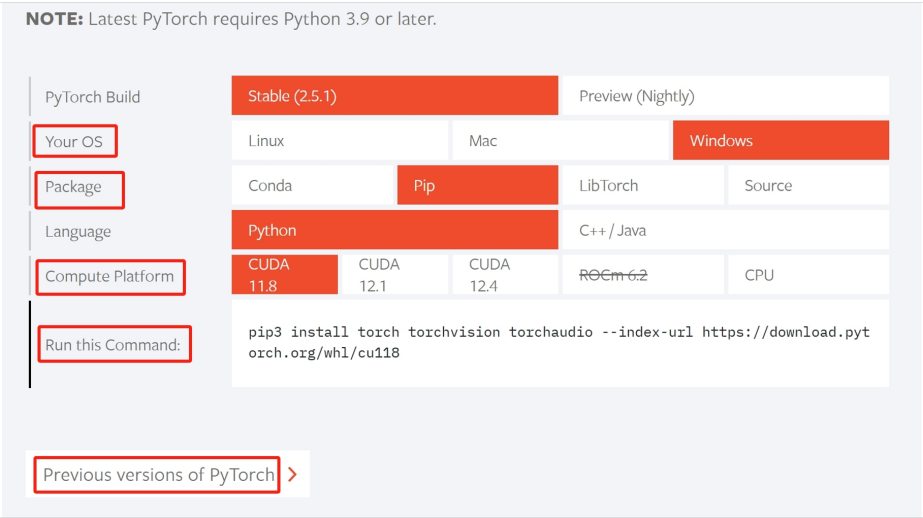
0.实验准备

- 环境搭建

```
# 创建虚拟环境
conda create -n llm_transformer python=3.8
# 激活虚拟环境
conda activate llm_transformer
# 安装必要的包
pip3 install torch torchvision torchaudio  # 根据具体硬件选择pytorch版本
```

pytorch官网



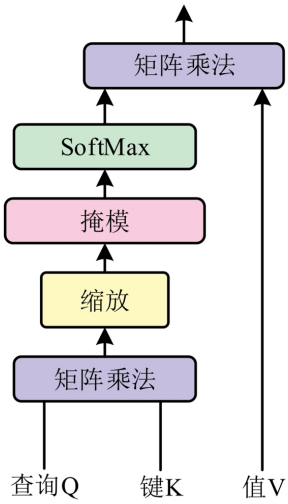
- 常用库导入

```
import os
import math
import json
import argparse

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset

import numpy as np

from tqdm import tqdm
```

1.Transformer

1.1 注意力模块

1.1.1 模块简介



缩放点积注意力模块的输入是查询Q、键K和值V，计算公式为：

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- 第1步：QK^T
- 第2步：$\frac{QK^T}{\sqrt{d_k}}$
- 第3步：$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$
- 第4步：$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

缩放点积注意力模块框架

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, d_k):
        super(ScaledDotProductAttention, self).__init__()
        self.d_k = d_k
    def forward(self, Q, K, V):
        ...
        ...
        ...
        ...
        return context, attn
```

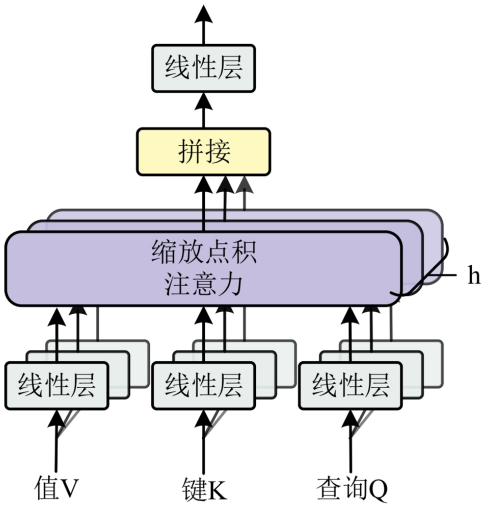
- **d_k**：d_k，键向量维度大小
- **Q, K, V**：**输入的查询向量、键向量和值向量，维度为$[batch_size \times n_heads \times len_sentence \times d_k]$

1.1.2 实验

根据缩放点积注意力的原理，补全**缩放点积注意力模块**框架。

1.2 多头注意力模块

1.2.1 模块简介



多头注意力模块的输入同样为查询Q、键K和值V，计算公式为：

$$MultiHead_h(Q,K,V) = Concat(head_1,\ldots,head_h)W^O \; head_i = Attention(QW_i^O, KW_i^O, VW_i^O)$$

多头注意力模块框架

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super(MultiHeadAttention, self).__init__()
        self.d_model = d_model
        self.n_heads = n_heads
        self.head_dim = d_model // n_heads
        ...
        ...
    def forward(self, Q, K, V):
        ...
        ...
        ...
        ...
        return output, attn
```

