智能问答系统项目报告书

七月在线机器学习集训营第 14 期

姓名：邓再勇

学号：667742

目录

一、 项目概述

智能问答是NLP领域落地最多的场景，其商业价值较高，能有效解决业务问题，降低人力成本。智能问答有以下优势：

1、24小时值守，即使非工作时间也能服务客户，避免客户流失。

2、分担高峰压力，双11期间，客服无法接待所有客户。

3、自动导购，结合用户画像等数据，对用户的兴趣加深了解，进而推荐Item。

因此，智能问答也成了目前各大AI厂商的主推产品。

智能问答包括封闭域与开放域，封闭域即特定领域的问答，例如经融、教育、医疗等，如下图所示：

开放域即闲聊与任务型问题，如下图所示：

本项目我们将实现封闭域与开放域(闲聊)结合的问答系统，其整体的流程如下：

1、判断用户的意图，即所问的问题是封闭域问题还是聊，这一步被称为意图识别。

2、对于封闭域的问题，我们采用检索式问答，即匹配最相似的问题，返回其对应的答案。

3、对于闲聊，我们采用生成式模型来动态生成答案。

二、 项目设计

1、从零搭建一个框架，实现基于word2vec的检索式问答。

2、采用预训练模型(elmo、bert、sentence-transformers)来替代word2vec，进一步的效果提升。

3、实现文本分类模型(TextCNN、Bert fine-tune），判断是封闭域问题还是开放域问题。

4、文本匹配模型(ESIM、Bert fine-tune)，针对召回的文本做精排。

5、实现开放域模型(GPT2)，把整个流程走通，进行项目整合。

三、 项目实现

1、 环境搭建

本项目的代码全部放在远程 GPU 服务器上运行，所以需要提前在 GPU 服务器上搭建 conda 虚拟环境，并安装所需要的包 Numpy、Pandas、gensim、jieb、ELmoForManyLangs、transformers、sentence-transformers、Pytorch 等，具体操作在这里不再进行详述。

开发工具采用开源的 VSCode，通过安装 Remote Development 插件，让 VSCode 可以方便地在本地进行开发和在远程进行调试。

2、 准备数据集

Word2Vec 词向量训练数据下载：
https://pan.baidu.com/s/1cWt2qqH5ym0vLcjihehfGg 提取码：z6dt

ELMo 预训练模型下载：
http://39.96.43.154/zhs.model.tar.bz2

Bert 预训练模型下载：
https://huggingface.co/hfl/chinese-roberta-wwm-ext

sentence-transformers 预训练模型下载：
https://public.ukp.informatik.tu-darmstadt.de/reimers/sentence-transformers/v0.2/paraphrase-multilingual-MiniLM-L12-v2.zip

chitchat 闲聊预训练模型下载：
https://pan.baidu.com/share/init?surl=iEu_-Avy-JTRsO4aJNiRi 提取码：ju6m

3、 Word2Vec 实现检索式问答

⑴ 词向量生成

最早的检索式问答大部分都是基于 elastic search，es 实际上是一个数据库，其主要功能是用来在海量的数据中进行检索，添加插件即可对文本自动分词。使用 es 做问答实际上就是找相同的词，并计算其相似度。es 最大的缺点就是没办法很好的处理同义句。

在文本表示上，one-hot 较为稀疏，tf-idf 维度又太大，Word2Vec 能有效解决以上问题，并且能更有效的表示字词的语义，但是多义词没办法处理。

使用 Language Model 可以实时生成字词向量，唯一缺点就是速度慢。

有了文本表示方法之后，即可计算文本的相似度，找到相似度最高的问题，返回其答案。

Word2Vec 常用的训练方法即 CBOW 与 skip-gram，但是如果词典较大，其 embedding 层参数较多，当我们用 BP 反向传播的时候，实际只需要更新部分词的权重，而不需要更新所有的词。

传统的神经网络词向量语言模型有三层，输入层（词向量），隐藏层和输出层（softmax 层），神经网络模型最大的问题在于从隐藏层到输出的 softmax 层的计算量很大，因为要计算所有词的 softmax 概率，再去找概率最大的值。

为了避免要计算所有词的 softmax 概率，Word2Vec 采用了哈夫曼树来代替从隐藏层到输出 softmax 层的映射，在哈夫曼树中，隐藏层到输出层的 softmax 映射是沿着哈夫曼树一步步完成的，因此这种 softmax 取名为"Hierarchical Softmax"。

在 Word2Vec 中，我们采用二元逻辑回归的方法，即规定沿着哈夫曼树左子树走，那么就是负类(树节点编码 1)，沿着右子树走，那么就是正类(树节点编码 0)，判别正类和负类的方法是使用 sigmoid 函数。

使用最大似然法来寻找所有节点的词向量和所有内部节点 θ，当训练样本里的中心词 w 是一个很生僻的词时，就需要在哈夫曼树中寻找很久，这个时候就可以通过 Negative Sampling 进行负采样来解决。

使用 gensim 训练词向量，实现代码如下：

```
from gensim.models.word2vec import LineSentence
import multiprocessing

input_file = './word2vec/wiki.txt'
model_file_name = './word2vec/wiki.model'

model = Word2Vec(sentences=LineSentence(input_file),
                 vector_size=100, #词向量的维度
                 window=5,
                 min_count=5, #少于 5 的词舍去
                 workers=multiprocessing.cpu_count(),
                 sg=1, #使用 skip-gram
                 hs=0, #使用 negative
                 negative=5 #每次负采样 5 条
    )
```

```
model.save(model_file_name)
```

词向量训练完成后保存在 wiki.model 文件中，后续就可以直接加载使用。

⑵ 句向量生成

常用的生成句向量的方式即加权平均，例如对于动词和名词给一个稍大的权重，对于特定领域重要的词给一个稍大的权重。当然也可以让权重相等，直接取一个均值，在代码实现中为对词向量取均值。

⑶ 计算文本相似度

文本相似度计算的方法有很多种，常用的有欧氏距离、曼哈顿距离、汉明距离、余弦相似度，在这里我们使用余弦相似度，余弦相似度的值域在[-1,1]之间，方便度量，计算公式如下：

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}.$$

⑷ 评估标准

检索式问答的评估标准采用 Mean reciprocal rank（MPR），MRR 是指多个查询语句的排名倒数的均值。公式如下：

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}$$

其中|Q|表示问题个数，rank_i 表示第 i 个召回的结果的排序位置。

⑷ 检索式问答实现

```
import pandas as pd
import numpy as np
import jieba
from gensim.models import Word2Vec

df = pd.read_csv('./data/qa_data.csv')
questions = df['question'].values
```

```
answers = df['answer'].values

model_path = './word2vec/wiki.model'
model = Word2Vec.load(model_path)

# 句子转化为向量
def sen2vec(sentence):
    # 对句子进行分词
    segment = list(jieba.cut(sentence))
    vec = np.zeros(100)
    for s in segment:
        try:
            # 取出 s 对应的向量相加
            vec += model.wv[s]

        #出现 oov 问题，词不在词典中
        except:
            print(f"{s} 不在词典中")
            pass
    # 采用加权平均求句向量
    vec /= len(segment)
    return vec

# 生成所在问题的句向量
question_vec = []
for q in questions:
    question_vec.append(sen2vec(q))

# 计算余弦相似度
def cosine(a, b):
    # 矩阵的积除以矩阵模的积
    return np.matmul(a, np.array(b).T) / (np.linalg.norm(a) *
        np.linalg.norm(b, axis=1))

def qa(text):
    vec = sen2vec(text)

    # 计算输入的问题和问题库中问题的相似度
    similarity = cosine(vec, question_vec)

    # 取最大相似度
    max_similarity = max(similarity)
    print("最大相似度：", max_similarity)
```

```
index = np.argmax(similarity)
if max_similarity < 0.8:
    print(max_similarity)
    print('没有找到对应的问题，您问的是不是：
        ', questions[index])
    return f"没有找到对应的问题，您问的是不是：
        {questions[index]}"

print('最相似的问题：', questions[index])
print('答案：', answers[index])
return answers[index]
```

4、 NLP 语言模型

⑴ 什么是语言模型

语言模型简单来说就是一串词序列的概率分布。具体来说，语言模型的作用是为一个长度为 m 的文本确定一个概率分布 P，表示这段文本存在的可能性。

语言模型可以用来判断语言序列是否为正常语句，也可以用来预测下一个字或词即生成文本。

在实践中，如果文本的长度较长，P(wi | w1, w2, . . . , wi−1)的估算会非常困难。因此使用一个简化模型：n 元模型（n-gram model）。在 n 元模型中估算条件概率时，只需要对当前词的前 n-1 个词进行计算。当 n 较大时，就会存在数据稀疏问题，导致估算结果不准确。因此，一般在百万词级别的语料中，一般也就用到三元模型（tri-gram）。

在 n-gram 中不管 n 怎么选取，实际上都是近似值，因此可以使用神经网络来建模使得结果更精确。

先给每个词在连续空间中赋予一个向量(词向量)，再通过神经网络去学习这种分布式表征。利用神经网络去建模当前词出现的概率与其前 n-1 个词之间的约束关系。很显然这种方式相比 n-gram 具有更好的泛化能力，只要词表征足够好。

⑵ 什么是预训练语言模型

预训练语言模型即预先使用大规模的语料训练一个语言模型，在下游任务上，只需要做一个 feature base 或 微调 fine tuning 即可得到很好的效果。

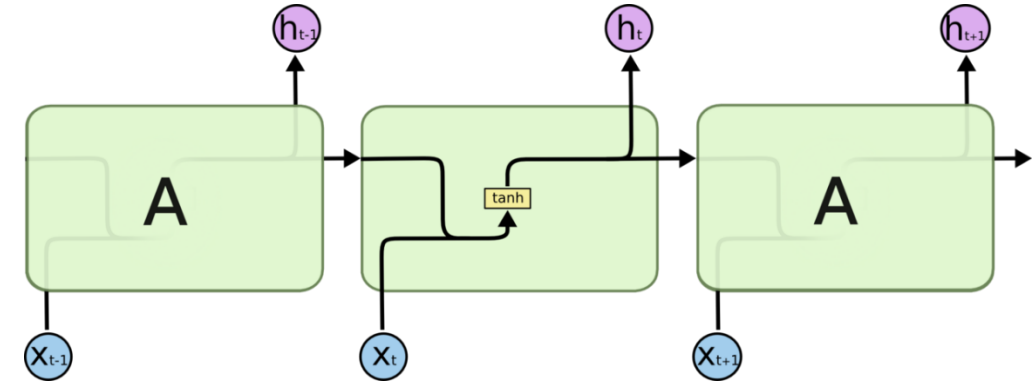
在没有预训练模型之前，我们可能需要几十万的训练数据，有了预

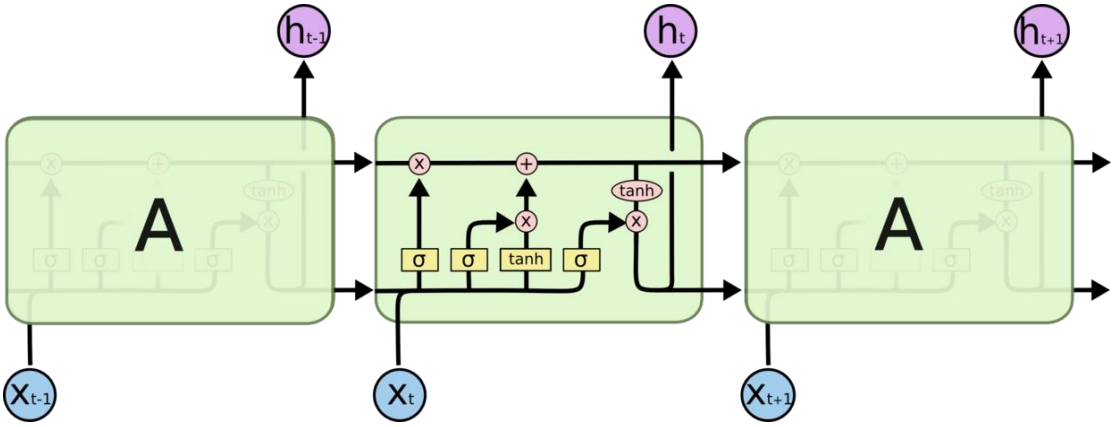
训练模型之后，我们可能只需要几千条训练数据，并且训练时间和效果都有提升。

5、 LSTM 模型

LSTM 是 RNN 的变种，RNN 的问题是容易出现梯度消失，梯度消失的原因是当序列很长时反向求导多个 0-1 之间的数累乘结果会接近于零，而 LSTM 被称为长短期记忆网络，其能有效学习到长期依赖的关系，也能有效缓解梯度消失的问题，缓解的办法是反向传播求导时让累乘变成累加。

RNN 和 LSTM 的模型结构对比如下图：

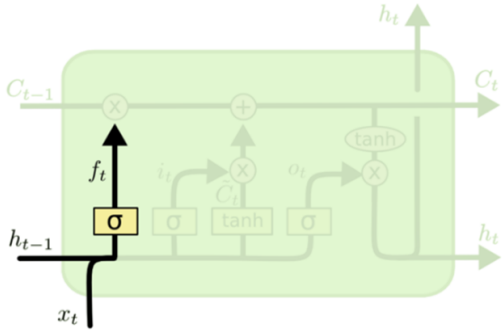




LSTM 中主要有三种门，遗忘门，输入门，输出门。

⑴ 遗忘门

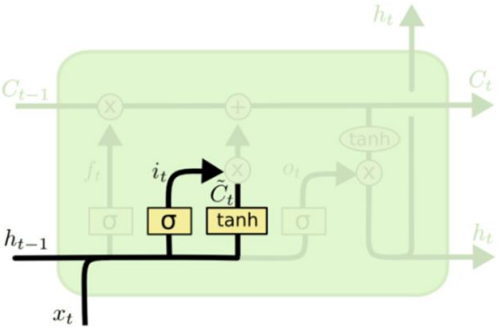
用来决定要抛弃哪些细胞的状态信息，具体实现是，输入 h(t-1)与 x(t)经过 sigmoid()函数后，输出了一个 0-1 之间的值，0 代表完全移植，1 代表完全保留，这样细胞状态 C(t-1)就有些被保留有些被抛弃了。



