**《自然语言处理导论》实验报告**

学院（系）：信息科学技术学院 专业：智能科学与技术 班级：2022-2

姓 名： 孙伟豪 学号： 2220223194 实验项目：深度学习扩展实验

目录

[1 实验目的和要求 2](#_Toc184413568)

[2 实验原理和内容 2](#_Toc184413569)

[3 实验设备 2](#_Toc184413570)

[4 实验核心代码 2](#_Toc184413571)

[5 实验结果与分析 20](#_Toc184413572)

[6 实验结果讨论分析 21](#_Toc184413573)

# 实验目的和要求

使用TEXTCNN进行文本分类实验。

使用svm设计自然语言处理的一个实验：毒蘑菇分类实验。这个项目是kaggle上开源的一个分类项目，数据集质量很高。数据集：https://www.kaggle.com/datasets/uciml/mushroom-classification

使用谷歌公司开发的tansformer架构来设计实验，主要利用transformer网络进行文本翻译任务和词嵌入任务，领略下自注意力机制对于理解上下文的有效性，对于自然语言处理中做出的杰出贡献。

# 实验原理和内容

* TextCNN 是一种基于卷积神经网络（CNN）的文本分类方法，来源于传统的图像分类网络，TextCNN 中，卷积层的作用是从文本中提取局部特征，池化层帮助我们从这些局部特征中获得最重要的信息，最后通过全连接层输出分类结果。这个模型在一些文本数据，如情感分析、新闻分类表现良好。
* 支持向量机（SVM）是一种经典的监督学习算法，广泛应用于分类和回归问题。其基本思想是在高维特征空间中找到一个超平面，以最大化分类边界。LSTM 是一种改进的循环神经网络（RNN），其最大特点是能够有效地处理和学习序列数据中的长期依赖问题。传统 RNN 在处理长序列时容易出现梯度消失或梯度爆炸的问题，而 LSTM 通过引入门控机制来解决这个问题。设计毒蘑菇分类实验来验证这两个模型的优秀效果。
* Transformer是现代NLP领域的核心模型，通过自注意力机制进行设计，表现相当优秀，设计一个简单的文本翻译内容对其进行训练。

# 实验设备

在本地的3060laptop进行实验。

# 实验核心代码

* textNN文本分类

import torch

import torch.nn as nn

import torch.optim as optim

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from torch.utils.data import DataLoader, TensorDataset

from torch.nn import functional as F

import numpy as np

# 1. 数据预处理（假设我们有一个文本和标签的简单数据集）

texts = [

"I love programming",

"Deep learning is amazing",

"I hate bugs",

"Machine learning is fascinating",

"Python is a great language",

"I enjoy coding in Python",

"Algorithms are fun",

"I love solving problems",

"I hate debugging",

"Artificial intelligence is the future"

] # 文本数据

labels = ["positive", "positive", "negative", "positive", "positive", "positive", "positive", "positive", "negative",

"positive"] # 标签

# 2. 标签编码（将文本标签转化为数值）

label\_encoder = LabelEncoder()

labels = label\_encoder.fit\_transform(labels)

# 3. 文本预处理（分词和填充）

# 使用torch的Tokenizer进行文本向量化

from torchtext.data.utils import get\_tokenizer

tokenizer = get\_tokenizer('basic\_english')

sequences = [tokenizer(text) for text in texts]

vocab = set([word for seq in sequences for word in seq])

word\_to\_idx = {word: idx + 1 for idx, word in enumerate(vocab)} # 从1开始，0为填充符

word\_to\_idx['<pad>'] = 0 # 添加pad标记

# 将文本转换为整数序列

sequences\_idx = [[word\_to\_idx[word] for word in seq] for seq in sequences]

max\_len = max(len(seq) for seq in sequences\_idx) # 获取最大句子长度

# 填充序列，使得所有句子长度相同

X = [seq + [0] \* (max\_len - len(seq)) for seq in sequences\_idx] # 填充0

X = np.array(X)

# 4. 切分数据集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, labels, test\_size=0.2, random\_state=42)

# 将数据转化为tensor

X\_train\_tensor = torch.tensor(X\_train, dtype=torch.long)

X\_test\_tensor = torch.tensor(X\_test, dtype=torch.long)

y\_train\_tensor = torch.tensor(y\_train, dtype=torch.long)

y\_test\_tensor = torch.tensor(y\_test, dtype=torch.long)

# 将数据加载为DataLoader格式

train\_data = TensorDataset(X\_train\_tensor, y\_train\_tensor)

test\_data = TensorDataset(X\_test\_tensor, y\_test\_tensor)

train\_loader = DataLoader(train\_data, batch\_size=2, shuffle=True)

test\_loader = DataLoader(test\_data, batch\_size=2)

# 5. 定义TextCNN模型

class TextCNN(nn.Module):

def \_\_init\_\_(self, vocab\_size, embed\_dim, num\_classes, kernel\_sizes=[3, 4, 5], num\_filters=128):

super(TextCNN, self).\_\_init\_\_()

self.embedding = nn.Embedding(vocab\_size, embed\_dim)

# 定义多个卷积层

self.convs = nn.ModuleList([

nn.Conv2d(1, num\_filters, (K, embed\_dim)) for K in kernel\_sizes

])

