simpleNet_PredictDistribution

2023 年 2 月 19 日

1 设计简单神经网络预测分布

1.0.1 介绍

给定数据集有五个字符的单词，希望通过神经网络学习到这些单词的对应的答对人数的分布。

1.0.2 导入相关库

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchinfo import summary
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

1.0.3 定义网络

[2]:
```
# Define the network
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(5, 50)
        self.fc2 = nn.Linear(50, 50)
        self.fc3 = nn.Linear(50, 7)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # x = torch.sigmoid(self.fc1(x))
```

```
        x = F.relu(self.fc1(x))
        # x = torch.sigmoid(self.fc2(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = self.softmax(x) # 将数据转换为概率分布
        return x
```

1.0.4 加载数据集

[3]:
```
import pandas as pd
df = pd.read_excel("Problem_C_Data_Wordle_new.xlsx")
df.head()
```

[3]:

	Date	Contest number	Word	Number of reported results	Number in hard mode	1 try	2 tries	3 tries	4 tries	5 tries	6 tries	7 or more tries (X)	normal_value
0	2022-12-31	560	manly	20380	1899	0	2	17	37	29	12	2	0.343806
1	2022-12-30	559	molar	21204	1973	0	4	21	38	26	9	1	0.491583
2	2022-12-29	558	havoc	20001	1919	0	2	16	38	30	12	2	0.097901
3	2022-12-28	557	impel	20160	1937	0	3	21	40	25	9	1	0.187709
4	2022-12-27	556	condo	20879	2012	0	2	17	35	29	14	3	0.308737

[4]:
```
# 构造输入输出数据
df_Word = df.loc[:, 'Word']
df_Word.head()
```

[4]:
```
0    manly
1    molar
2    havoc
3    impel
4    condo
Name: Word, dtype: object
```

[5]:
```
df_Distribution = df.iloc[:, 5:12]
df_Distribution.head()
```

[5]:

	1 try	2 tries	3 tries	4 tries	5 tries	6 tries	7 or more tries (X)
0	0	2	17	37	29	12	2
1	0	4	21	38	26	9	1
2	0	2	16	38	30	12	2
3	0	3	21	40	25	9	1
4	0	2	17	35	29	14	3

[6]:
```
# print shape of the data
print('shape of the df_Word is: ', df_Word.shape)
print('shape of the df_Distribution is: ', df_Distribution.shape)
```

```
shape of the df_Word is:  (359,)
shape of the df_Distribution is:  (359, 7)
```

1.0.5 转换数据格式

[7]:
```
# 将数据转换为 numpy 数组
Word = df_Word.values # 1D array (359,)
```

[8]:
```
def word2vec(word):
    # 将单词转换为向量
    # 这里由于单词长度为 5，所以将单词转换为长度为 5 的向量是很简单的
    # 而在自然语言处理中，单词的长度是不固定的，所以需要将单词转换为固定长度的向量
    # 还要做 embedding
    vec = np.zeros(5)
    for i in range(len(word)):
        vec[i] = ord(word[i]) - ord('a')
    return vec
```

```
def vec2word(vec):
    # 将向量转换为单词
    word = ''
    for i in range(len(vec)):
        word += chr(int(vec[i]) + ord('a'))
    return word
```

[9]:
```
# 我们希望将 Word 转换为 2D array , 又因为每个 word 的长度都一样所以不需要做
padding, 直接转换即可 (359,)-->(359,1)-->(359,5)
Word = np.array([word2vec(word) for word in Word])
```

[10]:
```
print('shape of the Word is: ', Word.shape)
print('the preview of the Word is:\n', Word[:5])
print('the true Word is:\n', df_Word[:5])
```

```
shape of the Word is:  (359, 5)
the preview of the Word is:
 [[12.  0. 13. 11. 24.]
 [12. 14. 11.  0. 17.]
 [ 7.  0. 21. 14.  2.]
 [ 8. 12. 15.  4. 11.]
 [ 2. 14. 13.  3. 14.]]
the true Word is:
 0    manly
1    molar
2    havoc
3    impel
4    condo
Name: Word, dtype: object
```