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

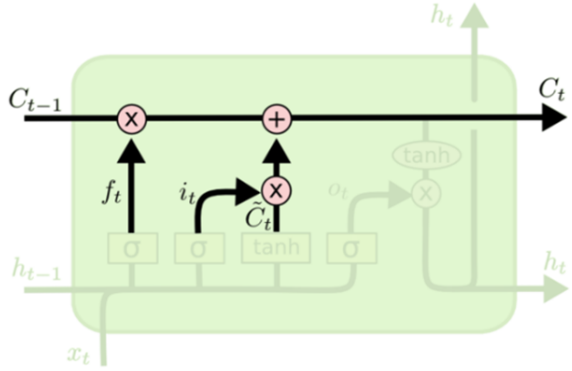
⑵ 输入门

用来决定当前的输入哪些应该保留哪些应该遗弃，这个过程主要分两步，首先是 sigmoid，同样是输入 0-1 之间的值，从而决定我们要更新哪些值，随后 tanh 生成了一个新的候选向量 Ct，将这两个值相乘得到需要输入的值。



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

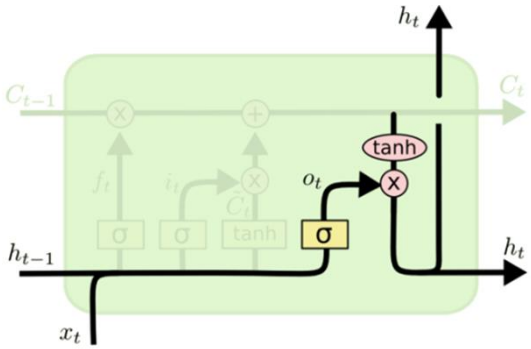
接下来，就更新细胞状态，将旧的细胞状态 Ct-1 与 ft 相乘，忘记那些需要忘记的细胞状态，然后加上输入门的值 it*Ct，得到的结果便是新的细胞状态。



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

⑶ 输出门

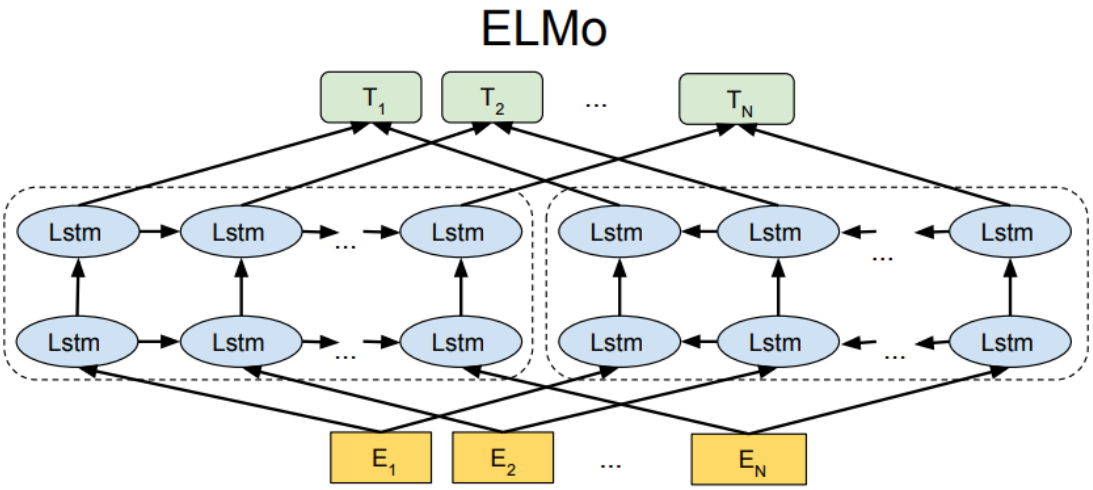
用来决定哪些细胞状态应该输出哪些不输出，首先运行 sigmoid 层决定细胞状态输出哪一部分，然后再把细胞状态经过 tanh 函数，将输出值保持在[-1, 1]之间，再乘以 sigmoid 的输出值，就得到输出门的输出值。



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

6、 ELMo 实现检索式问答

Elmo 是一种双向的 LSTM 语言模型，即包含前向与反向，结构如下图：



ELMo 中表示词向量的方式有多种。
第一种：取第二层 LSTM 的输出。
第二种：给每一层计算一个权重，取加权平均。

本项目我们使用哈工大开源的多语言 Elmo 模型，下载地址为：
https://github.com/HIT-SCIR/ELMoForManyLangs

现在使用 ELMo 替换 Word2Vec 生成句向量，实现检索式问答，具体代码如下：

```
import pandas as pd
import numpy as np
import jieba
from elmoformanylangs import Embedder

df = pd.read_csv('./data/qa_data.csv')
questions = df['question'].values
answers = df['answer'].values
```

```
elmo_model_path = './elmo/zhs.model'
elmo_model = Embedder(elmo_model_path)

def elmo2vec(sentence):
    '''
    output_layer 参数.
    0 for the word encoder
    1 for the first LSTM hidden layer
    2 for the second LSTM hidden layer
    -1 for an average of 3 layers. (default)
    -2 for all 3 layers
    '''
    if isinstance(sentence, str):
        segment = list(jieba.cut(sentence))
        # 用 elmo 转换词向量
        vec = elmo_model.sents2elmo([segment], output_layer=-1)
    elif isinstance(sentence, np.ndarray):
        segment = [jieba.cut(s) for s in sentence]
        # 用 elmo 转换词向量
        vec = elmo_model.sents2elmo(segment, output_layer=-1)

    # 句向量取均值
    return [np.mean(v, axis=0) for v in vec]

# 生成所有问题的句向量
question_vec = []
for q in questions:
    question_vec.extend(elmo2vec(q))

# 计算余弦相似度
def cosine(a, b):
    # 矩阵的积除以矩阵模的积
    return np.matmul(a, np.array(b).T) / (np.linalg.norm(a) * np
        .linalg.norm(b, axis=1))

def qa(text):
    vec = elmo2vec(text)[0]

    # 计算输入的问题和问题库中问题的相似度
    similarity = cosine(vec, question_vec)

    # 取最大相似度
    max_similarity = max(similarity)
```

```
        print("最大相似度：", max_similarity)

        index = np.argmax(similarity)
        if max_similarity < 0.8:
            print(max_similarity)
            print('没有找到对应的问题，您问的是不是：', questions[index])
            return f"没有找到对应的问题，您问的是不是：
                {questions[index]}"

        print('最相似的问题：', questions[index])
        print('答案：', answers[index])
        return answers[index]

    if __name__ == '__main__':
        for i in range(2):
            text = input('请输入您的问题：')
            qa(text)
```

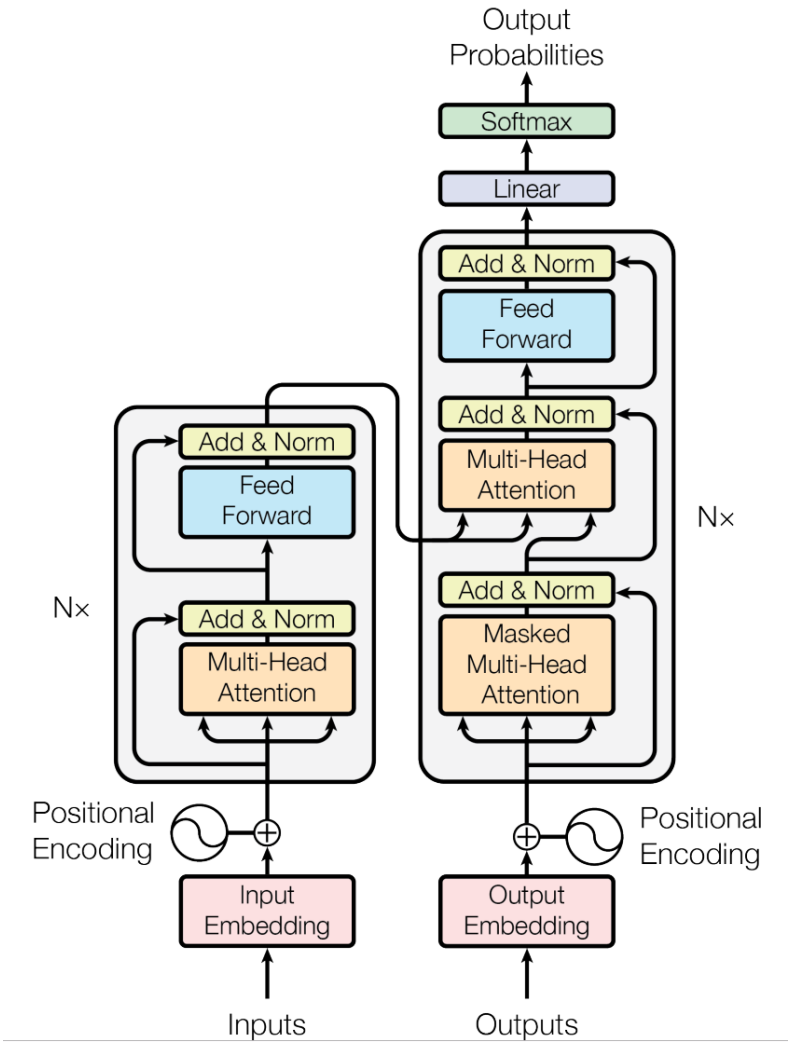
7、 Transformer 模型

Transformer 是 encoder-decoder 的结构

encoder 是由 6 个相同结构的层堆叠在一起，每一个层包含两个子层，分别是 multi-head attention 层与一个全连接层，每个子层跟了一个 residual connection 与一个 layer normalization。

decoder 也是由 6 个相同结构的层堆叠在一起，除了和 encoder 一样的两个子层外，还有一个 masked multi-head attention。

模型结构如下图：



(1) Scaled Dot-Product Attention

Transformer 模型的核心点就在于 Attention 机制，在 Transformer 中使用的是 Scaled Dot-Product Attention。

常用的 attention 有两种，additive attention 和 dot-product attention，两者的效果都差不多， dot-product 更简单一些。

假如 q 与 k 的初始化的值满足均值为 0，方差为 1，那么点乘后的结果均值为 0，方差为d_k，这就导致不同的值之间相差过大， softmax 的输出值之和为 1，如果有一个输出单元的值增大了，那么其他所有单元的值都将会受到抑制，导致出现全盘通吃的情况。

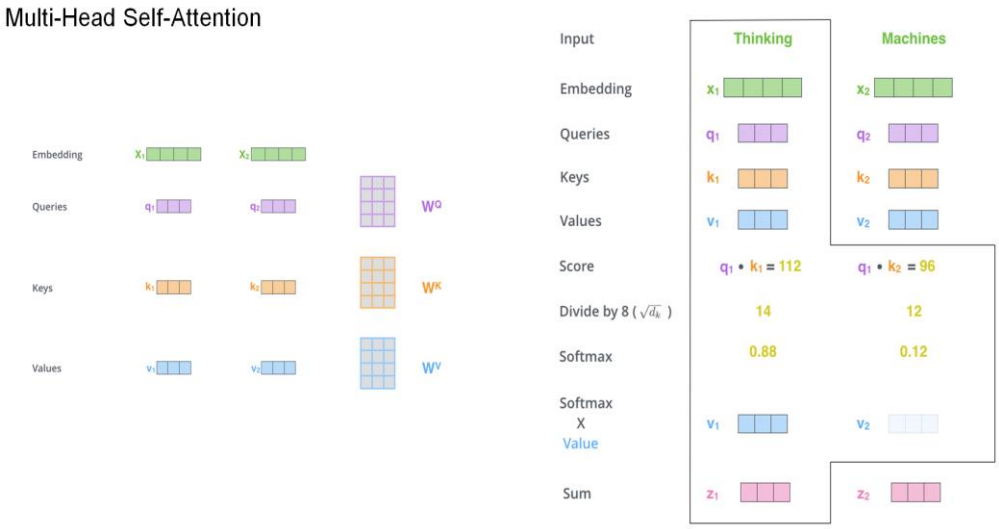
Scaled Dot-Product Attention 就是在 dot-product attention 的基础上加上了一个缩放因子。

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

⑵ Self Attention

在 Self Attention 中 Q、K、V 是相等的，Q*K 相乘后的结果实际上就是一个权重矩阵，再除以缩放因子，经过 softmax 变换后再和 V 进行相乘，实际上是对 V 进行加权。

