中山大学计算机学院

人工智能

本科生实验报告

（2023学年春季学期）

课程名称：Artificial Intelligence

教学班级	学号	专业（方向）	姓名
计科1班	21307035	计算机科学与技术	邓栩瀛

一、实验题目

基于CNN/RNN的文本分类

- 使用CNN或RNN进行文本分类任务
- 词表处理， 词清洗：低频词，标点，数字等
- 预训练的词嵌入数据（glove.6b.50d），包含50维度的词嵌入
- 数据目录为20ns，其中train.txt用于训练，valid.txt是验证集，test.txt是测试集
- 通过在验证集上尝试不同的参数等来筛选准确率最好的一组参数，并将该过程记录在实验报告中

二、实验内容

1.算法原理

CNN文本分类

- 词向量化：将文本中的每个词转换为一个向量表示，每个文本看作一个词向量矩阵。
- 卷积层：输入词向量矩阵，通过多个不同大小的卷积核，对每个文本进行卷积操作，提取文本中的局部信息。其中，卷积核的大小可以根据任务需求来设定。
- 池化层：将卷积层的输出进行下采样，减少数据量和计算量。
- 全连接层：将池化层的输出展平成一维向量，通过一系列全连接层进行分类或回归。
- 激活函数：在卷积层和全连接层之后，添加激活函数，如ReLU函数，进行非线性变换，优化模型。
- Dropout：防止模型过拟合，在全连接层之间添加Dropout层，随机将一部分神经元输出置为零。
- Softmax：在最后一个全连接层之后，添加Softmax函数，将每个类别的得分转换为概率，用于分类。
- 损失函数：例如交叉熵损失函数（Cross-entropy Loss）。
- 训练：通过反向传播算法，计算损失函数对模型参数的梯度，来更新模型参数，从而最小化损失函数。

2.伪代码

初始化 `__init__` ：定义模型的各个层，包括一个嵌入层、多个卷积层、多个池化层和一个全连接层。

- `vocab_size` ：词汇表大小
- `embedding_dim` ：嵌入向量的维度
- `num_classes` ：分类数
- `kernel_sizes` 卷积核的大小
- `num_filters` 卷积核的数量

`forward` 方法：定义模型的前向传播过程。

- 输入张量 `x` ，形状为 `(batch_size, max_length)` ， `max_length` 表示输入文本的最大长度
- 首先将输入张量 `x` 传入嵌入层，再将输出的张量添加一个维度变为 `(batch_size, max_length, embedding_dim, 1)` ，再将卷积层和池化层按顺序应用到输出张量上，并将多个池化层的输出张量连接在一起，通过全连接层得到最终的分类结果

```
class TextCNN(tf.keras.Model):
```

```
    def __init__(self, vocab_size, embedding_dim, num_classes, kernel_sizes, num_filters):
        super(TextCNN, self).__init__()
        # 嵌入层
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        # 卷积层
        self.conv_layers = [tf.keras.layers.Conv1D(filters=num_filters, kernel_size=k, activation='relu')
        # 池化层
        self.pool_layers = [tf.keras.layers.MaxPooling1D() for _ in kernel_sizes]
        # 全连接层
        self.fc = tf.keras.layers.Dense(num_classes, activation='softmax')

    def forward(self, x):
        # x: (batch_size, max_length)
        # (batch_size, max_length, embedding_dim)
        x = self.embedding(x)
        # (batch_size, max_length, embedding_dim, 1)
        x = tf.expand_dims(x, axis=-1)
        result = []
        for conv, pool in zip(self.conv_layers, self.pool_layers):
            conv_out = conv(x)
            pool_out = pool(conv_out)
            out.append(pool_out)
        result = tf.concat(out, axis=-1)
        result = self.fc(out)
        return result
```

3.关键代码展示（带注释）

本项目使用的深度学习框架是 `tensorflow`

part1：CNN模型相关

模型的创建

- 嵌入层 `Embedding(vocab_size, embedding_dim, input_length=max_length, weights=[embedding_matrix], trainable=False)`
- - `vocab_size` :词汇表大小
  - `embedding_dim` :嵌入向量的维度
  - `input_length` :输入文本的最大长度
  - `weights` :预训练的嵌入矩阵
  - `trainable` :是否在训练过程中更新嵌入权重
- 两个卷积层 `Conv1D(128, 5, activation='relu')` 和 `Conv1D(64, 5, activation='relu')` ，分别包含 128 个和 64 个卷积核，每个卷积核的大小为 5，激活函数为 ReLU
- 两个最大池化层 `MaxPooling1D(5)`
- 展平层 `Flatten` ，将池化层的输出张量展平成一个一维向量
- `Dropout` 层 `Dropout(0.5)` ，随机丢弃一定比例的神经元，防止过拟合。
- 全连接层 `Dense(64, activation='relu')` ，包含 64 个神经元，激活函数为 ReLU，用于将展平后的特征向量映射到更高维度的空间中
- 输出层 `Dense(len(labels), activation='softmax')` ， `len(labels)` 表示分类数，激活函数为 `softmax` ，将全连接层的输出转换为分类概率

```
model = Sequential([
    # 嵌入层：将输入的单词序列转换为低维稠密向量表示
    Embedding(vocab_size, embedding_dim, input_length=max_length, weights=[embedding_matrix], trainable=False),
    # 卷积层1
    Conv1D(128, 5, activation='relu'),
    # 池化层1
```

```
    MaxPooling1D(5),
    # 卷积层2
    Conv1D(64, 5, activation='relu'),
    # 池化层2
    MaxPooling1D(5),
    # 展平层
    Flatten(),
    # Dropout层
    Dropout(0.5),
    # 全连接层
    Dense(64, activation='relu'),
    # 输出层
    Dense(len(labels), activation='softmax')
```

