# NLP 大作业报告
# 基于 Bert 的中文商品评论分类模型

西安交通大学，数学与统计学院，强基数学

吴天阳[a]，马煜璇[b]，孙思雨[c]，陈江河[d]

2204210460[a]，2204220461[b]，2206124483[c]，2204411197[d]

2022 年 12 月 23 日

# 目录

# 1 实验目的

使用 GitHub 上商品评论的数据集，总共 10 个商品类别，包含两种情绪，总计 62773 条数据. 正向评论：31727 条，负向评论：31046 条.

使用 Bert 预训练模型，对每条评论进行词向量转换，输出层连接一个有 11 个神经元的全连接神经网络对文本进行分类.

调整超参数，包括：不同的预处理序列长度，优化器的选择，输出层前神经网络的层数. 自定义验证数据，验证模型的泛化性.

# 2 实验原理

Bert 模型原理参考文章：

1. 图解 BERT 模型：从零开始构建 BERT 这篇文章详细的介绍了 Bert 的原理、transformer 模型和 Attention 机制.（以下图片均来自该博客）
2. 什么是 BERT? 这两篇文章都介绍了 Bert 模型的输入输出还有 Bert 的结构，第二篇更简短一些. 其预训练部分使用的原理，MLM 和 NSP 过程，不用很具体的讲解，只需要大致介绍他们俩分别起到什么功能.

Bert 模型是一种**无监督学习的预训练模型**（能进行迁移学习的模型，用于各种 NLP 问题），模型主要就是将 transformer 模型进行堆叠而形成的，其输入与输出的维度相同，过程类似于词编码过程，将低维的词向量进行编码，并将其特征进行放大，与其他特征进行分离.

## 2.1 模型结构

NLP，BERT，Transformer，Attention 之间的关系，如图1所示.

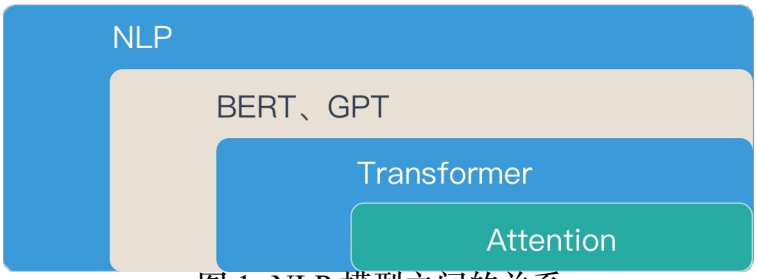



图 1: NLP 模型之间的关系

### 2.1.1 Attention 机制

Attention 机制主要思路就是：**通过机器学习得到单词之间的权重分布，然后作用在特征上**.

Attention 机制有三种实现方式：RNN+Attention，CNN+Attention，纯 Attention，第三种就是 Google 团队在 2017 发表的论文Attention is All you need中提到的，上述模型都是使用该思路搭建的.

这里以单个文字的 Attention 权重计算为例，首先将该句话总每个文字使用神经网络转化为向量表示形式（词向量），取定一个文字作为当前的 Query 目标，将上下文的文字作为 Key，并同时另存到 Value 值内. 然后计算 Query 值和 Key 值的相关性（利用内积进行计算），并通过 softmax 函数得到 Attention 权重（归一化），最后再对 Value 向量使用 Attention 权重加权求和，即可得到 Attention 机制后的输出. 如图2所示.

我们再对该句话中每一个文字都进行如上操作即可得到整句话的 Attention 输出，由于只融合了该句话字之间的相关性，所以也称为 Self-Attention，如图3所示.