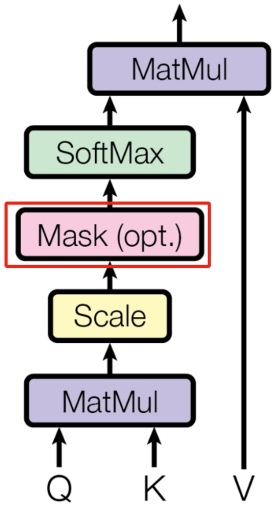
- **d_model：** 模型中向量的维度大小
- **n_heads：** head的数量
- **head_dim：** 切分后单个注意力模块输入向量的维度大小
- **W_Q, W_K, W_V：** 查询投影矩阵、键投影矩阵和值投影矩阵
- **atten：** 缩放点积注意力
- **linear：** 线性层
- **layer_norm：** 层归一化
- **Q, K, V：** 输入的查询向量、键向量和值向量，维度为$[batch_size \times len_sentence \times d_{model}]$

1.2.2 实验

根据多头注意力的原理，补全**多头注意力模块**框架。

1.3 掩码多头注意力模块

1.3.1 模块简介



$$mask = \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}$$

掩码多头注意力模块的关键是两个操作

- 生成一个上三角掩码矩阵，用于掩码操作；
- 在缩放点积注意力模块中添加掩码操作--即将上三角掩码矩阵与输入矩阵按元素相乘。

上三角掩码矩阵生成函数：

```
def get_attn_subsequent_mask(seq):
    ...
    ...
    ...
```

- **seq：** 输入向量，维度为$[batch_size \times len_sentence]$

1.3.2 实验

根据掩码多头注意力原理，补全**掩码矩阵生成函数**框架，并修改**缩放点积注意力模块**与**多头注意力模块**以实现掩码操作。

1.4 位置编码

1.4.1 模块简介

$$P = \begin{bmatrix} sin(\frac{pos}{10000^0}) & cos(\frac{pos}{10000^0}) & sin(\frac{pos}{10000^{2/3}}) \\ sin(\frac{pos}{10000^0}) & cos(\frac{pos}{10000^0}) & sin(\frac{pos}{10000^{2/3}}) \\ sin(\frac{pos}{10000^0}) & cos(\frac{pos}{10000^0}) & sin(\frac{pos}{10000^{2/3}}) \end{bmatrix}$$

位置编码是一个固定矩阵，只需要根据输入向量维度大小选取所需矩阵即可

位置编码模块框架

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        ...
        ...
        ...
        self.register_buffer('pe', pe)
    def forward(self, x):
        ...
        ...
```

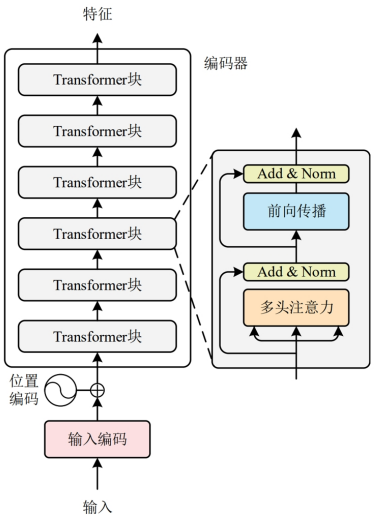
- **d_model：** 模型中向量的维度大小
- **dropout：** dropout函数参数
- **max_len：** 输入向量最大长度

1.4.2 实验

填写位置编码所需的固定矩阵生成代码，补全**位置编码模块**框架。

1.5Transformer编码器

1.5.1 模块简介



Transformer编码器由输入编码、位置编码和N个Transformer编码器层组成，其中Transformer编码器层包含多头注意力模块和前馈层，可以从三个层次完成Transformer编码器模块：

- **前馈层：** 由一维卷积层和层归一化构成
- **Transformer编码器层：** 由多头注意力模块和前馈层构成
- **Transformer编码器：** 由输入编码、位置编码和N个Transformer编码器层构成

掩码生成函数框架

