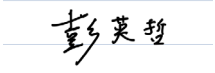
模式识别实验报告

专业：	人工智能
学号：	58119106
年级：	大二
姓名：	彭英哲

签名： 

时间：2021/6/20

实验一 隐马尔科夫模型分词任务

1. 问题描述

在人民日报分词语料库上统计语料信息，对隐马尔科夫模型进行训练。利用训练好的模型，对以下语句进行分词测试：

1）今天的天气很好。

2）学习模式识别课程是有难度的事情。

3）我是东南大学的学生。

2. 实现步骤与流程

隐马尔可夫模型 （Hidden Markov Model）是一种统计模型，用来描述一个含有隐含未知参数的**马尔可夫过程**。

隐马尔科夫模型由状态集，观测集，初始状态转移概率，状态转移概率，以及发射概率确定。

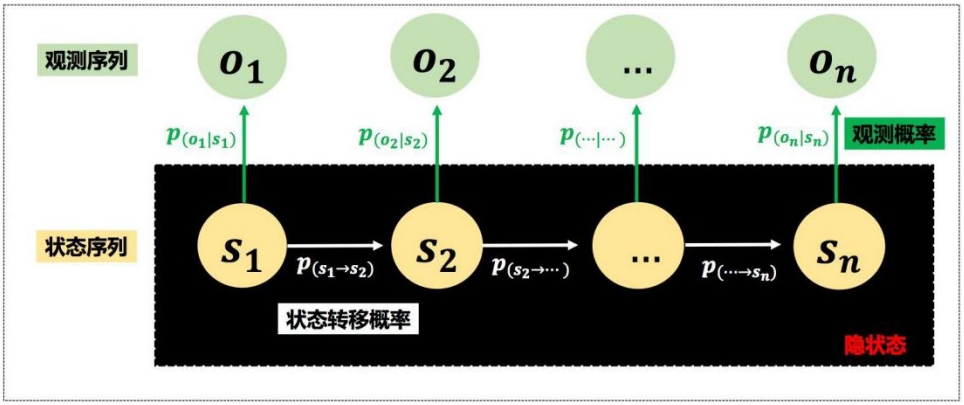


图 1. 隐马尔可夫模型图示

对于中文分词，首先要确立隐状态集合：{B,M,E,S}。四个字母分别代表一个字在，四个字母分别代表一个字在/中间/结尾/或者单字成词，这样可以将输入的中文句子编为一段状态序列，然后或者单字成词，这样可以将输入的中文句子编为一段状态序列。同时我们应当注意到：B,M后面不会接B与S，E,S后面不会有M与E。

训练时，通过统计语料库中相关信息训练 HMM 中的三个参数 PI、A 和 B。A 表示字的词位状态转移矩阵，B 表示词位到词的混淆矩阵。从语料库中可以获得每个词位出现的次数，每个字符出现的次数。再通过频率来代替概率获取三个参数的取值：

$$A_{ij}=P(Z_j/Z_i)=\frac{P(Z_i,Z_j)}{P(Z_i)}\approx\frac{freq(Z_i,Z_j)}{freq(Z_i)} \qquad (1)$$

$$B_{ij}=P(O_j/Z_i)=\frac{P(O_j,Z_i)}{P(Z_i)}\approx\frac{freq(O_j,Z_i)}{freq(Z_i)} \qquad (2)$$

$$A=\begin{bmatrix} 0.0 & p(M/B) & p(E/B) & 0.0 \\ 0.0 & p(M/M) & p(E/M) & 0.0 \\ p(B/E) & 0.0 & 0.0 & p(S/E) \\ p(B/S) & 0.0 & 0.0 & p(S/S) \end{bmatrix}$$

$$B=\begin{bmatrix} p(o_1/B) & p(o_2/B) & \cdots & p(o_n/B) \\ p(o_1/M) & p(o_2/M) & \cdots & p(o_n/M) \\ p(o_1/E) & p(o_2/E) & \cdots & p(o_n/E) \\ p(o_1/S) & p(o_2/S) & \cdots & p(o_n/S) \end{bmatrix}$$

由于我们训练集的汉字总数比较多，因此为了防止下溢，应当对所有的参数进行对数取值。对于取值0的概率我们将其设置为-3.14e100。

得到模型的三个参数后，可以使用Viterbi算法来求出最有可能的状态序列，进而来进行对未知句子的分词结果。只要找到概率最大的转移路劲，我们就可以预测未知句子的分词状态。

3. 实验结果与分析

在基础要求的句子外，还测试了一些其他的句子，结果如下：

原始句子:
今天我来到了东南大学。
分词后句子:
今天/我来/到/了/东南/大学/。/

原始句子:
模式识别课程是一门有趣的课程。
分词后句子:
模式/识别/课程/是/一门/有趣/的/课程/。/

原始句子:
我认为完成本次实验是一个挑战。
分词后句子:
我/认为/完成/本次/实验/是/一个/挑战/。/

> 原始句子：
> 我们生活在一个安定团结的国家中。
> 分词后句子：
> 我们/生活/在/一个/安定/团结/的/国家/中/。/

1.

原始句子：
我今天花了整整3小时辛辛苦苦终于写出来了隐马尔可夫模型的代码！
分词后句子：
我/今天/花/了/整/整3小时/辛辛苦苦/终于/写/出来/了/隐/马尔/可夫/模型/的/代码/！/

原始句子：
人生犹如花草，容颜终将在岁月的风尘中老去。能在时光的雕刻刀下，让自己保留清晨阳光般的笑容，端庄厚重的气度，深刻内敛的内涵，那将是上苍赐予我们一生最宝贵的财富。
分词后句子：
人生/犹如/花草/，/容/颜终/将/在/岁月/的/风/尘/中/老去/。/能/在/时光/的/雕刻/刀/下/，/让/自己/保留/清晨/阳/光般/的/笑容/，/端庄/厚重/的/气度/，/深刻/内敛/的/内涵/，/那/将/是/上/苍/赐/予/我们/一生/最宝贵/的/财富/。/

原始句子：
烟水两茫茫，蒹葭复苍苍，你就是那一位伫立于水之涠的俏佳人，着一袭素裳霓裙，黛眉如远山，一双含情眸如一池碧波，秋水盈盈，潋滟妩媚，清雅逼人，皓肤若凝脂，冰肌似玉骨！
分词后句子：
烟水/两/茫茫/，蒹葭复/苍苍/，/你就/是那/一位/伫立/于水/之涠的俏佳人/，/着一/袭素/裳霓裙/，/黛/眉如/远山/，/一/双含/情眸如/一池/碧波/，/秋水/盈盈/，/潋滟妩媚/，/清雅/逼人/，/皓肤若/凝脂/，/冰/肌似/玉骨/！/

根据结果可以看出，预测的分词结果基本正确，但是对于一些四字名词大多都被分成 BEBE 的形式，比如东南大学，模式识别，安定团结等等，同时对于一些古诗词分词也存在些许问题。但是对于常见的词分词效果较好，比如“今天”，“终于”，“是”，“的”等等，这些词都被正确划分开来。

