# NLP 期末实践报告

姓　　名　陈静

学　　号　2272072

实验时间：   1   月   14   日          星期 六

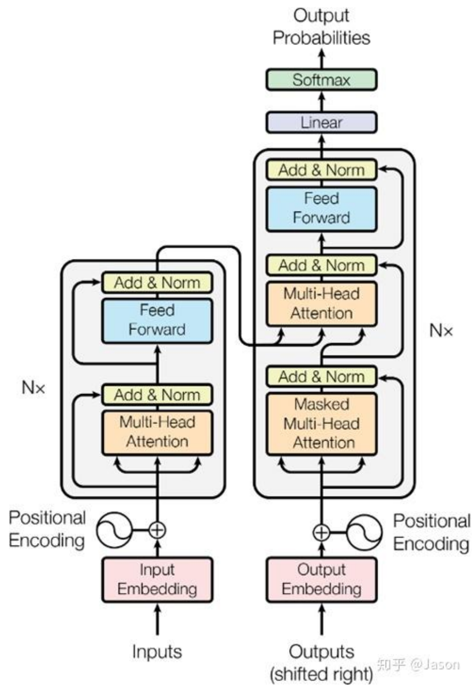
实验目的：运用 Transformer 模型实现机器翻译并识别实体

实验环境：

Python 3.7.2
Torch 1.13.1
Pycharm 2019.3.3
Git

实验理论：

在机器翻译任务中使用 Transformer 模型，将一种语言的一个句子作为输入，然后将其翻译成另一种语言的一个句子作为输出。Transformer 本质上是一个 Encoder-Decoder 架构。因此中间部分的 Transformer 可以分为两个部分：编码组件和解码组件。其中，编码组件由多层编码器（Encoder）组成。解码组件也是由相同层数的解码器(Decoder)组成.每个编码器由两个子层组成:Self-Attention 层(自注意力层)和 Position-wise Feed Forward Network(前馈网络,缩写为 FFN)每个编码器的结构都是相同的，但是它们使用不同的权重参数。解码器也有编码器中这两层，但是它们之间还有一个注意力层（即 Encoder-Decoder Attention），其用来帮忙解码器关注输入句子的相关部分（类似于 seq2seq 模型中的注意力）。通过解码器解码后再进行线性变换和归一化输出预测的结果。




实体识别的任务是对每一个 token 都进行分类。比如，识别这个 token 是不是一个人名、组织名或地名。命名实体识别的一个数据集是 CoNLL-2003，这个数据集完全契合这个任务。

实验步骤：

一、环境搭建

(1) 下载 python3.7.2
(2) 安装 Anaconda3
(3) 下载 torch1.13.1
(4) 安装 Pycharm2019.3.3
PyCharm 是一种 Python IDE（Integrated Development Environment，集成开发环境）
(5) 安装 git