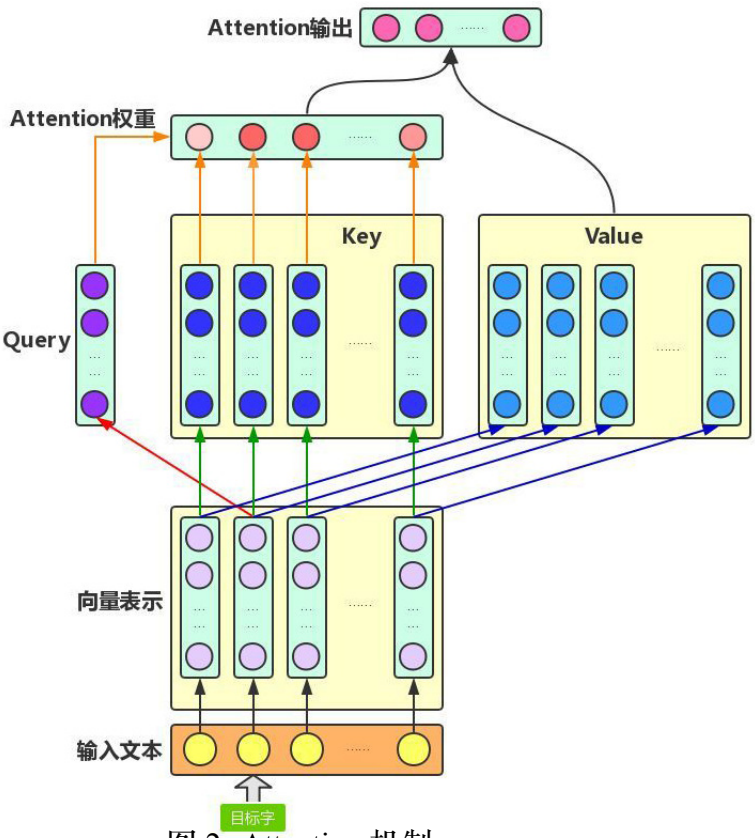



图 2: Attention 机制

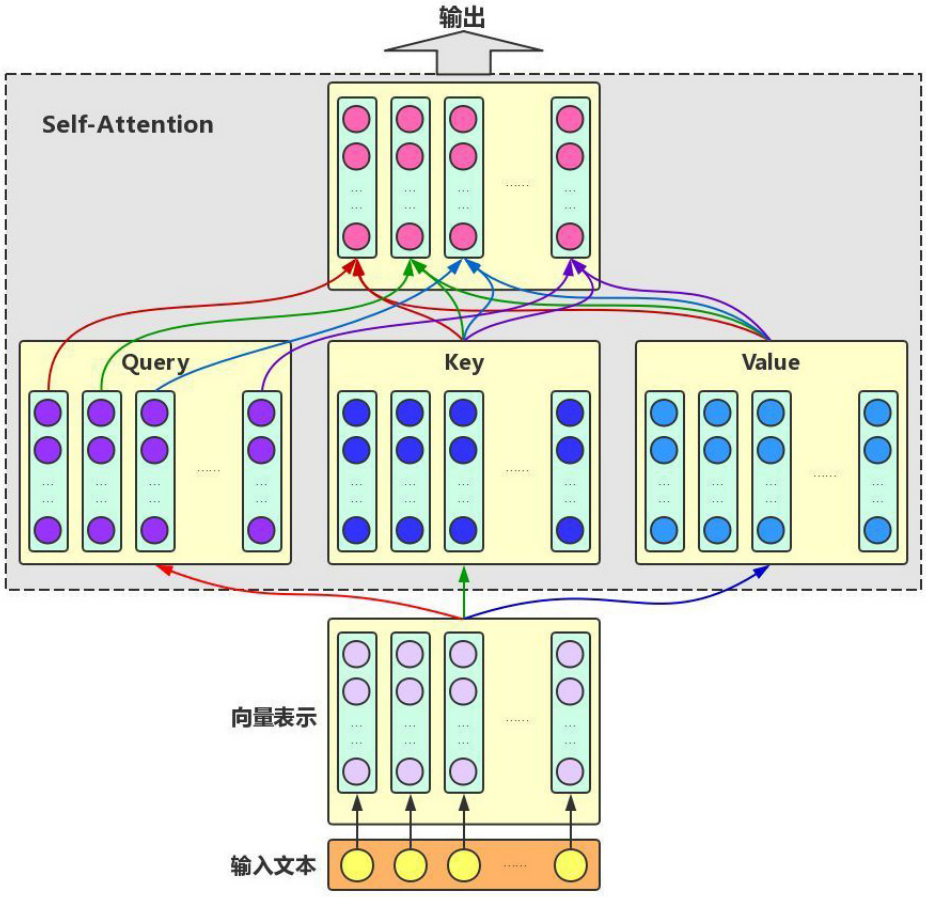



图 3: Self-Attention 模型

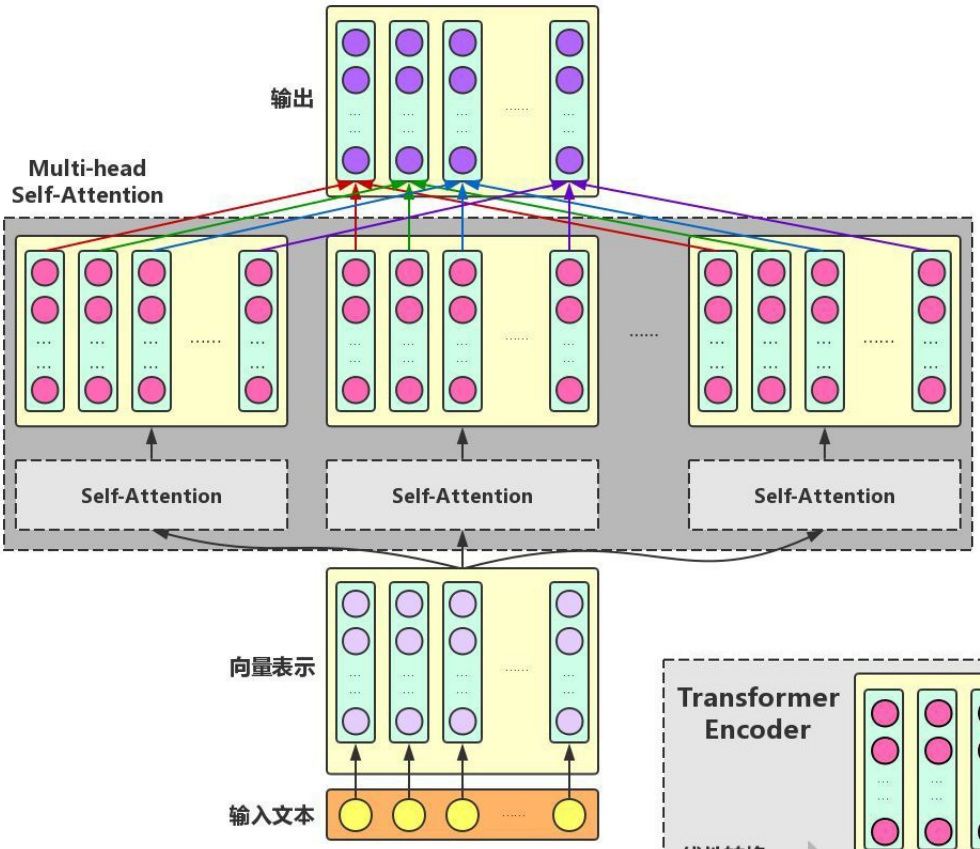



图 4: Multi-head-Self-Attention

为了进一步提高 Attention 处理的多样性，处理不同语义空间下的增强向量，Multi-head Self-Attention 模型中进一步叠加 Self-Attention，最后连接神经网络保持输出层和原始向量长度相同，这就得到了 Multi-head Self-Attention，如图4所示.

### 2.1.2 Transformer Encoder

由于 Berd 中只是用了 Transformer 的编码部分，所以只对其进行介绍. Transformer 主要是在 Multi-head Self-Attention 的基础上加入了三个操作 (模型如图5所示)：1. 残差连接（Residual Connection）：此处使用的思路应该是来自 2015 年 ImageNet 图像识别比赛第一名的ResNet，其主要用于解决深度神经网络在深度过高之后数据过度离散的问题，主要解决了过多的非线性函数导致网络难以实现恒等变换的问题，同样该操作使得网络变得更加容易训练.