我认为出现的错误的原因有以下几点：

1） 训练集的大小较小，包含词句较少，比如东南大学与模式识别并为在文章中出现，同时考虑到训练集是从报纸文章中收集，考虑到文章类型，文言文，古诗句出现较少。导致降低了对古诗词的分词效果。

2） 对于“安定团结”文章中虽然出现，但是依旧被分成 BEBE 的形式。推测是相对于 BEBE 形式，BMME 形式的概率连乘结果更容易比 BEBE 小。因此四字短语出现概率比 BEBE 更低。

综上所述，本次实验基本达到使用的要求。如果增加训练集的数量，分词结果肯定会更加出色。

4. 代码附录

```
"""
使用已分词的文本进行训练
使用 HMM 解码算法对测试集进行分词
输出分词结果并写入 test_result.txt
58119106 彭英哲 2021.5.30
"""
import numpy as np
import pandas as pd
import re
import itertools
import os
inf = -3.14e100  # 设置 log（0）


def coding(string):
    """
    计算已分好词的字符串的隐状态序列
    :param string:已分好词的字符串
    :return:string 对应的隐状态徐磊
    """
    obs = ''
    temp = string.split(' ')
    for word in temp:
        if word == '*':
            obs += word
            continue
        if len(word) == 1:
            obs += 'S'
        elif len(word) == 2:
            obs += 'BE'
        elif len(word) > 2:
            obs += 'B' + (len(word) - 2) * 'M' + 'E'
    return obs


def div_str(path, string, sep='/'):
    """
    依据 path，对 string 进行分词
    :param sep: 字符串分割符
    :param path: string 对应的隐状态序列
    :param string: 需要分词的字符串
```

```
        :return: 分词后的字符串
        """
        result = ''
        path = path[1:]  # 由于在 decode 函数求解 path 时没有用到 path[0] 来存储，所以要舍弃第一个元素
        for char, s in zip(path, string):
            result += s
            if char == 'S' or char == 'E':  # 只会在 S，E 后才可能分词结束
                result += sep
        return result


def cal_code_data(train_data):
    """
    使用*来对每一个句子划分，计算其隐状态序列
    :param train_data:
    :return: train_data 中每一个句子对应的隐状态序列，划分的 train_data
    """
    data = train_data.split('\n')
    data = [s for s in data]
    data = ' * '.join(data)
    return coding(data), data


class HMM:
    def __init__(self, train_data):
        """
        初始化 HMM 模型
        :param train_data: 文本训练集
        """
        self.index = ['B', 'M', 'E', 'S']
        A = self.cal_A(train_data)
        B, self.word_set = self.cal_B(train_data)
        S = self.cal_S(train_data)
        self.state_num = A.shape[0]
        self.view_num = B.shape[1]

        self.col = list(B.columns)
        self.A = pd.DataFrame(A, index=self.index, columns=self.index)
        self.B = B
        self.S = pd.Series(S, index=self.index)

    def cal_A(self, train_data):
        """
```

```
85            计算状态转移矩阵
86            :param train_data: 训练集
87            :return: 状态转移矩阵
88            """
89            code_data = cal_code_data(train_data)[0]
90            kind_list = list(itertools.product(*[self.index, self.index]))
    # 生成四个隐状态的排列
91            m = []
92            for kind in kind_list:  # 计算每一种排列的出现的频率
93                pattern = '{}(?={})'.format(kind[0], kind[1])
94                pattern2 = '{}(?!,)'.format(kind[0])
95                p = len(re.findall(pattern, code_data)) / len(re.findall(pat
    tern2, code_data))
96                m.append(p)
97
98            A = np.array(m).reshape(4, 4)
99            A = np.log(A)  # 对得到的状态转移矩阵取对数
100           A[A == -np.inf] = inf  # 对log(0)取设定的负无穷
101           return A
102
103       def cal_B(self, train_data):
104           """
105           计算混淆矩阵
106           :param train_data: 训练集
107           :return: 混淆矩阵，训练集中的字符集合
108           """
109           code_data, data = cal_code_data(train_data)
110           com_data = {
111               'word': list(data.replace(' ', '')),
112               'code_data': list(code_data)
113           }
114           df = pd.DataFrame(com_data)
115           df = df[df['word'] != '*']  # 去掉设定的分隔符
116           group_all = df.groupby(['word', 'code_data'])  # 对每个组合分组
117           z = pd.DataFrame(group_all.size())
118           word_set = np.unique([word[0] for word in z.index])
119           B = pd.DataFrame(np.zeros((4, word_set.shape[0])), index=self.in
    dex, columns=word_set)
120           for i in z.index:
121               B.loc[i[1], i[0]] = z.loc[i, 0]
122           new_B = B.T.apply(lambda x: x / x.sum()).T.apply(np.log)
123           new_B[new_B == -np.inf] = inf
124           return new_B, word_set
125
```

```
126         def cal_S(self, train_data):
127             """
128             计算初始状态概率矩阵
129             :param train_data: 训练集
130             :return: 初始状态概率矩阵
131             """
132             S = pd.Series(np.zeros(4), index=self.index)
133             code_data = cal_code_data(train_data)[0]
134             title = pd.Series([s[0] for s in code_data.split('*')[:-
    1] if s != ''])
135             title = pd.DataFrame(title.value_counts())
136             for i in title.index:
137                 S[i] = title.loc[i]
138             new_S = np.log(S / S.sum())
139             new_S[new_S == -np.inf] = inf
140             return new_S
141 
142         def decode(self, V):
143             """
144             定义：HMM 的解码问题就是给定一个观测序列  ,找到最有可能（例如最能够解
    释观测序列，等等）的隐藏状态序列
145             :param V: 已知的观测序列
146             :return: S 最有可能的隐藏序列
147             """
148             time = len(V)
149             view_col = list(range(1, time + 1))
150             trellis = pd.DataFrame(np.zeros((self.state_num, time)), index=s
    elf.index, columns=view_col)
151             l = pd.Series(V, index=view_col, name='Time')
152             path = pd.DataFrame(np.zeros((self.state_num, time)), index=self
    .index, columns=view_col)
153 
154             trellis.loc[:, 1] = self.S + self.B.loc[:, self.word_set[l[1]]]
     # trellis 的初始化
155             for t in range(2, time + 1):
156                 for j in self.index:
157                     temp = trellis.loc[:, t - 1] + self.A.loc[:, j]
158                     # t-1 时刻的状态 i 的 trellis 乘状态转移参数 a_{i,j}转移致 j
    状态的概率
159                     # 得到隐状态的个数的概率结果，我们只要取最大的即可
160                     trellis.loc[j, t] = temp.max() + self.B.at[j, self.word_
    set[l[t]]]
161                     # 计算转移至 j 状态发出 t 时刻可见状态的概率
162                     path.loc[j, t] = temp.idxmax()
```

```
                    # 保存此时的状态名
            p_time = pd.Series(np.zeros(time))
            p_time[time] = trellis.loc[:, time].idxmax()
            for i in range(1, time):
                p_time[time - i] = path.loc[p_time[time - i + 1], time - i + 1]
            return p_time

        def get_index(self, string, word_set):
            """
            返回每个字符对应的索引序列
            :param string: 预测用的字符串
            :param word_set: 训练集的字符集
            :return: 字符串对应的字符序列
            """
            index = []
            for i in string:
                if i in word_set:
                    index.append(np.where(word_set == i)[0][0])
                else:  # 如果不存在字符集，则加入到字符集中，并指定为M
                    self.B.loc['M', i] = 0
                    self.word_set = np.append(self.word_set, i)
                    index.append(self.B.shape[1] - 1)
                    self.view_num = self.B.shape[1]
                    self.col = list(self.B.columns)
            return index


current_path = os.getcwd()
train_path = current_path + os.sep + 'data' + os.sep + 'RenMinData.txt_utf8'
with open(train_path, encoding='utf8')as f:
    text = f.read()
h = HMM(text)

str_list = [
    '今天我来到了东南大学。',
    '模式识别课程是一门有趣的课程。',
    '我认为完成本次实验是一个挑战。',
    '我今天花了整整 3 小时辛辛苦苦终于写出来了隐马尔可夫模型的代码！',
    '人生犹如花草，容颜终将在岁月的风尘中老去。能在时光的雕刻刀下，让自己保留清晨阳光般的笑容，端庄厚重的气度，深刻内敛的内涵，那将是上苍赐予我们一生最宝贵的财富。',
```

```
202         '烟水两茫茫，蒹葭复苍苍，你就是那一位伫立于水之湄的俏佳人，着一袭素裳霓
    裙，黛眉如远山，一双含情眸如一池碧波，秋水盈盈，潋滟妩媚，清雅逼人，皓肤若凝
    脂，冰肌似玉骨！',
203         '我们生活在一个安定团结的国家中。'
204     ]
205     for s in str_list:
206         res = div_str(h.decode(h.get_index(s, h.word_set)), s)
207         with open('test_result.txt','a')as f:
208             f.write('原始句子：'+'\n'+s+'\n'+'分词后句子：'+'\n'+res+'\n')
209             f.write('\n')
```