单个维度的 Self Attention 计算流程如下图：

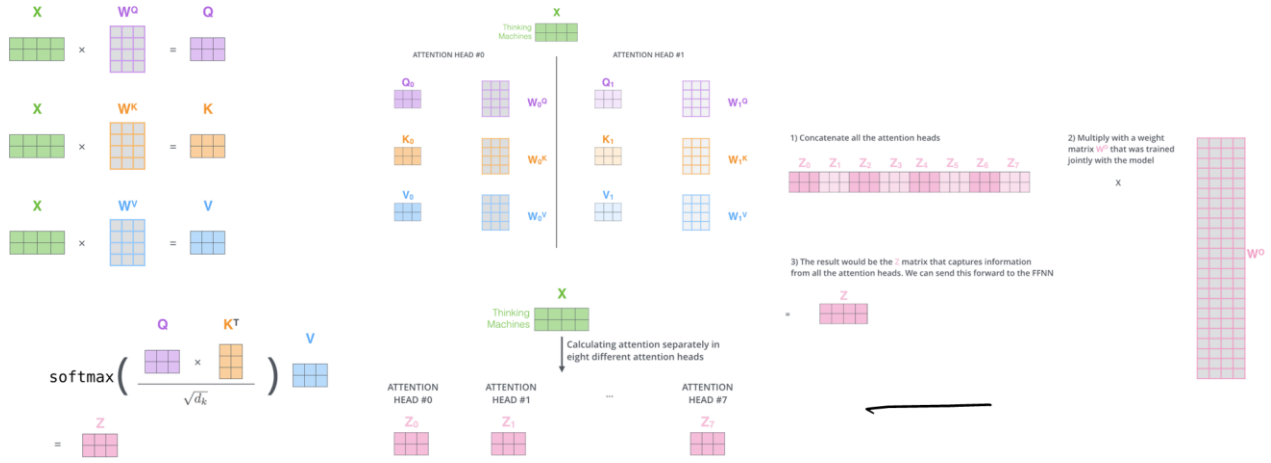


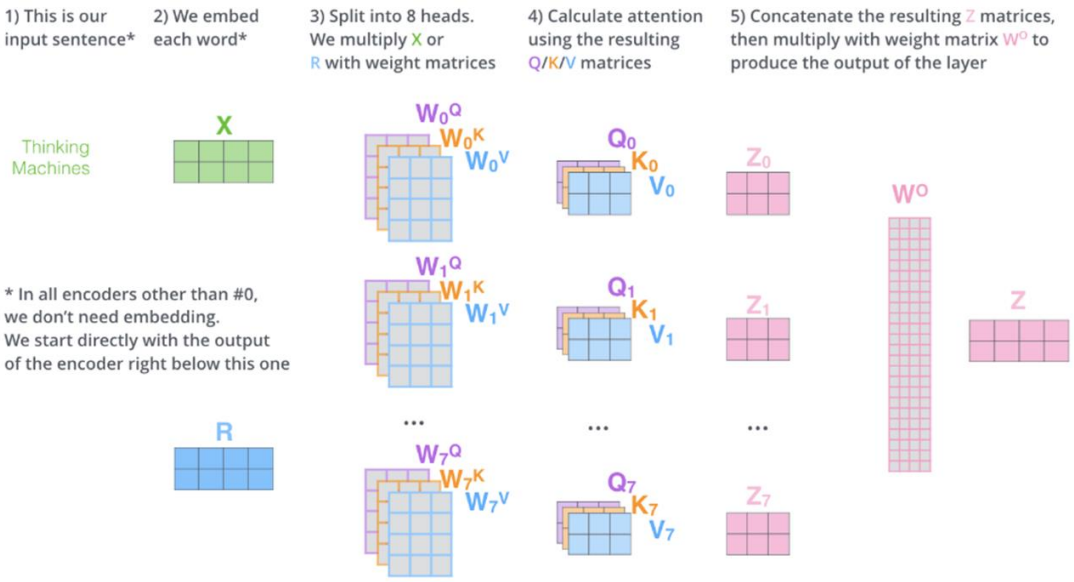
⑶ Multi-Head Attention

单个 Attention 只能考虑到一个维度，在 NLP 任务中一个维度是不够的，所以要应用 Multi-Head Attention，每个维度都有对应的 Q、K、V 值，这样每个维度就会有对应输出的值，再把每个维度输出的值 concatenate 在一起，最后再乘以一个权重矩阵把维度降回原来的输入维度。

Multi-Head Attention 的计算流程如下图：

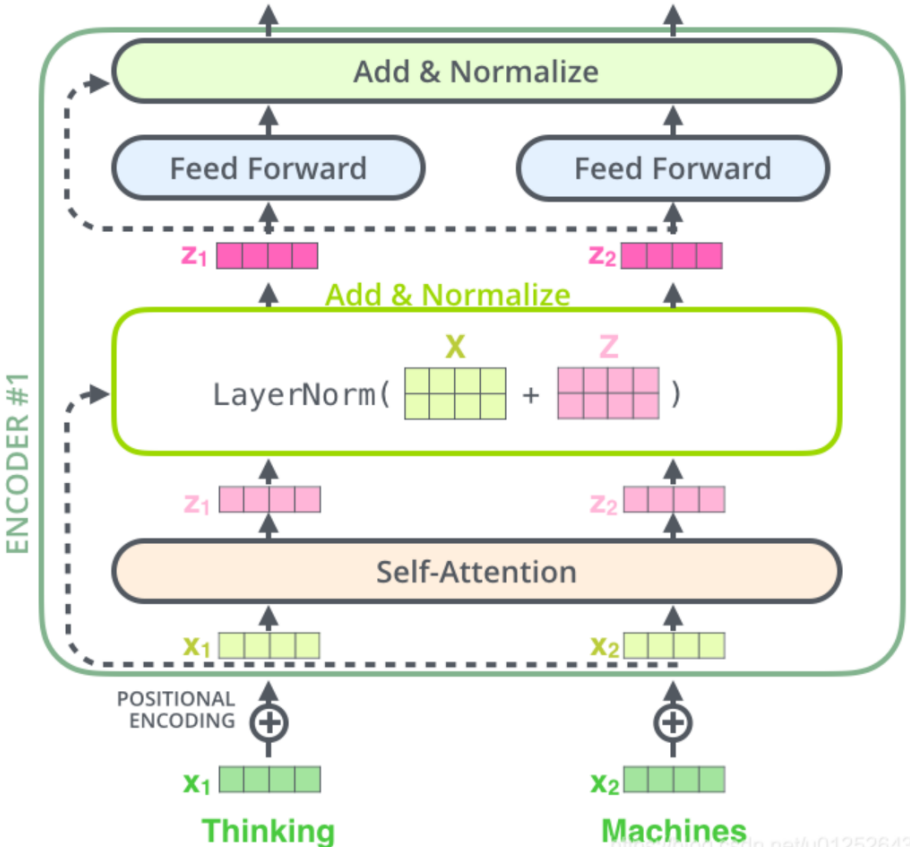
Multi-Head Self-Attention



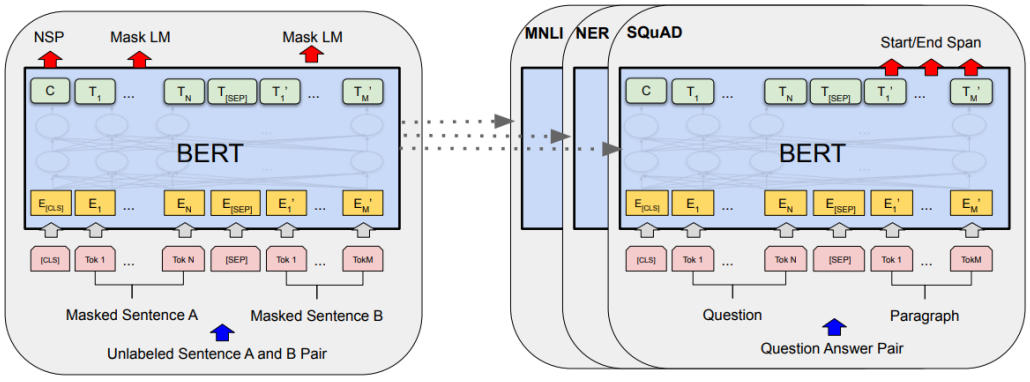


8、 BERT 模型

BERT 主要结构是 Transformer 的 encoder 层，其网络结构如下图：



BERT 包括两个训练阶段，预训练阶段与 fine-tuning 阶段。



Pre-training　　Fine-Tuning

BERT 的预训练包括两个任务，Masked Language Model 与 Next Sentence Prediction。

(1) Masked Language Model

Masked Language Model 可以理解为完形填空，随机 mask 每一个句子中 15%的词，用其上下文来做预测。

例如：my dog is hairy → my dog is [MASK]

此处将 hairy 进行了 mask 处理，然后采用非监督学习的方法预测 mask 位置的词是什么，但是该方法有一个问题，因为是 mask15%的词，其数量已经很高了，这样就会导致某些词在 fine-tuning 阶段从未见过。

80%的是采用[mask]，my dog is hairy → my dog is [MASK]。

10%的是随机取一个词来代替 mask 的词，my dog is hairy -> my dog is apple。

10%的保持不变，my dog is hairy -> my dog is hairy。

这是因为 transformer 要保持对每个输入 token 分布式的表征，否则 Transformer 很可能会记住这个[MASK]就是“hairy”。至于使用随机词带来的负面影响，论文中认为所有其他的 token(即非"hairy"的 token)共享 15%*10% = 1.5%的概率，其影响是可以忽略不计的。

⑵ Next Sentence Prediction

选择一些句子对 A 与 B，其中 50%的数据 B 是 A 的下一条句子，剩余 50%的数据 B 是语料库中随机选择的，学习其中的相关性，添加这样的预训练的目的是目前很多 NLP 的任务比如 QA 和 NLI 都需要理解两个句子之间的关系，从而能让预训练的模型更好的适应这样的任务。

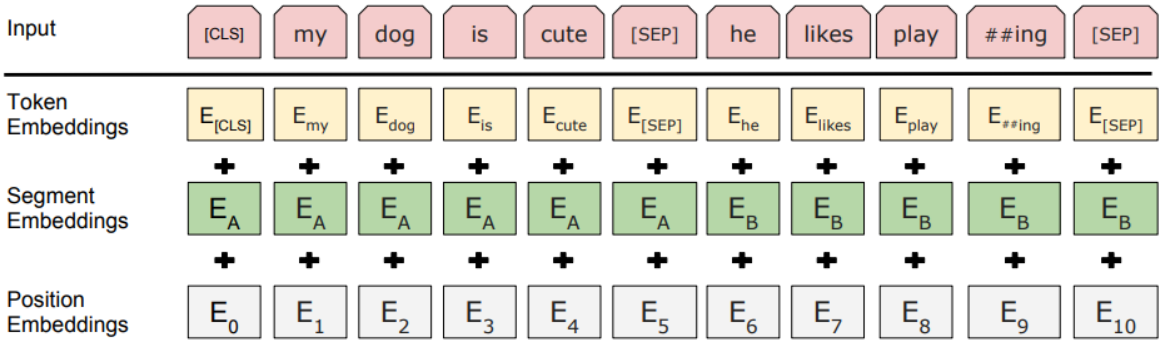
序列的头部会填充一个[CLS]标识符，该符号对应的 bert 输出值通常用来直接表示句向量，不同的序列之间以[SEP]标识符进行填充表示，序列尾部也以[SEP]进行填充，如下图：

$$\begin{aligned}\text{Input} &= \texttt{[CLS] the man went to [MASK] store [SEP]}\\ &\quad\ \texttt{he bought a gallon [MASK] milk [SEP]}\\ \text{Label} &= \texttt{IsNext}\end{aligned}$$

$$\begin{aligned}\text{Input} &= \texttt{[CLS] the man [MASK] to the store [SEP]}\\ &\quad\ \texttt{penguin [MASK] are flight ##less birds [SEP]}\\ \text{Label} &= \texttt{NotNext}\end{aligned}$$

⑶ BERT 的输入值

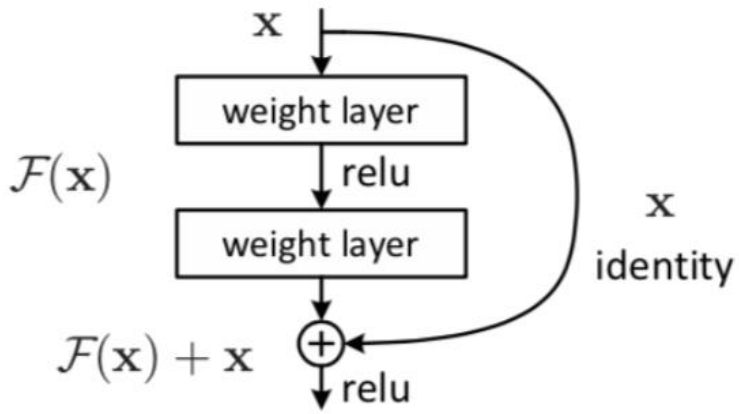
输入值包括了三个部分，分别是 token embedding 词向量，segment embedding 段落向量，position embedding 位置向量，这三个部分相加形成了最终的 bert 输入向量。



⑷ 残差模块

随着网络越来越深，训练变得原来越难，网络的优化变得越来越难，残差网络就可以解决这个问题，用残差模块能让训练变得更加简单。

如果输入值 x 和输出值 F(x) 的差值过小，那么梯度可能会过小，导致出现一些梯度消失的情况，残差网络的好处在于当残差为 0 时，该层神经元只是对前层进行一次现行堆叠，使得网络梯度不容易消失，性能不会下降，这是最差的情况，实际上残差不可能为 0。