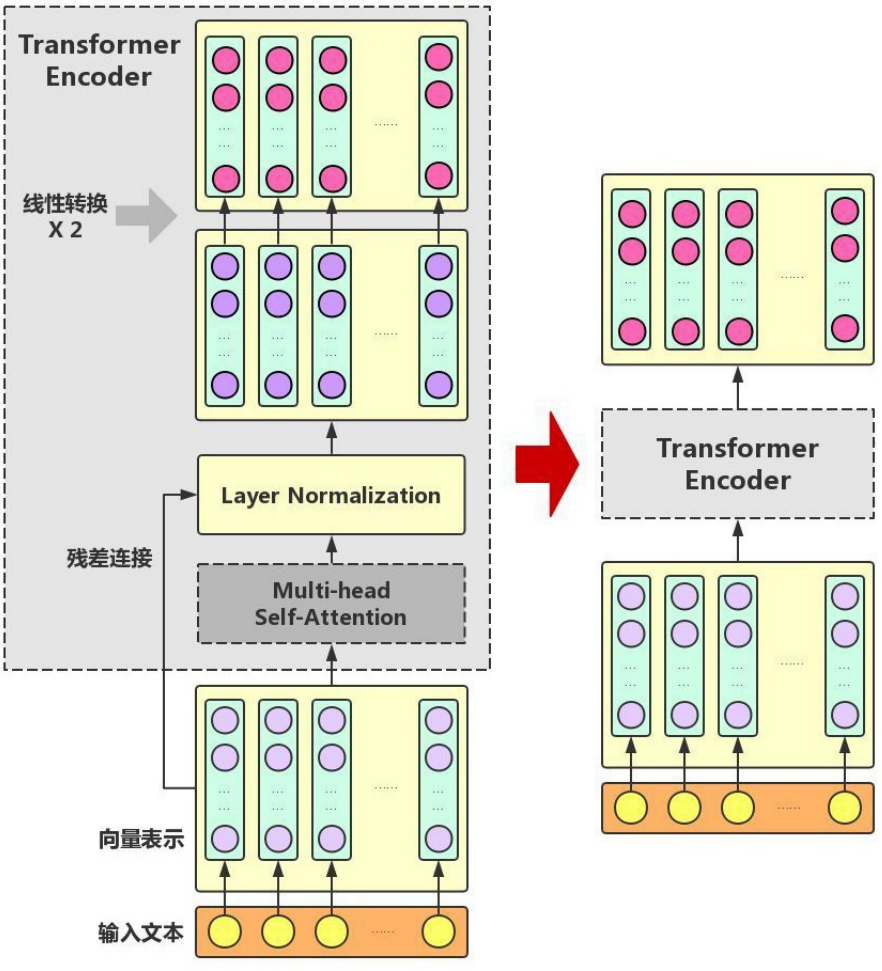



图 5: Transformer 模型

2. 层标准化（Layer Normalization）：对某一层神经网络做均值为 0 方差为 1 的标准化操作，主要为了避免网络过深导致 loss 值过小的问题.

3. 两次线性变换：两层神经网络处理，增强模型的表达能力（保持输入与输出长度相同）.

### 2.1.3 Bert 模型

在 Transformer 模型基础上对其进行堆叠，最后连接全连接神经网络降低特征维数，这样就完成了 Bert 模型基本框架，模型如图6所示.（论文中使用的堆叠层数为 12 层和 24 层，我们将使用 12 层的 Bert 模型进行训练）

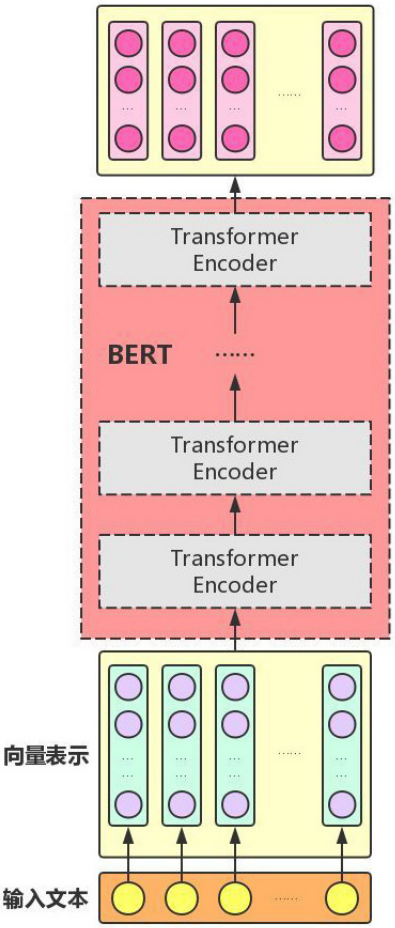


图 6: Bert 模型

## 2.2 预训练任务

有了 Bert 模型之后，为了使该模型具有泛化能力，能够用于处理各种 NLP 问题，论文作者以 Wiki 作为数据集对模型进行预训练（如同在读懂文章之前，学会如何理解句式，学习语言的本身）. Bert 模型主要由以下两个预训练模型构成：

1. Masked Language Model（MLM），在一句话中随机掩去该句话中的几个字，通过剩余的字去预测掩去的字是什么，类似英文中的完形填空，本质是在模仿人类学习语言的方法，这样的好处在于迫使机器去依赖上下文预测词汇，增强上下文词汇之间的关联性，并赋予其一定的纠错能力. 如图7所示.