实验二 贝叶斯分类

1. 问题描述

基于贝叶斯分类原理，实现一个贝叶斯分类器。在训练集中进行训练，尽可能提高模型准确率，并在测试集上进行测试。在朴素贝叶斯分类模型中，当属性是连续型时，有两种方法可以计算属性的类条件概率：第一种方法是把一个连续的属性离散化，然后用相应的离散区间替换连续属性值，之后用频率去表示类条件概率，但这种方法不好控制离散区间划分的粒度；第二种方法是假设连续变量服从某种概率分布，然后使用训练数据估计分布的参数，例如可以使用高斯分布来表示连续属性的类条件概率分布，通过高斯分布估计出类条件概率。

本实验规定采用高斯分布估计类条件概率。其中，均值和方差分别用训练集的样本均值和样本方差估计。

实验报告要求对贝叶斯分类模型的过程进行推导，并计算各个属性各个类别的类条件密度（高斯分布），同时，给出测试集的预测准确率。

测试集预测结果文件需要包含每个测试样例的预测类别及分属于三个类别的概率值

2. 实现步骤与流程

模型推导：

不妨假设样本有 c 个特征，共有 n 个样本。 $x_i^{(j)}$ 表示第 i 个样本的第 j 个特征。

不同的类别用 w_i 表示

在贝叶斯分类器中，我们所期望的应该是选择最大的后验概率。即：

$$\boldsymbol{argmax}\ \ \boldsymbol{P(w_i|x_i)}$$

$$\boldsymbol{P(w_i|x_i) = \alpha P(x_i|w_i)p(w_i)}$$

$$\boldsymbol{P(w_i|x_i) = \alpha P\left(x_i^{(1)}, x_i^{(2)}, \dots, x_i^{(c)} \middle| w_i\right) p(w_i)}$$

又由于各个特征条件独立

$$\boldsymbol{P(w_i|x_i) = \alpha \prod_j P(x_i^{(j)}|w_i)p(w_i)}$$

同时，我们认为每个特征的类条件概率密度服从高斯分布，用 MLE 可以计算出其均值与方差的参数。

$$\boldsymbol{P(w_i|x_i) = \alpha \prod_j \frac{1}{\sqrt{(2\pi {\sigma_i}^2)}} exp - \frac{(x-\mu_i)^2}{2{\sigma_i}^2} p(w_i)}$$

所以对不同的类别求其后验概率再取其最大的后验概率的那一个类别我们就可以完成分类为了计算上式

1） 首先要计算不同类别的不同特征的均值与方差。

2） 再计算出每一个类别的先验概率。

3） 测试时，只要对每一个类别的每一个特征的概率密度相乘再乘以先验我们就可以计算出其后验概率的相对大小。再对计算出来的值进行标准化处理即可。我们就可以算出每一个类的不同后验概率。取最大的后验概率即可。

3. 实验结果与分析

为了检测模型的性能，对训练集进行交叉验证。

```
开始交叉验证检验模型 k = 5
正确率列表:
[0.958, 1.0, 0.958, 1.0, 1.0]
正确率均值:
0.9832000000000001
开始交叉验证检验模型 k = 10
正确率列表:
[1.0, 1.0, 0.833, 1.0, 1.0, 0.917, 1.0, 1.0, 1.0, 1.0]
正确率均值:
0.975
```

图 2.进行 5 折和 10 折交叉训练结果

由交叉验证可以发现模型的性能十分好，能取得 97%以上的正确率。说明模型的泛化能力较好。

```
测试集准确率 0.9482758620689655
文件保存完成
```

图 3.贝叶斯分类测试准确度

程序运行得到，测试集正确率达到 94.8%，表现良好。

在对结果标准化后。保留三位小数，存储到 **test_prediction.csv 文件中：**

1	1	1	0	0
2	1	1	0	0
3	1	1	0	0
4	1	1	0	0
5	1	0.999	0.001	0
6	1	1	0	0
7	1	1	0	0
8	1	1	0	0
9	1	1	0	0
10	1	1	0	0
11	1	1	0	0
12	1	1	0	0
13	1	1	0	0
14	1	1	0	0
15	1	1	0	0
16	1	1	0	0
17	1	1	0	0
18	1	1	0	0
19	1	1	0	0
20	2	0	1	0
21	2	0	1	0
22	3	0	0.166	0.834
23	2	0.001	0.999	0
24	2	0.001	0.999	0
25	2	0	1	0
26	2	0.012	0.988	0
27	1	0.937	0.063	0
28	2	0	1	0
29	2	0	0.991	0.009
30	2	0	1	0
31	2	0	0.757	0.243
32	2	0.153	0.847	0
33	2	0	1	0
34	1	0.908	0.092	0
35	2	0.002	0.998	0
36	2	0	1	0
37	2	0	1	0
38	2	0	1	0

