基于蛋白质序列的相互作用预测任务报告

任务描述

基于蛋白质序列的相互作用预测

数据集包含约 24000 组蛋白质序列对，每个序列对对应一个标签，提示两个蛋白之间是否存在相互作用。搭建深度学习模型，实现预测两个蛋白之间是否存在相互作用。

任务流程

1. 数据预处理

首先，我们定义一个函数`load_data()`来读取三个数据集。

```
import json
import numpy as np

# 读取数据函数
def load_data(filename):
    with open(filename, "r", encoding="utf-8") as file:
        data = json.load(file)
    seq_1 = [seq["seq_1"] for seq in data]
    seq_2 = [seq["seq_2"] for seq in data]
    labels = [seq["label"] for seq in data]
    return seq_1, seq_2, labels

# 加载数据
train_seq_1, train_seq_2, train_labels = load_data("hppi_train.json")
valid_seq_1, valid_seq_2, valid_labels = load_data("hppi_valid.json")
test_seq_1, test_seq_2, test_labels = load_data("hppi_test.json")
```

由于测试集和验证集的大小过小，影响模型性能，在这里我们分别将训练集的10%数据划分给测试集和验证集。

```
import random

# 将10%的训练数据划分为测试集和验证集
def split_data(train_seq_1, train_seq_2, train_labels):
    # 确定10%的数据量
    split_size = int(0.1 * len(train_seq_1))

    # 生成测试集和验证集的索引
    indices = list(range(len(train_seq_1)))
    random.shuffle(indices)
    test_indices = indices[:split_size]
    valid_indices = indices[split_size:2*split_size]

    # 初始化测试集和验证集
    test_seq_1 = [train_seq_1[i] for i in test_indices]
    test_seq_2 = [train_seq_2[i] for i in test_indices]
    test_labels = [train_labels[i] for i in test_indices]
```

```
    valid_seq_1 = [train_seq_1[i] for i in valid_indices]
    valid_seq_2 = [train_seq_2[i] for i in valid_indices]
    valid_labels = [train_labels[i] for i in valid_indices]

    # 从训练集中移除测试集和验证集的样本
    train_seq_1 = [train_seq_1[i] for i in range(len(train_seq_1)) if i not in
test_indices and i not in valid_indices]
    train_seq_2 = [train_seq_2[i] for i in range(len(train_seq_2)) if i not in
test_indices and i not in valid_indices]
    train_labels = [train_labels[i] for i in range(len(train_labels)) if i not
in test_indices and i not in valid_indices]

    return train_seq_1, train_seq_2, train_labels, valid_seq_1, valid_seq_2,
valid_labels, test_seq_1, test_seq_2, test_labels

# 划分数据集
train_seq_1, train_seq_2, train_labels, valid_seq_1, valid_seq_2, valid_labels,
test_seq_1, test_seq_2, test_labels = split_data(train_seq_1, train_seq_2,
train_labels)

# 打印各个数据集的大小
print("训练集大小:", len(train_seq_1))
print("验证集大小:", len(valid_seq_1))
print("测试集大小:", len(test_seq_1))
```

现在，各个数据集的大小分别为：

训练集大小: 18476

验证集大小: 2309

测试集大小: 2309

2. k-mer 模式

将蛋白质序列划分为不同长度的子序列，称为 k-mer。然后统计每种 k-mer 在序列中出现的频率作为特征。在这里，经过调试和选择，我们采用超参数k=2。我们测试了k=1和k=3的情况，当k=1时，模型不能很好地提取生物学特征，当k=3时，模型参数过多，会导致训练的效果变差。我们采用将氨基酸编码成数字的方式来表征序列，并用0将所有序列补充至相同长度。根据k=2的选择，我们可以确定超参数vocab_size=1+20*20=401（20种氨基酸+0的插补）。