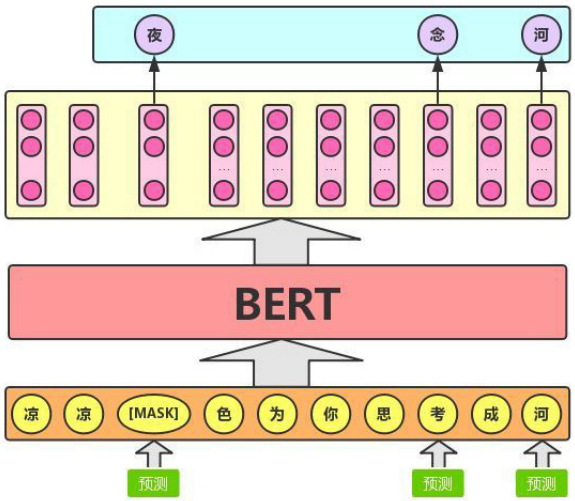


图 7: MLM 任务

2. Next Sentence Prediction（NSP）：通过给出文章中的两句话，判断第一句话是否出现在第二句话之后，类似高中语文的古诗词默写和英文的段落重排，该训练可以使模型学习到整篇文章内容之间的关联性，更准确的刻画语句之间的信息. 如图8所示.

## 2.3 模型应用

对于不同的现实场景 Bert 模型通过构建不同的输出层维度从而完成不同的分类问题，例如：单文本分类（通过在文章的开头加入 [CLS] 符号表示文章的语义信息），语义场景分类（使用 [SEP] 分隔符作为两句话之间的分隔），序列标注问题等等，参考图8中的输入部分.（在程序设计中的预处理部分会详细解释）

更一般的，可以通过对 Bert 模型的输出进行微调从而完成各种分类问题，即对编码后的向量连接全连接神经网络到输出层，对模型进行训练.

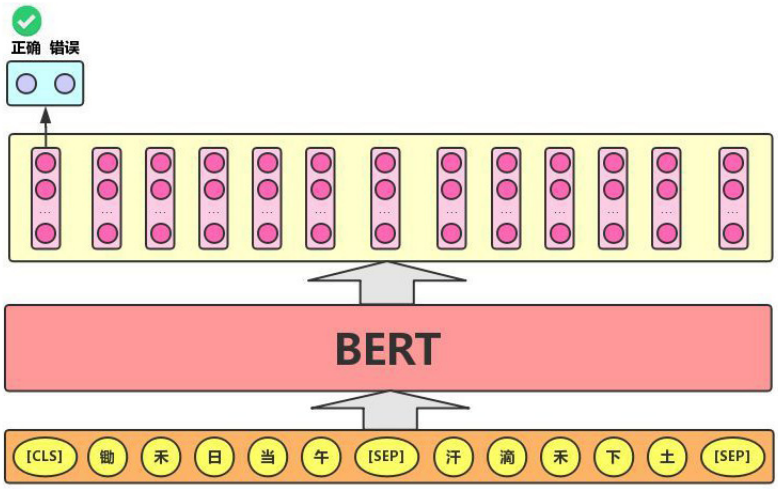



图 8: NSP 任务

# 3 实验步骤与结果分析

试验所使用的 Python 环境为 3.9.15，神经网络框架为 TensorFlow2.9.1 及以上版本，使用 Bert 模型必备库为 `tensorflow_hub`，`tensorflow_text`（使用 pip 安装对应 TensorFlow 版本的即可），其他常用库为 `numpy`，`pandas`，`matplotlib`，`tqdm`.

## 3.1 数据处理

使用 `pandas` 库做数据处理部分，首先删去评论中的无效数据项

```
df = pd.read_csv(data_handle)  # 数据读入
df = df[df.review.isna() == False]  # 删去 Nan 项
```

对数据类别进行编号，构造从类别空间到 $\mathbb{Z}^{10}$ 的双射

```
class2idx = {}  # 从类别到编号的映射
idx2class = {}  # 从编号到类别的映射
for idx, name in enumerate(class_names):
    class2idx[name] = idx
    idx2class[idx] = name
```

不同类别的正负样例数目不同，如下表所示

| 类别 | 书籍 | 平板 | 手机 | 水果 | 洗发水 | 热水器 | 蒙牛 | 衣服 | 计算机 | 酒店 |
|---|---|---|---|---|---|---|---|---|---|---|
| 总数目 | 3851 | 10000 | 2323 | 10000 | 10000 | 574 | 2033 | 10000 | 3992 | 10000 |
| 正例 | 2100 | 5000 | 1165 | 5000 | 5000 | 474 | 992 | 5000 | 1996 | 5000 |
| 负例 | 1751 | 5000 | 1158 | 5000 | 5000 | 100 | 1041 | 5000 | 1996 | 5000 |

首先将每种类别商品的数据按照 3:1 划分为训练集和验证集，然后对训练集进行补全到 7500 个数据，正负样例分别为 3750 个，若不足，则随机选取补齐.

```
train_ds, val_ds = None, None  # 训练集与验证集
for name in class_names:
    tmp = df[df.cat==name]
    pos_num, neg_num = tmp[tmp.label==1].shape[0], tmp[tmp.label==0].shape[0]  # 获取当前类别下正例与负例数目
    pos_ds = tf.data.Dataset.from_tensor_slices((tmp[tmp.label==1]['review'], [(1, class2idx[name]) for _ in range(pos_num)]))  # 正例集合
    neg_ds = tf.data.Dataset.from_tensor_slices((tmp[tmp.label==0]['review'], [(0, class2idx[name]) for _ in range(neg_num)]))  # 负例集合

    pos_num = pos_num * val_split // 100  # 划分给验证集的正例数目
    neg_num = neg_num * val_split // 100  # 划分给验证集的负例数目
    pos_val = pos_ds.take(pos_num)  # 正例验证集
    pos_train = pos_ds.skip(pos_num)  # 正例训练集
    neg_val = neg_ds.take(neg_num)  # 负例验证集
    neg_train = neg_ds.skip(neg_num)  # 负例训练集

    train_marge = pos_train.concatenate(neg_train).shuffle(10000, seed=seed).repeat(-1)  # 合并正负数据，再补齐到 7500 个数据
    train_ds = train_marge.take(7500) if train_ds is None else train_ds.concatenate(train_marge.take(7500))  # 合并训练集
    val_ds = pos_val.concatenate(neg_val) if val_ds is None else val_ds.concatenate(pos_val).concatenate(neg_val)  # 合并验证集
```

## 3.2 模型搭建

Bert 模型被分为两部分：`Bert_preprocessor`，`Bert_encoder`，前者对输入特征进行预处理操作，加入 [CLS] 符号与 [SEP] 符号，进行 mask 操作与段落分段任务的标记（对应 MLM 和 NSP 任务）；后者是 Bert 模型的主要部分，包括 12 个 Transformer 块和最后输出的全连接神经网络. 下面我们详细分析每一部分的具体实现方法.