⑸ Layer Normalization

Normalization 有很多种，作用就是把输入转化成均值为 0 方差为 1 的数据，让网络更稳定，不至于出现梯度消失，不容易过拟合。

在把数据输入激活函数之前进行 normalization（归一化），这样就可以避免输入数据落在激活函数的饱和区。

Batch Normalization：在每一个 batch 上进行 normalization，不适合文本，batch 太小也不合适，一般用在 CNN 模型。

Layer Normalization：在每一条数据上进行 normalization，一般用在 RNN 模型。

⑹ Bert 实现检索式问答

```
import numpy as np
import pandas as pd
import torch
from transformers import AutoTokenizer, AutoModel

df = pd.read_csv('./data/qa_data.csv')
questions = df['question'].values
answers = df['answer'].values

bert_model_path = "./chinese-roberta-wwm-ext"

tokenizer = AutoTokenizer.from_pretrained(bert_model_path)
bert_model = AutoModel.from_pretrained(bert_model_path)

# 平均池化
def mean_pooling(model_output, attention_mask):
    token_embeddings = model_output[0]
    input_mask_expanded = attention_mask.unsqueeze(-1).expand(to
        ken_embeddings.size()).float()
    sum_embeddings = torch.sum(token_embeddings * input_mask_exp
        anded, 1)
    sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
    return sum_embeddings / sum_mask

# 用 transformers 转换句向量
def bert_to_vec(sentence):
    print('bert encode start')
    if isinstance(sentence, np.ndarray):
        encoded_input = tokenizer(list(sentence), padding=True,
            truncation=True, max_length=128, return_tensors='pt')
    else:
        encoded_input = tokenizer(sentence, padding=True, trunca
            tion=True, max_length=128, return_tensors='pt')

    with torch.no_grad():
        model_output = bert_model(**encoded_input)
    sentence_embeddings = mean_pooling(model_output, encoded_inp
        ut['attention_mask'])
    print('bert encode finish')
    return sentence_embeddings.numpy()

# 将所有问题库中问题转换为句向量
```

```
question_vec = bert_to_vec(questions)

# 计算余弦相似度
def cosine(a, b):
    # 矩阵的积除以矩阵模的积
    return np.matmul(a, np.array(b).T) / (np.linalg.norm(a) * np
        .linalg.norm(b, axis=1))

def qa(text):
    vec = bert_to_vec(text)

    # 计算输入的问题和问题库中问题的相似度
    similarity = cosine(vec, question_vec)

    # 取最大相似度
    max_similarity = max(max(similarity))
    print("最大相似度：", max_similarity)

    index = np.argmax(similarity)
    if max_similarity < 0.8:
        print(max_similarity)
        print('没有找到对应的问题，您问的是不是：', questions[index])
        return f"没有找到对应的问题，您问的是不是：
           {questions[index]}"

    print('最相似的问题：', questions[index])
    print('答案：', answers[index])
    return answers[index]

if __name__ == '__main__':
    for i in range(2):
        text = input('请输入您的问题：')
        qa(text)
```

⑺ Sentence Transformers 实现检索式问答

```
import numpy as np
import pandas as pd
from sentence_transformers import SentenceTransformer

df = pd.read_csv('./data/qa_data.csv')
questions = df['question'].values
answers = df['answer'].values
```

```
sentence_model_path = "./paraphrase-multilingual-MiniLM-L12-v2"
sentence_model = SentenceTransformer(sentence_model_path, device
    ='cuda:0')

# 用 sentence_transformers 转换句向量
def sentence_to_vec(sentence):
    embedding = sentence_model.encode(sentence)
    return embedding

# 将问题库中所有问题转换为句向量
question_vec = sentence_model.encode(questions, batch_size=64, s
    how_progress_bar=True, device='cuda:0')

# 计算余弦相似度
def cosine(a, b):
    # 矩阵的积除以矩阵模的积
    return np.matmul(a, np.array(b).T) / (np.linalg.norm(a) * np
        .linalg.norm(b, axis=1))

def qa(text):
    vec = sentence_to_vec(text)

    # 计算输入的问题和问题库中问题的相似度
    similarity = cosine(vec, question_vec)

    # 取最大相似度
    max_similarity = max(similarity)
    print("最大相似度：", max_similarity)

    index = np.argmax(similarity)
    if max_similarity < 0.8:
        print(max_similarity)
        print('没有找到对应的问题，您问的是不是：', questions[index])
        return f"没有找到对应的问题，您问的是不是：
            {questions[index]}"

    print('最相似的问题：', questions[index])
    print('答案：', answers[index])
    return answers[index]

if __name__ == '__main__':
    for i in range(2):
        text = input('请输入您的问题：')
        qa(text)
```

9、 从零搭建 NLP 模型

⑴ 模型的输入

NLP 模型常见的输入包括字词向量与先验特征。

字向量：即分字，分字词典小，不容易出现 OOV 的问题，但是表述会略逊于词向量。

词向量：效果受分词的影响，训练和预测阶段的分词必须要一样，容易出现 OOV 问题。

动态字词向量：使用预训练好的字词向量初始化 embedding matrix，并在训练的过程中更新。

静态字词向量：使用预训练好的字词向量初始化 embedding matrix，训练的过程中不更新。

随机字词向量：随机初始化 embedding matrix，训练的过程中更新。

⑵ 先验特征

先验特征即特征工程，是预先提出出来的人工特征，例如在相似度匹配模型中，有多少相同的词，文本的长度相差多少，在文本匹配中，是否包含一些特定的关键词（这些词能明显区分文本类别）。

⑶ 词典的构建

NLP 模型第一步就是构建词典，首先确定好使用什么形式的输入，如果是分词，就对语料进行分词，并根据词频进行排序，分字同理。

输入的时候输入的是字词在词典中的下标。

NLP 模型的第一层，通常都是 embedding 层，这一层能够把 index 自动转成 one-hot 的形式，然后和 embedding matrix 进行相乘，得到字词向量。

在词典中，通常有四个特殊标记位

<PAD>　　补充位
<UNK>　　未知词
<START>　起始词
<END>　　终止词

⑷ 输入长度的统一

训练语料有长有短，但是在训练的过程中，是以 batch 的形式进行训练，所以需要统一语料的长度的，例如在一个 batch 中，最长的语料是 50，最短的语料是 10，可以设置一个中间值，通常取 90%的分位数。那么对于短的语料会补长，对于长的语料会截取。

补长通常补<PAD>标记位的下标，补长可以补前面，也可以补后面，截取也是同样的道理，通常是补长后面。

⑸ 数据加载

NLP 语料数据通常很多，文件也非常大，所以最好用生成器，动态的从本地以 batch 的形式读取数据，如果全部读到内存或者显存中，肯定是放不下的。

可以使用 pytorch 的 TensorDataset 与 DataLoader 来进行读取，也可以自定义一个生成器。

⑹ 模型构建

在 pytorch 中，如果要构建一个模型，定义一个类，继承自 nn.Module，重写构造方法与 forward 方法。

构造方法中主要是初始化一些值，并定义好会用到的一些 layer，forward 方法实际上就是前向传播，把定义好的 layer 全部串起来。

⑺ 实现代码

```
import pandas as pd
import jieba
import numpy as np
import os
from collections import defaultdict

path = os.path.dirname(__file__)

# 获取 QA 项目根目录
root_path =os.path.dirname(os.path.dirname(os.path.abspath(__f
    ile__)))

# 对句子进行分词
def tokenize(string):
    res = list(jieba.cut(string, cut_all=False))
```

```
    return res

# 构建词典并保存
def build_vocab(del_word_frequency):
    data = pd.read_csv(root_path + '/data/classification.csv')
    segment = data['sentence'].apply(tokenize)

    word_frequency = defaultdict(int)

    # 统计每个词出现的次数
    for row in segment:
        for i in row:
            word_frequency[i] += 1

    # 按词频降序排序
    word_sort = sorted(word_frequency.items(), key=lambda x: x
       [1], reverse=True)

    f = open(path + '/vocab.txt', 'w', encoding='utf-8')
    f.write('[PAD]' + "\n" + '[UNK]' + "\n")

    # 词频小的不保存到词典中
    for d in word_sort:
        if d[1] > del_word_frequency:
            f.write(d[0] + "\n")
    f.close()

vocab = {}
# 转换字典索引
if os.path.exists(path + '/vocab.txt'):
    with open(path + '/vocab.txt', encoding='utf-8') as file:
        for line in file.readlines():
            vocab[line.strip()] = len(vocab)

# 转换句中的词在字典中的索引
def seq2index(seq):
    seg = tokenize(seq)
    seg_index = []
    for s in seg:
        seg_index.append(vocab.get(s, 1))
    return seg_index

# 统一长度
def padding_seq(X, max_len=10):
```

```
    return np.array([
        np.concatenate([x, [0]*(max_len-len(x))]) if len(x) <
            max_len else x[:max_len] for x in X
    ])
```

10、 文本分类模型

文本分类模型在本项目的作用是用来把封闭域问题和闲聊问题区分开，对于封闭域问题使用检索式模型，对于闲聊问题使用生成式模型。

常见的文本分类模型有两种。
基于文本表示的方法：词向量、语言模型

基于神经网络的文本分类模型：

分类较为明显：fastText
短文本：TextCNN
长文本：HAN
通吃：精调的 BiLSTM+Attention
终极方案：bert+fine tuning

⑴ fastText

fastText 模型架构和 word2vec 中的 CBOW 很相似， 不同之处是 fastText 是预测标签的类别，而 CBOW 是用上下文预测中间词，两者模型架构类似但是模型的任务不同。

⑵ TextCNN

TextCNN 是将卷积神经网络 CNN 应用到文本分类任务，利用多个不同 size 的 kernel 来提取句子中的关键信息（类似于多窗口大小的 ngram），从而能够更好地捕捉局部相关性。

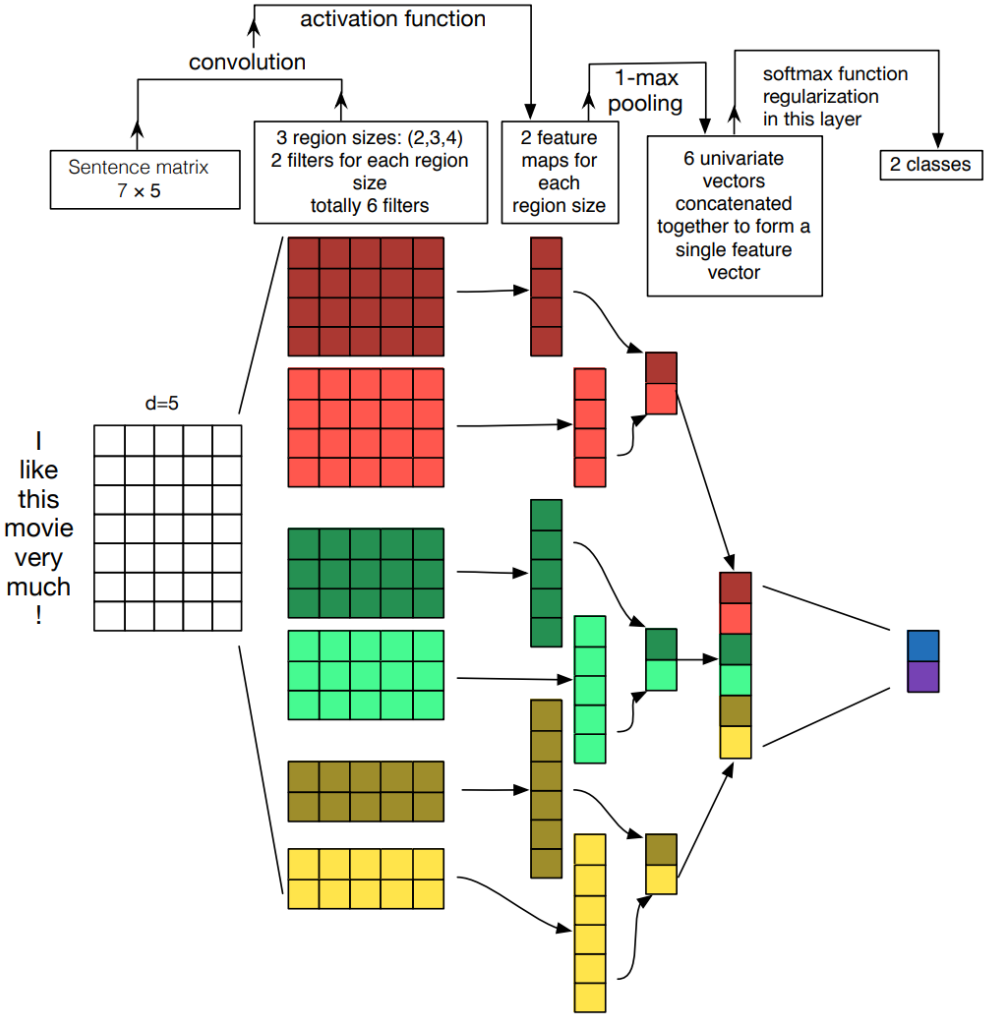
Embedding：第一层是图中最左边的 7 乘 5 的句子矩阵，每行是词向量，维度=5，这个可以类比为图像中的原始像素点。