模型的编译

- 损失函数：交叉熵损失函数 `sparse_categorical_crossentropy`
- 优化器：RMSprop
- 监测指标：accuracy

```
model.compile(loss='sparse_categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

模型的训练

使用 `fit()` 函数对模型进行训练

- `train_padded` 训练集经过填充后的输入张量
- `train_targets` 训练集标签
- `batch_size` 每次训练时使用的样本大小
- `epochs` 迭代次数
- `validation_data` 用于验证模型性能的验证集数据，包括填充后的输入张量 `valid_padded` 和标签 `valid_targets`
- `callbacks` 回调函数，调用 `history` 记录训练和验证集的准确率和损失函数值。

```
model.fit(train_padded, np.array(train_targets), batch_size=batch_size, epochs=epochs,
          validation_data=(valid_padded, np.array(valid_targets)), callbacks=[history])
```

模型的评估与预测

`model.evaluate()` 用于对模型在验证集和测试集上进行评估，返回指定的评估指标

在验证集上评估模型

- `valid_padded` 填充后的验证集输入张量
- `valid_targets` 验证集标签

在测试集上验证模型

- `test_padded` 填充后的测试集输入张量
- `test_targets` 测试集的标签
- `batch_size` 每次评估时使用的样本大小

```
# 在验证集上评估模型
valid_loss, valid_accuracy = model.evaluate(valid_padded, np.array(valid_targets), batch_size=batch_size)
print("Validation accuracy: {:.2f}%".format(valid_accuracy * 100))
# 在测试集上验证模型
test_loss, test_accuracy = model.evaluate(test_padded, np.array(test_targets), batch_size=batch_size)
print("Test accuracy: {:.2f}%".format(test_accuracy * 100))
```

part2：数据预处理相关

读取预训练的词嵌入数据

返回值：字典 `embeddings_index`，字典的键为单词，值为对应的词向量。

```
def load_embeddings(embedding_file):
    embeddings_index = {} # 创建空字典
    with open(embedding_file, encoding='utf-8') as f:
        for line in f: # 逐行读取
            value = line.split() # 按空格分割
            word = value[0] # 单词
            # 词向量值转换为Numpy数组
            coef = np.asarray(value[1:], dtype='float32')
            embeddings_index[word] = coef
    return embeddings_index
```

读取标签数据

返回值：标签列表

```
def load_labels(label_file):
    with open(label_file, 'r') as f:
        # 逐行读取，使用strip()去除行末的换行符
        labels = [label.strip() for label in f.readlines()]
    return labels
```

清洗文本数据

```
def clean_text(text):
    # 将非字母、数字、标点替换成空格
    text = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ", text)
    text = re.sub(r'\d+', '', text)  # 去除数字
    text = re.sub(r'[^\w\s]', '', text)  # 去除标点
    # 去除转义字符
    text = re.sub(r"\'s", " 's", text)
    text = re.sub(r"\'ve", " 've", text)
    text = re.sub(r"n\'t", " n't", text)
    text = re.sub(r"\'re", " 're", text)
    text = re.sub(r"\'d", " 'd", text)
    text = re.sub(r"\'ll", " 'll", text)
    # 标点符号前后添加空格
    text = re.sub(r",", " , ", text)
    text = re.sub(r"!", " ! ", text)
    text = re.sub(r"\(", " ( ", text)
    text = re.sub(r"\)", " ) ", text)
    text = re.sub(r"\?", " ? ", text)
    # 将多余的空格替换成单个空格
    text = re.sub(r"\s{2,}", " ", text)
    # 去除首尾的空格，并全部转换为小写后返回
    return text.strip().lower()
```

输入数据的预处理

返回值：两个列表 `texts` 和 `targets`，分别为文本和对应标签

```
def load_data(file, labels):
    texts = [] # 存储文本数据
    targets = [] # 存储标签信息
    with open(file, 'r', encoding='utf-8') as f:
        for line in f: # 逐行读取
            label, text = line.strip().split('\t') # 分隔标签和文本
            text = clean_text(text) # 清洗文本
            texts.append(text) # 将文本添加到texts列表
            # 将标签转换为对应的索引，并添加到targets中
            targets.append(labels.index(label))
    return texts, targets
```

将文本转换为数字序列

- 创建一个 `Tokenizer` 对象，指定 `num_words` 参数为词汇表大小， `oov_token` 参数为词汇表外的单词使用的标记。
- 使用 `fit_on_texts()` 函数将训练集文本转换成数字序列并训练分词器
- 使用 `texts_to_sequences()` 函数将训练集、验证集和测试集的文本数据转换成数字序列
- 对于每个文本， `texts_to_sequences()` 方法会将其分词并将每个单词映射成一个整数。如果单词不在词汇表中，则将其被标记为 `oov_token`
- `texts_to_sequences()` 返回一个整数序列列表，每个整数序列对应一个文本数据

```
# 创建分词器
tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(train_texts)

# 文本转为序列
train_sequences = tokenizer.texts_to_sequences(train_texts) # 训练集
valid_sequences = tokenizer.texts_to_sequences(valid_texts) # 验证集
test_sequences = tokenizer.texts_to_sequences(test_texts) # 测试集
```

序列填充

使用 `pad_sequences` 函数，将序列填充或截断到相同长度 `max_length`

- `padding_type` 填充方式， `pre` （在序列前面填充）或 `post` （在序列后面填充）
- `trunc_type` 截断方式， `pre` （从序列前面截断）或 `post` （从序列后面截断）