[11]:
```
# 对 Word 进行 normalization
# 方法一
# mean = np.mean(Word, axis=0)
# std = np.std(Word, axis=0)
# Word = (Word - mean) / std
# 方法二
Word = Word / 26
```

[12]:
```
Distribution = df_Distribution.values # 2D array (359,7)
```

[13]:
```
# 将 Distribution 转换为 float 类型，并除于 100
Distribution = Distribution.astype(np.float32)
Distribution = Distribution / 100
print('shape of the Distribution is: ', Distribution.shape)
print('the preview of the Distribution is:\n', Distribution[:5])
```

```
shape of the Distribution is:  (359, 7)
the preview of the Distribution is:
 [[0.   0.02 0.17 0.37 0.29 0.12 0.02]
 [0.   0.04 0.21 0.38 0.26 0.09 0.01]
 [0.   0.02 0.16 0.38 0.3  0.12 0.02]
 [0.   0.03 0.21 0.4  0.25 0.09 0.01]
 [0.   0.02 0.17 0.35 0.29 0.14 0.03]]
```

1.0.6 构造训练集和测试集

[14]:
```
# 分割数据
train_size = int(0.7 * len(Word))
train_X = Word[:train_size]
train_Y = Distribution[:train_size]
test_X = Word[train_size:]
test_Y = Distribution[train_size:]
```

[15]:
```
train_X = torch.from_numpy(train_X).float()
train_Y = torch.from_numpy(train_Y).float()
test_X = torch.from_numpy(test_X).float()
test_Y = torch.from_numpy(test_Y).float()
```

[16]:
```
# 构造数据迭代器
batch_size = 32


train_dataset = torch.utils.data.TensorDataset(train_X, train_Y)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
 ↪batch_size=batch_size, shuffle=True)
test_dataset = torch.utils.data.TensorDataset(test_X, test_Y)
```

```
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,␣
↪batch_size=batch_size, shuffle=False)
```

1.0.7 网络参数

[17]:
```
# 定义网络
model = SimpleNet()
# 定义损失函数
criterion = nn.MSELoss()
# 定义优化器
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

[18]:
```
summary(model, input_size=(32,5))
```

[18]:
```
==========================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
==========
SimpleNet                                [32, 7]                   --
 Linear: 1-1                             [32, 50]                  300
 Linear: 1-2                             [32, 50]                  2,550
 Linear: 1-3                             [32, 7]                   357
 Softmax: 1-4                            [32, 7]                   --
==========================================================================
==========
Total params: 3,207
Trainable params: 3,207
Non-trainable params: 0
Total mult-adds (M): 0.10
==========================================================================
==========
Input size (MB): 0.00
Forward/backward pass size (MB): 0.03
Params size (MB): 0.01
Estimated Total Size (MB): 0.04
==========================================================================
```

==========

1.0.8 训练网络

[37]:
```
# 训练网络
num_epochs = 2000
optimizer = optim.SGD(model.parameters(), lr=1e-4)
for epoch in range(num_epochs):
    for i, (word, distribution) in enumerate(train_loader):
        # 前向传播
        outputs = model(word)
        loss = criterion(outputs, distribution)
        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if (epoch + 1) % 200 == 0:
        print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch + 1, num_epochs, loss.
 ↪item()))
```

```
Epoch [200/2000], Loss: 0.0027
Epoch [400/2000], Loss: 0.0030
Epoch [600/2000], Loss: 0.0034
Epoch [800/2000], Loss: 0.0017
Epoch [1000/2000], Loss: 0.0027
Epoch [1200/2000], Loss: 0.0031
Epoch [1400/2000], Loss: 0.0031
Epoch [1600/2000], Loss: 0.0032
Epoch [1800/2000], Loss: 0.0027
Epoch [2000/2000], Loss: 0.0024
```

1.0.9 测试网络

[38]:
```
# 测试网络
model.eval()
with torch.no_grad():
    loss = 0
    for word, distribution in test_loader:
```