Convolution：然后经过 kernel_sizes=(2,3,4) 的一维卷积层，每个 kernel_size 有两个输出 channel。

MaxPolling：第三层是一个 1-max pooling 层，这样不同长度句子经过 pooling 层之后都能变成定长的表示。

FullConnection and Softmax：最后接一层全连接的 softmax 层，输出每个类别的概率。

TextCNN 最大优势网络结构简单，因此参数数目少，计算量少，训练速度快。



(3) HAN

HAN 主要有 word encoder，word attention，sentence encoder，sentence attention 阶段。

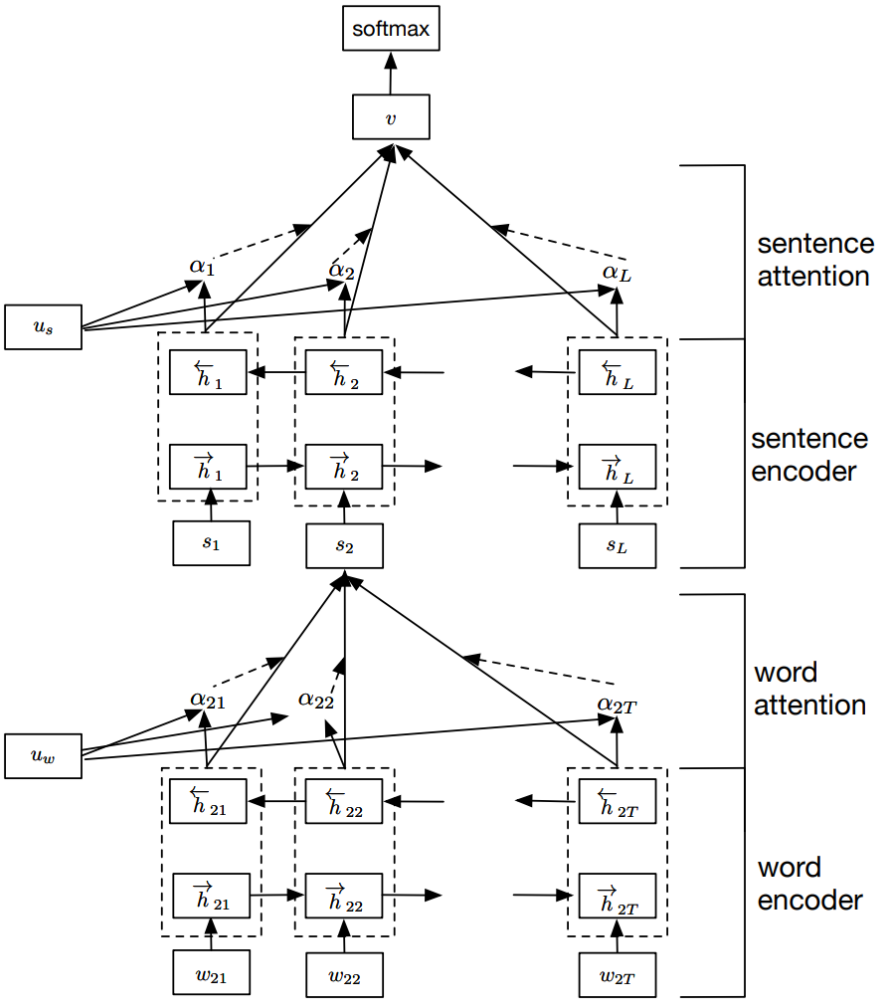
word encoder：对词汇进行编码，建立词向量。接着用双向 GRU 从单词的两个方向汇总信息来获取单词的注释，因此将上下文信息合并到句子向量中。

word attention：对句子向量使用 Attention 机制。

sentence encoder：与上面一样，根据句子向量，使用双向 GRU 构建文档向量。

sentence attention：对文档向量使用 Attention 机制。

softmax：输出分类结果。



⑷ TextCNN 实现文本分类

TextCNN 模型的实现代码如下：

```
import torch
from torch import nn

class TextCNN(nn.Module):
    def __init__(self, vocab_len,
                 embedding_size,
                 max_length=10,
                 kernel_sizes=[3,4,5],
                 out_channels=2):
        super().__init__()
```

```
        self.embedding = nn.Embedding(vocab_len, embedding_size)

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=out_c
           hannels, kernel_size=(kernel_sizes[0], embedding_size
           ))

        self.conv2 = nn.Conv2d(in_channels=1, out_channels=out_c
           hannels, kernel_size=(kernel_sizes[1], embedding_size
           ))

        self.conv3 = nn.Conv2d(in_channels=1, out_channels=out_c
             hannels, kernel_size=(kernel_sizes[2], embedding_si
             ze))

        self.max_pool1 = nn.MaxPool1d(kernel_size=max_length - k
           ernel_sizes[0] + 1)

        self.max_pool2 = nn.MaxPool1d(kernel_size=max_length - k
           ernel_sizes[1] + 1)

        self.max_pool3 = nn.MaxPool1d(kernel_size=max_length - k
           ernel_sizes[2] + 1)

        self.drop_out = nn.Dropout(0.2)
        self.dense = nn.Linear(3*out_channels, 1)

    def forward(self, x):
        embedding = self.embedding(x)
        embedding = embedding.unsqueeze(1)

        conv1_out = self.conv1(embedding).squeeze(-1)
        conv2_out = self.conv2(embedding).squeeze(-1)
        conv3_out = self.conv3(embedding).squeeze(-1)

        out1 = self.max_pool1(conv1_out)
        out2 = self.max_pool2(conv2_out)
        out3 = self.max_pool3(conv3_out)

        out = torch.cat([out1, out2, out3], dim=1).squeeze(-1)
        out = self.drop_out(out)
        out = self.dense(out)
        out = torch.sigmoid(out).squeeze(-1)

        return out
```

TextCNN 模型的训练实现如下：

```
import sys
import os
from numpy.core.defchararray import mod
import torch
from torch import FloatStorage, nn
import pandas as pd
from text_classification.text_cnn import TextCNN
from text_classification.data_process import *
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import train_test_split

# 获取 QA 项目根目录
root_path =os.path.dirname(os.path.dirname(os.path.abspath(__fil
    e__)))

def load_data(batch_size=64):
    #读取分类数据
    df = pd.read_csv(root_path + '/data/classification.csv')
    X = df['sentence']
    y = df['label']

    # 划分训练集与验证集
    X_train, X_valid, y_train, y_valid = train_test_split(X, y,
        test_size=0.3, shuffle=True, random_state=True)

    # 每个句子进行编码并统一长度
    X_train = padding_seq(X_train.apply(seq2index))
    X_valid = padding_seq(X_valid.apply(seq2index))

    y_train = np.array(y_train)
    y_valid = np.array(y_valid)

    train_data_set = TensorDataset(torch.from_numpy(X_train),
                                   torch.from_numpy(y_train))

    train_data_loader = DataLoader(train_data_set,
                                   batch_size=batch_size,
                                   shuffle=True)

    return train_data_loader, X_valid, y_valid
```

```
def train():
    vocab_size = buff_count(os.path.dirname(__file__) + '/vocab.
        txt')
    model = TextCNN(vocab_len=vocab_size, embedding_size=100, ou
        t_channels=100)

    train_data_loader, X_valid, y_valid = load_data(batch_size=6
        4)

    X_valid = torch.from_numpy(X_valid)
    if torch.cuda.is_available():
        model = model.cuda()
        X_valid = X_valid.cuda().long()

    optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
    loss_func = nn.BCELoss()

    best_accuracy = 0.0

    for epoch in range(100):
        for step, (b_x, b_y) in enumerate(train_data_loader):
            if torch.cuda.is_available():
                b_x = b_x.cuda().long()
                b_y = b_y.cuda()
            output = model(b_x)
            loss = loss_func(output, b_y.float())

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if step % 20 == 0:
                valid_output = model(X_valid)
                # 计算验证集上的精度
                y_pred = (valid_output.cpu().data.numpy() >0.5).
                    astype(int)
                valid_accuracy = float((y_pred == y_valid).astyp
                    e(int).sum()) / float(y_valid.size)

                # 保存验证精度最好的模型
                if valid_accuracy > best_accuracy:
                    best_accuracy = valid_accuracy
                    torch.save(model, os.path.dirname(__file__)
                        + '/text_cnn.model')
```

```
                    print('save best valid accuracy: %.3f' %best
                        _accuracy)
                print('epoch:', epoch, '| train loss: %.3f' %los
                    s.cpu().data.numpy(),
                        '| valid accuracy: %.3f' %best_accuracy)

if __name__ == '__main__':
    train()
```

TextCNN 模型的预测实现如下：

```
import os
import torch
from text_classification.data_process import seq2index, padding_
    seq

path = os.path.dirname(__file__)

model = torch.load(path + '/text_cnn.model')
model.eval()

# 对输入的句子
def classification_predict(s):
    s = seq2index(s)
    s = padding_seq([s])

    x = torch.from_numpy(s).cuda().long()
    out = model(x)
    return out.cpu().data.numpy()

if __name__ == '__main__':
    while True:
        s = input('输入句子：').strip()
        print('分类：', classification_predict(s)[0])
```

⑸ BERT fine-tune 实现文本分类

BERT fine-tune 的训练代码实现如下：

```
import os
import numpy as np
import pandas as pd
import torch
from torch import nn
```

```
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset, DataLoader
from transformers import BertTokenizer
from transformers import BertForSequenceClassification
from transformers.optimization import AdamW
from sklearn.metrics import accuracy_score

# 获取 QA 项目根目录
root_path = os.path.dirname(os.path.dirname(os.path.abspath(__fi
    le__)))
bert_model_path = root_path + "/chinese-roberta-wwm-ext"

tokenizer = BertTokenizer.from_pretrained(bert_model_path)

def load_data(batch_size=32):
    #读取分类数据
    df = pd.read_csv(root_path + '/data/classification.csv')
    X = df['sentence']
    y = df['label']

    # 划分训练集与验证集
    X_train, X_valid, y_train, y_valid = train_test_split(X, y,
        test_size=0.3, shuffle=True, random_state=True)

    # 对训练集和验证集数据进行编码
    encoded_input_train = tokenizer(list(X_train), max_length=10
       , padding=True, return_tensors='pt')
    encoded_input_valid = tokenizer(list(X_valid), max_length=10
       , padding=True, return_tensors='pt')

    # 获取 tokenizer 结果的 input_ids 和 attention_mask
    train_inputs = encoded_input_train['input_ids']
    train_masks = encoded_input_train['attention_mask']

    valid_inputs = encoded_input_valid['input_ids']
    valid_masks = encoded_input_valid['attention_mask']

    # Series 转换成 numpy
    y_train = np.array(y_train)
    y_valid = np.array(y_valid)

    # numpy 转换为 tensor
    y_train = torch.tensor(y_train)
    y_valid = torch.tensor(y_valid)
```

```
    # 构建训练集的 TensorDataset 和 DataLoader
    train_data_set = TensorDataset(train_inputs, train_masks, y_
        train)
    train_data_loader = DataLoader(train_data_set,
                                   batch_size=batch_size,
                                   shuffle=True)

    # 构建验证集的 TensorDataset 和 DataLoader
    valid_data_set = TensorDataset(valid_inputs, valid_masks, y_
        valid)
    valid_data_loader = DataLoader(valid_data_set,
                                   batch_size=batch_size,
                                   shuffle=True)

    return train_data_loader, valid_data_loader

def train():
    # 用 Bert 分类模型加载预训练语言模型
    model = BertForSequenceClassification.from_pretrained(bert_m
        odel_path)

    train_data_loader, valid_data_loader = load_data(batch_size=
        32)

    # 模型搬到 GPU 上
    if torch.cuda.is_available():
        model = model.cuda()

    optimizer = AdamW(model.parameters(), lr=5e-5, eps=1e-8)
    loss_func = nn.CrossEntropyLoss()
    best_accuracy = 0.0

    for epoch in range(10):
        # 训练模式
        model.train()

        for step, (b_inputs, b_masks, b_y) in enumerate(train_da
            ta_loader):
            optimizer.zero_grad()

            # 数据搬到 GPU 上
            if torch.cuda.is_available():
                b_inputs = b_inputs.cuda().long()
```

```
            b_masks = b_masks.cuda().long()
            b_y = b_y.cuda()

        # 计算 loss 并进行反向传播
        output = model(b_inputs, b_masks)
        train_loss = loss_func(output.logits, b_y)
        train_loss.backward()
        optimizer.step()

        with torch.no_grad():
            # 计算训练集上的准确率
            preds = torch.argmax(output.logits, dim=1).flatt
                en()
            train_accuracy = accuracy_score(b_y.cpu().data.n
                umpy(),
                preds.cpu().data.numpy())

        if step % 10 == 0:
            print('epoch:', epoch,
                'step:', step,
                '| train loss: %.3f' %train_loss.cpu().data.
                    numpy(),
                '| train accuracy: %.3f' %train_accuracy)
    # 评估模式
    model.eval()
    for step, (b_inputs, b_masks, b_y) in enumerate(valid_da
         ta_loader):
        # 数据搬到 GPU 上
        if torch.cuda.is_available():
            b_inputs = b_inputs.cuda().long()
            b_masks = b_masks.cuda().long()
            b_y = b_y.cuda()

        # 计算验证集上的准确率
        output = model(b_inputs, b_masks)
        valid_loss = loss_func(output.logits, b_y)
        preds = torch.argmax(output.logits, dim=1).flatten()
        valid_accuracy = accuracy_score(b_y.cpu().data.numpy
             (),
            preds.cpu().data.numpy())

        if step % 10 == 0:
            print('epoch:', epoch,
                'step:', step,
```

```
                    '| valid loss: %.3f' %valid_loss.cpu().data.
                    numpy(),
                    '| valid accuracy: %.3f' %valid_accuracy)

            # 保存验证精度最好的模型
            if valid_accuracy > best_accuracy:
                best_accuracy = valid_accuracy
                torch.save(model, os.path.dirname(__file__)
                    + '/bert_text_classification.model')
                print('save best valid accuracy: %.3f' %best_acc
                    uracy)

        print('epoch:', epoch,
            '| train loss: %.3f' %train_loss.cpu().data.numpy(),
            '| train accuracy: %.3f' %train_accuracy,
            '| valid loss: %.3f' %valid_loss.cpu().data.numpy(),
            '| valid accuracy: %.3f' %valid_accuracy)

if __name__ == '__main__':
    train()
```

```
epoch: 0 step: 0 | train loss: 0.729 | train accuracy: 0.531
epoch: 0 step: 10 | train loss: 0.044 | train accuracy: 1.000
epoch: 0 step: 20 | train loss: 0.003 | train accuracy: 1.000
epoch: 0 step: 30 | train loss: 0.001 | train accuracy: 1.000
epoch: 0 step: 40 | train loss: 0.001 | train accuracy: 1.000
save best valid accuracy: 1.000
epoch: 0 step: 0 | valid loss: 0.000 | valid accuracy: 1.000
epoch: 0 step: 10 | valid loss: 0.000 | valid accuracy: 1.000
epoch: 0 | train loss: 0.001 | train accuracy: 1.000 | valid loss: 0.651 | valid accuracy: 0.909
epoch: 1 step: 0 | train loss: 0.001 | train accuracy: 1.000
epoch: 1 step: 10 | train loss: 0.001 | train accuracy: 1.000
epoch: 1 step: 20 | train loss: 0.000 | train accuracy: 1.000
epoch: 1 step: 30 | train loss: 0.001 | train accuracy: 1.000
epoch: 1 step: 40 | train loss: 0.000 | train accuracy: 1.000
epoch: 1 step: 0 | valid loss: 0.000 | valid accuracy: 1.000
epoch: 1 step: 10 | valid loss: 0.000 | valid accuracy: 1.000
epoch: 1 | train loss: 0.000 | train accuracy: 1.000 | valid loss: 0.000 | valid accuracy: 1.000
```