```
train_padded = pad_sequences(train_sequences, maxlen=max_length, padding=padding_type, truncating=trunc_type)
valid_padded = pad_sequences(valid_sequences, maxlen=max_length, padding=padding_type, truncating=trunc_type)
test_padded = pad_sequences(test_sequences, maxlen=max_length, padding=padding_type, truncating=trunc_type)
```

创建词嵌入矩阵

- 创建矩阵 `embedding_matrix` ，存储词汇表中每个单词对应的词向量
- - `vocab_size` ：词汇表的大小
  - `embedding_dim` ：每个词向量的维度
- 词汇表中的单词 `word`
- - 单词的索引 `i` 小于 `vocab_size` ，尝试从预训练好的词向量中获取该单词对应的词向量 `embedding_vector`
  - 如果预训练的词向量中存在该单词的词向量，则存储在 `embedding_matrix` 的第 `i` 行，如果没有，则将第 `i` 行设为0向量
- `embedding_matrix` 中的每一行对应词汇表中的一个单词，值为单词对应的词向量

```
embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, i in tokenizer.word_index.items():
    if i < vocab_size:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

三、实验结果及分析

1.实验结果展示示例

本案例中的最优模型

- 使用向量维度为300的嵌入词 `glove.6B.300d.txt`
- 两层卷积层，两层池化层

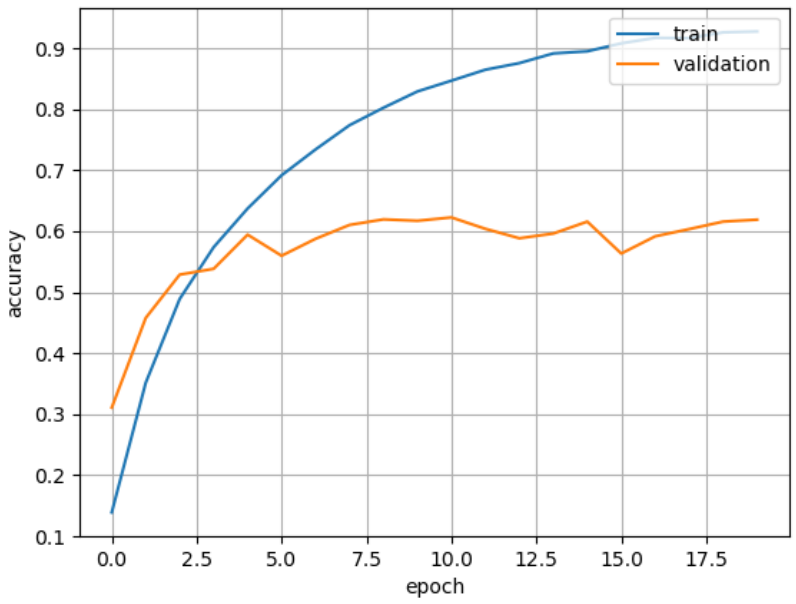
```
    Conv1D(128, 5, activation='relu'),
    MaxPooling1D(5),
    Conv1D(64, 5, activation='relu'),
    MaxPooling1D(5),
    Flatten(),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(len(labels), activation='softmax')  # 全连接层
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 200, 300)          1500000

 conv1d (Conv1D)             (None, 196, 128)          192128

 max_pooling1d (MaxPooling1D  (None, 39, 128)          0
 )

 conv1d_1 (Conv1D)           (None, 35, 64)            41024

 max_pooling1d_1 (MaxPooling  (None, 7, 64)            0
 1D)

 flatten (Flatten)           (None, 448)               0

 dropout (Dropout)           (None, 448)               0

 dense (Dense)               (None, 64)                28736

 dense_1 (Dense)             (None, 20)                1300

=================================================================
```

- 填充序列长度 `max_length=200`
- 迭代次数 `epoch=20`
- 损失函数 `sparse_categorical_crossentropy`
- 优化器 `RMSprop`
- 每次评估时的样本大小 `batch_size=32`

模型的训练效果

```
69/69 [==============================] - 0s 6ms/step - loss: 2.3057 - accuracy: 0.6190
Validation accuracy: 61.90%

229/229 [==============================] - 1s 6ms/step - loss: 2.4183 - accuracy: 0.6117
Test accuracy: 61.17%
processing time: 73.690666 seconds
```

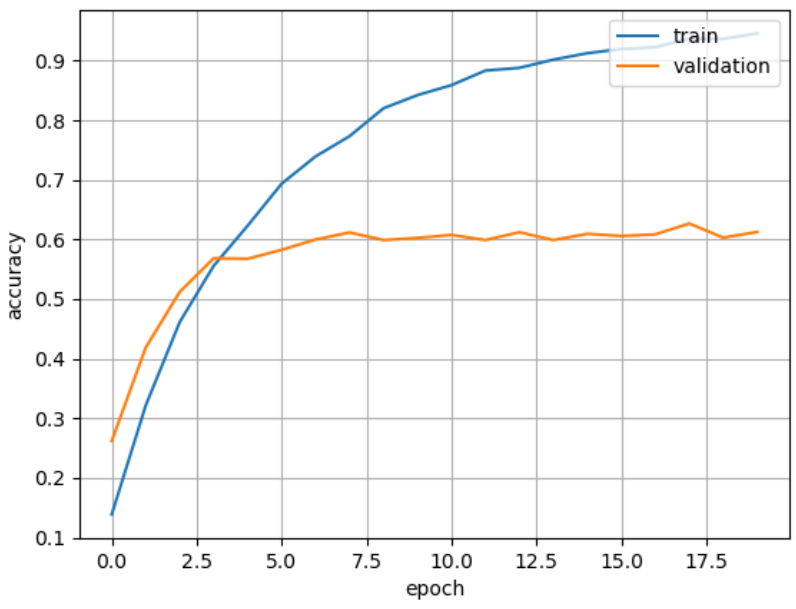


2.评测指标展示及分析

基准模型

- 使用向量维度为300的词嵌入数据
- 两层卷积层，两层池化层
- 填充序列长度为200
- epoch=20
- 损失函数 `sparse_categorical_crossentropy`
- 优化器 `Adam`

```
69/69 [==============================] - 0s 6ms/step - loss: 2.2415 - accuracy: 0.6127
Validation accuracy: 61.27%

229/229 [==============================] - 1s 6ms/step - loss: 2.3120 - accuracy: 0.6091
Test accuracy: 60.91%
processing time: 65.044593 seconds
```