```
        outputs = model(word)
        loss += criterion(outputs, distribution)
    loss /= len(test_loader)
    print('Test Loss: {:.4f}'.format(loss))
```

```
Test Loss: 0.0030
```

1.0.10 加载模型

[39]:
```
#  保存模型
torch.save(model.state_dict(), 'model.ckpt')
```

[39]:
```
<All keys matched successfully>
```

[18]:
```
# 加载模型
model.load_state_dict(torch.load('model.ckpt'))
```

[18]:
```
<All keys matched successfully>
```

[19]:
```
# 预测
model.eval()
```

[19]:
```
SimpleNet(
  (fc1): Linear(in_features=5, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=50, bias=True)
  (fc3): Linear(in_features=50, out_features=7, bias=True)
  (softmax): Softmax(dim=1)
)
```

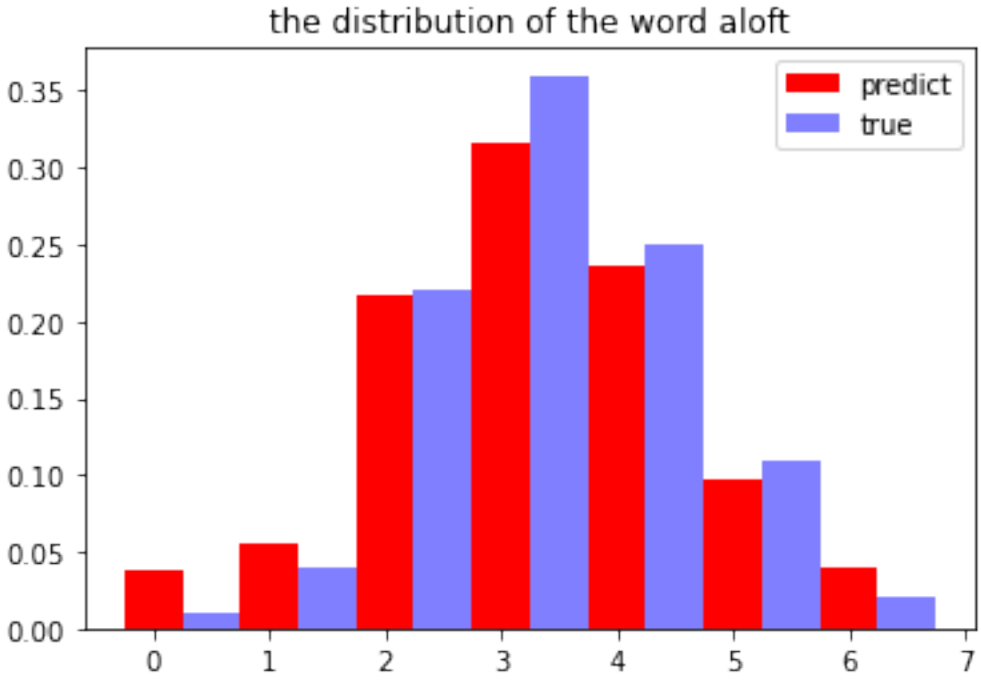
[20]:
```
idx = np.random.randint(0, len(test_X))
vec2word(test_X[idx])
```

[20]:
```
'aaaaa'
```

[61]:
```
## 从测试集中随机抽取一个单词，预测其分布
with torch.no_grad():
    # 预测单个单词
    index = np.random.randint(0, len(test_X))
    # 还原单词
    vec = test_X[index] * 26
```