图 4.测试结果

39	2	0	1	0
40	2	0	1	0
41	3	0	0.002	0.998
42	3	0	0	1
43	3	0	0	1
44	3	0	0	1
45	3	0	0	1
46	3	0	0	1
47	3	0	0	1
48	3	0	0	1
49	3	0	0	1
50	3	0	0.013	0.987
51	3	0	0.017	0.983
52	3	0	0.001	0.999
53	3	0	0	1
54	3	0	0	1
55	3	0	0	1
56	3	0	0	1
57	3	0	0	1
58	3	0	0	1

图 5.测试结果（续）

4. 代码附录

```
"""
使用 wine 葡萄酒数据集进行贝叶斯分类
并对测试集进行预测
输出预测结果以及相关的概率写入 test_prediction.csv
58119106 彭英哲 2021.5.31
"""

import pandas as pd
import numpy as np
import os

current_path = os.getcwd()
train_path = current_path + os.sep + 'data/train_data.csv'
test_path = current_path + os.sep + 'data/test_data.csv'


class Bayes:
    def __init__(self, train_x, train_y):
        """
        初始化模型
```

```
        :param train_x:训练集的特征，pandas.DataFrame 格式
        :param train_y:训练集的标签，pandas.Series 格式
        """
        self.train_x = train_x
        self.train_y = train_y
        self.pdf = {}  # 计算每一个类别不同特征的条件概率密度
        self.labels = train_y.unique()
        for label in self.labels:
            index = train_y[train_y == label].index
            self.pdf[label] = {
                'mean': train_x.loc[index].mean(),
                'std': train_x.loc[index].std()
            }
        self.pri = self.cal_pri(self.labels) #计算不同类别的先验概率

    def cal_pri(self, data_label):
        """
        计算每个类别的先验概率
        :param data_label:训练集所有标签
        :return: 所有标签的先验概率
        """
        label_num = []
        for attr_label in self.labels:
            label_num.append(data_label[data_label == attr_label].shape[0])
        return pd.DataFrame(label_num, index=self.labels) # 将标签设置为index

    def predict(self, data):
        """
        预测未知数据
        :param data:需要预测的样本矩阵，pandas.DataFrame 格式
        :return: 1)预测对应的标签
                 2）分属于 3 个类别的概率
        """
        pred = []
        all_prob = []
        for sample in data.index:
            prob = pd.Series(np.zeros_like(self.labels), index=self.labels)
            x = data.loc[sample]
            for label in self.labels:
                total_p = 1
```

```
                for attr in self.train_x.columns:
                    total_p *= Gaussian(mu=self.pdf[label]['mean'][attr], gamma=self.pdf[label]['std'][attr], x=x[attr])
                    # 乘以每一个特征对于attr（label）的pdf
                prob.loc[label] = total_p * self.pri.loc[label, 0]
                #最后再乘以先验概率
            pred.append(prob.idxmax())
            all_prob.append(prob)
        return pred, all_prob

    def score(self, x, y):
        """
        计算预测的正确率
        :param x: 预测所用的样本
        :param y: 预测数据对于标签
        :return: 准确率
        """
        pred = self.predict(x)[0]
        res = [i for i in range(len(pred)) if pred[i] == y[i]]
        return len(res) / len(pred)


def Gaussian(mu, gamma, x):
    """
    计算 1 维高斯分布的概率
    :param mu: 均值
    :param gamma: 方差
    :param x: 输入样本的特征
    :return: 高斯分布的对应概率
    """
    return 1 / np.sqrt(2 * np.pi * gamma ** 2) * np.exp(-np.square(x - mu) / 2 / np.square(gamma))


def load_data(data_path):
    """
    加载数据，并分成标签和特征矩阵
        ！！数据的标签在第一列，后面 13 全为特征！！
        :param data_path: 数据对应地址,str
        :return: 返回整个数据矩阵，特征矩阵，所有标签
        """
        df = pd.read_csv(data_path, header=None, names=['label'] + [i for i in range(13)])
```

```
        x = df.iloc[:, 1:]
        y = df.loc[:, 'label']
        return df, x, y


    train_x, train_y = load_data(train_path)[1:]
    model = Bayes(train_x, train_y)
    test_x, test_y = load_data(test_path)[1:]
    print('测试集准确率', model.score(test_x, test_y))

    pred, prob = model.predict(test_x)
    pred = pd.DataFrame(pred)
    prob = pd.DataFrame(prob).apply(lambda x: x / x.sum(), axis=1
   ).apply(lambda x: round(x, 4))
    cat = pd.concat([pred, prob], axis=1)
    cat.to_csv('test_prediction.csv', header=False, index=False)
    print('文件保存完成')
```

实验三 神经网络分类任务

1. 问题描述

基于华为 AI 框架 MindSpore，构建神经网络对 CIFAR-10 数据集中的测试集进行分类。

1）基于 BP 算法，在给定训练集上使用华为 MindSpore 框架自行设计并实现神经网络模型进行训练，随后对测试集进行分类，在实验报告中记录并分析所设计网络的分类准确率等性能指标。

2）记录神经网络在一个训练轮次（epoch）中训练损失值及分类准确率随训练步数（step）的变化，绘制并保存为图表，可参考“sample_dynamics.png”，也可自行设计。

3）从测试集中随机选取若干图像，基于训练后的神经网络对该组图像的类别标签进行预测分类，将结果绘制并保存为图表，可参考“sample_predict.png”，也可自行设计。

2. 实现步骤与流程

1）配置相关的实验环境，新建一个 python 3.7.5 的虚拟环境安装 mindspore。

2）观察实验数据，CIFAR-10 数据总计 60,000 张 32×32 的 RGB 彩色图像。该数据集由训练数据集和测试数据集两部分组成，训练数据集包含 50,000 张样本图像及其类别标签，测试数据集包含 10,000 张样本图像及其类别标签。

本次实验数据有 10 个类别的图片。

3）本次实验决定使用 Lenet 网络来进行训练。

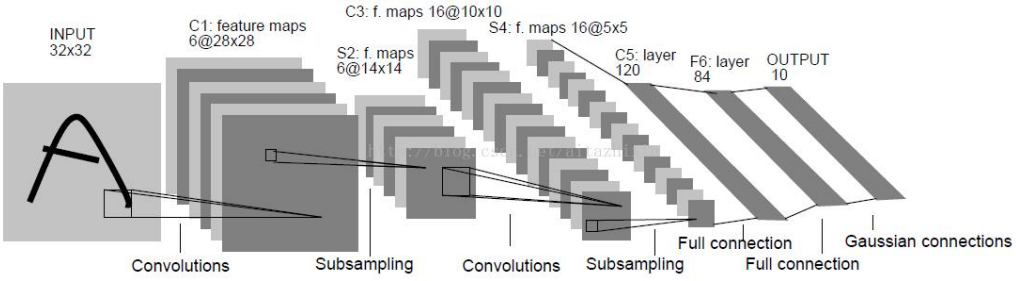


图 6.Lenet 网络结构

4）数据处理：

i. 对于标签先要将其类型转换为 int32 类型

ii. 对于图片数据，为了加快训练速度，要对其进行归一化的处理。并且为了符合网络的输入，应当转换其格式，比如通道的变化等等。同时为了获得更好的训练效果，我决定对其进行数据增强的处理。本次实验对训练集的图片采用了随机翻转和随机裁剪，同时在最后进行 shuffle 和 batch 操作。