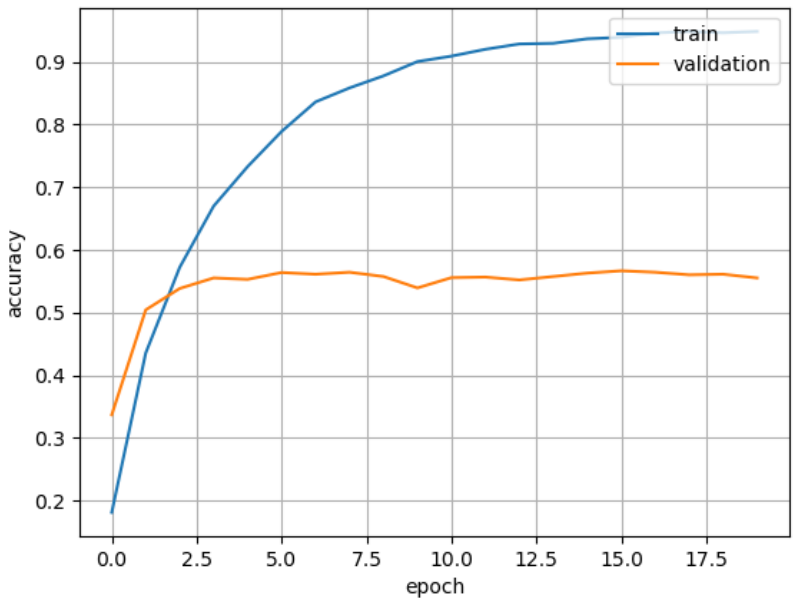
使用不同的卷积层的对比

使用单层卷积层和单层池化层

```
    Conv1D(128, 5, activation='relu'),
    MaxPooling1D(5),
    Flatten(),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(len(labels), activation='softmax')
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 200, 300)	1500000
conv1d (Conv1D)	(None, 196, 128)	192128
max_pooling1d (MaxPooling1D)	(None, 39, 128)	0
flatten (Flatten)	(None, 4992)	0
dropout (Dropout)	(None, 4992)	0
dense (Dense)	(None, 64)	319552
dense_1 (Dense)	(None, 20)	1300

```
69/69 [==============================] - 0s 6ms/step - loss: 2.2212 - accuracy: 0.5553
Validation accuracy: 55.53%

229/229 [==============================] - 1s 6ms/step - loss: 2.2489 - accuracy: 0.5685
Test accuracy: 56.85%
processing time: 61.209846 seconds
```

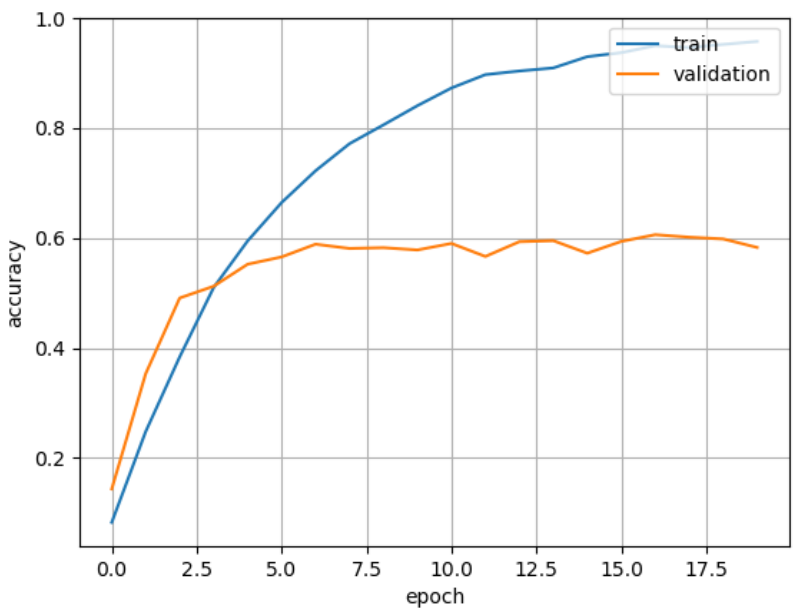


使用三层卷积层和三层池化层

```
    Conv1D(256, 5, activation='relu'),
    MaxPooling1D(3),
    Conv1D(128, 5, activation='relu'),
    MaxPooling1D(3),
    Conv1D(64, 5, activation='relu'),
    MaxPooling1D(3),
    Flatten(),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(len(labels), activation='softmax')
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 200, 300)          1500000

 conv1d (Conv1D)             (None, 196, 256)          384256

 max_pooling1d (MaxPooling1D  (None, 65, 256)          0
 )

 conv1d_1 (Conv1D)           (None, 61, 128)           163968

 max_pooling1d_1 (MaxPooling  (None, 20, 128)          0
 1D)

 conv1d_2 (Conv1D)           (None, 16, 64)            41024

 max_pooling1d_2 (MaxPooling  (None, 5, 64)            0
 1D)

 flatten (Flatten)           (None, 320)               0

 dropout (Dropout)           (None, 320)               0

 dense (Dense)               (None, 64)                20544

 dense_1 (Dense)             (None, 20)                1300

=================================================================
```

```
69/69 [==============================] - 1s 9ms/step - loss: 3.2349 - accuracy: 0.5831
Validation accuracy: 58.31%