```
    word = vec2word(vec)
    vec = test_X[index]
    vec = vec.view(1, -1)
    output = model(vec)
    print('the distribution of the word {} is: {}'.format(word, output))
    print('the true distribution of the word {} is: {}'.format(word,␣
↪test_Y[index]))
    # 画出预测的分布图
    plt.figure()
    plt.bar(np.arange(7), output[0].numpy(), width=0.5, color='r', ␣
↪label='predict')
    plt.bar(np.arange(7) + 0.5, test_Y[index].numpy(), width=0.5, color='b',␣
↪alpha=0.5, label='true')
    plt.title('the distribution of the word {}'.format(word))
    plt.legend()
    plt.show()
```

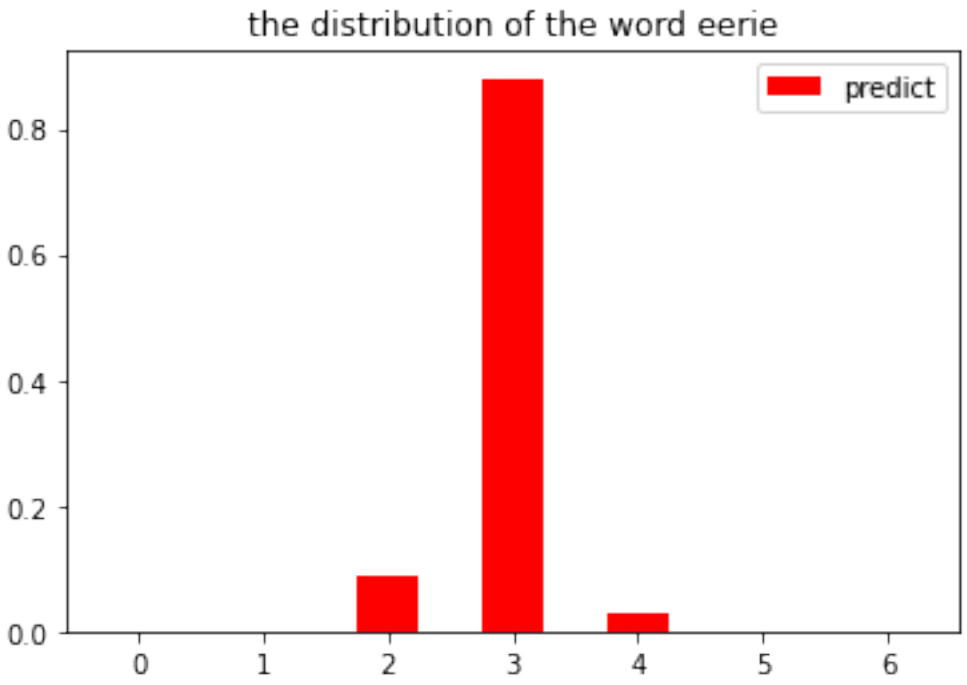
```
the distribution of the word aloft is: tensor([[0.0378, 0.0548, 0.2172, 0.3169,
0.2354, 0.0977, 0.0401]])
the true distribution of the word aloft is: tensor([0.0100, 0.0400, 0.2200,
0.3600, 0.2500, 0.1100, 0.0200])
```



1.0.11 预测单词 eerie 的分布

[69]:
```
with torch.no_grad():
    # 预测单个单词
    word = 'eerie'
    vec = word2vec(word)
    vec = torch.from_numpy(vec).float()
    vec = vec.view(1, -1)
    output = model(vec)
    print('the distribution of the word {} is: {}'.format(word, output))
    # 画出预测的分布图
    plt.figure()
    plt.bar(np.arange(7), output[0].numpy(), width=0.5, color='r', ␣
↪label='predict')
    plt.title('the distribution of the word {}'.format(word))
    plt.legend()
    plt.show()
```

```
the distribution of the word eerie is: tensor([[5.7472e-07, 3.8120e-06,
8.9551e-02, 8.8023e-01, 2.9478e-02, 7.3104e-04,
         8.9817e-07]])
```


the distribution of the word eerie
predict
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6