# 全连接层

self.fc = nn.Linear(num\_filters \* len(kernel\_sizes), num\_classes)

def forward(self, x):

# 输入x的形状: [batch\_size, seq\_len]

x = self.embedding(x) # [batch\_size, seq\_len, embed\_dim]

x = x.unsqueeze(1) # 增加一个维度，变为 [batch\_size, 1, seq\_len, embed\_dim] 以适应卷积层

conv\_outputs = []

for conv in self.convs:

conv\_out = conv(x) # [batch\_size, num\_filters, seq\_len-K+1, 1]

conv\_out = F.relu(conv\_out).squeeze(3) # [batch\_size, num\_filters, seq\_len-K+1]

pooled\_out = F.max\_pool1d(conv\_out, conv\_out.size(2)).squeeze(2) # [batch\_size, num\_filters]

conv\_outputs.append(pooled\_out)

# 拼接所有卷积池化的输出

x = torch.cat(conv\_outputs, 1) # [batch\_size, num\_filters \* len(kernel\_sizes)]

x = self.fc(x) # [batch\_size, num\_classes]

return x

# 6. 创建模型

vocab\_size = len(word\_to\_idx) # 词汇表大小

embed\_dim = 128

num\_classes = len(np.unique(y\_train)) # 标签种类数量

model = TextCNN(vocab\_size, embed\_dim, num\_classes)

# 7. 定义损失函数和优化器

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# 8. 训练模型

epochs = 5

for epoch in range(epochs):

model.train()

running\_loss = 0.0

correct = 0

total = 0

for X\_batch, y\_batch in train\_loader:

optimizer.zero\_grad()

# 前向传播

outputs = model(X\_batch)

loss = criterion(outputs, y\_batch)

# 反向传播和优化

loss.backward()

optimizer.step()

running\_loss += loss.item()

# 计算精度

\_, predicted = torch.max(outputs, 1)

total += y\_batch.size(0)

correct += (predicted == y\_batch).sum().item()

train\_accuracy = correct / total \* 100

print(

f"Epoch [{epoch + 1}/{epochs}], Loss: {running\_loss / len(train\_loader):.4f}, Accuracy: {train\_accuracy:.2f}%")

# 9. 模型评估

model.eval()

correct = 0

total = 0

with torch.no\_grad():

for X\_batch, y\_batch in test\_loader:

outputs = model(X\_batch)

\_, predicted = torch.max(outputs, 1)

total += y\_batch.size(0)

correct += (predicted == y\_batch).sum().item()

test\_accuracy = correct / total \* 100

print(f"Test Accuracy: {test\_accuracy:.2f}%")

# 10. 预测示例

test\_text = ["I love solving new problems"]

test\_seq = tokenizer(test\_text[0])

test\_seq\_idx = [word\_to\_idx.get(word, 0) for word in test\_seq] # 未知单词使用0

test\_seq\_idx = test\_seq\_idx + [0] \* (max\_len - len(test\_seq\_idx)) # 填充到最大长度

test\_tensor = torch.tensor([test\_seq\_idx], dtype=torch.long)

model.eval()

with torch.no\_grad():

output = model(test\_tensor)

\_, pred\_label = torch.max(output, 1)

pred\_label = label\_encoder.inverse\_transform([pred\_label.item()])

print(f"Predicted Label: {pred\_label[0]}")

* svm实现毒蘑菇分类

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from sklearn.svm import SVC

import joblib

from sklearn.metrics import accuracy\_score

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import f1\_score

from sklearn.metrics import precision\_score

from sklearn.metrics import recall\_score

mushrooms=pd.read\_csv('mushroom.csv')

mushrooms.columns=['class','cap-shape','cap-surface','cap-color','ruises','odor','gill-attachment','gill-spacing','gill-size','gill-color','stalk-shape','stalk-root','stalk-surface-above-ring','stalk-surface-below-ring','stalk-color-above-ring','stalk-color-below-ring','veil-type','veil-color','ring-number','ring-type','spore-print-color','population','habitat']

#print(mushrooms)

print(mushrooms.shape)

mushrooms.isnull().sum()

mushrooms['class'].value\_counts()

# 利用众数填充缺失值

#imputer = SimpleImputer(strategy='most\_frequent')

#mushrooms\_filled = pd.DataFrame(imputer.fit\_transform(mushrooms), columns=mushrooms.columns)

mushrooms = mushrooms.fillna(mushrooms.mode().iloc[0])

from sklearn.preprocessing import OneHotEncoder

from sklearn.preprocessing import LabelEncoder

labelencoder=LabelEncoder()

onehotencoder = OneHotEncoder()

for col in mushrooms.columns:

mushrooms[col] = labelencoder.fit\_transform(mushrooms[col])

X = mushrooms.drop('class', axis=1)

y = mushrooms['class']

X = onehotencoder.fit\_transform(X).toarray()

print(X.shape)

# 将数据集拆分为训练集和测试集，按照8:2划分。

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

print(X\_train.shape)

print(X\_test.shape)

# 创建 SVM 模型并进行训练

svm = SVC()

svm.fit(X\_train, y\_train)

joblib.dump(svm, r'mushroom\_svm\_model-kuochong.pkl')