229/229 [==============================] - 2s 9ms/step - loss: 3.4106 - accuracy: 0.5923
Test accuracy: 59.23%
processing time: 71.84473200000001 seconds
```

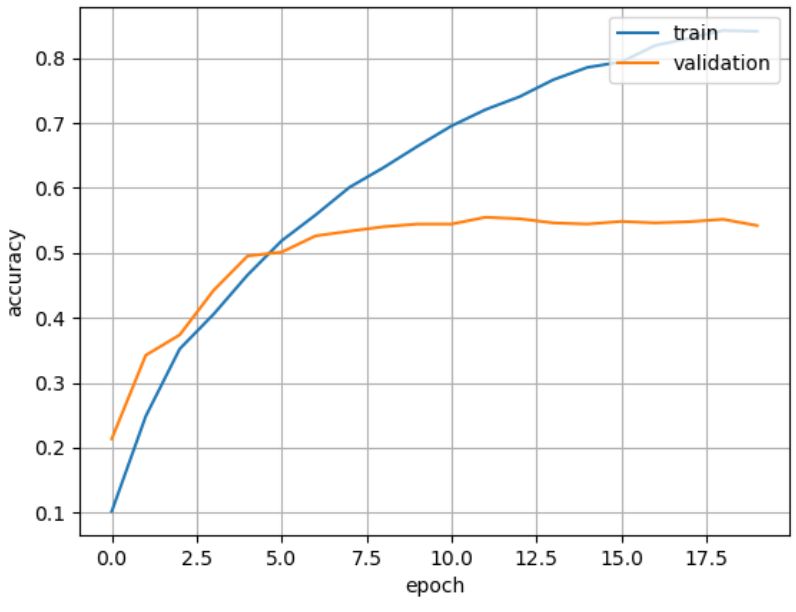


总体上来看，三层的训练效果要优于单层的训练效果，但是不如两层的训练效果好，实际上训练效果都比较接近

使用不同向量维度的词嵌入数据的对比

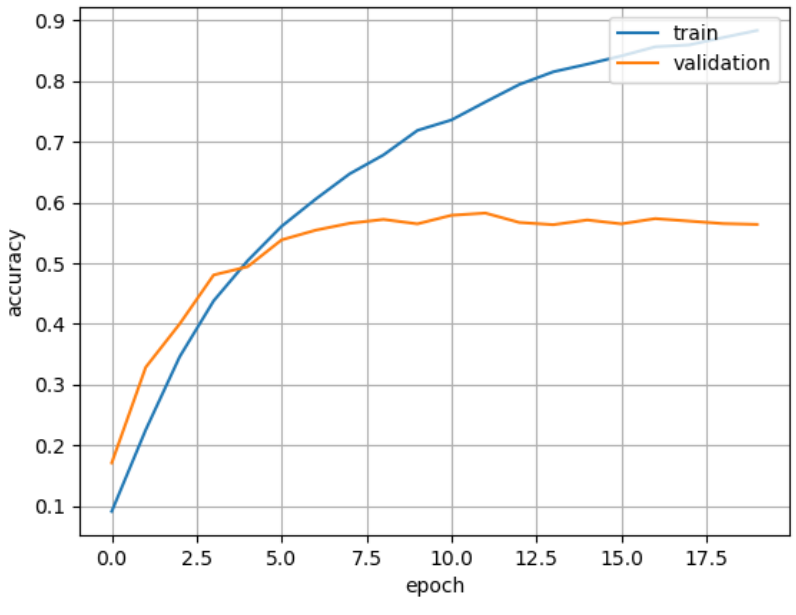
`glove.6B.50d.txt`

```
69/69 [==============================] - 0s 5ms/step - loss: 1.8899 - accuracy: 0.5421
Validation accuracy: 54.21%

229/229 [==============================] - 1s 5ms/step - loss: 1.9176 - accuracy: 0.5388
Test accuracy: 53.88%
processing time: 61.101999 seconds
```



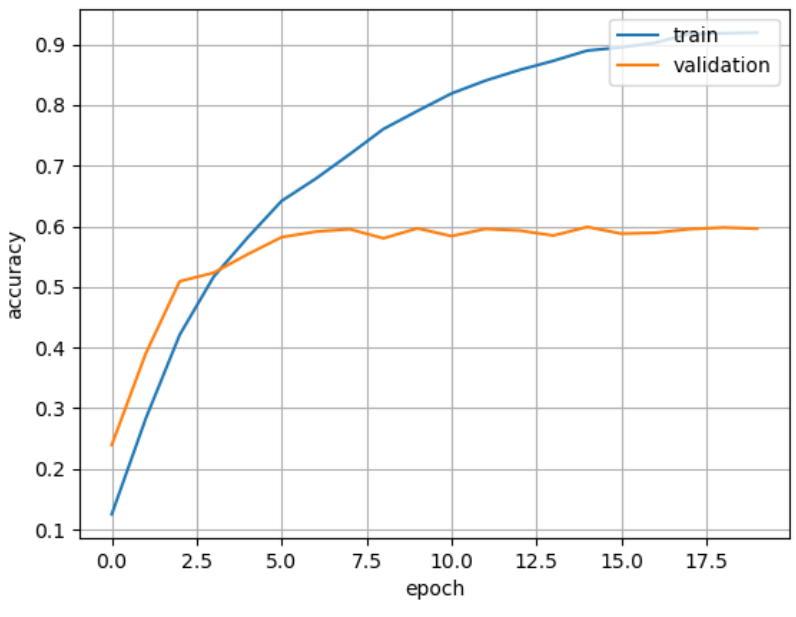
`glove.6B.100d.txt`

