, embed_size, max_position_embeddings)
        self.transformer_blocks = [TransformerBlock(embed_size, num_heads, dropout_rate) for _ in range(num_layers)]
        self.mlp_units = mlp_units
        self.dropout_rate = dropout_rate

    def call(self, inputs, training=None):
        input_ids, segment_ids = inputs

        # 嵌入层
        embeddings = self.embedding_layer([input_ids, segment_ids])

        # Transformer 编码器
        for block in self.transformer_blocks:
            attention_output = block([embeddings, embeddings, embeddings, None])
            # 加和归一化层
            embeddings = AddNorm(embeddings, attention_output, None, self.dropout_rate)

        return embeddings

# 修改 AddNorm 层的 call 方法
class AddNorm(tf.keras.layers.Layer):
    def __init__(self, emb_size, dropout_rate):
        super(AddNorm, self).__init__()

        self.norm = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.dropout_layer = Dropout(dropout_rate)

    def call(self, inputs, training=None):
        x, sublayer, mask = inputs

        # 残差连接
        residual = x

        # 更新这行代码，确保在训练时使用 dropout
        mlp_output = self.mlp_linear(sublayer(x))
        x = residual + self.dropout_layer(mlp_output, training=training)

        # 层规范化
        x = self.norm(x)
        return x
```

更改之后在训练过程中的区别不大，准确率仍在 69%上下，原因可能是更改中层规范化操作的位置对模型的影响较小，不属于关键因素。