根据对数据的观察，最长序列长度为1024，采用序列全长，会导致0值过多，在这里经过1024、512、256、128的尝试，我们最终决定采用效果最好的256作为超参数max_length的值，代表我们这里利用每一条序列的前256位投入模型。

```
from collections import Counter

vocab_size = 401
max_length = 256

def count_kmer(seq, k):
    kmers = [seq[i:i+k] for i in range(len(seq) - k + 1)]
    return Counter(kmers)

# 统计训练集中 k-mer=2 的出现频率，并按频率顺序构建字典
train_kmer_counts = Counter()
for seq in train_seq_1 + train_seq_2:
```

```
    train_kmer_counts += count_kmer(seq, k=2)

sorted_train_kmer_counts = sorted(train_kmer_counts.items(), key=lambda x: x[1],
reverse=True)

# 构建字典，将每个蛋白质映射到一个索引值
protein_dict = {protein: idx + 1 for idx, (protein, _) in
enumerate(sorted_train_kmer_counts)}

# 编码函数，将蛋白质序列转换为编码序列
def encode_sequence(seq):
    encoded_seq = [protein_dict[seq[i:i+2]] for i in range(0, len(seq)-1)]
    # 补齐到长度为1024
    encoded_seq += [0] * (1024 - len(encoded_seq))
    return encoded_seq

# 对所有蛋白质序列进行编码
train_encoded_seq_1 = [encode_sequence(seq)[:max_length] for seq in train_seq_1]
train_encoded_seq_2 = [encode_sequence(seq)[:max_length] for seq in train_seq_2]
valid_encoded_seq_1 = [encode_sequence(seq)[:max_length] for seq in valid_seq_1]
valid_encoded_seq_2 = [encode_sequence(seq)[:max_length] for seq in valid_seq_2]
test_encoded_seq_1 = [encode_sequence(seq)[:max_length] for seq in test_seq_1]
test_encoded_seq_2 = [encode_sequence(seq)[:max_length] for seq in test_seq_2]

# 打印编码后的序列长度
print("Train Encoded Sequence Length:", len(train_encoded_seq_1[0]))

# 转化成向量形式
train_seq_1_np = np.array(train_encoded_seq_1)
train_seq_2_np = np.array(train_encoded_seq_2)
valid_seq_1_np = np.array(valid_encoded_seq_1)
valid_seq_2_np = np.array(valid_encoded_seq_2)
test_seq_1_np = np.array(test_encoded_seq_1)
test_seq_2_np = np.array(test_encoded_seq_2)
train_labels_np = np.array(train_labels)
valid_labels_np = np.array(valid_labels)
test_labels_np = np.array(test_labels)
```

```
Train Encoded Sequence Length: 256
```

展示其中一组数据：

```
print(train_seq_1_np)
```

```
array([[193,  49,  11, ..., 213, 256, 233],
       [234,  40,  49, ...,  74, 110, 227],
       [193, 106, 145, ...,  54, 357, 374],
       ...,
       [193,  24,  70, ..., 191,  86,  90],
       [278,  12,  14, ...,   0,   0,   0],
       [208,   3,  52, ...,  72, 100, 209]])
```