```
69/69 [==============================] - 0s 5ms/step - loss: 1.8473 - accuracy: 0.5640
Validation accuracy: 56.40%
229/229 [==============================] - 1s 5ms/step - loss: 1.9100 - accuracy: 0.5689
Test accuracy: 56.89%
processing time: 62.24169200000001 seconds
```



`glove.6B.200d.txt`

```
69/69 [==============================] - 0s 6ms/step - loss: 2.0733 - accuracy: 0.5963
Validation accuracy: 59.63%
229/229 [==============================] - 1s 6ms/step - loss: 2.0977 - accuracy: 0.5972
Test accuracy: 59.72%
processing time: 63.836767 seconds
```

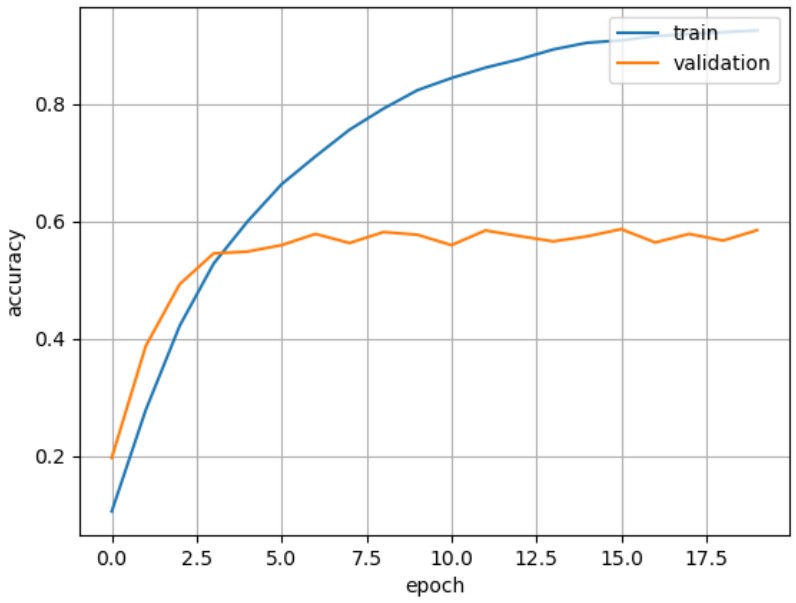


随着向量维度增加，验证集的正确率不断提高，但提升的幅度比较小

不同填充序列长度的对比

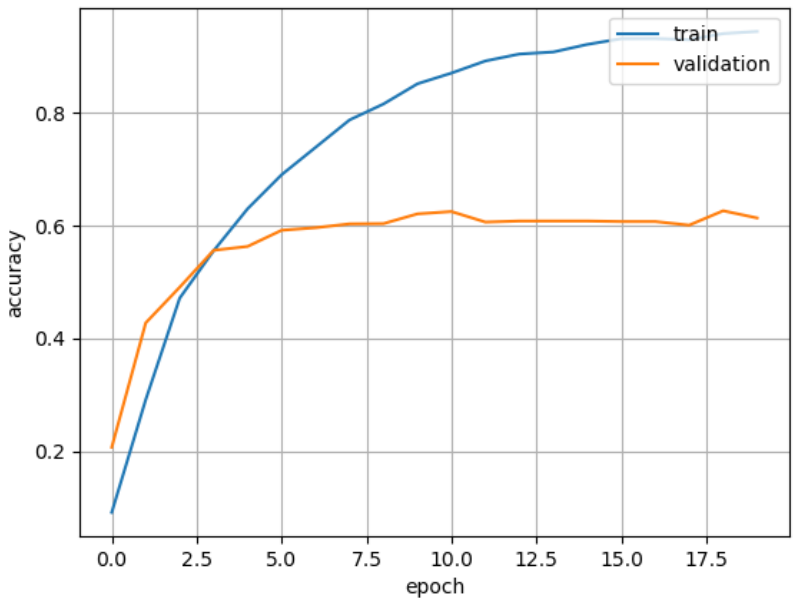
`max_length=100`

```
69/69 [==============================] - 0s 7ms/step - loss: 2.3153 - accuracy: 0.5853
Validation accuracy: 58.53%

229/229 [==============================] - 2s 8ms/step - loss: 2.3915 - accuracy: 0.5722
Test accuracy: 57.22%
processing time: 67.568326 seconds
```



`max_length=300`

```
69/69 [==============================] - 1s 7ms/step - loss: 2.1912 - accuracy: 0.6136
Validation accuracy: 61.36%