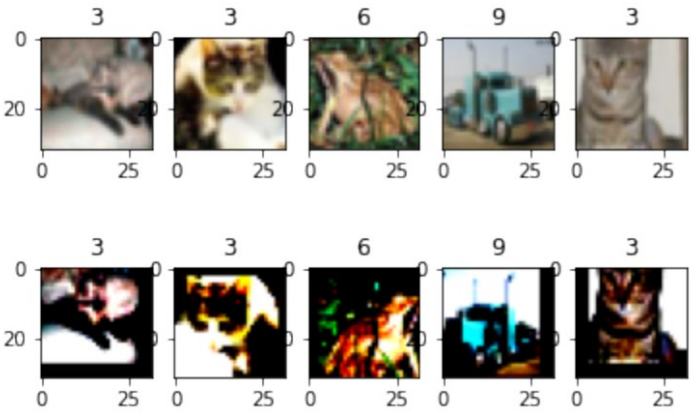


图 7.数据处理前后的照片（上：处理前，下：处理后）

5）对数据进行训练：

在训练时应当记录相关的信息，比如每个 epoch 的时间，loss 值，准确率等等，最后绘制相关的图像。

6）将训练好的网络运用到测试集查看正确率。

3. 实验结果与分析

本次实验先使用了标准 Lenet 网络进行训练，其中 epoch 取 20，learn rate 取 0.01 Batch size 取 32。

训练效果一般：

```
train acc:  {'Accuracy': 0.5573383482714469}
test acc {'Accuracy': 0.5649}
```

图 8. Lenet 网络在训练集和测试机上的准确率

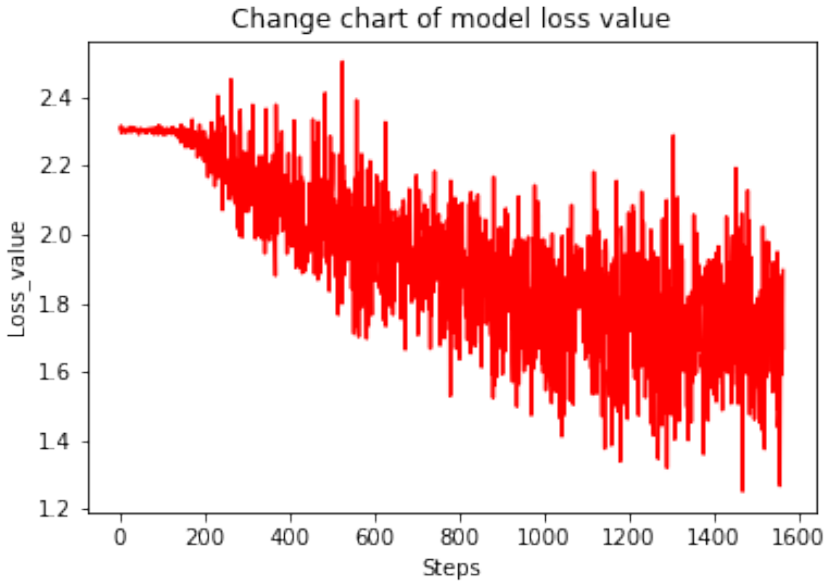


图 9. Lenet 网络其中一个 epoch 的 loss 变化

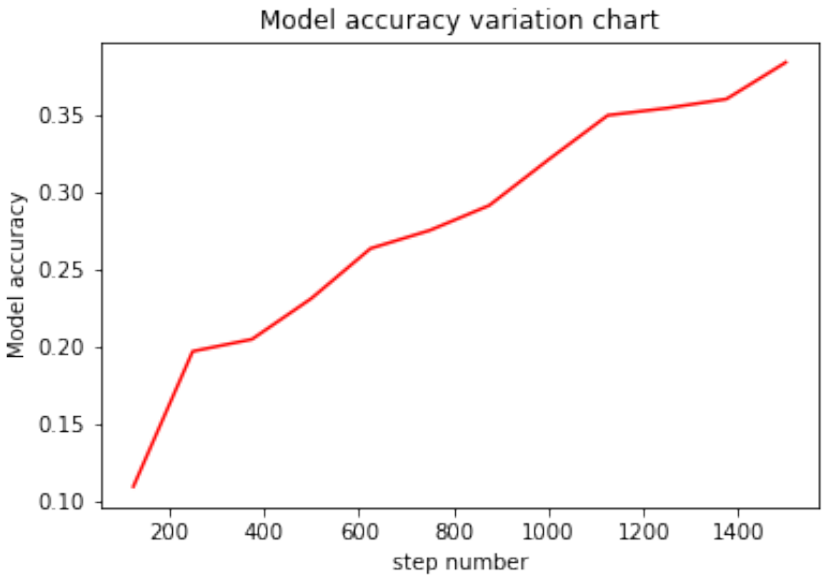


图 10. Lenet 网络其中一个 epoch 的 accuracy 变化

图 11. Lenet 网络预测例子展示

可以发现 loss 值下降幅度很小，且不太稳定，推测原因有二，一是 epoch 较小，二是 learn rate 较大。因此我决定修正 learn rate 和 epoch。降低 learn rate 的取值，同时增大训练次数。

将 learn rate 降低一个数量级，设置为 1e-3， 将 epoch 增加成 50，查看相应训练效果：

```
train acc:  {'Accuracy': 0.6761963828425096}
test acc {'Accuracy': 0.6833}
```