二、代码设计与实现

(1)代码设计部分

```
i#
#  数据构建
import math
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as Data
from transformers import AutoModelForTokenClassification, AutoTokenizer


device = 'cpu'
# device = 'cuda'

# transformer.py epochs
epochs = 100

#  这里手动输入了两对中文→英语的句子
# S:  显示解码输入开始的符号
# E:  显示解码输出开始的符号
# P:  如果当前批处理数据大小小于时间步长，将填充空白序列的符号


label_list = [
    "O",            # Outside of a named entity
    "B-PER",      # Beginning of a person's name right after another person's name
    "I-PER",      # Person's name
```

```
    "B-ORG",    # Beginning of an organisation right after another organisation
    "I-ORG",    # Organisation
    "B-LOC",    # Beginning of a location right after another location
    "I-LOC"     # Location
]


# 训练集
sentences = [
    #                                                                   enc_input
dec_input
dec_output
    ['亚 马 逊 公 司 是 美 国 最 大 的 一 家 网 络 电 子 商 务 公 司 P', 'S Amazon is the largest online e-commerce company in the US . ', 'Amazon is the largest online e-commerce company in the US . E'],
    ['亚 马 逊 位 于 华 盛 顿 州 的 西 雅 图 P P P P P P P P P', 'S Amazon is located in Seattle , Washington . . . . ', 'Amazon is located in Seattle , Washington . . . . E']
]

# 测试集（
# 输入："亚 马 逊 公 司 是 美 国 最 大 的 一 家 网 络 电 子 商 务 公 司"
# 输出："Amazon is the largest online e-commerce company in the US."

# 分别建立中文和英文词库
src_vocab = {'P': 0, '亚': 1, '马': 2, '逊': 3, '公': 4, '司': 5, '是': 6, '美': 7, '国': 8, '最': 9,
             '大': 10, '的': 11 ,'一': 12, '家': 13, '网': 14, '络': 15, '电': 16, '子': 17, '商': 18, '务': 19,
             '位': 20, '于': 21, '华': 22, '盛': 23, '顿': 24, '州': 25, '西': 26, '雅': 27, '图': 28 }
src_idx2word = {i: w for i, w in enumerate(src_vocab)}
src_vocab_size = len(src_vocab)

tgt_vocab = {'P': 0, 'Amazon': 1, 'is': 2, 'the': 3, 'largest': 4, 'online': 5, 'e-commerce': 6,
             'company': 7, 'in': 8, 'US': 9, 'located': 10, 'Seattle': 11, 'Washington': 12,
             'S': 13, 'E':14, '.':15, ',':16 }
idx2word = {i: w for i, w in enumerate(tgt_vocab)}
tgt_vocab_size = len(tgt_vocab)

src_len = 22   # enc_input max sequence length
tgt_len = 14   # dec_input max sequence length

# 超参数
d_model = 512   # Embedding Size（token embedding 和 position 编码的维度）
d_ff = 2048   # 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_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()]]    # [[1, 2, 3, 4, 0],
[1, 2, 3, 5, 0]]
        dec_input = [[tgt_vocab[n] for n in sentences[i][1].split()]]    # [[6, 1, 2, 3, 4, 8],
[6, 1, 2, 3, 5, 8]]
        dec_output = [[tgt_vocab[n] for n in sentences[i][2].split()]]    # [[1, 2, 3, 4, 8,
7], [1, 2, 3, 5, 8, 7]]

        enc_inputs.extend(enc_input)
        dec_inputs.extend(dec_input)
        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

    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,   5,   6,   7,   8,   9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19,   4,   5,   0]]
    pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)   # [batch_size, 1, len_k], True is
masked
    return pad_attn_mask.expand(batch_size, len_q, len_k)   # [batch_size, len_q, len_k]
构成一个立方体(batch_size 个这样的矩阵)


def get_attn_subsequence_mask(seq):
    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):

        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）
        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 = torch.matmul(attn, V)    # context: [batch_size, n_heads, len_q, d_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):

    def __init__(self):
        super(MultiHeadAttention, self).__init__()
        self.W_Q = nn.Linear(d_model, d_k * n_heads, bias=False)
        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):

        residual, batch_size = input_Q, input_Q.size(0)

        Q = self.W_Q(input_Q).view(batch_size, -1, n_heads, d_k).transpose(1, 2)
        K = self.W_K(input_K).view(batch_size, -1, n_heads, d_k).transpose(1, 2)
        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


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):

        residual = inputs
        output = self.fc(inputs)
        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):

        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs, enc_inputs, enc_self_attn_mask)
        enc_outputs = self.pos_ffn(enc_outputs)

        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_outputs, dec_self_attn = self.dec_self_attn(dec_inputs, dec_inputs, dec_inputs,dec_self_attn_mask)
        dec_outputs, dec_enc_attn = self.dec_enc_attn(dec_outputs, enc_outputs, enc_outputs,dec_enc_attn_mask)
        dec_outputs = self.pos_ffn(dec_outputs)
        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)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)])

    def forward(self, enc_inputs):

        enc_outputs = self.src_emb(enc_inputs)
        enc_outputs = self.pos_emb(enc_outputs.transpose(0, 1)).transpose(0, 1)
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs)
        enc_self_attns = []
        for layer in self.layers:
            enc_outputs, enc_self_attn = layer(enc_outputs,enc_self_attn_mask)
            enc_self_attns.append(enc_self_attn)
        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_outputs = self.tgt_emb(dec_inputs)
```

```
        dec_outputs = self.pos_emb(dec_outputs.transpose(0, 1)).transpose(0, 1).to(device)
        dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs).to(device)
        dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs).to(device)

        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)
        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):

        # 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, 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)



# ====================================================
for epoch in range(epochs):
    for enc_inputs, dec_inputs, dec_outputs in loader:

        enc_inputs, dec_inputs, dec_outputs = enc_inputs.to(device),
dec_inputs.to(device), dec_outputs.to(device)
        outputs, enc_self_attns, dec_self_attns, dec_enc_attns = model(enc_inputs,
dec_inputs)
        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):

    enc_outputs, enc_self_attns = model.encoder(enc_input)
    dec_input = torch.zeros(1, 0).type_as(enc_input.data)  # 初始化一个空的 tensor:
tensor([], size=(1, 0), dtype=torch.int64)
    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
            next_word = prob.data[-1]
            next_symbol = next_word
            if next_symbol == tgt_vocab["E"]:
                terminal = True

        greedy_dec_predict = dec_input[:, 1:]
        return greedy_dec_predict


# 预测阶段
# 测试集
sentences = [
        # enc_input                    dec_input                    dec_output
        ['亚 马 逊 公 司 是 美 国 最 大 的 一 家 网 络 电 子 商 务 公 司 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("="*45)
print("利用训练好的 Transformer 模型将中文句子'亚 马 逊 公 司 是 美 国 最 大
的 一 家 网 络 电 子 商 务 公 司 ' 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()])

dec_predict=[idx2word[n.item()] for n in greedy_dec_predict.squeeze()]
sequence=" ".join(dec_predict)
print(sequence)

cache_dir = "./transformersModels/ner"
""""
,cache_dir = cache_dir
""""
model                                                                    =
```