3. 构建模型

我们的模型结构如下：

1. **输入层**：模型接受两个序列作为输入，这两个序列分别代表两条蛋白质。每个序列的长度为超参数`max_length`的值。
2. **Embedding 层**：这个层将输入的离散型数据转换为密集的低维向量表示。这里使用了一个共享的 Embedding 层，因为两个序列均为蛋白质序列，其嵌入应该在语义上是相似的，这样可以在参数较少的情况下共享信息。
3. **Dropout 层**：为了防止过拟合，我们将Dropout 添加到 Embedding 输出上。Dropout 在训练期间会随机丢弃一部分神经元的输出，有助于提高模型的泛化能力。我们选择dropout的丢失率为0.2，表示每次随机屏蔽20%的隐藏单元。
4. **LSTM 层**：长短期记忆（LSTM）是一种适用于序列数据的循环神经网络（RNN）变体，可以捕捉序列中的长期依赖关系。这里的 LSTM 层会处理嵌入后的序列数据，并生成一个固定长度的向量作为输出。这个向量的长度经过调试，最终我们选择32。
5. **全连接层**：经过 LSTM 处理后的两个序列被连接在一起，然后通过两个全连接层进行进一步的特征提取和非线性变换。全连接层有助于学习更高级别的特征表示。这里，经过全连接层数和全连接层的单位数的调试，我们选择两个全连接层，隐藏单元数分别为64和32，激活函数在调试后选择`relu`函数。
6. **输出层**：最后，通过一个具有 sigmoid 激活函数的密集层进行二元分类，输出表示是否存在蛋白质相互作用的概率。

```
from keras.layers import Input, Embedding, LSTM, Dense, concatenate, Dropout
from keras.models import Model

# 输入层
input_1 = Input(shape=(max_length,))
input_2 = Input(shape=(max_length,))

# Embedding 层
embedding_layer = Embedding(input_dim=vocab_size, output_dim=64)

# 序列编码
encoded_seq_1 = embedding_layer(input_1)
encoded_seq_2 = embedding_layer(input_2)

# 添加 Dropout
dropout_rate = 0.2
dropout_layer = Dropout(dropout_rate)

encoded_seq_1 = dropout_layer(encoded_seq_1)
encoded_seq_2 = dropout_layer(encoded_seq_2)

# LSTM 层
lstm_layer = LSTM(units=32, dropout=0.2, recurrent_dropout=0.2)

# 序列处理
processed_seq_1 = lstm_layer(encoded_seq_1)
processed_seq_2 = lstm_layer(encoded_seq_2)

# 合并处理后的序列
merged_sequences = concatenate([processed_seq_1, processed_seq_2], axis=-1)

# 全连接层
dense_layer1 = Dense(units=64, activation='relu')
```

```
merged_sequences = dense_layer1(merged_sequences)

dense_layer2 = Dense(units=32, activation='relu')
merged_sequences = dense_layer2(merged_sequences)

# 输出层
output_layer = Dense(units=1, activation='sigmoid')