# 预测并评估模型

y\_pred = svm.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

f1 = f1\_score(y\_test, y\_pred)

confusion = confusion\_matrix(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Precision:", precision)

print("Recall:", recall)

print("F1:", f1)

print("Confusion matrix:\n", confusion)

* Transformer代码

import math

import torch

import numpy as np

import torch.nn as nn

import torch.optim as optim

import torch.utils.data as Data

device = 'cpu'

# transformer epochs

epochs = 100

# epochs = 1000

# 这里我没有用什么大型的数据集，而是手动输入了两对中文→英语的句子

# 还有每个字的索引也是我手动硬编码上去的，主要是为了降低代码阅读难度

# S: Symbol that shows starting of decoding input

# E: Symbol that shows starting of decoding output

# P: Symbol that will fill in blank sequence if current batch data size is short than time steps

# 训练集

sentences = [

# 中文和英语的单词个数不要求相同

# enc\_input dec\_input dec\_output

['我 有 零 个 好 朋 友 P', 'S I have zero good friend .', 'I have zero good friend . E'],

['我 有 零 个 女 朋 友 P', 'S I have zero girl friend .', 'I have zero girl friend . E'],

['我 有 一 个 男 朋 友 P', 'S I have a boy friend .', 'I have a boy friend . E'],

['我 有 零 个 男 朋 友 P', 'S I have zero boy friend .', 'I have zero boy friend . E'],

['我 有 一 个 好 朋 友 P', 'S I have a good friend .', 'I have a good friend . E']

]

# 测试集（希望transformer能达到的效果）

# 输入："我 有 一 个 女 朋 友"

# 输出："i have a girlfriend"

# 中文和英语的单词要分开建立词库

# Padding Should be Zero

src\_vocab = {'P': 0, '我': 1, '有': 2, '一': 3,

'个': 4, '好': 5, '朋': 6, '友': 7, '零': 8, '女': 9, '男': 10}

src\_idx2word = {i: w for i, w in enumerate(src\_vocab)}

src\_vocab\_size = len(src\_vocab)

tgt\_vocab = {'P': 0, 'I': 1, 'have': 2, 'a': 3, 'good': 4,

'friend': 5, 'zero': 6, 'girl': 7, 'boy': 8, 'S': 9, 'E': 10, '.': 11}

idx2word = {i: w for i, w in enumerate(tgt\_vocab)}

tgt\_vocab\_size = len(tgt\_vocab)

src\_len = 8 # （源句子的长度）enc\_input max sequence length

tgt\_len = 7 # dec\_input(=dec\_output) max sequence length

# Transformer Parameters

d\_model = 512 # Embedding Size（token embedding和position编码的维度）

# FeedForward dimension (两次线性层中的隐藏层 512->2048->512，线性层是用来做特征提取的），当然最后会再接一个projection层

d\_ff = 2048

d\_k = d\_v = 64 # dimension of K(=Q), V（Q和K的维度需要相同，这里为了方便让K=V）

n\_layers = 6 # number of Encoder of Decoder Layer（Block的个数）

n\_heads = 8 # number of heads in Multi-Head Attention（有几套头）

# ==============================================================================================

# 数据构建

def make\_data(sentences):

"""把单词序列转换为数字序列"""

enc\_inputs, dec\_inputs, dec\_outputs = [], [], []

for i in range(len(sentences)):

enc\_input = [[src\_vocab[n] for n in sentences[i][0].split()]]

dec\_input = [[tgt\_vocab[n] for n in sentences[i][1].split()]]

dec\_output = [[tgt\_vocab[n] for n in sentences[i][2].split()]]

# [[1, 2, 3, 4, 5, 6, 7, 0], [1, 2, 8, 4, 9, 6, 7, 0], [1, 2, 3, 4, 10, 6, 7, 0]]

enc\_inputs.extend(enc\_input)

# [[9, 1, 2, 3, 4, 5, 11], [9, 1, 2, 6, 7, 5, 11], [9, 1, 2, 3, 8, 5, 11]]

dec\_inputs.extend(dec\_input)

# [[1, 2, 3, 4, 5, 11, 10], [1, 2, 6, 7, 5, 11, 10], [1, 2, 3, 8, 5, 11, 10]]

dec\_outputs.extend(dec\_output)

return torch.LongTensor(enc\_inputs), torch.LongTensor(dec\_inputs), torch.LongTensor(dec\_outputs)

enc\_inputs, dec\_inputs, dec\_outputs = make\_data(sentences)

class MyDataSet(Data.Dataset):

"""自定义DataLoader"""

def \_\_init\_\_(self, enc\_inputs, dec\_inputs, dec\_outputs):

super(MyDataSet, self).\_\_init\_\_()

self.enc\_inputs = enc\_inputs

self.dec\_inputs = dec\_inputs

self.dec\_outputs = dec\_outputs

def \_\_len\_\_(self):

return self.enc\_inputs.shape[0]

def \_\_getitem\_\_(self, idx):

return self.enc\_inputs[idx], self.dec\_inputs[idx], self.dec\_outputs[idx]

loader = Data.DataLoader(

MyDataSet(enc\_inputs, dec\_inputs, dec\_outputs), 2, True)