```
def get_attn_pad_mask(seq_q, seq_k):
    batch_size, len_q = seq_q.size()
    batch_size, len_k = seq_k.size()
```

```
    pad_attn_mask = seq_k.data.eq(2).unsqueeze(1)
    return pad_attn_mask.expand(batch_size, len_q, len_k)
```

前馈层框架

```
class PoswiseFeedForwardNet(nn.Module):
    def __init__(self, d_model, d_ff):
        super(PoswiseFeedForwardNet, self).__init__()
        ...
        ...
        ...
    def forward(self, inputs):
        ...
        ...
        ...
```

- **d_model**： 模型中向量的特征维度大小
- **d_ff**： 前馈层中特征维度大小
- **inputs**： 输入向量，维度为$[batch_size \times len_sentence \times d_{model}]$

编码器层框架

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super(EncoderLayer, self).__init__()
        ...
        ...
    def forward(self, enc_inputs, enc_self_attn_mask):
        ...
        ...
```

- **d_model**： 模型中向量的特征维度大小
- **n_heads**： head数量
- **d_ff**： 前馈层中特征维度大小
- **enc_inputs**： 输入向量，维度为$[batch_size \times len_sentence \times d_{model}]$
- **enc_self_attn_mask**： 掩码矩阵，维度为$[batch_size \times len_sentence \times len_sentence]$

编码器框架

```
class Encoder(nn.Module):
    def __init__(self, src_vocab_size, d_model, n_layers, n_heads,
dim_feedforward, dropout):
        super(Encoder, self).__init__()
        ...
        ...
        ...
```

```
    def forward(self, enc_inputs):
        ...
        ...
        ...
```

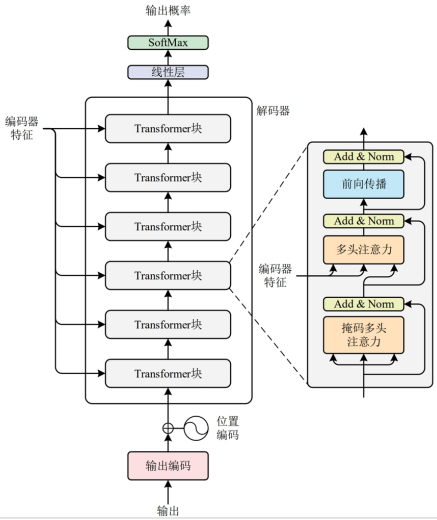
- **src_vocab_size：** 输入词的词表大小
- **d_model：** 模型中向量的特征维度大小
- **n_layers：** Transformer编码器中编码器层数量
- **n_heads：** head数量
- **dim_feedforward：** 前馈层中特征维度大小
- **dropout：** dropout参数
- **enc_inputs：** 输入向量，维度为$[batch_size \times len_sentence \times d_{model}]$

1.5.2 实验

根据所给出的各个模块框架，结合之前所写的各个模块的框架，实现完整的Transformer编码器模块

1.6 Transformer解码器

1.6.1 模块简介



Transformer解码器由输出编码、位置编码和N个Transformer解码器层组成，其中Transformer解码器层包含掩码多头注意力模块、交叉多头注意力模块和前馈层，可以从三个层次实现Transformer解码器模块：

- **前馈层：** 由一维卷积层和层归一化构成
- **Transformer解码器层：** 由掩码多头注意力模块、交叉多头注意力模块和前馈层构成，其中交叉多头注意力模块即多头注意力模块，只由输入不同
- **Transformer解码器：** 由输出编码、位置编码和N个Transformer解码器层构成

解码器层框架

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super(DecoderLayer, self).__init__()
        ...
        ...
```