# 输出
output = output_layer(merged_sequences)

# 定义模型
model = Model(inputs=[input_1, input_2], outputs=output)

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

model.summary()
```

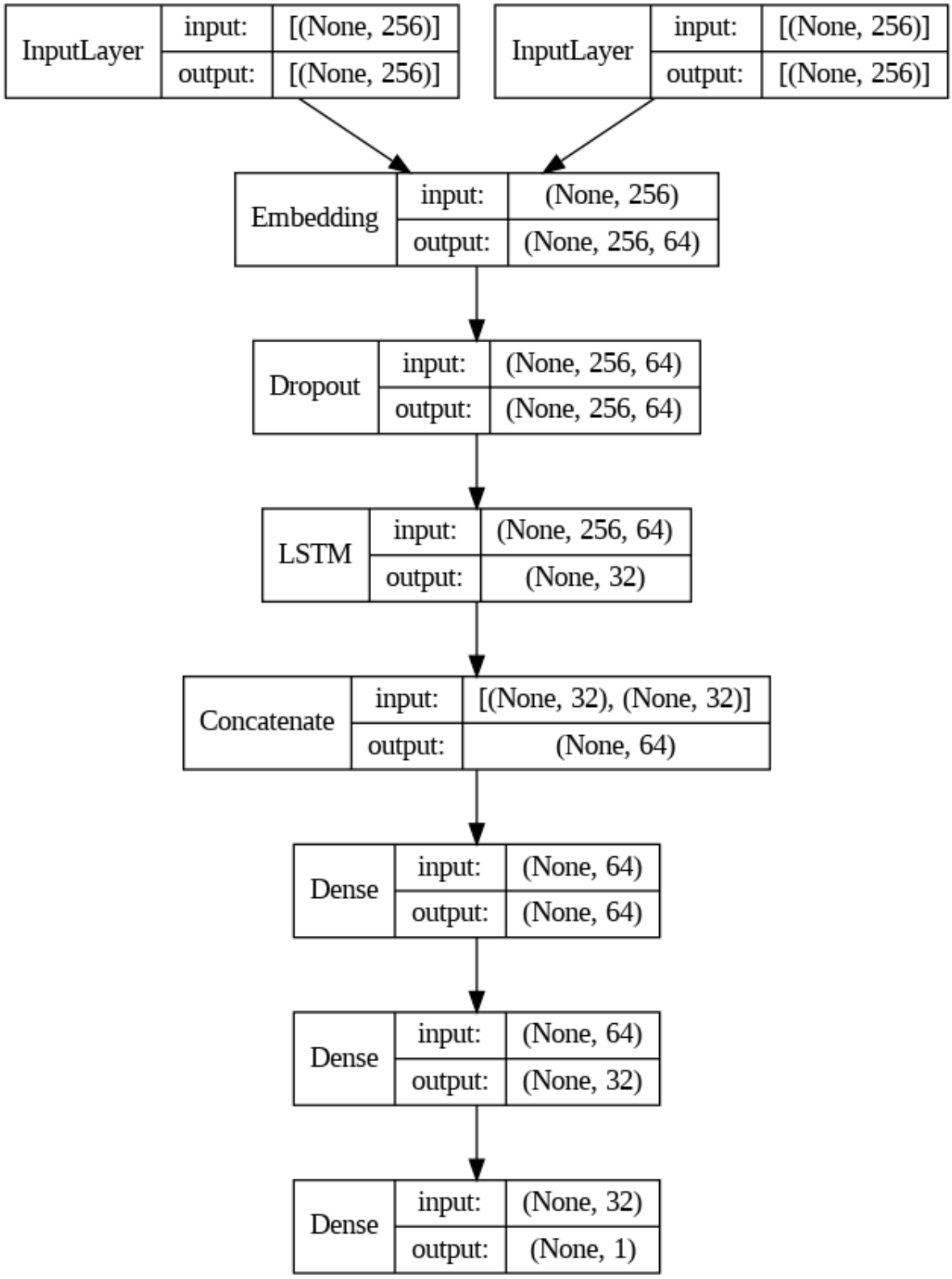
模型总结信息如下：

```
Model: "model"
__________________________________________________________________________________
_________________
 Layer (type)                Output Shape                 Param #   Connected to

==================================================================================
==================
 input_1 (InputLayer)        [(None, 256)]                0         []



 input_2 (InputLayer)        [(None, 256)]                0         []



 embedding (Embedding)       (None, 256, 64)              25664     ['input_1[0]
[0]',
                                                                     'input_2[0]
[0]']


 dropout (Dropout)           (None, 256, 64)              0
['embedding[0][0]',

'embedding[1][0]']


 lstm (LSTM)                 (None, 32)                   12416     ['dropout[0]
[0]',
                                                                     'dropout[1]
[0]']


 concatenate (Concatenate)   (None, 64)                   0         ['lstm[0]
[0]',
```

```
                                                                 'lstm[1]
[0]']

 dense (Dense)               (None, 64)                4160
['concatenate[0][0]']

 dense_1 (Dense)             (None, 32)                2080      ['dense[0]
[0]']

 dense_2 (Dense)             (None, 1)                 33        ['dense_1[0]
[0]']

==================================================================================
==================
Total params: 44353 (173.25 KB)
Trainable params: 44353 (173.25 KB)
Non-trainable params: 0 (0.00 Byte)
```