从日志可以看出，经过一个 epoch 后，模型在验证集上的精度达到了 100%，可见 BERT fine-tune 比 TextCNN 模型收敛的速度要更快，分类的效果要更好。

BERT fine-tune 的预测代码实现如下：

```
import os
import torch
from transformers import BertTokenizer

path = os.path.dirname(__file__)
```

```
# 获取 QA 项目根目录
root_path = os.path.dirname(os.path.dirname(os.path.abspath(__fi
    le__)))
bert_model_path = root_path + "/chinese-roberta-wwm-ext"

tokenizer = BertTokenizer.from_pretrained(bert_model_path)
model = torch.load(path + '/bert_text_classification.model')
model.eval()

# 对输入的句子进行预测
def classification_predict(s):
    encoded_input = tokenizer(s, max_length=10, padding=True, re
       turn_tensors='pt')
    input_ids = encoded_input['input_ids']
    attention_mask = encoded_input['attention_mask']

    if torch.cuda.is_available():
        input_ids = input_ids.cuda().long()
        attention_mask = attention_mask.cuda().long()

    output = model(input_ids, attention_mask)
    preds = torch.argmax(output.logits, dim=1).flatten()
    return preds.cpu().data.numpy()

if __name__ == '__main__':
    while True:
        s = input('输入句子：').strip()
        print('分类：', classification_predict(s)[0])
```

11、 文本相似度计算模型

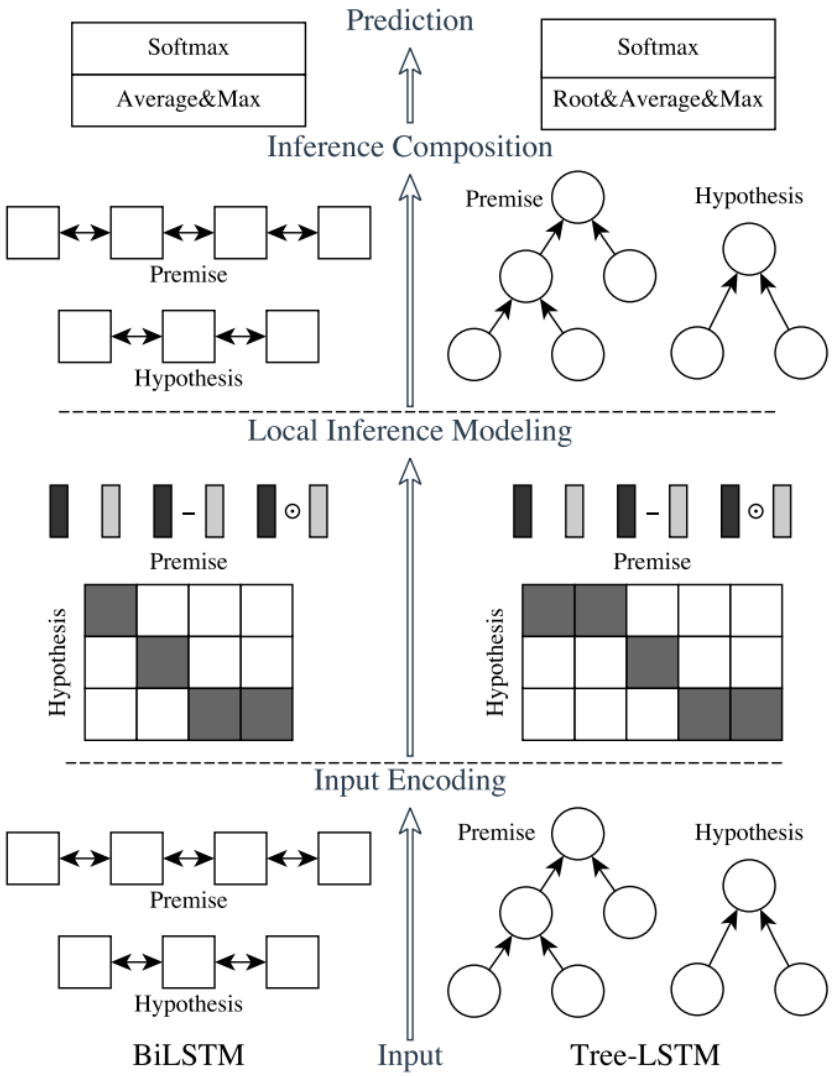
文本相似度计算模型主要作用在于判断两个句子是否是同义句，其结构基本都是孪生网络，其实就是一个二分类的问题。

相似度计算模型有很多种，主要有 DSSM、ESIM、ABCNN、BiMPM、DIIN、DRCN，本项目选用 ESIM 模型用来计算文本相似度。

⑴ ESIM 模型原理

ESIM 是一个表常用的文本匹配模型，其速度和效果都不错。

主要结构如下图所示：



可以看到，ESIM 有两种结构，一个是基于 LSTM，一个是基于 Tree-LSTM，Tree-LSTM 需要先构建依存树，本项目使用的是 BiLSTM 结构。

Input Enconding：是一个经过了 embedding 层之后的双向 LSTM，输出的值是默认把正向与反向的输出值拼接在一起的。

Local Inference Modeling：是对输入的两个句子 p 与 q 相乘，再用相乘的结果分别和 p 与 q 相乘，即 attention 操作，得到加权之后的 p’与 q’。

接下来把[p、p’、p-p’、p*p’]，[q、q’、q-q’、q*q’]拼接在一起作为这层的输出。

Inference Composition：是一个普通的双向 LSTM，使用的是上面一层拼接的结果作为输入，再把双向 LSTM 的输出值给到下一层。

Predict 层：首先把上一层的输出做了一个 Pooling 的操作，取均值与最大值，之后把结果拼接送入全连接层与 softmax 层。

⑵ ESIM 模型实现

ESIM 模型的实现代码如下：

```
import torch
from torch import nn

class ESIM(nn.Module):
    def __init__(self,
                 char_vocab_size,
                 char_dim,
                 char_hidden_size,
                 hidden_size,
                 max_word_len):
        super().__init__()

        # char 的 embedding 层
        self.char_embedding = nn.Embedding(char_vocab_size, char_d
           im)

        # char 的双向 lstm 层
        self.char_lstm = nn.LSTM(input_size=char_dim,
                                 hidden_size=char_hidden_size,
                                 num_layers=1,
                                 bidirectional=True,
                                 batch_first=True)

        # context 的双向 lstm 层
        # inpuput_size 计算方法
        # char_lstm 的双向 lstm 输出为 char_hidden_size*2
        # 经过 attenion 后拼接的输出为 char_hidden_size*2*4
        self.context_lstm = nn.LSTM(input_size=char_hidden_size*8,
                                    hidden_size=hidden_size,
                                    num_layers=1,
                                    bidirectional=True,
                                    batch_first=True)

        # 最大池化层
        self.max_pool = nn.MaxPool2d((max_word_len, 1))

        # 全连接层
        self.dense1 = nn.Linear(char_hidden_size*2*4, hidden_size)
        self.dense2 = nn.Linear(hidden_size, 1)
```

```
        self.drop_out = nn.Dropout(0.2)

    def forward(self, char_p, char_q):
        # 先进入 embedding 层
        p_embedding = self.char_embedding(char_p)
        q_embedding = self.char_embedding(char_q)

        # 经过双向 LSTM
        # 输出为 out, (hidden_state, ceil_state)
        p_embedding, _ = self.char_lstm(p_embedding)
        q_embedding, _ = self.char_lstm(q_embedding)

        # dropout 防止过拟合
        p_embedding = self.drop_out(p_embedding)
        q_embedding = self.drop_out(q_embedding)

        # attention 处理
        # p_embedding:[batch_size, seq_len, char_hidden_size*2]
        # e: [batch_size, seq_len, seq_len]
        e = torch.matmul(p_embedding, torch.transpose(q_embedding,
          1, 2))

        # 对权重矩阵 e 按行进行 softmax,再与 q_embedding 相乘
        # p_hat:[batch_size, seq_len, char_hidden_size*2]
        p_hat = torch.matmul(torch.softmax(e, dim=2), q_embedding)

        # 对权重矩阵 e 按列进行 softmax,再与 p_embedding 相乘
        q_hat = torch.matmul(torch.softmax(e, dim=1), p_embedding)


        # 拼接维度变化
        # p_embedding:[batch_size, seq_len, char_hidden_size*2]
        # p_hat:[batch_size, seq_len, char_hidden_size*2]
        # p_cat:[batch_size, seq_len, char_hidden_size*2*4]
        p_cat = torch.cat([p_embedding, p_hat, p_embedding-p_hat,
           p_embedding*p_hat], dim=-1)
        q_cat = torch.cat([q_embedding, q_hat, q_embedding-q_hat,
           q_embedding*q_hat], dim=-1)

        # Inference Composition 的双向 LSTM
        p, _ = self.context_lstm(p_cat)
        q, _ = self.context_lstm(q_cat)

        # 最化池化，降维把 seq_len 维度去掉
```

```
        p_max = self.max_pool(p).squeeze(dim=1)
        q_max = self.max_pool(q).squeeze(dim=1)

        # 平均池化，直接调用 mean 函数
        p_mean = torch.mean(p, dim=1)
        q_mean = torch.mean(q, dim=1)

        # 按 seq_len 这个维度进行拼接
        x = torch.cat([p_max, p_mean, q_max, q_mean], dim=1)
        x = self.drop_out(x)

        # 经过两个全连接层，再经过 sigmoid 输出二分类
        x = torch.tanh(self.dense1(x))
        x = self.drop_out(x)
        x = torch.sigmoid(self.dense2(x))

        return x.squeeze(dim=-1)
```

ESIM 模型的训练代码如下：

```
import os
import numpy as np
import pandas as pd
import torch
from torch import nn
from text_classification.data_process import *
from text_similarity.esim import ESIM
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset, DataLoader

# 获取 QA 项目根目录
root_path = os.path.dirname(os.path.dirname(os.path.abspath(__fi
le__)))

def load_data(batch_size=32):
    # 读取同义句数据集
    df = pd.read_csv(root_path + '/data/LCQMC.csv')
    X = df[['sentence1', 'sentence2']]
    y = df['label']

    # 划分训练集与验证集
    X_train, X_valid, y_train, y_valid = train_test_split(X, y,
        test_size=0.05, shuffle=True, random_state=True)
```

```
    X_train_p = X_train['sentence1']
    X_train_q = X_train['sentence2']

    X_valid_p = X_valid['sentence1']
    X_valid_q = X_valid['sentence2']

    # 每个句子进行编码并统一长度
    X_train_p = padding_seq(X_train_p.apply(seq2index))
    X_train_q = padding_seq(X_train_q.apply(seq2index))

    X_valid_p = padding_seq(X_valid_p.apply(seq2index))
    X_valid_q = padding_seq(X_valid_q.apply(seq2index))

    y_train = np.array(y_train)
    y_valid = np.array(y_valid)

    train_data_set = TensorDataset(torch.from_numpy(X_train_p),
                                   torch.from_numpy(X_train_q),

                                   torch.from_numpy(y_train))

    train_data_loader = DataLoader(train_data_set,
                                   batch_size=batch_size,
                                   shuffle=True)

    return train_data_loader, [X_valid_p, X_valid_q], y_valid


def train():
    vocab_size = buff_count(os.path.dirname(__file__) + '/vocab.
txt')
    model = ESIM(char_vocab_size=vocab_size, char_dim=100,
                 char_hidden_size=128, hidden_size=128,
                 max_word_len=10)

    train_data_loader, X_valid, y_valid = load_data(batch_size=1
28)
    X_valid_p = X_valid[0]
    X_valid_q = X_valid[1]

    X_valid_p = torch.from_numpy(X_valid_p)
    X_valid_q = torch.from_numpy(X_valid_q)
    if torch.cuda.is_available():
        model = model.cuda()
```

```
            X_valid_p = X_valid_p.cuda().long()
            X_valid_q = X_valid_q.cuda().long()

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

        loss_func = nn.BCELoss()

        best_accuracy = 0.0

        for epoch in range(30):
            running_loss = 0.0
            for step, (b_x_p, b_x_q, b_y) in enumerate(train_data_lo
                 ader):
                if torch.cuda.is_available():
                    b_x_p = b_x_p.cuda().long()
                    b_x_q = b_x_q.cuda().long()
                    b_y = b_y.cuda()
                output = model(b_x_p, b_x_q)
                loss = loss_func(output, b_y.float())

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                if step % 50 == 0:
                    valid_output = model(X_valid_p, X_valid_q)
                    # 计算验证集上的精度
                    y_pred = (valid_output.cpu().data.numpy() >0.5).
                      astype(int)
                    valid_accuracy = float((y_pred == y_valid).astyp
                        e(int).sum()) / float(y_valid.size)

                    # 保存验证精度最好的模型
                    if valid_accuracy > best_accuracy:
                        best_accuracy = valid_accuracy
                        torch.save(model, os.path.dirname(__file__)
                            + '/esim.model')
                        print('save best valid accuracy: %.3f' %best
                            _accuracy)
                    print('epoch:', epoch,
                        '| train loss: %.3f' %loss.cpu().data.numpy(
                        ), '| valid accuracy: %.3f' %valid_accuracy)

if __name__ == '__main__':
```

```
train()
```