229/229 [==============================] - 2s 7ms/step - loss: 2.3347 - accuracy: 0.5908
Test accuracy: 59.08%
processing time: 66.378976 seconds
```



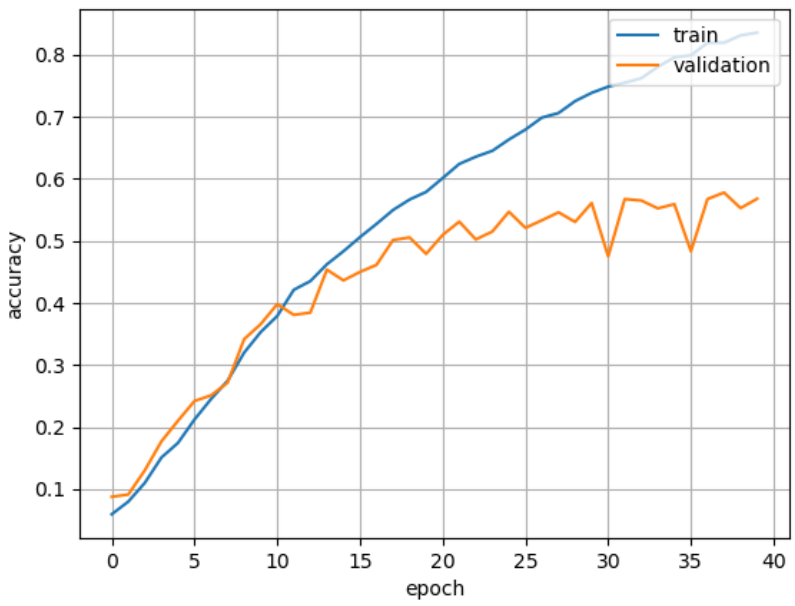
随着填充序列长度的增加，模型的训练效果有一定的提升，但填充序列长度增加到一定程度以后，模型的训练效果几乎没有变化

不同优化器的对比

SGD随机梯度下降

$$
\begin{gathered}
w_{t+1}=w_t-\eta \nabla_w J\left(w_t ; x_i, y_i\right) \\
w_t：第t个时间步的模型参数 \\
\eta：学习率 \\
J\left(w_t ; x_i, y_i\right)：损失函数 \\
\nabla_w J\left(w_t ; x_i, y_i\right)：损失函数对模型参数w的一阶导数，即模型在样本\left(x_i, y_i\right)处的梯度
\end{gathered} \tag{1}
$$

```
69/69 [==============================] - 0s 6ms/step - loss: 1.4333 - accuracy: 0.5680
Validation accuracy: 56.80%

229/229 [==============================] - 1s 6ms/step - loss: 1.4893 - accuracy: 0.5654
Test accuracy: 56.54%
processing time: 106.075675 seconds
```



当迭代次数`epoch=20`时，运行时间为

```
processing time: 56.754341999999994 seconds
```

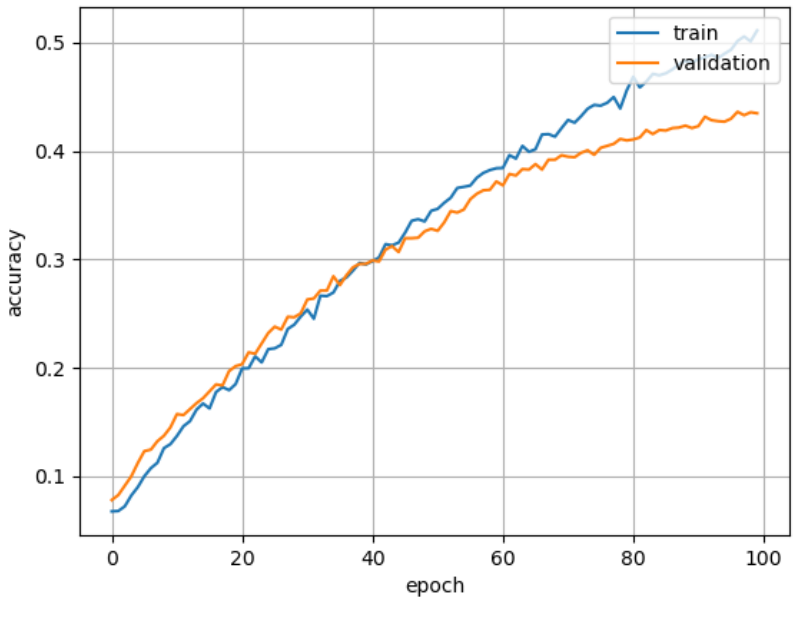
样本的梯度信息是随机的，且只考虑了单个样本的信息，因此SGD优化器通常会出现较大的更新方差和收敛过程中的震荡

SGD优化器原理比较简单，但是训练时间比较长，训练效果一般，随着迭代次数的增加，验证集的正确率变化趋于平缓

Adagrad自适应学习率优化器

$$
\begin{gathered}
g_{t, i}=\left(\nabla_{w_i} J\left(w_{t, i}\right)\right)^2 \\
w_{t+1, i}=w_{t, i}-\frac{\eta}{\sqrt{\sum_{\tau=1}^t g_{\tau, i}+\epsilon}} \nabla_{w_i} J\left(w_{t, i}\right) \\
g_{t,i}：第i个参数在前t个时间步的梯度平方和 \\
\eta：学习率 \\
\epsilon：一个小常数，避免除数为零的情况
\end{gathered} \tag{2}
$$

```
69/69 [==============================] - 0s 6ms/step - loss: 1.7135 - accuracy: 0.4347
Validation accuracy: 43.47%

229/229 [==============================] - 1s 6ms/step - loss: 1.7444 - accuracy: 0.4457
Test accuracy: 44.57%
processing time: 266.367936 seconds
```



当迭代次数 `epoch=20` 时，运行时间为