我们通过流程图的形式来展示模型，这样更加直观清晰：

```
import tensorflow as tf
tf.keras.utils.plot_model(model, to_file='model.png', show_shapes=True,
show_layer_names=False)
```



我们的模型在GPU上进行训练：

```
import tensorflow as tf
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

Num GPUs Available: 1

模型训练过程

经过调优，指定超参数epoch=15，batch_size=64。

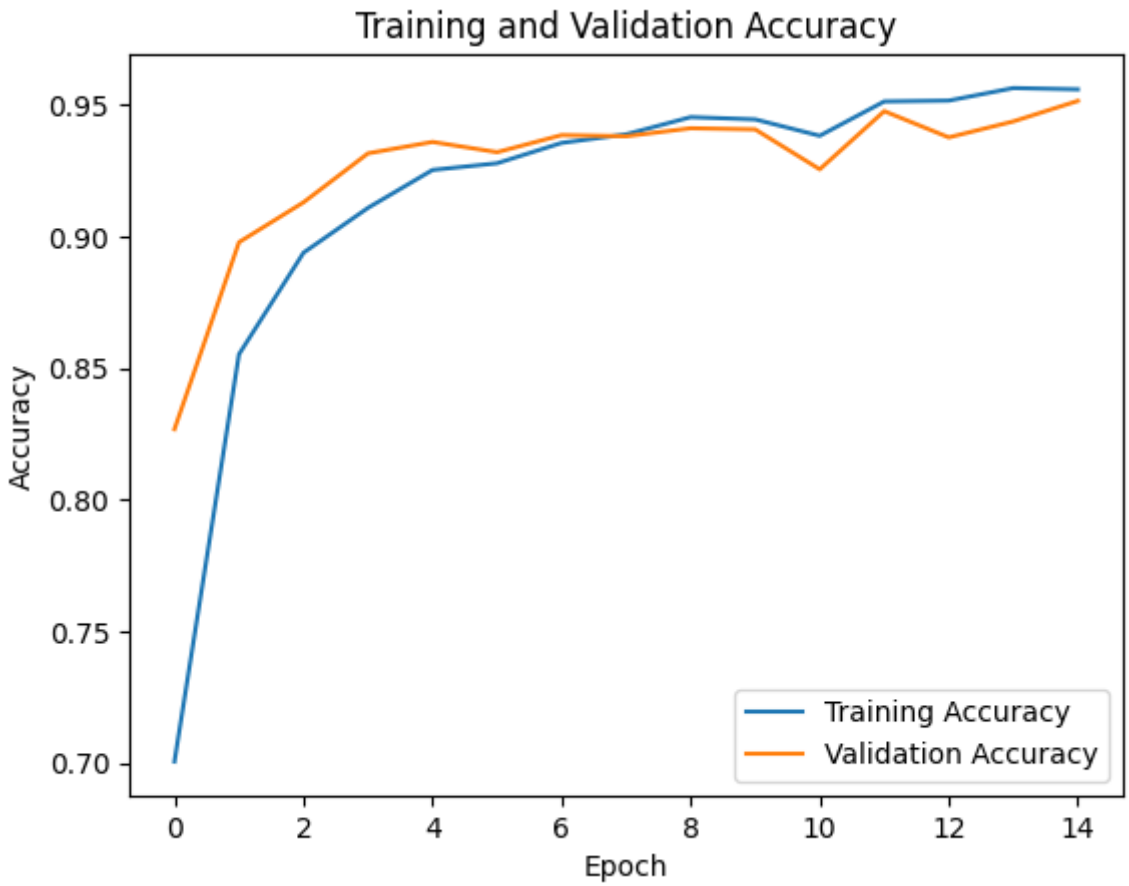
```
# 训练模型
history = model.fit(x=[train_seq_1_np, train_seq_2_np], y=train_labels_np,
          validation_data=([valid_seq_1_np, valid_seq_2_np], valid_labels_np),
          epochs=15, batch_size=64)
```

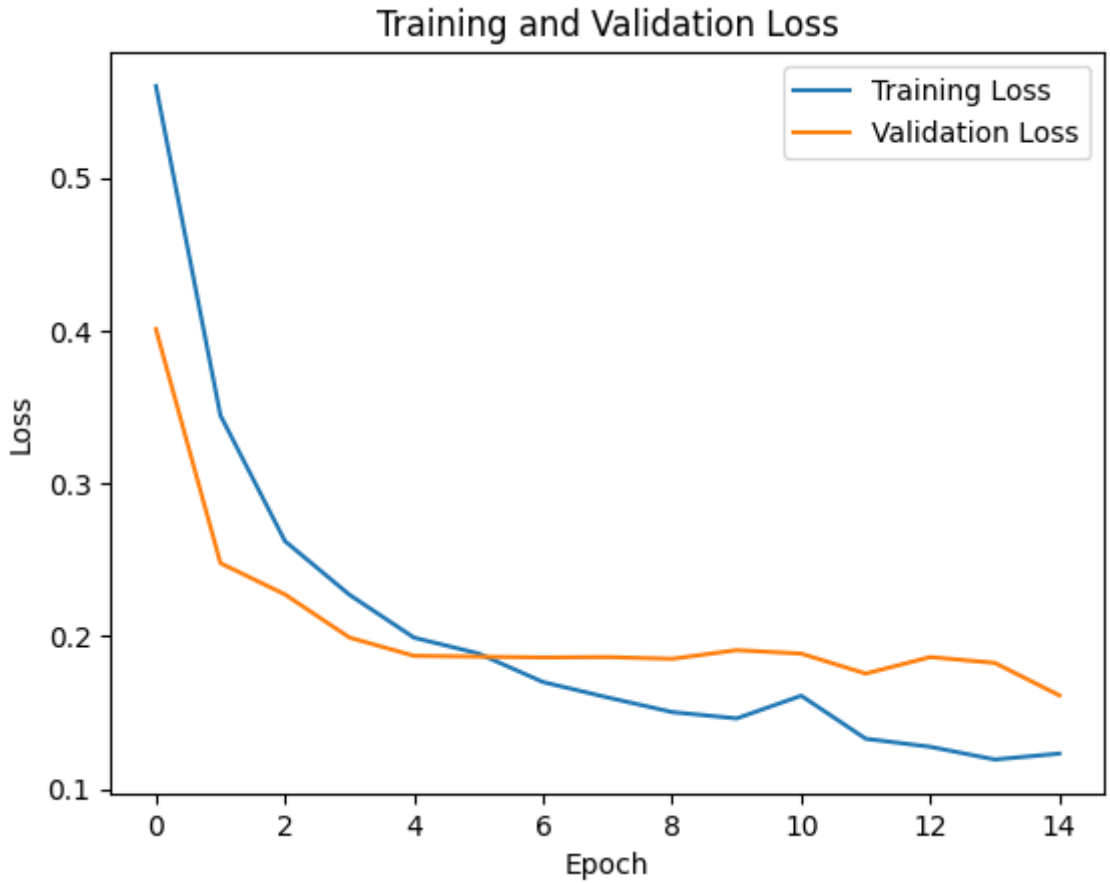
```
Epoch 1/15
289/289 [==============================] - 453s 2s/step - loss: 0.5601 -
accuracy: 0.7005 - val_loss: 0.4009 - val_accuracy: 0.8268
Epoch 2/15
289/289 [==============================] - 437s 2s/step - loss: 0.3445 -
accuracy: 0.8552 - val_loss: 0.2478 - val_accuracy: 0.8978
Epoch 3/15
289/289 [==============================] - 438s 2s/step - loss: 0.2621 -
accuracy: 0.8938 - val_loss: 0.2273 - val_accuracy: 0.9129
Epoch 4/15
289/289 [==============================] - 434s 2s/step - loss: 0.2270 -
accuracy: 0.9109 - val_loss: 0.1990 - val_accuracy: 0.9316
Epoch 5/15
289/289 [==============================] - 434s 1s/step - loss: 0.1990 -
accuracy: 0.9253 - val_loss: 0.1871 - val_accuracy: 0.9359
Epoch 6/15
289/289 [==============================] - 436s 2s/step - loss: 0.1886 -