图 12. 改变 learn rate 和 epoch 之后的准确率

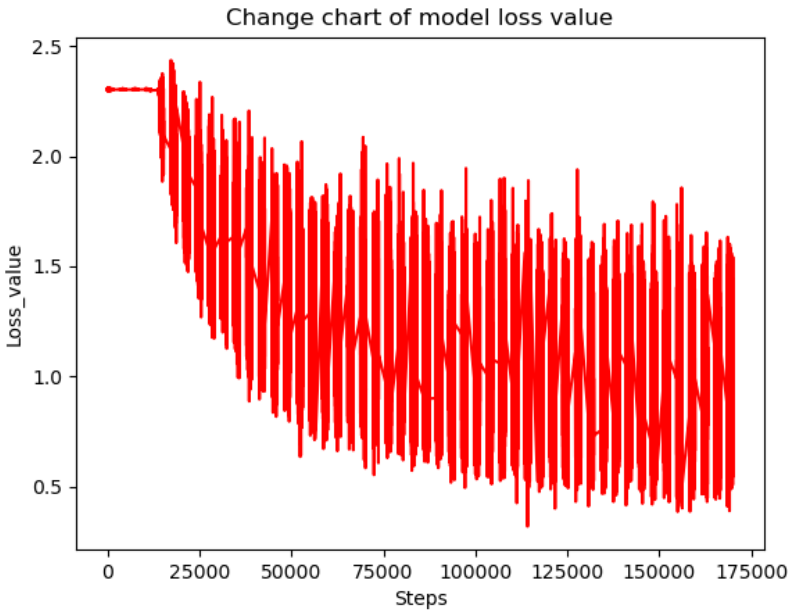


图 13. 改变 learn rate 和 epoch 之后的整个训练过程的 loss 变化值

图 14. 改变 learn rate 和 epoch 之后的预测例子展示

可以看到 loss 值有了一个下降的趋势，并且比之前的更加稳定。同时训练集和测试集的准确率得到很大的提升。

再次改变 epoch=100：

```
train acc:  {'Accuracy': 0.712187900128041}
test acc {'Accuracy': 0.7065}
```

图 15. 再次改变 learn rate 和 epoch 之后的准确率

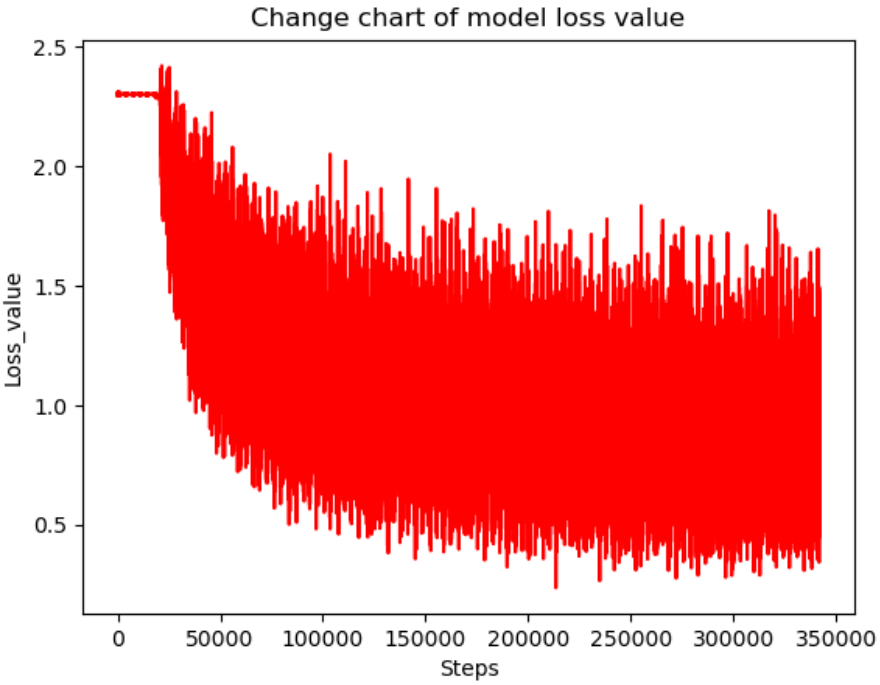


图 16. 再次改变 learn rate 和 epoch 之后的整个训练过程的 loss 变化值

图 17. 再次改变 learn rate 和 epoch 之后的预测例子展示

可以发现随着 epoch 的次数越多，准确率也随之上升，loss 减小的趋势也越明显。

除此之外，我想了解之前做的数据增强是否有效果，因此我取消了随机裁剪和随机翻转，重新训练一次模型。此时 epoch=50，learn rate = 1e-3 。

```
train acc:  {'Accuracy': 0.7052856914212549}
test acc {'Accuracy': 0.5886}
```

图 18. 取消数据增强之后的测试集和训练集准确率

从这里可以发现，模型在训练集上正确率较高，但是在测试集上模型正确率比较低。可以判断出模型存在过拟合现象，而与之前的实验结果相比较，可以发现使用的数据增强操作很好的避免了过拟合现象。

而当 epoch 继续增大，learn rate 减小时正确率提升似乎不是很明显，因此我决定更改网络架构，尝试加深神经网络的深度，并且增加卷积核的数量，从而使模型可以提取更多的特征。由于网络变复杂随之而来的就是时间的提升，因此应该避免网络过于复杂导致训练时间过长。

修改后的网络架构：

1） Input 层（32*32）

2） 卷积层，选 32 个卷积核，每个卷积核 3*3，输出 30*30*32

3） 池化层，输出 15*15*64

4） 卷积层，64 个卷积核，每个卷积核 3*3，输出 13*13*64

5） 池化层，输出 6*6*128

6） 卷积层，128 个卷积核，每个卷积核 3*3，输出 4*4*128

7） 池化层，输出 2*2*128

8） 全连接层，输出 120 个神经元

9） 全连接层，输出 80 个神经元

10） 输出：10 个神经元

修改后再次进行训练计算准确率：

```
train acc:  {'Accuracy': 0.7907130281690141}
test acc {'Accuracy': 0.7909}
```

图 19. 修改网络结构后的测试集和训练集的准确率

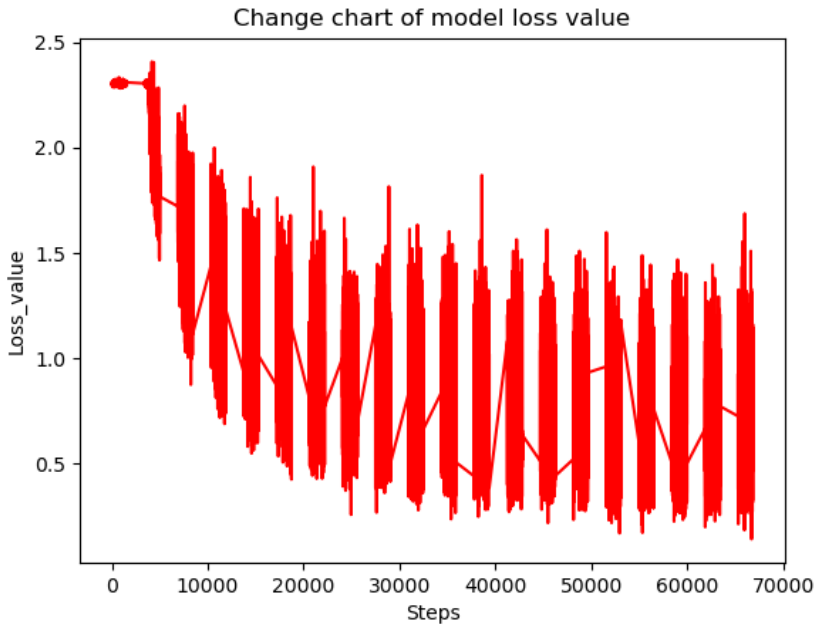


图 20. 修改网络架构后的整个训练过程 loss 变化

图 21. 修改网络架构后的预测例子展示

可以发现，加深神经网络后，模型的表现更加优良，loss 值下降程度更大，同时准确率达到了 80%，同时在测试集上与训练集正确率几乎一致，表面这个模型具有良好的泛化能力。但是同时训练速度变慢了许多，训练 20epoch 所花费时间与之前 100epoch 的原本网络所消耗时间一致。