经过 30 个 epoch 迭代，ESIM 模型最好的 Valid Accuracy 在 0.885。

⑶ Bert fine-tune 实现

Bert 也是可以实现文本相似度计算的，和文本分类一样也是在预训练模型的基础上进行 fine-tune，用的也是 BertForSequenceClassification 这个模型，只是在做文本相似度计算的时候模型的输入有所不同，多了 token_type_ids，用来区分前后两个句子。

Bert fine-tune 文本相似度计算代码实现如下：

```
import os
import numpy as np
import pandas as pd
import torch
from torch import nn
from torch.nn.utils import clip_grad_norm_
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset, DataLoader
from transformers import BertTokenizer
from transformers import BertForSequenceClassification
from transformers.optimization import AdamW

# 获取 QA 项目根目录
root_path = os.path.dirname(os.path.dirname(os.path.abspath(__fi
    le__)))
bert_model_path = root_path + "/chinese-roberta-wwm-ext"

tokenizer = BertTokenizer.from_pretrained(bert_model_path)

# 截短和 padding 操作
def truncate_and_pad(token_seq_1, token_seq_2, max_seq_len=100):
    if len(token_seq_1) > max_seq_len:
        token_seq_1 = token_seq_1[0:max_seq_len]

    if len(token_seq_2) > max_seq_len:
        token_seq_1 = token_seq_2[0:max_seq_len]

    # 用[SET]分割两个句子
    seq = ['[CLS]'] + token_seq_1 + ['[SEP]'] + token_seq_2 + ['
       [SEP]']
```

```
    # 0 为第二个句子 1 为第二个句子
    seq_segment = [0] * (len(token_seq_1) + 2) + [1] * (len(toke
        n_seq_2) + 1)

    # 转换为 ID
    seq = tokenizer.convert_tokens_to_ids(seq)

    # 需要补 0 的数量
    padding = [0] * (max_seq_len - len(seq))

    # 有效位为 1，其他为 0
    seq_mask = [1] * len(seq) + padding

    # 超过两个句子长度的补 0
    seq = seq + padding
    seq_segment = seq_segment + padding

    return seq, seq_mask, seq_segment

"""
用[CLS][SEP]方式分割句子
用 tokenize 方式分词
手动构建 input_ids, attenion_mask, token_type_ids
"""
def load_data_2(batch_size=32):
    #读取分类数据
    df = pd.read_csv(root_path + '/data/LCQMC.csv')
    X = df[['sentence1', 'sentence2']]
    y = df['label']

    # 划分训练集与验证集
    X_train, X_valid, y_train, y_valid = train_test_split(X, y,
        test_size=0.1, shuffle=True, random_state=42)

    train_sentences_1 = X_train['sentence1']
    train_sentences_2 = X_train['sentence2']

    valid_sentences_1 = X_valid['sentence1']
    valid_sentences_2 = X_valid['sentence2']

    # 对训练集句子进行分词
    train_tokens_seq_1 = list(map(tokenizer.tokenize, train_sent
        ences_1))
```

```
train_tokens_seq_2 = list(map(tokenizer.tokenize, train_sent
    ences_2))

# 对训练集句子进行分词
valid_tokens_seq_1 = list(map(tokenizer.tokenize, valid_sent
    ences_1))
valid_tokens_seq_2 = list(map(tokenizer.tokenize, valid_sent
    ences_2))

# 截短和 Padding 操作
train_pad = list(map(truncate_and_pad, train_tokens_seq_1, t
    rain_tokens_seq_2))
valid_pad = list(map(truncate_and_pad, valid_tokens_seq_1, v
    alid_tokens_seq_2))

# 取出 input_ids, attention_mask, token_type_ids
train_seqs = torch.tensor([i[0] for i in train_pad])
train_seq_masks = torch.tensor([i[1] for i in train_pad])
train_seq_segments = torch.tensor([i[2] for i in train_pad])

valid_seqs = torch.tensor([i[0] for i in valid_pad])
valid_seq_masks = torch.tensor([i[1] for i in valid_pad])
valid_seq_segments = torch.tensor([i[2] for i in valid_pad])

# Series 转换成 numpy
y_train = np.array(y_train)
y_valid = np.array(y_valid)

# numpy 转换为 tensor
y_train = torch.tensor(y_train)
y_valid = torch.tensor(y_valid)

# 构建训练集的 TensorDataset 和 DataLoader
train_data_set = TensorDataset(train_seqs, train_seq_masks,
    train_seq_segments, y_train)
train_data_loader = DataLoader(train_data_set,
                               batch_size=batch_size,
                               shuffle=True)

# 构建验证集的 TensorDataset 和 DataLoader
valid_data_set = TensorDataset(valid_seqs, valid_seq_masks,
     valid_seq_segments, y_valid)
valid_data_loader = DataLoader(valid_data_set,
                               batch_size=batch_size,
```

```
                                    shuffle=True)

    return train_data_loader, valid_data_loader

"""
直接用 tokenizer 方式编码
生成 input_ids, attention_mask, token_type_ids
"""
def load_data(batch_size=32):
    #读取分类数据
    df = pd.read_csv(root_path + '/data/LCQMC.csv')
    X = df[['sentence1', 'sentence2']]
    y = df['label']

    # 划分训练集与验证集
    X_train, X_valid, y_train, y_valid = train_test_split(X, y,
        test_size=0.1, shuffle=True, random_state=42)

    X_train_p = X_train['sentence1']
    X_train_q = X_train['sentence2']

    X_valid_p = X_valid['sentence1']
    X_valid_q = X_valid['sentence2']

    # 对训练集和验证集数据进行编码
    encoded_train = tokenizer(list(X_train_p), list(X_train_q),
        max_length=100, padding=True, return_tensors='pt')

    encoded_valid = tokenizer(list(X_valid_p), list(X_valid_q),

        max_length=100, padding=True, return_tensors='pt')

    # 获取 tokenizer 结果的 input_ids 和 attention_mask
    train_inputs = encoded_train['input_ids']
    train_masks = encoded_train['attention_mask']
    train_segments = encoded_train['token_type_ids']

    valid_inputs = encoded_valid['input_ids']
    valid_masks = encoded_valid['attention_mask']
    valid_segments = encoded_valid['token_type_ids']

    # Series 转换成 numpy
    y_train = np.array(y_train)
```

```
    y_valid = np.array(y_valid)

    # numpy 转换为 tensor
    y_train = torch.tensor(y_train)
    y_valid = torch.tensor(y_valid)

    # 构建训练集的 TensorDataset 和 DataLoader
    train_data_set = TensorDataset(train_inputs, train_masks, tr
        ain_segments, y_train)
    train_data_loader = DataLoader(train_data_set,
                                   batch_size=batch_size,
                                   shuffle=True)

    # 构建验证集的 TensorDataset 和 DataLoader
    valid_data_set = TensorDataset(valid_inputs, valid_masks, va
        lid_segments, y_valid)
    valid_data_loader = DataLoader(valid_data_set,
                                   batch_size=batch_size,
                                   shuffle=True)


    return train_data_loader, valid_data_loader

def train():
    # 用 Bert 分类模型加载预训练语言模型
    model = BertForSequenceClassification.from_pretrained(bert_m
        odel_path, num_labels=2)

    train_data_loader, valid_data_loader = load_data(batch_size=
        64)

    # 模型搬到 GPU 上
    if torch.cuda.is_available():
        model = model.cuda()

    # 待优化的参数
    param_optimizer = list(model.named_parameters())
    no_decay = ['bias', 'LayerNorm.bias', 'LayerNorm.weight']

    # no_decy 里面的不进行 weight_decay
    optimizer_grouped_parameters = [{
        'params':[p for n, p in param_optimizer if not any(nd in
          n for nd in no_decay)],
        'weight_decay':0.01},
```

```
    {'params':[p for n, p in param_optimizer if any(nd in n
       for nd in no_decay)],
    'weight_decay':0.0}]

optimizer = AdamW(optimizer_grouped_parameters, lr=2e-05)
best_accuracy = 0.0

for epoch in range(10):
    # 训练模式
    model.train()

    # 每个 epoch 总的 loss
    train_loss = 0.0

    # 每个 epoch 总计预测正确数量
    train_correct_preds = 0

    # 每个 epoch 总计处理过多少个问题
    train_question_count = 0

    for step, (b_inputs, b_masks, b_segments, b_y) in enumer
       ate(train_data_loader):
        optimizer.zero_grad()

         # 数据搬到 GPU 上
        if torch.cuda.is_available():
            b_inputs = b_inputs.cuda().long()
            b_masks = b_masks.cuda().long()
            b_segments = b_segments.cuda().long()
            b_y = b_y.cuda()

        output = model(input_ids=b_inputs,
                       attention_mask=b_masks,
                       token_type_ids=b_segments,
                       labels=b_y)
        # 取出 loss, logits
        loss, logits = output.loss, output.logits

        # 对 logits 进算 softmax
        probabilities = torch.softmax(logits, dim=-1)
        loss.backward()
        # 进行梯度裁剪
        clip_grad_norm_(model.parameters(), max_norm=10.0)
```

```
        optimizer.step()

        # 每个 step 的 loss 相加，便于后面计算每个 epoch 的 loss
        train_loss += loss.item()

        # 取出最大预测值
        _, out_classes = probabilities.max(dim=1)

        # 计算每个 step 预测正确的数量
        correct = (out_classes == b_y).sum()

        # 每个 epoch 预测正确的数量累加
        train_correct_preds += correct.item()

        # 每个 epoch 预测的总量累加
        train_question_count += len(b_y)

        # 每个 epoch 预测正确的数量 / 每个 epoch 预测的总数量
        train_accuracy = train_correct_preds / train_questio
            n_count

        if step % 5 == 0:
            print('epoch:', epoch,
                'train step:', step,
                '| train loss: %.3f' %loss.item(),
                '| train accuracy: %.3f' %train_accuracy)

    # 评估模式
    model.eval()

    valid_loss = 0.0
    valid_correct_preds = 0
    valid_question_count = 0

    for step, (b_inputs, b_masks, b_segments, b_y) in enumer
         ate(valid_data_loader):

        # 数据搬到 GPU 上
        if torch.cuda.is_available():
            b_inputs = b_inputs.cuda().long()
            b_masks = b_masks.cuda().long()
            b_segments = b_segments.cuda().long()
            b_y = b_y.cuda()
```

```
        output = model(input_ids=b_inputs,
                       attention_mask=b_masks,
                       token_type_ids=b_segments,
                       labels=b_y)
        loss, logits = output.loss, output.logits
        probabilities = torch.softmax(logits, dim=-1)

        valid_loss += loss.item()
        _, out_classes = probabilities.max(dim=1)
        correct = (out_classes == b_y).sum()
        valid_correct_preds += correct.item()
        valid_question_count += len(b_y)
        valid_accuracy = valid_correct_preds / valid_questio
            n_count

        if step % 5 == 0:
            print('epoch:', epoch,
                'valid step:', step,
                '| valid loss: %.3f' %loss.item(),
                '| valid accuracy: %.3f' %valid_accuracy)

        # 保存验证精度最好的模型
        if valid_accuracy > best_accuracy:
            best_accuracy = valid_accuracy
            torch.save(model, os.path.dirname(__file__)
                + '/bert_text_similarity.model')
            print('save best valid accuracy: %.3f' %best_acc
                uracy)

    # 计算每个 epoch 的 train loss, train accuracy
    epoch_train_loss = train_loss / len(train_data_loader)
    epoch_train_accuracy = train_correct_preds / len(train_d
        ata_loader.dataset)

    # 计算每个 epoch 的 valid loss, valid_accuracy
    epoch_valid_loss = valid_loss / len(valid_data_loader)
    epoch_valid_accuracy = valid_correct_preds / len(valid_d
        ata_loader.dataset)

    print('epoch:', epoch,
        '| train loss: %.3f' %epoch_train_loss,
        '| train accuracy: %.3f' %epoch_train_accuracy,
        '| valid loss: %.3f' %epoch_valid_loss,
        '| valid accuracy: %.3f' %epoch_valid_accuracy)
```

```
if __name__ == '__main__':
    train()
```

经过 10 个 epoch 训练，最好的 Valid Accuary 为 0.925，比 ESIM 模型的精度提高了一点。

Ber fine-tune 文本相似度计算预测代码实现如下：

```
import os
import torch
from transformers import BertTokenizer

path = os.path.dirname(__file__)

# 获取 QA 项目根目录
root_path = os.path.dirname(os.path.dirname(os.path.abspath(__fi
    le__)))
bert_model_path = root_path + "/chinese-roberta-wwm-ext"

tokenizer = BertTokenizer.from_pretrained(bert_model_path)
model = torch.load(path + '/bert_text_similarity.model')
model.eval()

# 对输入的句子进行预测
def bert_similarity_predict(sentence1, sentence2):

    encoded_input = tokenizer(list(sentence1), list(sentence2),
        max_length=100, padding=True,
        return_tensors='pt')

    input_ids = encoded_input['input_ids']
    attention_mask = encoded_input['attention_mask']
    token_type_ids = encoded_input['token_type_ids']

    if torch.cuda.is_available():
        input_ids = input_ids.cuda().long()
        attention_mask = attention_mask.cuda().long()
        token_type_ids = token_type_ids.cuda().long()

    output = model(input_ids=input_ids,
                   attention_mask=attention_mask,
                   token_type_ids=token_type_ids)
    logits = output.logits
```

```
    probabilities = torch.softmax(logits, dim=-1)
    similarity = probabilities.max(dim=1)
    preds = similarity[0].cpu().data.numpy()

    return preds

if __name__ == '__main__':
    sentence1 = ["老年人怎么买保险？"]
    sentence2 = ["70 岁以上老人怎么买保险？"]

    print('文本相似度：
        ', bert_similarity_predict(sentence1, sentence2)[0])
```