accuracy: 0.9278 - val_loss: 0.1865 - val_accuracy: 0.9320
Epoch 7/15
289/289 [==============================] - 434s 2s/step - loss: 0.1699 -
accuracy: 0.9355 - val_loss: 0.1860 - val_accuracy: 0.9385
Epoch 8/15
289/289 [==============================] - 433s 1s/step - loss: 0.1598 -
accuracy: 0.9388 - val_loss: 0.1863 - val_accuracy: 0.9381
Epoch 9/15
289/289 [==============================] - 436s 2s/step - loss: 0.1502 -
accuracy: 0.9453 - val_loss: 0.1851 - val_accuracy: 0.9411
Epoch 10/15
289/289 [==============================] - 433s 1s/step - loss: 0.1461 -
accuracy: 0.9445 - val_loss: 0.1908 - val_accuracy: 0.9407
Epoch 11/15
289/289 [==============================] - 434s 1s/step - loss: 0.1609 -
accuracy: 0.9382 - val_loss: 0.1885 - val_accuracy: 0.9255
Epoch 12/15
289/289 [==============================] - 434s 2s/step - loss: 0.1328 -
accuracy: 0.9513 - val_loss: 0.1754 - val_accuracy: 0.9476
Epoch 13/15
289/289 [==============================] - 432s 1s/step - loss: 0.1275 -
accuracy: 0.9516 - val_loss: 0.1862 - val_accuracy: 0.9376
Epoch 14/15
289/289 [==============================] - 427s 1s/step - loss: 0.1192 -
accuracy: 0.9564 - val_loss: 0.1825 - val_accuracy: 0.9437
Epoch 15/15
289/289 [==============================] - 427s 1s/step - loss: 0.1231 -
accuracy: 0.9559 - val_loss: 0.1612 - val_accuracy: 0.9515
```

可以粗略地观察到，模型在经过15个epoch的训练后，训练集和验证集均达到了0.95以上的较好的预测准确率。

我们展示模型训练过程中的loss和accuracy随epoch的变化曲线：