### 3.2.1 预处理模型

模型来自TF-hub: bert_zh_preprocess，对句子的预处理的输出包括三个部分：

1. `input_mask`：二进制掩码，1 表示模型可以知道该处的字，0 表示模型无法知道该处的字.（用于 MLM 任务）
2. `input_word_ids`：将每个字进行编码，其中`101`表示 [CLS]，`102`表示 [SEP].
3. `input_type_ids`：只有在有两句话判断上下文时有用，用于告诉模型两句话的位置. 第一段 0 表示第一句话，第一段 1 表示第二句话.（用于 NSP 任务）

模型搭建部分如下，seq_length 为自定义转化的最大字符串长度.

```
# preprocessor_handle 为预加载模型的本地路径，或者为官网下载链接
def bert_preprocessor(sentence_features, seq_length=128):
    text_inputs = [layers.Input(shape=(), dtype=tf.string, name=ft)
                   for ft in sentence_features]  # 设定文本的输入

    preprocessor = hub.load(preprocessor_handle)  # 加载预处理模型
    # tokenize 可以将每个句子划分为单个的字
    tokenize = hub.KerasLayer(preprocessor.tokenize, name='tokenizer')
    tokenized_inputs = [tokenize(segment) for segment in text_inputs]

    # packer 加载预处理模型中将 tokenize 结果进一步打包输出为上述三种形式
    packer = hub.KerasLayer(
        preprocessor.bert_pack_inputs,
        arguments=dict(seq_length=seq_length),
        name='packer'
    )
    encoder_inputs = packer(tokenized_inputs)
    return keras.Model(text_inputs, encoder_inputs, name='preprocessor')
preprocessor = bert_preprocessor(['input1'])  # 构建一个输入句子的预处理模型
```

对于输入的字符串" 你好 "，预处理部分会在字符串的开头和结尾处加上开始符号 [CLS] 和分隔符号 [SEP]，转化为"[CLS] 你好 [SEP]"，从而在原有序列长度上增加 2.

下面我们对其进行验证（101,102分别表示 [CLS]，[SEP]）:

```
x = tf.constant(['你好呀', '不好但是'])
x_preprocessed = preprocessor(x)
print(f"{'Keys':<15}: {list(x_preprocessed.keys())}")
print(f"{'Shape Word Ids':<15}: {x_preprocessed['input_word_ids'].shape}")
print(f"{'Word Ids':<15}: {x_preprocessed['input_word_ids'][0,:12]}")
print(f"{'Shape Mask':<15}: {x_preprocessed['input_mask'].shape}")
print(f"{'Input Mask':<15}: {x_preprocessed['input_mask'][0,:12]}")
print(f"{'Shape Type Ids':<15}: {x_preprocessed['input_type_ids'].shape}")
print(f"{'Type Ids':<15}: {x_preprocessed['input_type_ids'][0,:12]}")
""" 输出结果
Keys           : ['input_mask', 'input_word_ids', 'input_type_ids']
Shape Word Ids : (2, 128)
Word Ids       : [101 872 1962 1435 102  0  0  0  0  0  0  0]
Shape Mask     : (2, 128)
Input Mask     : [1 1 1 1 1 0 0 0 0 0 0 0]
Shape Type Ids : (2, 128)
Type Ids       : [0 0 0 0 0 0 0 0 0 0 0 0]
"""
```

使用 tf.keras.utils.plot_modelapi 输出模型的架构图，如图9所示.

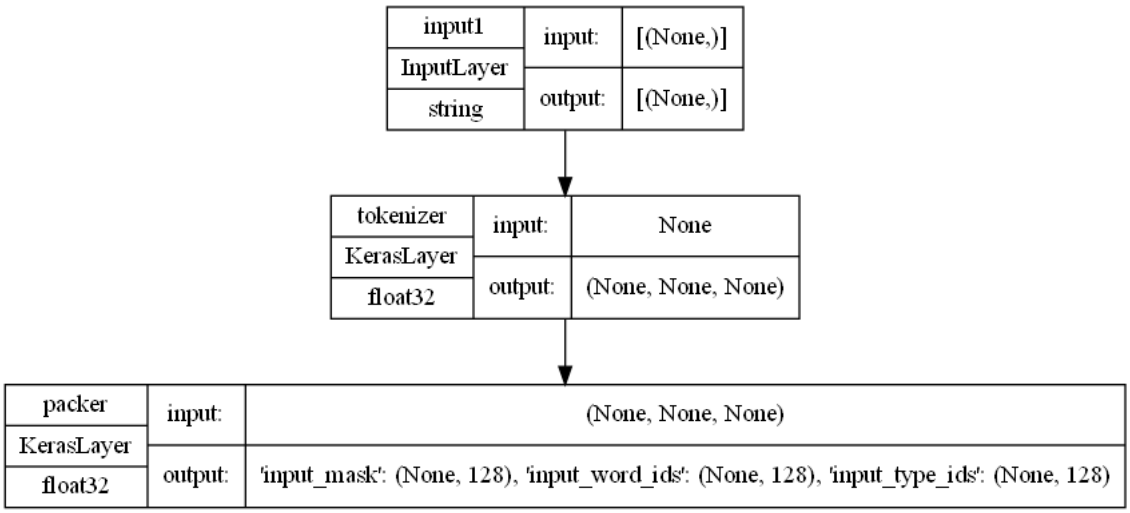



图 9: Preprocessor 模型

### 3.2.2 Bert 分类模型

已预训练的 Bert 编码模型来自TF-hub: bert_zh_L-12_H-768_A-12，从模型论文BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding 中可知，模型编号中 L 为 Transformer 块个数，H 为隐藏层大小 (输出维度)，A 为每个 Transormer 中 Self-Attention 的个数.