# ====================================================================================================

# Transformer模型

class PositionalEncoding(nn.Module):

def \_\_init\_\_(self, d\_model, dropout=0.1, max\_len=5000):

super(PositionalEncoding, self).\_\_init\_\_()

self.dropout = nn.Dropout(p=dropout)

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)

div\_term = torch.exp(torch.arange(

0, d\_model, 2).float() \* (-math.log(10000.0) / d\_model))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0).transpose(0, 1)

self.register\_buffer('pe', pe)

def forward(self, x):

"""

x: [seq\_len, batch\_size, d\_model]

"""

x = x + self.pe[:x.size(0), :]

return self.dropout(x)

def get\_attn\_pad\_mask(seq\_q, seq\_k):

# pad mask的作用：在对value向量加权平均的时候，可以让pad对应的alpha\_ij=0，这样注意力就不会考虑到pad向量

"""这里的q,k表示的是两个序列（跟注意力机制的q,k没有关系），例如encoder\_inputs (x1,x2,..xm)和encoder\_inputs (x1,x2..xm)

encoder和decoder都可能调用这个函数，所以seq\_len视情况而定

seq\_q: [batch\_size, seq\_len]

seq\_k: [batch\_size, seq\_len]

seq\_len could be src\_len or it could be tgt\_len

seq\_len in seq\_q and seq\_len in seq\_k maybe not equal

"""

batch\_size, len\_q = seq\_q.size() # 这个seq\_q只是用来expand维度的

batch\_size, len\_k = seq\_k.size()

# eq(zero) is PAD token

# 例如:seq\_k = [[1,2,3,4,0], [1,2,3,5,0]]

# [batch\_size, 1, len\_k], True is masked

pad\_attn\_mask = seq\_k.data.eq(0).unsqueeze(1)

# [batch\_size, len\_q, len\_k] 构成一个立方体(batch\_size个这样的矩阵)

return pad\_attn\_mask.expand(batch\_size, len\_q, len\_k)

def get\_attn\_subsequence\_mask(seq):

"""建议打印出来看看是什么的输出（一目了然）

seq: [batch\_size, tgt\_len]

"""

attn\_shape = [seq.size(0), seq.size(1), seq.size(1)]

# attn\_shape: [batch\_size, tgt\_len, tgt\_len]

subsequence\_mask = np.triu(np.ones(attn\_shape), k=1) # 生成一个上三角矩阵

subsequence\_mask = torch.from\_numpy(subsequence\_mask).byte()

return subsequence\_mask # [batch\_size, tgt\_len, tgt\_len]

# ==========================================================================================

class ScaledDotProductAttention(nn.Module):

def \_\_init\_\_(self):

super(ScaledDotProductAttention, self).\_\_init\_\_()

def forward(self, Q, K, V, attn\_mask):

"""

Q: [batch\_size, n\_heads, len\_q, d\_k]

K: [batch\_size, n\_heads, len\_k, d\_k]

V: [batch\_size, n\_heads, len\_v(=len\_k), d\_v]

attn\_mask: [batch\_size, n\_heads, seq\_len, seq\_len]

说明：在encoder-decoder的Attention层中len\_q(q1,..qt)和len\_k(k1,...km)可能不同

"""

scores = torch.matmul(Q, K.transpose(-1, -2)) / \

np.sqrt(d\_k) # scores : [batch\_size, n\_heads, len\_q, len\_k]

# mask矩阵填充scores（用-1e9填充scores中与attn\_mask中值为1位置相对应的元素）

# Fills elements of self tensor with value where mask is True.

scores.masked\_fill\_(attn\_mask, -1e9)

attn = nn.Softmax(dim=-1)(scores) # 对最后一个维度(v)做softmax

# scores : [batch\_size, n\_heads, len\_q, len\_k] \* V: [batch\_size, n\_heads, len\_v(=len\_k), d\_v]

# context: [batch\_size, n\_heads, len\_q, d\_v]

context = torch.matmul(attn, V)

# context：[[z1,z2,...],[...]]向量, attn注意力稀疏矩阵（用于可视化的）

return context, attn

class MultiHeadAttention(nn.Module):

"""这个Attention类可以实现:

Encoder的Self-Attention

Decoder的Masked Self-Attention

Encoder-Decoder的Attention

输入：seq\_len x d\_model

输出：seq\_len x d\_model

"""

def \_\_init\_\_(self):

super(MultiHeadAttention, self).\_\_init\_\_()

self.W\_Q = nn.Linear(d\_model, d\_k \* n\_heads,

bias=False) # q,k必须维度相同，不然无法做点积

self.W\_K = nn.Linear(d\_model, d\_k \* n\_heads, bias=False)

self.W\_V = nn.Linear(d\_model, d\_v \* n\_heads, bias=False)

# 这个全连接层可以保证多头attention的输出仍然是seq\_len x d\_model

self.fc = nn.Linear(n\_heads \* d\_v, d\_model, bias=False)

def forward(self, input\_Q, input\_K, input\_V, attn\_mask):