综上，我们可以得到几个结论：

1）我们可以发现对图像进行数据处理是十分重要的，可以很好的避免模型的过拟合现象，同时对准确率的提升也有所帮助。

2）发现 learn rate 对训练的影响比较大，learn rate 较大会导致模型收敛速度变慢甚至不收敛。

3）模型网络越深，卷积核数目越多，模型的拟合效果就会越好。

4. 代码附录

train.py:

```
from netSet import *

data_path = 'cifar-10-batches-bin/train'
batch_size = 32
status = "train"
epoch = 1
learn_rate = 0.01
para = {
    'epoch': epoch,
     'lr': learn_rate
}

network = LeNet5(10)
train_data = get_data(data_path, batch_size=batch_size, status=status)
model,steps_loss,steps_eval = train_model(network, train_data, para)

test_path = './cifar-10-batches-bin/test'
batch_size = 1
status = "test"
test_data = get_data(datapath=test_path, batch_size=batch_size, status=status)
test_model(model,test_data)
pic(steps_loss,model, model_name='LeNet')
eval_show(steps_eval, model_name='LeNet')


batch_size = 32
```

```
status = "train"
epoch = 20
learn_rate = 0.001
para = {
    'epoch': epoch,
    'lr': learn_rate
}
network = my_LeNet5(10)
train_data = get_data(data_path, batch_size=batch_size, status=status)
model, steps_loss, steps_eval = train_model(network, train_data, para)

test_path = './cifar-10-batches-bin/test'
batch_size = 1
status = "test"
test_data = get_data(datapath=test_path, batch_size=batch_size, status=status)
test_model(model,test_data)
pic(steps_loss, model,model_name='my_LeNet')
eval_show(steps_eval, model_name='my_LeNet')
```

netSet.py:

```
import mindspore.dataset.transforms.c_transforms as C
import mindspore.dataset.vision.c_transforms as CV
from mindspore.common import dtype as mstype
import mindspore
import mindspore.dataset as ds
from mindspore import Tensor
import mindspore.nn as nn
from mindspore.train.callback import Callback
from mindspore.common.initializer import TruncatedNormal
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig, LossMonitor, TimeMonitor
from mindspore.train import Model
from mindspore.nn.metrics import Accuracy
import numpy as np
import matplotlib.pyplot as plt
import os


def get_data(datapath, batch_size=32, status='train'):
```

```
    """
    获取数据
    :param datapath: 数据路径
    :param batch_size: batch 大小
    :param status: train or test 对应不同的数据处理
    :return: 处理好的数据
    """
    data = ds.Cifar10Dataset(datapath)
    rescale = 1.0 / 255.0
    shift = 0.0
    rescale = CV.Rescale(rescale, shift)  # 归一化与平移
    # 对于RGB三通道分别设定mean和std
    normalize = CV.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    # 通道变化
    channel_swap = CV.HWC2CHW()
    # 类型变化
    typecast = C.TypeCast(mstype.int32)
    data = data.map(input_columns="label", operations=typecast)
    if status == "train":
        random_horizontal = CV.RandomHorizontalFlip()  # 随机翻转
        random_crop = CV.RandomCrop([32, 32], [4, 4, 4, 4])  # 随机裁剪
        data = data.map(input_columns="image", operations=random_crop)
        data = data.map(input_columns="image", operations=random_horizontal)
    data = data.map(input_columns="image", operations=rescale)
    data = data.map(input_columns="image", operations=normalize)
    data = data.map(input_columns="image", operations=channel_swap)

    # shuffle
    data = data.shuffle(buffer_size=1000)
    # 切分数据集到batch_size
    data = data.batch(batch_size, drop_remainder=True)

    return data