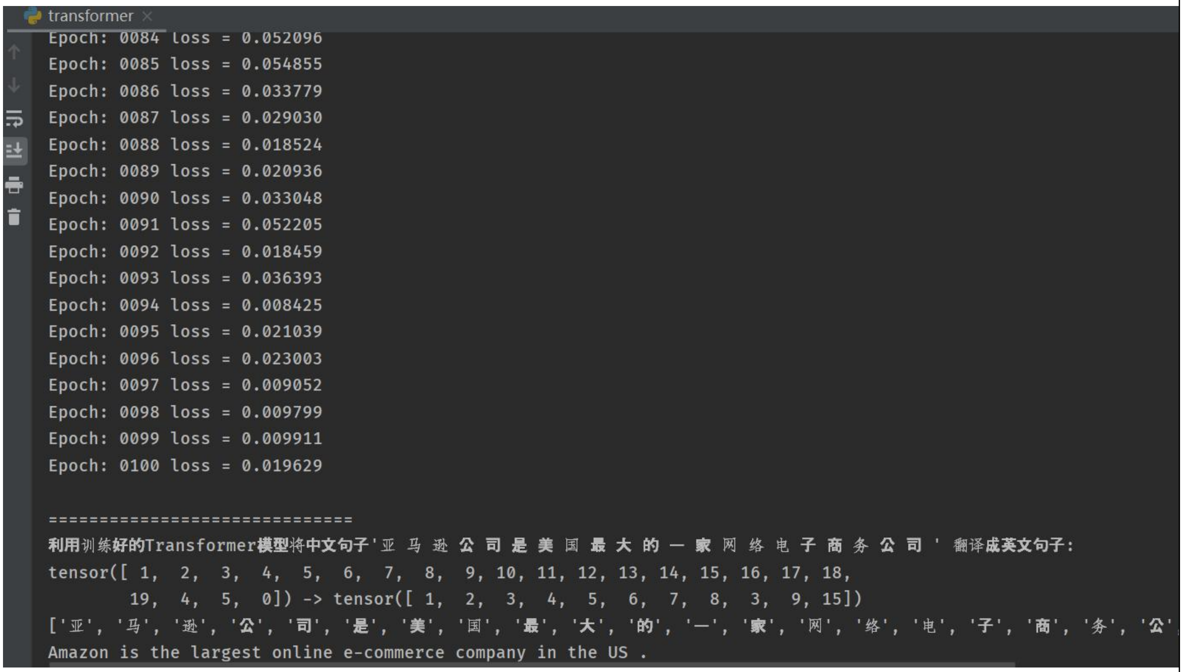
```
AutoModelForTokenClassification.from_pretrained("dbmdz/bert-large-cased-finetuned-co
nll03-english",

cache_dir=cache_dir, return_dict=True)
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased", cache_dir=cache_dir)

# Bit of a hack to get the tokens with the special tokens
tokens = tokenizer.tokenize(tokenizer.decode(tokenizer.encode(sequence)))
inputs = tokenizer.encode(sequence, return_tensors="pt")

outputs = model(inputs).logits
predictions = torch.argmax(outputs, dim=2)

for token, prediction in zip(tokens, predictions[0].numpy()):
    print(token, label_list[prediction])
```