"""

input\_Q: [batch\_size, len\_q, d\_model]

input\_K: [batch\_size, len\_k, d\_model]

input\_V: [batch\_size, len\_v(=len\_k), d\_model]

attn\_mask: [batch\_size, seq\_len, seq\_len]

"""

residual, batch\_size = input\_Q, input\_Q.size(0)

# 下面的多头的参数矩阵是放在一起做线性变换的，然后再拆成多个头，这是工程实现的技巧

# B: batch\_size, S:seq\_len, D: dim

# (B, S, D) -proj-> (B, S, D\_new) -split-> (B, S, Head, W) -trans-> (B, Head, S, W)

# 线性变换 拆成多头

# Q: [batch\_size, n\_heads, len\_q, d\_k]

Q = self.W\_Q(input\_Q).view(batch\_size, -1,

n\_heads, d\_k).transpose(1, 2)

# K: [batch\_size, n\_heads, len\_k, d\_k] # K和V的长度一定相同，维度可以不同

K = self.W\_K(input\_K).view(batch\_size, -1,

n\_heads, d\_k).transpose(1, 2)

# V: [batch\_size, n\_heads, len\_v(=len\_k), d\_v]

V = self.W\_V(input\_V).view(batch\_size, -1,

n\_heads, d\_v).transpose(1, 2)

# 因为是多头，所以mask矩阵要扩充成4维的

# attn\_mask: [batch\_size, seq\_len, seq\_len] -> [batch\_size, n\_heads, seq\_len, seq\_len]

attn\_mask = attn\_mask.unsqueeze(1).repeat(1, n\_heads, 1, 1)

# context: [batch\_size, n\_heads, len\_q, d\_v], attn: [batch\_size, n\_heads, len\_q, len\_k]

context, attn = ScaledDotProductAttention()(Q, K, V, attn\_mask)

# 下面将不同头的输出向量拼接在一起

# context: [batch\_size, n\_heads, len\_q, d\_v] -> [batch\_size, len\_q, n\_heads \* d\_v]

context = context.transpose(1, 2).reshape(

batch\_size, -1, n\_heads \* d\_v)

# 这个全连接层可以保证多头attention的输出仍然是seq\_len x d\_model

output = self.fc(context) # [batch\_size, len\_q, d\_model]

return nn.LayerNorm(d\_model).to(device)(output + residual), attn

# Pytorch中的Linear只会对最后一维操作，所以正好是我们希望的每个位置用同一个全连接网络

class PoswiseFeedForwardNet(nn.Module):

def \_\_init\_\_(self):

super(PoswiseFeedForwardNet, self).\_\_init\_\_()

self.fc = nn.Sequential(

nn.Linear(d\_model, d\_ff, bias=False),

nn.ReLU(),

nn.Linear(d\_ff, d\_model, bias=False)

)

def forward(self, inputs):

"""

inputs: [batch\_size, seq\_len, d\_model]

"""

residual = inputs

output = self.fc(inputs)

# [batch\_size, seq\_len, d\_model]

return nn.LayerNorm(d\_model).to(device)(output + residual)

class EncoderLayer(nn.Module):

def \_\_init\_\_(self):

super(EncoderLayer, self).\_\_init\_\_()

self.enc\_self\_attn = MultiHeadAttention()

self.pos\_ffn = PoswiseFeedForwardNet()

def forward(self, enc\_inputs, enc\_self\_attn\_mask):

"""E

enc\_inputs: [batch\_size, src\_len, d\_model]

enc\_self\_attn\_mask: [batch\_size, src\_len, src\_len] mask矩阵(pad mask or sequence mask)

"""

# enc\_outputs: [batch\_size, src\_len, d\_model], attn: [batch\_size, n\_heads, src\_len, src\_len]

# 第一个enc\_inputs \* W\_Q = Q

# 第二个enc\_inputs \* W\_K = K

# 第三个enc\_inputs \* W\_V = V

enc\_outputs, attn = self.enc\_self\_attn(enc\_inputs, enc\_inputs, enc\_inputs,

enc\_self\_attn\_mask) # enc\_inputs to same Q,K,V（未线性变换前）

enc\_outputs = self.pos\_ffn(enc\_outputs)

# enc\_outputs: [batch\_size, src\_len, d\_model]

return enc\_outputs, attn

class DecoderLayer(nn.Module):

def \_\_init\_\_(self):

super(DecoderLayer, self).\_\_init\_\_()

self.dec\_self\_attn = MultiHeadAttention()

self.dec\_enc\_attn = MultiHeadAttention()

self.pos\_ffn = PoswiseFeedForwardNet()

def forward(self, dec\_inputs, enc\_outputs, dec\_self\_attn\_mask, dec\_enc\_attn\_mask):

"""

dec\_inputs: [batch\_size, tgt\_len, d\_model]

enc\_outputs: [batch\_size, src\_len, d\_model]

dec\_self\_attn\_mask: [batch\_size, tgt\_len, tgt\_len]

dec\_enc\_attn\_mask: [batch\_size, tgt\_len, src\_len]

"""