Bert 编码模型的输出包含三个部分：

1. `pooled_output`：输出的特征向量，最后经过全连接层后输出的结果，大小为 768 维.
2. `sequence_output`：最后一个 Transformer 块的输出结果，大小为 `[128, 768]` 维.
3. `encoder_outputs`：大小为 12 的 list，包含 Transformer 块的输出结果.

我们要用汇聚后的输出特征 `pooled_output`，连接一层 0.3 抛弃率的 `Dropout` 层，最后通过两个全连接神经网络，其中一个包含 1 个神经元作为情感分类，另一个包含 2 个神经元作为商品分类. 模型搭建代码如下

```
# encoder_handle 为 Bert 模型的本地路径，或者为官网下载链接
def build_classifier():
    text_input = layers.Input(shape=(), dtype=tf.string, name='input')
    text_preprocessed = preprocessor(text_input)  # 使用已搭建的预处理模型
    encoder = hub.KerasLayer(encoder_handle, trainable=True,
    ↪  name='BERT_encoder')  # 导入 Bert 模型
    x = encoder(text_preprocessed)['pooled_output']
    x = layers.Dropout(0.3)(x)
    x1 = layers.Dense(1, name='emotion')(x)
    x2 = layers.Dense(10, name='classifier')(x)
    return keras.Model(text_input, [x1, x2])
classifier_model = build_classifier()
```

还是使用 `tf.keras.utils.plot_model` 显示模型框架，如图10所示

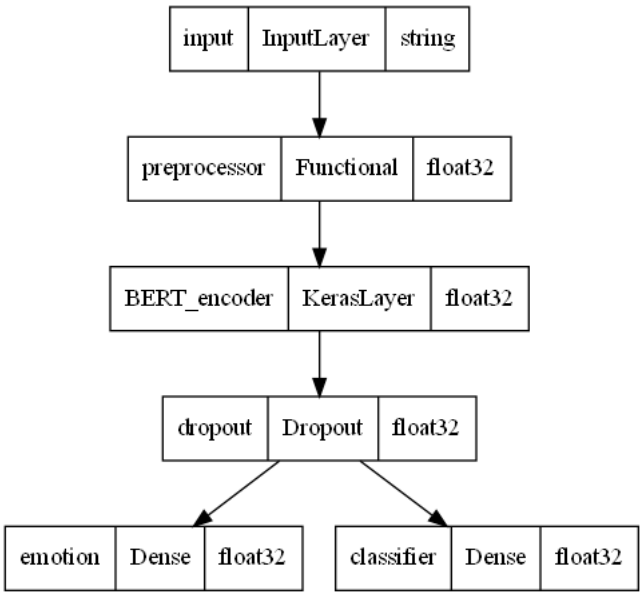



图 10: 基于 Bert 编码的分类模型

## 3.3 模型训练与验证

下面进行超参数配置和设置模型参数记录器：

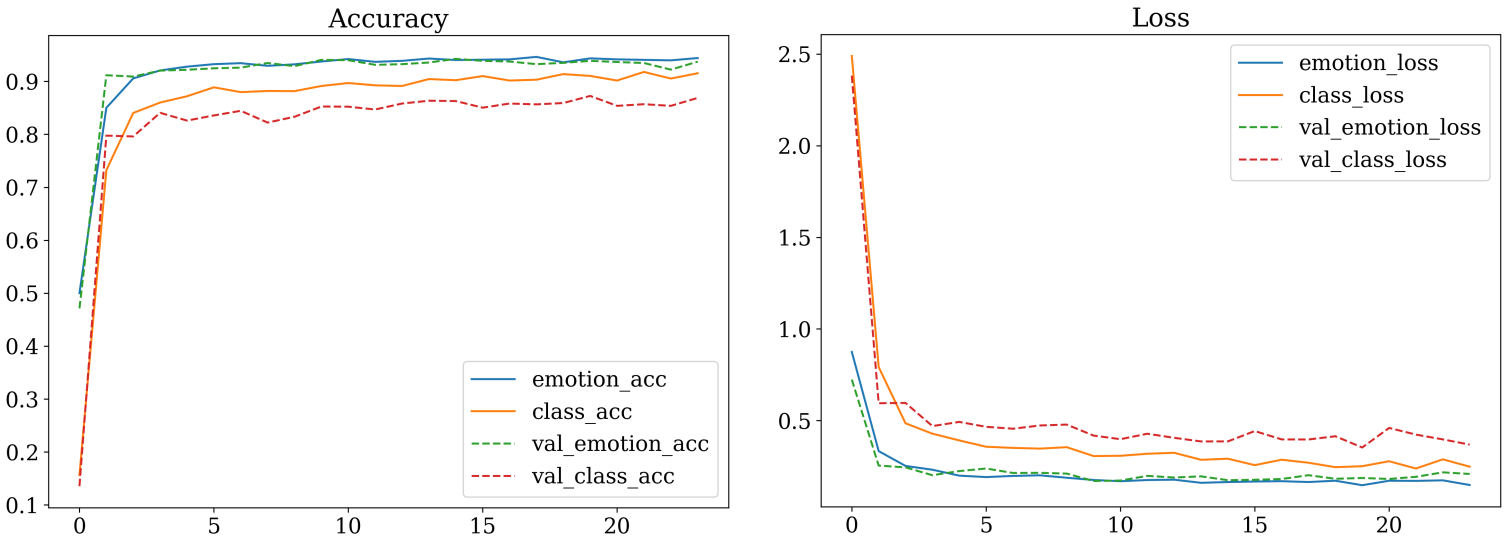
```
# 超参数配置
batch_size = 32
batch_N = 100000 / batch_size
epochs = 1
lr_change = None  # 根据 epoch 训练个数动态调整步长
optimizer = optimizers.Adam(learning_rate=1e-4)  # Adam 优化器，设定步长
binary_loss = losses.BinaryCrossentropy(from_logits=True)  # 损失函数
multi_loss = losses.SparseCategoricalCrossentropy(from_logits=True)
# 随机打乱样本，设定 batch 大小
dataset = ds.shuffle(100000).batch(batch_size).repeat(epochs)

# 情感分类上的准确率
emotion_acc = keras.metrics.BinaryAccuracy('emotion_acc')
# 情感分类上的平均损失
emotion_loss= keras.metrics.Mean('emotion_loss', dtype=tf.float32)
# 物品分类上的准确率
class_acc = keras.metrics.SparseCategoricalAccuracy('class_acc')
# 物品分类上的平均损失
class_loss = keras.metrics.Mean('class_loss', dtype=tf.float32)
metrics = [emotion_acc, emotion_loss, class_acc, class_loss]
```