(2) 测试与实验结果

当 epoch 很小比如为 6 时，Transformer 模型的 loss 还很大，如下：

```
"D:\Program Files\Python\Python37\python.exe" D:
Epoch: 0001 loss = 2.913607
Epoch: 0002 loss = 2.764294
Epoch: 0003 loss = 2.659649
Epoch: 0004 loss = 2.539561
Epoch: 0005 loss = 2.312232
Epoch: 0006 loss = 2.325197
```

当 epoch=100 时，会发现这时的 loss 已经很小了，模型拟合的比较好，已经能够输出结果了；

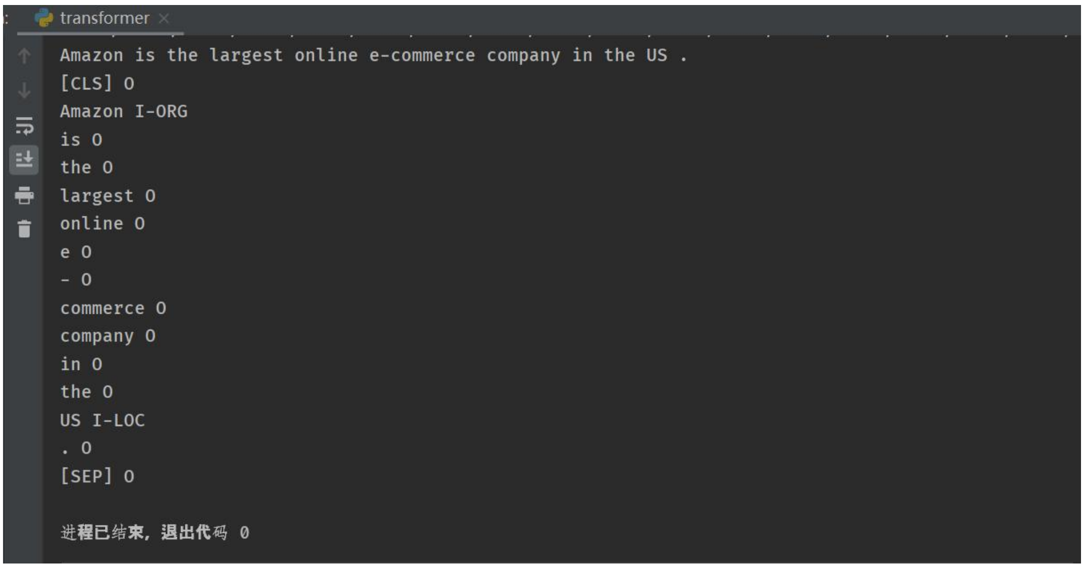



将上述模型获得的结果稍做处理后来进行标签预测。

获得的结果是 9 个分类的概率分布。

一般是使用最高概率的那个标签最为最终预测结果。

将每个标记与其预测标签一起打印出来。




然后将实践项目通过 git 上传到 github。

实验体会：

通过这次实验，我对 Transformer 模型有了更进一步的认识。

1、transformer 是编码器－解码器架构的一个实践，在实际情况中编码器或解码器可以单独使用。

2、在 transformer 中，多头自注意力用于表示输入序列和输出序列，不过解码器必须通过掩蔽机制来保留自回归属性。

3、transformer 中的残差连接和层规范化是训练非常深度模型的重要工具。

4、transformer 模型中基于位置的前馈网络使用同一个多层感知机，作用是对所有序列位置的表示进行转换。

在 transform 的基础上加入实体识别功能，通过 bert 模型实例化预训练模型和对应文本标记器，不过最后输出结果不能删除“O”实体，需要手动整理，仍需进一步改进。

在实验过程中对 Torch 也有了初步的理解，是一个有大量机器学习算法支持的科学计算框架，是一个与 Numpy 类似的张量（Tensor）操作库。因为刚开始不了解，在定义数据时就出现了差错：ValueError: expected sequence of length 22 at dim 1 (got 14)。发现这个问题出现在对 tensor 的转换中，tensor 的转换要求内部的数组维度相同。后修改数据长度一致即可。