```python
import matplotlib.pyplot as plt

# 绘制精确率随epoch变化的图
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.show()

# 绘制损失随epoch变化的图
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()
```





可见，经过15个epoch的训练，模型的loss接近收敛，表明15个epoch对于该模型来说已经足够。在训练的初期阶段（epoch较小时），训练集loss大于验证集，可能是由于模型的初始化差异或者验证集和训练集数据分布不一致，随着训练的进行，训练集loss低于验证集，但是差距不大，表明模型在训练集和验证集上都表现良好，并且具有一定的泛化能力。

4. 模型评估

我们把模型放在测试集上进行测试：

```
loss, accuracy = model.evaluate([test_seq_1_np, test_seq_2_np], test_labels_np)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

```
73/73 [==============================] - 13s 183ms/step - loss: 0.1585 -
accuracy: 0.9506
Test Loss: 0.1585349589586258
Test Accuracy: 0.9506279826164246
```

可见，测试集准确率达到0.9506279826164246，表现较为优秀。

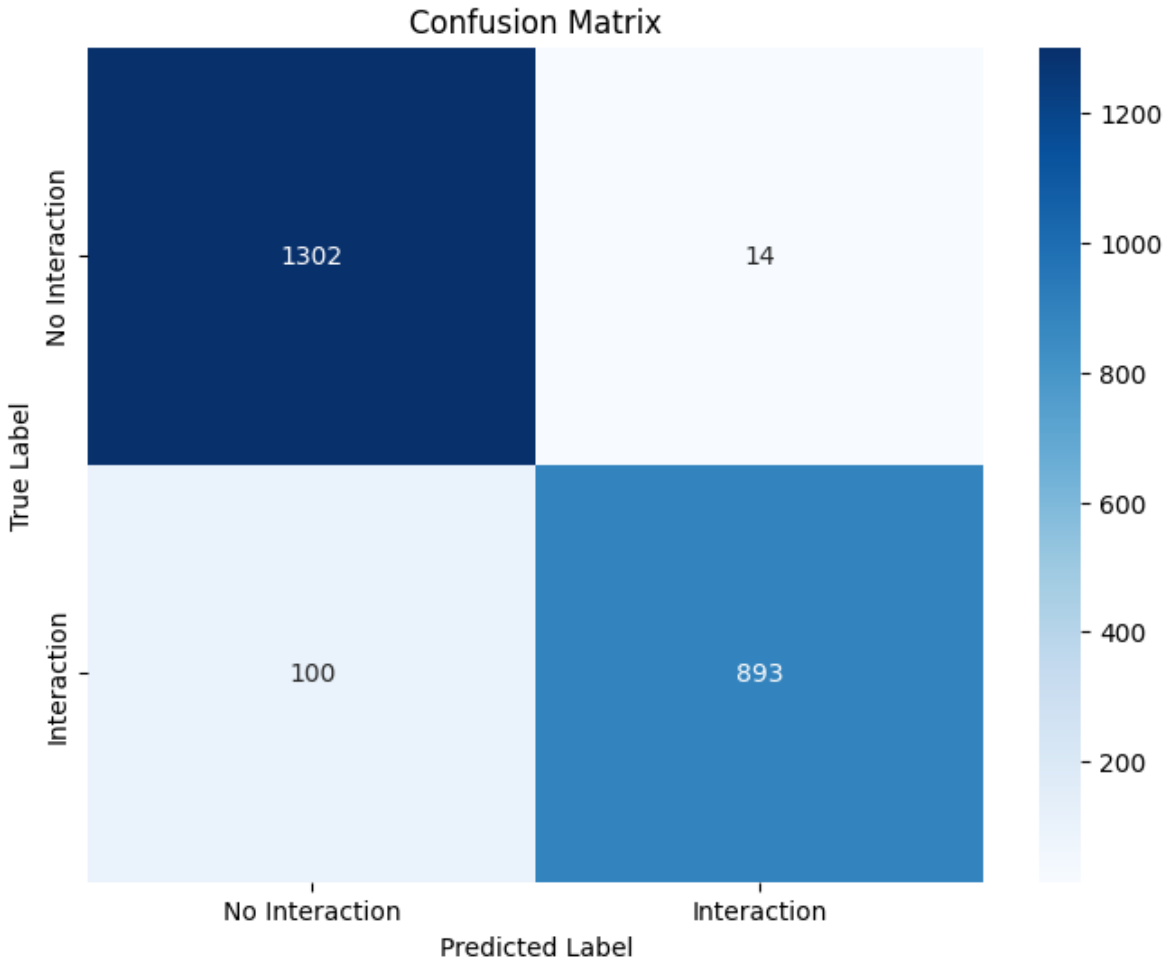
输出模型的分类报告，并将模型预测测试集的混淆矩阵进行可视化：

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# 预测测试集数据
predictions = model.predict([test_seq_1_np, test_seq_2_np])
predictions = (predictions > 0.5)  # 将概率值转换为二进制预测结果
```

```python
# 计算输出混淆矩阵
cm = confusion_matrix(test_labels_np, predictions)
print("Confusion Matrix:")
print(cm)
print("\nClassification Report:")
print(classification_report(test_labels_np, predictions))

# 可视化混淆矩阵
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['No Interaction', 'Interaction'],
            yticklabels=['No Interaction', 'Interaction'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
Confusion Matrix:
[[1302   14]
 [ 100  893]]