# dec\_outputs: [batch\_size, tgt\_len, d\_model], dec\_self\_attn: [batch\_size, n\_heads, tgt\_len, tgt\_len]

dec\_outputs, dec\_self\_attn = self.dec\_self\_attn(dec\_inputs, dec\_inputs, dec\_inputs,

dec\_self\_attn\_mask) # 这里的Q,K,V全是Decoder自己的输入

# dec\_outputs: [batch\_size, tgt\_len, d\_model], dec\_enc\_attn: [batch\_size, h\_heads, tgt\_len, src\_len]

dec\_outputs, dec\_enc\_attn = self.dec\_enc\_attn(dec\_outputs, enc\_outputs, enc\_outputs,

dec\_enc\_attn\_mask) # Attention层的Q(来自decoder) 和 K,V(来自encoder)

# [batch\_size, tgt\_len, d\_model]

dec\_outputs = self.pos\_ffn(dec\_outputs)

# dec\_self\_attn, dec\_enc\_attn这两个是为了可视化的

return dec\_outputs, dec\_self\_attn, dec\_enc\_attn

class Encoder(nn.Module):

def \_\_init\_\_(self):

super(Encoder, self).\_\_init\_\_()

self.src\_emb = nn.Embedding(src\_vocab\_size, d\_model) # token Embedding

self.pos\_emb = PositionalEncoding(

d\_model) # Transformer中位置编码时固定的，不需要学习

self.layers = nn.ModuleList([EncoderLayer() for \_ in range(n\_layers)])

def forward(self, enc\_inputs):

"""

enc\_inputs: [batch\_size, src\_len]

"""

enc\_outputs = self.src\_emb(

enc\_inputs) # [batch\_size, src\_len, d\_model]

enc\_outputs = self.pos\_emb(enc\_outputs.transpose(0, 1)).transpose(

0, 1) # [batch\_size, src\_len, d\_model]

# Encoder输入序列的pad mask矩阵

enc\_self\_attn\_mask = get\_attn\_pad\_mask(

enc\_inputs, enc\_inputs) # [batch\_size, src\_len, src\_len]

enc\_self\_attns = [] # 在计算中不需要用到，它主要用来保存你接下来返回的attention的值（这个主要是为了你画热力图等，用来看各个词之间的关系

for layer in self.layers: # for循环访问nn.ModuleList对象

# 上一个block的输出enc\_outputs作为当前block的输入

# enc\_outputs: [batch\_size, src\_len, d\_model], enc\_self\_attn: [batch\_size, n\_heads, src\_len, src\_len]

enc\_outputs, enc\_self\_attn = layer(enc\_outputs,

enc\_self\_attn\_mask) # 传入的enc\_outputs其实是input，传入mask矩阵是因为你要做self attention

enc\_self\_attns.append(enc\_self\_attn) # 这个只是为了可视化

return enc\_outputs, enc\_self\_attns

class Decoder(nn.Module):

def \_\_init\_\_(self):

super(Decoder, self).\_\_init\_\_()

self.tgt\_emb = nn.Embedding(

tgt\_vocab\_size, d\_model) # Decoder输入的embed词表

self.pos\_emb = PositionalEncoding(d\_model)

self.layers = nn.ModuleList([DecoderLayer()

for \_ in range(n\_layers)]) # Decoder的blocks

def forward(self, dec\_inputs, enc\_inputs, enc\_outputs):

"""

dec\_inputs: [batch\_size, tgt\_len]

enc\_inputs: [batch\_size, src\_len]

enc\_outputs: [batch\_size, src\_len, d\_model] # 用在Encoder-Decoder Attention层

"""

dec\_outputs = self.tgt\_emb(

dec\_inputs) # [batch\_size, tgt\_len, d\_model]

dec\_outputs = self.pos\_emb(dec\_outputs.transpose(0, 1)).transpose(0, 1).to(

device) # [batch\_size, tgt\_len, d\_model]

# Decoder输入序列的pad mask矩阵（这个例子中decoder是没有加pad的，实际应用中都是有pad填充的）

dec\_self\_attn\_pad\_mask = get\_attn\_pad\_mask(dec\_inputs, dec\_inputs).to(

device) # [batch\_size, tgt\_len, tgt\_len]

# Masked Self\_Attention：当前时刻是看不到未来的信息的

dec\_self\_attn\_subsequence\_mask = get\_attn\_subsequence\_mask(dec\_inputs).to(

device) # [batch\_size, tgt\_len, tgt\_len]

# Decoder中把两种mask矩阵相加（既屏蔽了pad的信息，也屏蔽了未来时刻的信息）

dec\_self\_attn\_mask = torch.gt((dec\_self\_attn\_pad\_mask + dec\_self\_attn\_subsequence\_mask),

0).to(device) # [batch\_size, tgt\_len, tgt\_len]; torch.gt比较两个矩阵的元素，大于则返回1，否则返回0

# 这个mask主要用于encoder-decoder attention层

# get\_attn\_pad\_mask主要是enc\_inputs的pad mask矩阵(因为enc是处理K,V的，求Attention时是用v1,v2,..vm去加权的，要把pad对应的v\_i的相关系数设为0，这样注意力就不会关注pad向量)

# dec\_inputs只是提供expand的size的

dec\_enc\_attn\_mask = get\_attn\_pad\_mask(

dec\_inputs, enc\_inputs) # [batc\_size, tgt\_len, src\_len]

dec\_self\_attns, dec\_enc\_attns = [], []

for layer in self.layers:

# dec\_outputs: [batch\_size, tgt\_len, d\_model], dec\_self\_attn: [batch\_size, n\_heads, tgt\_len, tgt\_len], dec\_enc\_attn: [batch\_size, h\_heads, tgt\_len, src\_len]

# Decoder的Block是上一个Block的输出dec\_outputs（变化）和Encoder网络的输出enc\_outputs（固定）

dec\_outputs, dec\_self\_attn, dec\_enc\_attn = layer(dec\_outputs, enc\_outputs, dec\_self\_attn\_mask,

dec\_enc\_attn\_mask)

dec\_self\_attns.append(dec\_self\_attn)

dec\_enc\_attns.append(dec\_enc\_attn)

# dec\_outputs: [batch\_size, tgt\_len, d\_model]

return dec\_outputs, dec\_self\_attns, dec\_enc\_attns

class Transformer(nn.Module):

def \_\_init\_\_(self):

super(Transformer, self).\_\_init\_\_()

self.encoder = Encoder().to(device)

self.decoder = Decoder().to(device)

self.projection = nn.Linear(

d\_model, tgt\_vocab\_size, bias=False).to(device)

def forward(self, enc\_inputs, dec\_inputs):

"""Transformers的输入：两个序列

enc\_inputs: [batch\_size, src\_len]

dec\_inputs: [batch\_size, tgt\_len]

"""

# tensor to store decoder outputs

# outputs = torch.zeros(batch\_size, tgt\_len, tgt\_vocab\_size).to(self.device)

# enc\_outputs: [batch\_size, src\_len, d\_model], enc\_self\_attns: [n\_layers, batch\_size, n\_heads, src\_len, src\_len]

# 经过Encoder网络后，得到的输出还是[batch\_size, src\_len, d\_model]

enc\_outputs, enc\_self\_attns = self.encoder(enc\_inputs)

# dec\_outputs: [batch\_size, tgt\_len, d\_model], dec\_self\_attns: [n\_layers, batch\_size, n\_heads, tgt\_len, tgt\_len], dec\_enc\_attn: [n\_layers, batch\_size, tgt\_len, src\_len]

dec\_outputs, dec\_self\_attns, dec\_enc\_attns = self.decoder(

dec\_inputs, enc\_inputs, enc\_outputs)

# dec\_outputs: [batch\_size, tgt\_len, d\_model] -> dec\_logits: [batch\_size, tgt\_len, tgt\_vocab\_size]

dec\_logits = self.projection(dec\_outputs)

return dec\_logits.view(-1, dec\_logits.size(-1)), enc\_self\_attns, dec\_self\_attns, dec\_enc\_attns

model = Transformer().to(device)

# 这里的损失函数里面设置了一个参数 ignore\_index=0，因为 "pad" 这个单词的索引为 0，这样设置以后，就不会计算 "pad" 的损失（因为本来 "pad" 也没有意义，不需要计算）

criterion = nn.CrossEntropyLoss(ignore\_index=0)

optimizer = optim.SGD(model.parameters(), lr=1e-3,

momentum=0.99) # 用adam的话效果不好

# ====================================================================================================

for epoch in range(epochs):

for enc\_inputs, dec\_inputs, dec\_outputs in loader:

"""

enc\_inputs: [batch\_size, src\_len]

dec\_inputs: [batch\_size, tgt\_len]

dec\_outputs: [batch\_size, tgt\_len]

"""

enc\_inputs, dec\_inputs, dec\_outputs = enc\_inputs.to(

device), dec\_inputs.to(device), dec\_outputs.to(device)

# outputs: [batch\_size \* tgt\_len, tgt\_vocab\_size]

outputs, enc\_self\_attns, dec\_self\_attns, dec\_enc\_attns = model(

enc\_inputs, dec\_inputs)

# dec\_outputs.view(-1):[batch\_size \* tgt\_len \* tgt\_vocab\_size]

loss = criterion(outputs, dec\_outputs.view(-1))

print('Epoch:', '%04d' % (epoch + 1), 'loss =', '{:.6f}'.format(loss))

optimizer.zero\_grad()

loss.backward()

optimizer.step()

def greedy\_decoder(model, enc\_input, start\_symbol):

"""贪心编码

For simplicity, a Greedy Decoder is Beam search when K=1. This is necessary for inference as we don't know the

target sequence input. Therefore we try to generate the target input word by word, then feed it into the transformer.