下面为训练与验证部分的核心代码，验证与训练的主要差别是无需计算梯度，并将模型训练关闭 `training=False`，这样才不会启用 Dropout 操作.

```
def test(val_ds):  # 模型测试
    for (x, y) in val_ds:
```

```
        emotion_y = tf.reshape(y[:, 0], [-1, 1])  # 情感标签
        classes_y = tf.reshape(y[:, 1], [-1, 1])  # 分类标签
        emotion, classes = classifier_model(x, training=False)
        loss1 = binary_loss(emotion_y, emotion)
        loss2 = multi_loss(classes_y, classes)

        emotion_acc.update_state(emotion_y, emotion)
        emotion_loss.update_state(loss1)
        class_acc.update_state(classes_y, classes)
        class_loss.update_state(loss2)

def train():  # 模型训练
    global optimizer  # 为了能修改全局变量
    for step, (x, y) in tqdm(enumerate(train_ds)):
        emotion_y = tf.reshape(y[:, 0], [-1, 1])  # 情感标签
        classes_y = tf.reshape(y[:, 1], [-1, 1])  # 分类标签
        with tf.GradientTape() as tape:
            emotion, classes = classifier_model(x, training=True)  # 预测
            loss1 = binary_loss(emotion_y, emotion)  # y=(batch, 2)
            loss2 = multi_loss(classes_y, classes)
            loss = tf.reduce_mean(loss1 + loss2)  # 将 loss 求和作为总损失
        grads = tape.gradient(loss, classifier_model.trainable_variables)
        optimizer.apply_gradients(zip(grads,
         ↪  classifier_model.trainable_variables))  # 更新网络参数

        emotion_acc.update_state(emotion_y, emotion)
        emotion_loss.update_state(loss1)
        class_acc.update_state(classes_y, classes)
        class_loss.update_state(loss2)

        if step % 100 == 0:
            save_metrics()  # 保存计数器结果
            metrics.reset_states()  # 重置计数器
            test(val_ds.take(100))  # 仅验证 100*batch_size 个数据
            draw_figure()  # 绘制结果图
            classifier_model.save_weights(ckp_save_handle)  # 保存模型权重
            print(f"Save in '{ckp_save_handle}'")

        if lr_change is not None and step == batch_N:  # 调整步长
            optimizer =
             ↪  optimizers.Adam(learning_rate=lr_change[step//batch_N])
            print(" 调整步长为", lr_change[step//batch_N])
```

模型训练参数变化图如下图所示 (epochs=1, Dropout=0.3，Seqlength=128)：




### 3.3.1 网络模型调整

我们尝试修改了输出端的网络参数，得到了以下的训练结果：(epoch=2, 在第一个 epoch 时使用 $10^{-4}$ 的步长, 在第二个 epoch 时使用 $10^{-5}$ 的步长)

1. batch=32, Dropout=0.3, Seqlength=128:

```
emotion_acc=0.978 emotion_loss=0.070 class_acc=0.952 class_loss=0.130
验证集上情感分类准确率: 94.57%
验证集上商品分类准确率: 90.63%
```

2. batch=32, Dropout=0.1, Seqlength=128:

```
emotion_acc=0.966 emotion_loss=0.095 class_acc=0.948 class_loss=0.146
验证集上情感分类准确率: 93.99%
验证集上商品分类准确率: 90.59%
```

3. batch=16, Dropout=0.3, Seqlength=256:

```
emotion_acc=0.957 emotion_loss=0.114 class_acc=0.940 class_loss=0.164
验证集上情感分类准确率: 93.26%
验证集上商品分类准确率: 89.45%
```

测试自定义评论可以使用如下方法

```
def binary_classifier(out):
    ret = []
    for item in out:
        ret.append(1 if item[0] > 0.5 else 0)
    return ret

def multi_classifier(out, num=3):  # 输出前 num 个的类别
```

```
    ret = []
    for item in out:
        classes = []
        arg = np.argsort(item.numpy())[::-1]
        for i in range(num):
            classes.append((idx2class[arg[i]],
             ↪  np.round(item[arg[i]].numpy(), 2)))
        ret.append(classes)
    return ret

x = [" 写的稀烂，太差了", " 写的真好，期待后续出版", ...]  # 自定义评论
emotion, classes = classifier_model(tf.constant(x), training=False)
emotion = binary_classifier(emotion)  # 转化为正负情感
classes = multi_classifier(classes, num=3)  # 显示排名前三的类别
for i in range(len(x)):
    print(f"\"{x[i]}\": {emotion[i]}, {classes[i]}")
""" 输出结果
" 写的稀烂，太差了": 0, [('书籍', 4.02), ('衣服', 3.67), ('酒店', 2.49)]
" 写的真好，期待后续出版": 1, [('书籍', 3.55), ('平板', 2.44), ('衣服', 2.08)]
"""
```

### 3.3.2 使用外部数据集作为验证集

我们打算使用 Amazon 商品数据集作进一步的验证，yf_amazon 数据集下载位置.