```

```
54. def conv_with_initialize(in_channels, out_channels, kern
   el_size, stride=1, padding=0):
55.     """
56.     返回初始化的卷积层
57.     :param in_channels: 输入通道数
58.     :param out_channels: 输出通道数
59.     :param kernel_size: 卷积核大小
60.     :param stride: stride 大小
61.     :param padding: padding 值
62.     :return: 初始化的卷积层
63.     """
64.     weight = TruncatedNormal(0.02)
65.     return nn.Conv2d(in_channels, out_channels,
66.                      kernel_size=kernel_size, stride=str
   ide, padding=padding,
67.                      weight_init=weight, has_bias=False,
    pad_mode="valid")
68.
69.
70. def fc_with_initialize(input_channels, out_channels):
71.     """
72.     返回初始化的全连接层
73.     :param input_channels: 输入个数
74.     :param out_channels: 输出个数
75.     :return: 初始化的全连接层
76.     """
77.     weight = TruncatedNormal(0.02)
78.     bias = TruncatedNormal(0.02)
79.     return nn.Dense(input_channels, out_channels, weight
   , bias)
80.
81.
82. class LeNet5(nn.Cell):
83.     def __init__(self, num_class=10, channel=3):
84.         super(LeNet5, self).__init__()
85.         self.num_class = num_class
86.         self.conv1 = conv_with_initialize(channel, 6, 5)
87.         self.conv2 = conv_with_initialize(6, 16, 5)
88.         self.fc1 = fc_with_initialize(16 * 5 * 5, 120)
89.         self.fc2 = fc_with_initialize(120, 84)
90.         self.fc3 = fc_with_initialize(84, self.num_class
   )
91.         self.relu = nn.ReLU()
```

```
92.         self.max_pool2d = nn.MaxPool2d(kernel_size=2, st
    ride=2)
93.         self.flatten = nn.Flatten()
94.
95.     def construct(self, x):
96.         x = self.max_pool2d(self.relu(self.conv1(x)))
97.         x = self.max_pool2d(self.relu(self.conv2(x)))
98.         x = self.flatten(x)
99.         x = self.relu(self.fc1(x))
100.            x = self.relu(self.fc2(x))
101.            x = self.fc3(x)
102.            return x
103.
104.
105.    class my_LeNet5(nn.Cell):
106.        def __init__(self, num_class=10, channel=3):
107.            super(my_LeNet5, self).__init__()
108.            self.num_class = num_class
109.            self.conv1 = conv_with_initialize(channel, 32
    , 3)
110.            self.conv2 = conv_with_initialize(32, 64, 3)
111.            self.conv3 = conv_with_initialize(64, 128, 3)
112.            self.fc1 = fc_with_initialize(512, 120)
113.            self.fc2 = fc_with_initialize(120, 80)
114.            self.fc3 = fc_with_initialize(80, self.num_cl
    ass)
115.            self.relu = nn.ReLU()
116.            self.max_pool2d = nn.MaxPool2d(kernel_size=2,
     stride=2)
117.            self.flatten = nn.Flatten()
118.
119.        def construct(self, x):
120.            x = self.max_pool2d(self.relu(self.conv1(x)))
121.            x = self.max_pool2d(self.relu(self.conv2(x)))
122.            x = self.max_pool2d(self.relu(self.conv3(x)))
123.            x = self.flatten(x)
124.            x = self.relu(self.fc1(x))
125.            x = self.relu(self.fc2(x))
126.            x = self.fc3(x)
127.            return x
128.
129.
130.    class StepLossAccInfo(Callback):
```

```
        def __init__(self, model, eval_dataset, steps_loss, steps_eval):
            self.model = model
            self.eval_dataset = eval_dataset
            self.steps_loss = steps_loss
            self.steps_eval = steps_eval

        def step_end(self, run_context):
            cb_params = run_context.original_args()
            cur_epoch = cb_params.cur_epoch_num
            cur_step = (cur_epoch - 1) * 1875 + cb_params.cur_step_num
            self.steps_loss["loss_value"].append(str(cb_params.net_outputs))
            self.steps_loss["step"].append(str(cur_step))
            if cur_step % 125 == 0:
                acc = self.model.eval(self.eval_dataset, dataset_sink_mode=False)
                self.steps_eval["step"].append(cur_step)
                self.steps_eval["acc"].append(acc["Accuracy"])


    def pic(steps_loss, model,model_name):
        """
        绘制 loss 变化图和预测例子图
        :param steps_loss: loss 的记录值
        :param model: 训练的模型
        :param model_name: 模型的名字
        :return: 无
        """
        steps = steps_loss["step"]
        loss_value = steps_loss["loss_value"]
        steps = list(map(int, steps))
        loss_value = list(map(float, loss_value))
        plt.plot(steps, loss_value, color="red")
        plt.xlabel("Steps")
        plt.ylabel("Loss_value")
        plt.title("Change chart of model loss value")
        plt.savefig('{}_loss.png'.format(model_name))
        plt.show()
        data_path = 'cifar-10-batches-bin/test'
        ds_test = get_data(datapath=data_path, batch_size=1, status='test').create_dict_iterator()
```

```
        for i, d in enumerate(zip(ds_test, ds.Cifar10Dataset(data_path).create_dict_iterator())):
            data = d[0]
            labels = data["label"].asnumpy()

            output = model.predict(Tensor(data['image']))
            pred = np.argmax(output.asnumpy(), axis=1)
            plt.subplot(4, 8, i + 1)
            color = 'blue' if pred == labels else 'red'
            plt.title("pre:{}".format(pred[0]), color=color, fontsize=10)
            pic = d[1]["image"].asnumpy()
            plt.imshow(pic)
            plt.axis("off")
            if i >= 31:
                break
        plt.savefig('{}_pre.png'.format(model_name))
        plt.show()


    def train_model(network, data, para):
        """
        训练模型
        :param network: 网络架构
        :param data: 训练数据
        :param para: 训练参数，dict 形式，包含 learn rate 和 epoch
        :return: 训练好的模型与训练过程中 loss 和准确率的变化
        """
        mindspore.context.set_context(mode=mindspore.context.GRAPH_MODE, device_target='CPU')
        loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
        opt = nn.Momentum(params=network.trainable_params(), learning_rate=para['lr'], momentum=0.9)
        time_cb = TimeMonitor(data_size=data.get_dataset_size())
        config_ck = CheckpointConfig(save_checkpoint_steps=1562, keep_checkpoint_max=10)
        checkpoint = ModelCheckpoint(prefix="checkpoint_lenet_original", directory='./checkpoint', config=config_ck)
        model = Model(network=network, loss_fn=loss, optimizer=opt, metrics={"Accuracy": Accuracy()})
```

```
        steps_loss = {"step": [], "loss_value": []}
        steps_eval = {"step": [], "acc": []}
        step_loss_acc_info = StepLossAccInfo(model, data, steps_loss, steps_eval)
        callback = [time_cb, checkpoint, LossMonitor(per_print_times=100), step_loss_acc_info]
        print("============== Starting Training ==========")
        model.train(epoch=para['epoch'], train_dataset=data, callbacks=callback, dataset_sink_mode=False)
        print("train acc: ", model.eval(data, dataset_sink_mode=False))
        return model,steps_loss,steps_eval


    def test_model(model, data):
        """
        测试模型
        :param model:训练好的模型
        :param data: 测试数据
        :return: 无
        """
        res = model.eval(data, dataset_sink_mode=False)
        print("test acc", res)


    def eval_show(steps_eval,model_name):
        """
        准确率变化图像绘制
        :param steps_eval:准确率的值
        :param model_name: 模型名字
        :return: 无
        """
        plt.xlabel("step number")
        plt.ylabel("Model accuracy")
        plt.title("Model accuracy variation chart")
        plt.plot(steps_eval["step"], steps_eval["acc"], "red")
        plt.savefig('{}_acc1.png'.format(model_name))
        plt.show()
```

心得体会

本次实验是第一次纯手写 PR 代码，过程虽然艰辛，但是看到自己手写的模型对数据有良好的效果，心里还是十分有成就感。尤其是在课堂上学到的知识能够得到运用，同时看到相应的效果，让这些知识对我来说不再是陌生而熟悉的。

写马尔可夫模型时，一次次观察维特比算法的计算过程，终于发现每一次循环中实际上计算的其实是两个向量的内积。再改进代码后速度也比单纯的 for 循环想加快了不少。

而在写朴素贝叶斯时，虽然原理简单，但是写代码时仍犯了不少的错误。这些看起来简单的原理在没经过一遍计算前对我来说也是一些陌生的知识，这一点也警醒我去对其他算法思想详细走一遍流程，来检验自己是否真正学懂了。

最后一个实验也是我认为最为困难的一个实验，神经网络。从我仅仅只是学习过最简单的神经网络的模型，到现在经过查阅大量的资料，研究相关的 API，学习代码的书写，最终写出来一个效果不错的模型。在这过程中，我学习到了很多知识。mindspore 也是一个易用的库。在大多数情况下我们要做的是思考模型的结构，数据的处理，而真正的训练已经被高度集成。不需要我们自己过多的关心。 虽然这次实验只是一个简单的神经网络，但是当我真正走完整个流程感觉十分自豪。看到数据增强对效果的提升让我欣喜若狂，自己修改网络结构最后产生效果更是要我激动不已。

总的来说，本次实验是一个很好的把所学知识投入到实践中的机会。除此之外，和同学的交流讨论，思维碰撞，也是一次对我很有帮助的经历。虽然花费了大量的时间，但是实验的结果和我学到的知识，并没有留下什么遗憾。感谢老师提供的这一次实践的机会！