```
processing time: 59.31329000000001 seconds
```

Adagrad在累加梯度平方和时，会将之前的所有梯度平方都累加起来，从而导致学习率的分母越来越大，学习率越来越小小，使得模型在后期训练时收敛速度变慢或停滞不前

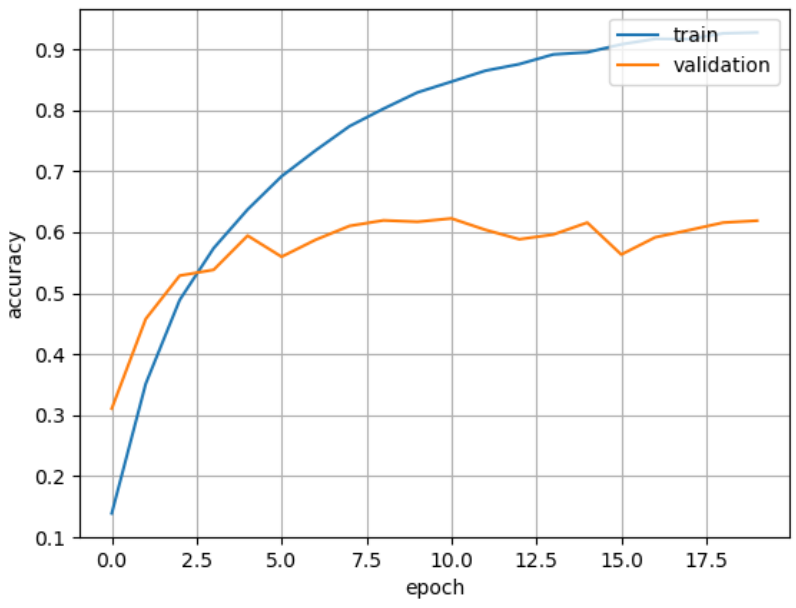
Adagrad优化器下的训练模型效果比较差，相比较而言，训练时间长，正确率低

RMSprop自适应学习率优化器

更新规则：

$$
\begin{gathered}
g_{t,i} = (\nabla_{w_i} J(w_{t,i}))^2 \\
s_{t,i} = \alpha s_{t-1,i} + (1-\alpha) g_{t,i} \\
w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{s_{t,i}} + \epsilon} \nabla_{w_i} J(w_{t,i}) \\
g_{t,i}：第i个参数在前t个时间步的梯度平方和 \\
s_{t,i}：第i个参数在前t个时间步的梯度平方和的指数加权移动平均 \\
\alpha：移动平均的衰减因子 \\
\eta：学习率 \\
\epsilon：一个小常数，避免除数为零的情况
\end{gathered} \tag{3}
$$

```
69/69 [==============================] - 0s 6ms/step - loss: 2.3057 - accuracy: 0.6190
Validation accuracy: 61.90%

229/229 [==============================] - 1s 6ms/step - loss: 2.4183 - accuracy: 0.6117
Test accuracy: 61.17%
processing time: 73.690666 seconds
```



在训练初期，梯度较大，学习率会相应地减小，从而减少参数的更新量；在训练后期，梯度较小，学习率会相应地增加，从而加快参数的更新速度。学习率自适应的方法可以更好地适应不同参数的更新需求，从而提高训练效果。

RMSprop在计算梯度平方和的指数加权移动平均时，会对之前的所有梯度平方进行平均，从而缓解了 Adagrad 学习率过小的问题。

虽然相比较而言，RMSprop的训练时间比较长，但是其训练效果是最好的（包括验证集和测试集）

Nadam优化器

与Adam优化器相比，Nadam优化器在处理高维非凸函数时具有更好的性能和泛化能力

$$
\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1-\beta_1) g_t \\
v_{t+1} &= \beta_2 v_t + (1-\beta_2) g_t^2 \\
m_{t+1}^* &= \frac{\beta_1}{1-\beta_1^t} m_{t+1} + \frac{1-\beta_1}{1-\beta_1^t} g_t \\
v_{t+1}^* &= \frac{\beta_2}{1-\beta_2^t} v_{t+1} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{v_{t+1}^*} + \epsilon} (\beta_1 m_{t+1}^* + \frac{(1-\beta_1) g_t}{1-\beta_1^t})
\end{aligned} \tag{4}
$$

g_t：第t个时间步的梯度

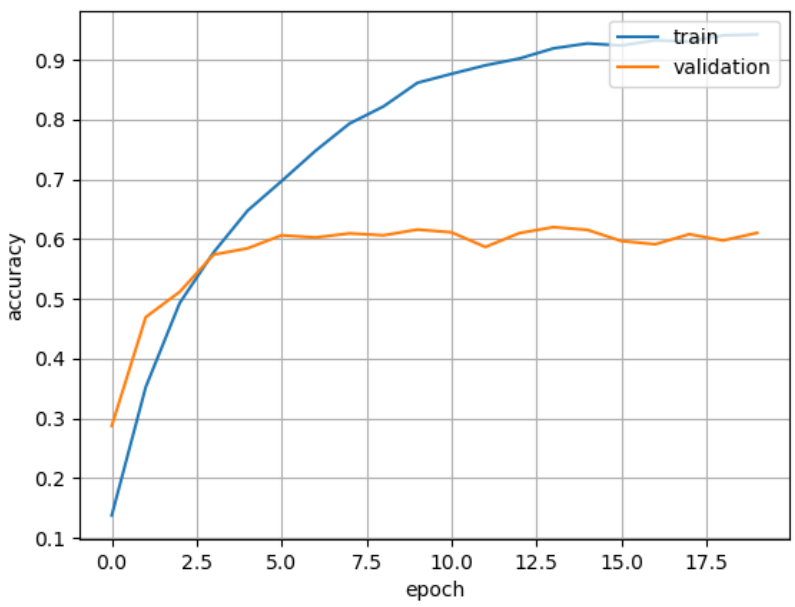
m_t和v_t：第t个时间步的梯度和梯度平方的指数加权移动平均

β_1和β_2：梯度和梯度平方的指数加权移动平均的衰减因子

η：学习率

ϵ：一个小常数，用于避免除数为零的情况。

```
69/69 [==============================] - 0s 6ms/step - loss: 2.2353 - accuracy: 0.6104
Validation accuracy: 61.04%

229/229 [==============================] - 1s 6ms/step - loss: 2.3629 - accuracy: 0.5980
Test accuracy: 59.80%
processing time: 106.53069099999999 seconds
```



Nadam优化器在计算梯度的指数加权移动平均时，使用Nesterov动量加速，以更好地利用历史梯度信息。

Nadam优化器下的训练效果良好，正确率仅仅略低于RMSprop优化器，但是训练时间是最长的