该数据集主要分为三个文件

1. rating.csv 中存储了用户评论的各种信息，包含评论具体内容与打分，可以通过 productID 找到对应的商品.

2. products.csv 中存储了每个 productID 对应的商品名称与类别编号 catIds.

3. categories.cvs 中存储了每种 catId 对应的类别名称.

使用方法:首先在 categories.cvs 中找到目标类别对应的 catId,然后在 products.csv 中找到包含该种分类编号的商品，最后从 rating.csv 中找到对应的评论. 我们期望将每种类别的文件保存到对应的文件当中.

```
amazon_df = pd.read_csv('../dataset/yf_amazon/ratings.csv')
products_df = pd.read_csv('../dataset/yf_amazon/products.csv')
cats_df = pd.read_csv('../dataset/yf_amazon/categories.csv')
cats = [[832], [642], [304], [-1], [1121], [702], [-1],
        [1100, 965, 903, 899, 875, 717, 585, 569, 534, 331, 245, 90],
         ↪  [1057], [-1]]  # 需要类别名称对应的类别编号
for cat, name in zip(cats, cat_names):
    def check(row):
        items = [int(a) for a in row['catIds'].split(',')]  # 将类别用逗号划
         ↪  分开
```

```
        for item in items:
            if item in cat:
                return True
        return False
    products = products_df[products_df.apply(check, axis=1)]['productId']
    item_df = amazon_df[amazon_df.apply(lambda row: row['productId'] in
    ↪ products, axis=1)][['rating', 'comment']]
    print(f"'{name}'类别中商品数目{products.shape[0]}, 评论数
    ↪ 目{item_df.shape[0]}")
    item_df = item_df.sort_values(by=['rating'], ascending=False,
    ↪ ignore_index=True)  # 按评分排序保存到文本中
    item_df.to_csv(f"./validation/{name}.csv", index=False)
```

每种类别对应的文件数目如下

```
'书籍'类别中商品数目 383500, 评论数目 2842292
'平板'类别中商品数目 363, 评论数目 13341
'手机'类别中商品数目 1861, 评论数目 81211
'水果'类别中商品数目 0, 评论数目 0
'洗发水'类别中商品数目 501, 评论数目 13923
'热水器'类别中商品数目 204, 评论数目 3853
'蒙牛'类别中商品数目 0, 评论数目 0
'衣服'类别中商品数目 5557, 评论数目 2332
'计算机'类别中商品数目 20112, 评论数目 359804
'酒店'类别中商品数目 0, 评论数目 0
```

我们发现服装数据中的数据有较多的书籍评论，所以将其删去. 由于对于平板和手机的评论十分类似，我们打算用预测结果中前 3 个中正确则认为预测正确. 预测代码如下所示

```
class_acc = [0, 0]  # 物品分类上的准确率

# 真实值在前 num 个中则认为正确
def update_acc(classes_y, classes, acc, num=3):
    for b in range(classes.shape[0]):
        acc[1] += 1
        pred = classes[b]
        arg = np.argsort(pred.numpy())[::-1]
        for i in range(num):
            if classes_y[b] == arg[i]:
                acc[0] += 1
                break

for (x, y) in tqdm(val_ds.batch(128)):
    emotion_y = tf.reshape(y[:, 0], [-1, 1])  # 情感标签
```

```
    classes_y = tf.reshape(y[:, 1], [-1, 1])  # 分类标签
    emotion, classes = classifier_model(x, training=False)
    emotion_acc.update_state(emotion_y, emotion)
    update_acc(classes_y, classes, class_acc)

print(f" 情感分类准确率: {emotion_acc.result().numpy():.2%}")
print(f" 商品分类准确率: {class_acc[0]/class_acc[1]:.2%}")
print(f" 总计验证数目: {emotion_acc.count.numpy()}")
```

最终测试如下三个模型在 Amazon 验证集上的正确率：
`bert_classifier_epoch1` 参数为 batch=32, epochs=1, Seqlength=128；
`bert_classifier_epoch2` 参数为 batch=32, epochs=2, Seqlength=128；
`bert_classifier_seq_256` 参数为 batch=16, epochs=2, Seqlength=256.

```
bert_classifier_epoch1
情感分类准确率: 73.95%
商品分类准确率: 84.04%

bert_classifier_epoch2
情感分类准确率: 78.74%
商品分类准确率: 81.75%

bert_classifier_seq_256
情感分类准确率: 76.38%
商品分类准确率: 83.71%
```

# 4 结论与讨论

全部代码均已上传至 GitHub 仓库：大作业代码，主要 Jupyter 代码main.ipynb，脚本代码main.py，可共部署的代码：Bert-Chinese-Text-Classifier-Tensorflow2.

通过本次实验我们学习了如何使用神经网络进行 NLP 的分类问题，首先我们学习了 Bert 的基本原理，与神经网络的编码基础知识；参考官方教程，学会如何使用 Bert 预训练模型进行文本的编码，对我们要训练的数据进行划分，训练集进行填充平均，使得每种类别的数据均匀分配，并搭建用于实际分类问题的神经网络；我们尝试了不同的输出层的神经网络，调整训练的次数、每次训练使用的步长、预处理文本的最大长度，最终比对不同模型的在验证集下的准确率，并自定义输入数据进行模型检验.

本次大作业提高了我们对复杂模型神经网络模型的训练，学会了数据的各种处理方式，为以后科研打下基础.