```
    def forward(self, dec_inputs, enc_outputs, dec_self_attn_mask,
dec_enc_attn_mask):
        ...
        ...
```

- **d_model**： 模型中向量的特征维度大小
- **n_heads**： head数量
- **d_ff**： 前馈层中特征维度大小
- **dec_inputs**： 解码器输入向量，维度为$[batch_size \times len_sentence \times d_{model}]$
- **enc_outputs**： 编码器输出向量，维度为$[batch_size \times len_sentence \times d_{model}]$
- **dec_self_attn_mask**： 掩码矩阵，维度为$[batch_size \times len_sentence \times len_sentence]$
- **dec_enc_attn_mask**： 掩码矩阵，维度为$[batch_size \times len_sentence \times len_sentence]$

解码器框架

- 解码器块

```
class Decoder(nn.Module):
    def __init__(self, tgt_vocab_size, d_model, n_layers, n_heads,
dim_feedforward, dropout):
        super(Decoder, self).__init__()
        ...
        ...
        ...
    def forward(self, dec_inputs, enc_inputs, enc_outputs):
        ...
        ...
        ...
```

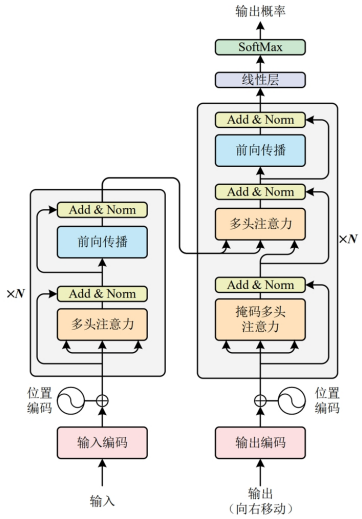
- **tgt_vocab_size**： 输出词的词表大小（为了方便，本例子中将输入输出统合在一个词表中）
- **d_model**： 模型中向量的特征维度大小
- **n_layers**： Transformer编码器中编码器层数量
- **n_heads**： head数量
- **dim_feedforward**： 前馈层中特征维度大小
- **dropout**： dropout参数
- **dec_inputs**： 解码器输入向量，维度为$[batch_size \times len_sentence \times d_{model}]$
- **enc_inputs**： 编码器输入向量，维度为$[batch_size \times len_sentence \times d_{model}]$，仅仅为交叉注意力层构造注意力图掩码
- **enc_outputs**： 编码器输出向量，维度为$[batch_size \times len_sentence \times d_{model}]$

1.6.2 实验

根据所给出的各个模块框架，结合之前所写的各个模块的框架，实现完整的Transformer解码器模块。

1.7 Transformer

1.7.1 模块简介



Transformer模块由Transformer编码器和Transformer解码器组成，在此基础上添加输出层，构成完整的Transformer模块。

Transformer模块框架

```
class Transformer(nn.Module):
    def __init__(self, vocab_size, d_model, nheads, dim_feedforward, nlayers,
dropout=0.5):
        super(Transformer, self).__init__()
        ...
        ...
        ...
    def forward(self, enc_inputs, dec_inputs):
        ...
        ...
        ...
```

- **vocab_size**： 词表大小（为了方便，本例子中将输入输出统合在一个词表中）
- **d_model**： 模型中向量的特征维度大小
- **n_layers**： Transformer编码器中编码器层数量
- **n_heads**： head数量
- **dim_feedforward**： 前馈层中特征维度大小
- **dropout**： dropout参数
- **dec_inputs**： 解码器输入向量，维度为$[batch_size \times len_sentence \times d_{model}]$
- **enc_inputs**： 编码器输入向量，维度为$[batch_size \times len_sentence \times d_{model}]$

1.7.2 实验

根据之前所写的各个模块的框架，实现完整的Transformer模块。

2.Transformer的训练与推理

2.1 数据处理

2.1.1 文本预处理

词元化，tokenize： 下面的tokenize函数将文本行列表（lines）作为输入， 列表中的每个元素是一个文本序列（如一条文本行）。 每个文本序列又被拆分成一个词元列表，词元（token）是文本的基本单位。 最后，返回一个由词元列表组成的列表，其中的每个词元都是一个字符串（string）。

```
def tokenize(self, path):
    assert os.path.exists(path)
    with open(path, 'r', encoding="utf8") as f:
        samples = []
        for line in f:
            ...
            ...
            ...
    data = torch.tensor(samples).type(torch.int64)
    return data
```

2.1.2 实验

补全tokenize方法代码。

> 完整代码见 main.py

2.2.模型训练

2.2.1 模型训练流程

部分训练代码