Starting Reference: http://nlp.seas.harvard.edu/2018/04/03/attention.html#greedy-decoding

:param model: Transformer Model

:param enc\_input: The encoder input

:param start\_symbol: The start symbol. In this example it is 'S' which corresponds to index 4

:return: The target input

"""

enc\_outputs, enc\_self\_attns = model.encoder(enc\_input)

# 初始化一个空的tensor: tensor([], size=(1, 0), dtype=torch.int64)

dec\_input = torch.zeros(1, 0).type\_as(enc\_input.data)

terminal = False

next\_symbol = start\_symbol

while not terminal:

# 预测阶段：dec\_input序列会一点点变长（每次添加一个新预测出来的单词）

dec\_input = torch.cat([dec\_input.to(device), torch.tensor([[next\_symbol]], dtype=enc\_input.dtype).to(device)],

-1)

dec\_outputs, \_, \_ = model.decoder(dec\_input, enc\_input, enc\_outputs)

projected = model.projection(dec\_outputs)

prob = projected.squeeze(0).max(dim=-1, keepdim=False)[1]

# 增量更新（我们希望重复单词预测结果是一样的）

# 我们在预测是会选择性忽略重复的预测的词，只摘取最新预测的单词拼接到输入序列中

# 拿出当前预测的单词(数字)。我们用x'\_t对应的输出z\_t去预测下一个单词的概率，不用z\_1,z\_2..z\_{t-1}

next\_word = prob.data[-1]

next\_symbol = next\_word

if next\_symbol == tgt\_vocab["E"]:

terminal = True

# print(next\_word)

# greedy\_dec\_predict = torch.cat(

# [dec\_input.to(device), torch.tensor([[next\_symbol]], dtype=enc\_input.dtype).to(device)],

# -1)

greedy\_dec\_predict = dec\_input[:, 1:]

return greedy\_dec\_predict

# ==========================================================================================

# 预测阶段

# 测试集

sentences = [

# enc\_input dec\_input dec\_output

['我 有 一 个 女 朋 友 P', '', '']

]

enc\_inputs, dec\_inputs, dec\_outputs = make\_data(sentences)

test\_loader = Data.DataLoader(

MyDataSet(enc\_inputs, dec\_inputs, dec\_outputs), 2, True)

enc\_inputs, \_, \_ = next(iter(test\_loader))

print()

print("=" \* 30)

print("利用训练好的Transformer模型将中文句子'我 有 一 个 女 朋 友' 翻译成英文句子: ")

for i in range(len(enc\_inputs)):

greedy\_dec\_predict = greedy\_decoder(model, enc\_inputs[i].view(

1, -1).to(device), start\_symbol=tgt\_vocab["S"])

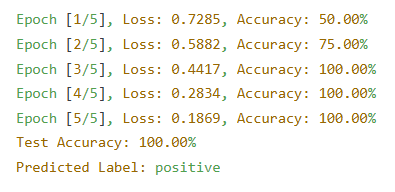
print(enc\_inputs[i], '->', greedy\_dec\_predict.squeeze())

print([src\_idx2word[t.item()] for t in enc\_inputs[i]], '->',

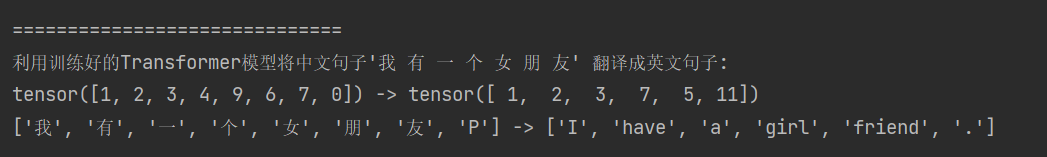
[idx2word[n.item()] for n in greedy\_dec\_predict.squeeze()])

# 实验结果与分析

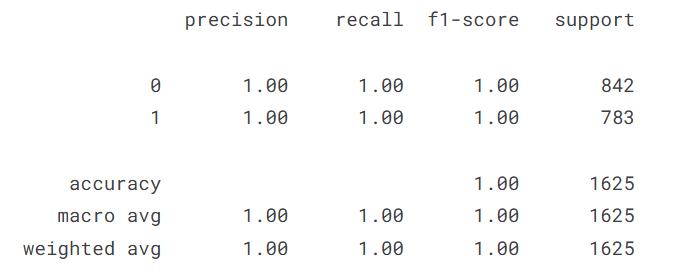
* 观察终端，看到textNN每个epoch的输出：



* 可以看到transformer进行机器翻译的结果，是非常正确的



* 在毒蘑菇分类实验中，svm支持向量机展示出了sota的效果，实际上，LSTM也可以达到一样的效果



# 实验结果讨论分析

对于文本分类任务，如果文本长度较短且数据量较小，可以尝试 TextCNN，它的卷积和池化操作能快速提取文本特征。对于数据较为复杂且特征维度高（例如文本特征），如果计算资源允许，可以使用 LSTM 来捕捉更复杂的时序和语义信息。如果需要高效且不依赖深度学习的解决方案，且数据集规模适中，SVM 是一个经典且有效的选择。

Transformer 模型以其强大的自注意力机制和并行处理能力，在 NLP 任务中取得了显著的成功。在实验中，正确选择数据集、合理设计模型架构、调优超参数并进行充分评估是非常重要的，学会使用transformer的基本模型进行机器翻译任务，或者调用其自注意力机制到模型中用来提升自己模型的效果都是很好的方向，transformer作为近年来nlp领域的核心，是所有工作的重点。

将transformer模型拿来做词嵌入，或者使用transformer作为自己模型的第一层，第二层，或者直接调用自注意力机制使用，增强自己模型的效果都可以得到不错的结果。本次实验仅对机器翻译任务进行实验，验证了transformer是一个极其优秀的模型，attention is all you need！