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.99      0.96      1316
           1       0.98      0.90      0.94       993

    accuracy                           0.95      2309
   macro avg       0.96      0.94      0.95      2309
weighted avg       0.95      0.95      0.95      2309
```



混淆矩阵的行代表了实际的类别，列代表了模型预测的类别。

(0,0) 单元格中的值1302表示模型将 1316 个真实类别为 0 的样本中的 1302 个正确预测为类别 0，而 (0,1) 单元格的值14代表将 14 个错误预测为类别 1。

同样，模型将 993 个真实类别为 1 的样本中的 893 个正确预测为类别 1，而将 100 个错误预测为类别 0。

分类报告提供了每个类别的精确度、召回率和 F1 分数以及该类别的支持数量。macro avg 是所有类别的精确度、召回率和 F1 分数的平均值，weighted avg 则是以每个类别的支持数量作为权重计算的平均值。

5. 结果讨论

从混淆矩阵和分类报告中可以看出，模型在预测类别 0（蛋白质之间不存在相互作用）方面表现很好，精确度和召回率都很高，F1 分数为 0.96。而在预测类别 1（蛋白质之间存在相互作用）方面，模型的表现稍稍低于类别 0，但仍然表现出较高的精确度、召回率和 F1 分数。

最终，模型在三个数据集上达到的最佳精确率分别为：

训练集：0.9564

验证集：0.9515

测试集：0.9506

三者均大于0.95，并且差距不大，表明模型基本不存在欠拟合或过拟合现象，并且具有较强的预测能力和模型泛化能力。

我们认为，在当前的数据量的情况下，模型已经表现得相当好，未来的改进方向有：

更多的数据：

尝试收集更多的数据或采用数据增强技术。更多的数据可以使我们能够训练更复杂的模型，帮助模型更好地学习数据的分布和规律，从而提高模型的性能。

在增大数据量的基础上，我们提出以下改进方式：

1. **调整嵌入层的维度和参数**：
    - 考虑调整嵌入层的输出维度（output_dim），64 维度在某些情况下可能过于简单。
    - 考虑使用预训练的词向量（如 Word2Vec 或 GloVe），这样可以更好地捕捉序列数据中的语义信息，提高模型性能。
2. **使用更复杂的循环神经网络结构**：
    - 使用LSTM 的更复杂的变体，如双向 LSTM、GRU 等。
3. **改变数据的特征提取方式**：
    - 考虑查询相关文献等，采用更具有生物学意义的方式来对蛋白质序列数据数据进行特征提取，就k-mers而言可以考虑尝试其他的k值。
4. **使用注意力机制**：
    - 考虑在模型中引入注意力机制，特别是当处理较长的序列时，注意力机制可以帮助模型更好地关注重要的部分，提高模型的表现。
5. **超参数调优**：
    - 考虑对模型的超参数进行更详细的调优，包括学习率、批量大小、优化器的选择等，或者使用网格搜索或随机搜索等技术来搜索最佳的超参数组合。
6. **集成学习**：
    - 考虑使用集成学习方法，如投票、堆叠等，结合多个模型的预测结果，以提高模型的稳健性和泛化能力。