```
train_dataset = Corpus(args.data, 'train')
train_loader = torch.utils.data.DataLoader(train_dataset,
                                batch_size=args.batch_size,
                                shuffle=True,
                                num_workers=args.num_workers)
# 加载数据集
vocab_size = len(train_dataset.dictionary)
model = Transformer(vocab_size,
                    args.d_model,
                    args.nhead,
                    args.dim_feedforward,
                    args.nlayers,
                    args.dropout).to(device)
# 加载模型
criterion = nn.CrossEntropyLoss()
# 创建损失函数
optimizer = optim.Adam(model.parameters(), lr=args.lr)
# 创建优化器
lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)
# 创建学习率调度器
...
...
for epoch in range(1, args.epochs + 1):  # 训练一轮（用数据集中所有数据训练一次）
```

```
        total_loss = 0
        train_iterator = tqdm(enumerate(train_loader), total=len(train_loader))
        for iteration, data in train_iterator:  # 用一个batch_size数据训练一次
            txt = data[0].to(device)
            label = data[1].to(device)

            optimizer.zero_grad()  # 清除所有被优化变量的梯度
            label_input = label[:, :-1]
            label_expected = label[:, 1:]

            output, _, _, _ = model(txt, label_input)  # 模型前向传播

            loss = criterion(output, label_expected.reshape(-1))  # 计算损失函数
            loss.backward()  # 反向传播
            optimizer.step()  # 根据计算出的梯度更新模型的参数
            total_loss += loss.item()*args.batch_size
            message = f'Epoch: {epoch}/{args.epochs}, \
            iter: {iteration+1}/{len(train_loader)}, lr:
{lr_scheduler.get_last_lr()[0]}, \
            loss: {total_loss/((iteration+1)*args.batch_size)}'
            train_iterator.set_description(message)
            torch.save(model.state_dict(), args.save)
        lr_scheduler.step()  # 更新学习率调度器
        torch.save(model.state_dict(), args.save)  # 存储模型参数
```

2.2.2 实验

利用完成的完整Transformer代码，实现Transformer的简单训练，并理解Transformer中数据的变换。

> 完整代码见 main.py

2.3 模型推理

2.3.1 模型推理流程

部分推理代码

```
    for number in number_list:  # 对输入文本进行处理
        number_str = str(number)
        words = [i for i in number_str]
        words = ['<sos>'] + words + ['<eos>']
        words.extend(['<pad>'] * (train_dataset.max_seq_length - len(words)))
        ids = []
        for word in words:
            ids.append(train_dataset.dictionary.word2idx[word])
        batch_inputs.append(ids)
        batch_y_inputs.append([train_dataset.dictionary.word2idx['<sos>']])

    batch_inputs = torch.tensor(batch_inputs).to(device)
    batch_y_inputs = torch.tensor(batch_y_inputs).to(device) # 解码器的第一次输入为
开始字符'<sos>'
```

```
    ...
    ...
    with torch.no_grad():
        for i in range(0, train_dataset.max_seq_length):  # 每次解码器的输出作为下次
解码器的输入
            output, _, _, _ = model(batch_inputs, batch_y_inputs)
            output = output.view(batch_inputs.size(0), -1, output.size(-1))
            output = output[:, -1:, :]
            output_ids = torch.argmax(output, -1)

            batch_y_inputs = torch.cat([batch_y_inputs, output_ids], dim=1)

            output_texts = train_dataset.decode(output_ids) # 根据id获取字符
            print(output_texts)
            results.append(output_texts)
```

2.3.2 实验

加载训练权重，实现模型推理流程。

> 完整代码见 inference.py

课后习题

1. 根据4.1节介绍的Transformer知识即提供的部分代码，尝试基于PyTorch框架实现完整的Transformer模型。

2. 随着社交媒体的普及，大量的文本数据被产生。文本情感分析是自然语言处理中的一个重要任务，旨在判断文本所表达的情感倾向（如积极、消极、中立）。尝试使用斯坦福大学的大型电影评论数据集实现完整的情感分析任务流程。