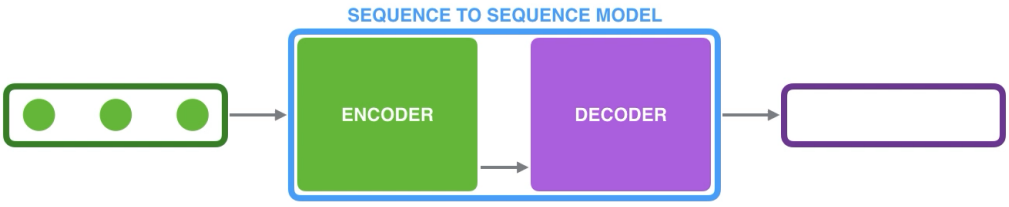
12、 闲聊模型

⑴ seq2seq 模型

sequence-to-sequence 模型是对序列的 item 建模，例如文字，单词，或者图像特征等，输出的内容是序列的 item，主要由 encoder 和 decoder 两部分组成。

encoder 用来处理每一个时刻的输入序列，encoder 会捕获输入序列的信息到一个 vector 中，该 vector 称做 context，在处理完整个 encoder 阶段的输入序列后，encoder 会把 context 发送给 decoder，decoder 一个接一个的生成新的输出序列。

context 是一个向量，在文本类任务中，通常是采用 encoder 的最后一个时刻的 hidden state 作为 context，这个 context 的大小是可以设置的，其维度等于 rnn 的 hidden state 的维度，可能是 256，512 甚至 1024。

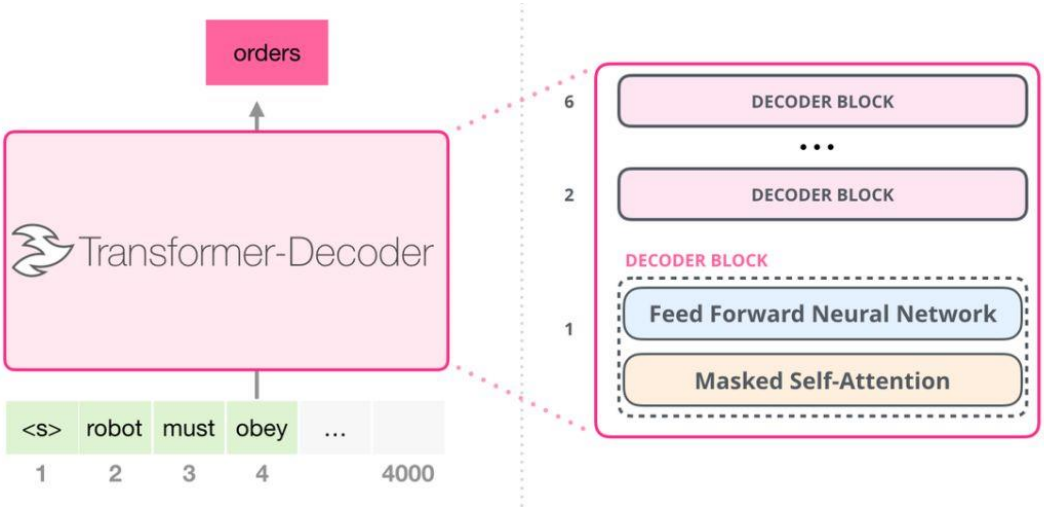


seq2seq 是一种多模态模型，可以是文本到文本（机器翻译）、图像到文本（看图说话）、文本到图像等。

seq2seq 模型是一种典型的 RNN 结构模型，由于 RNN 本身是串行结构，所以我们在这使用效率更高的并行结构模型 GPT 来做闲聊模型。

⑵ GPT 模型

GPT 是只包含了 Transformer 的 Decoder 结构，并且去除了第二个 Encoder Decoder Self-Attention 模块，使用 Mask Self-Attetion，模型结构如下图：



在 Bert 的 Self-Attention 中，由于 Q=K=V，因此可以看到 Q*K 相乘得到的 attention matrix 权重矩阵中所有的信息，也就是可以看到整个序列中的所有词，这样容易带来标签泄漏的问题。

而在 Mask Self-Attention 中只能看到当前输入的单词之前的词，对于一个序列，在 time_step 为 t 的时刻，我们的解码输出应该只能依赖于 t 时刻之前的输出，而不能依赖 t 之后的输出。

GPT 是利用上文预测下一个单词，ELMO 和 BERT 是根据上下文预测单词，因此 GPT 更加适合用于文本生成的任务，因为文本生成通常都是基于当前已有的信息，生成下一个单词，所以本项目使用 GPT 来做闲聊模型。

本项目使用 DialoGPT 来生成闲聊模型，DialoGPT 建模目标是：

$$p(T|S) = \prod_{n=m+1}^{N} p(x_n|x_1, \cdots, x_{n-1})$$

S 表示的是历史的对话信息，T 表示的是生成的回答。

在这里我们使用预训练好的 GPT 闲聊模型集成到项目中，下载地址：

https://github.com/yangjianxin1/GPT2-chitchat

⑶ 解码方法

Greedy search：所有的词都选择概率值最高，只考虑了局部最优。

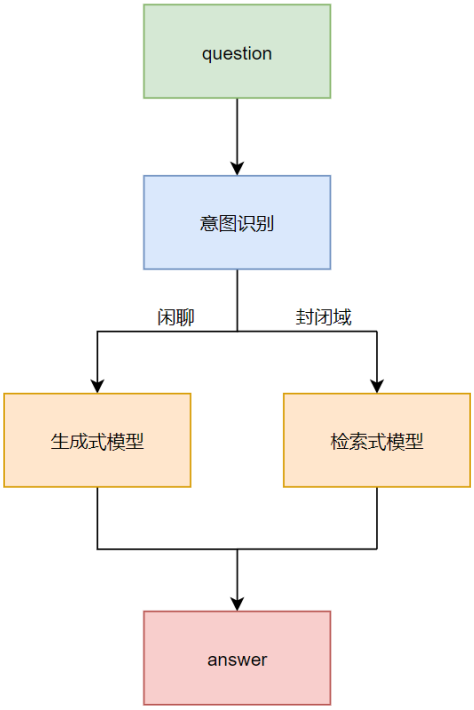
Beam search：每个词选择概率值的 Top k 个，概率相乘（实际中为取对数相加）最高的为最终的结果。Beam search 的问题在于过于严谨的回答，且容易出现重复的字词，因此可以采用一些采样机制，例如 topk 与 topp 采样。

随机采样 Topk：选取概率值最大的 K 个词，先进行归一化，再进行随机采样。

随机采样 Topp：选取概率值的和不超过 p 的词，对这些词进行归一化，再进行随机采样。

四、 项目整合

到目前为止，已经实现了本项目所有的功能模块，现在需要将这些功能模块整合到一起，实现一个完整的智能问答系统。



首先，输入一个问题，通过文本分类模型 TextCNN 或者 Bert fine-tune 训练的模型进行意图识别，判断是闲聊还是封闭域问题。

对于闲聊问题，用 DialoGPT 生成式模型，调用 chitchat 函数实现闲聊回答。

对于检索式模型，可以自由选用 Word2Vec、ELMo、Bert、sentence-transformers 这四种模型来生成句向量，用余弦相似度来计算和输入问题的相似度，选择一个相似度最高的问题的回答来作为答案。

接下来再对余弦相似度最高的 10 个问题，使用 ESIM 文本相似度模型和 Bert fine-tune 训练的模型分别选择一个和输入问题相似度最高的问题的回答作为答案，并可以对比 ESIM 和 Bert fine-tune 的选择。

项目整合实现的代码如下：

```
import os
import jieba
import numpy as np
import pandas as pd
import torch
from elmoformanylangs import Embedder
from gensim.models import Word2Vec
from sentence_transformers import SentenceTransformer
from transformers import AutoModel, AutoTokenizer

from bert_text_classification.predict import bert_classification
_predict
from chitchat.interact import chitchat
from text_classification.predict import classification_predict
from text_similarity.predict import similarity_predict
from bert_text_similarity.predict import bert_similarity_predict

# 获取 QA 项目根目录
root_path = os.path.dirname(__file__)

df = pd.read_csv(root_path + '/data/qa_data.csv')
questions = df['question'].values
answers = df['answer'].values

# 句子转化为句向量
def sen2vec(model, sentence):
    # 对句子进行分词
    segment = list(jieba.cut(sentence))
    vec = np.zeros(100)
    for s in segment:
        try:
            # 取出 s 对应的向量相加
            vec += model.wv[s]
```

```
            #出现 oov 问题，词不在词典中
            except:
                pass

        # 采用加权平均求句向量
        vec /= len(segment)
        return vec


def elmo2vec(model, sentence):
    '''
    output_layer 参数.
    0 for the word encoder
    1 for the first LSTM hidden layer
    2 for the second LSTM hidden layer
    -1 for an average of 3 layers. (default)
    -2 for all 3 layers
    '''
    if isinstance(sentence, str):
        segment = list(jieba.cut(sentence))
        # 用 elmo 转换词向量
        vec = model.sents2elmo([segment], output_layer=-1)
    elif isinstance(sentence, np.ndarray):
        segment = [jieba.cut(s) for s in sentence]
        # 用 elmo 转换词向量
        vec = model.sents2elmo(segment, output_layer=-1)

    # 句向量取均值
    return [np.mean(v, axis=0) for v in vec]

# 平均池化
def mean_pooling(model_output, attention_mask):
    token_embeddings = model_output[0]
    input_mask_expanded = attention_mask.unsqueeze(-1).expand(to
        ken_embeddings.size()).float()
    sum_embeddings = torch.sum(token_embeddings * input_mask_exp
        anded, 1)
    sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
    return sum_embeddings / sum_mask

# 用 transformers 转换句向量
def bert_to_vec(model, tokenizer, sentence):
    print('bert encode start')
    if isinstance(sentence, np.ndarray):
        encoded_input = tokenizer(list(sentence), padding=True,
```

```
            truncation=True, max_length=128, return_tensors='pt'
             )
    else:
        encoded_input = tokenizer(sentence, padding=True,
            truncation=True, max_length=128, return_tensors='pt'
             )

    with torch.no_grad():
        model_output = model(**encoded_input)
    sentence_embeddings = mean_pooling(model_output, encoded_inp
       ut['attention_mask'])
    print('bert encode finish')
    return sentence_embeddings.numpy()

# 用 sentence_transformers 转换句向量
def sentence_to_vec(model, sentence):
    if isinstance(sentence, np.ndarray):
        embedding = model.encode(sentence, batch_size=64, show_p
         rogress_bar=True, device='cuda:0')
    else:
        embedding = model.encode(sentence)
    return embedding

# 计算余弦相似度
def cosine(a, b):
    # 矩阵的积除以矩阵模的积
    return np.matmul(a, np.array(b).T) / (np.linalg.norm(a) * np
       .linalg.norm(b, axis=1))

if __name__ == '__main__':
    while True:
        text = input('请输入您的问题：').strip()

        # 判断是封闭域还是闲聊问题
        # TextCNN 和 Bert fine-tune 作对比
        prob_cnn = round(classification_predict(text)[0], 3)
        print("TextCNN 预测是闲聊的概率为：", prob_cnn)

        prob_bert = round(float(bert_classification_predict(text
           )[0]), 3)
        print("Bert 预测是闲聊的概率为：", prob_bert)

        if (prob_cnn > 0.5) or (prob_bert > 0.5):
            print("当前输入的问题为闲聊")
```

```
        print("闲聊回答：", chitchat(text))
        continue
    else:
        print("当前输入的问题为封闭域问题")

    while True:
        v = int(input("请选择句向量编码方式：\n" +
                "1. Word2Vec \n" +
                "2. ElMo \n" +
                "3. Bert \n" +
                "4. sentence-transformers \n",).strip())

        if v != 1 and v != 2 and v != 3 and v != 4:
            print("输入的句向量编码方式错误，请重新输入")
            continue
        else:
            break

    print("正在将问题库中的所有问题转换为句向
       量...")

    # 文本表示，转换为句向量
    vec = None

    # Word2Vec
    if v == 1:
        v_str = "Word2Vec"
        model_path = root_path + '/word2vec/wiki.model'
        model = Word2Vec.load(model_path)

        # 生成所有问题的句向量
        question_vec = []
        for q in questions:
            question_vec.append(sen2vec(model, q))

        # 生成当前输入问题的句向量
        vec = sen2vec(model, text)
    # Elmo
    elif v == 2:
        v_str = "ELMo"
        model_path = root_path + '/elmo/zhs.model'
        model = Embedder(model_path)

        # 生成所有问题的句向量
```

```
        question_vec = []
        for q in questions:
            question_vec.extend(elmo2vec(model, q))

        # 生成当前输入问题的句向量
        vec = elmo2vec(model, text)[0]
    # Bert
    elif v == 3:
        v_str = "Bert"
        model_path = root_path + "/chinese-roberta-wwm-ext"

        tokenizer = AutoTokenizer.from_pretrained(model_path
           )
        model = AutoModel.from_pretrained(model_path)

         # 生成所有问题的句向量
        question_vec = bert_to_vec(model, tokenizer, questio
           ns)

        # 生成当前输入问题的句向量
        vec = bert_to_vec(model, tokenizer, text)
    # sentence transformers
    elif v == 4:
        v_str = "sentence-transformers"
        model_path =  root_path + "/paraphrase-multilingual-
           MiniLM-L12-v2"
        model = SentenceTransformer(model_path, device='cuda
           :0')

        # 将所有问题库中问题转换为句向量
        question_vec = sentence_to_vec(model, questions)

        # 生成当前输入问题的句向量
        vec = sentence_to_vec(model, text)

    # 计算输入的问题和问题库中问题的相似度
    similarity = cosine(vec, question_vec)

    # Bert 的是二维数组，需要拉成一维数组
    if v == 3:
        similarity = similarity.ravel()

    # 取最大相似度
    max_similarity = max(similarity)
```

```
print(v_str + " 最大相似度：", max_similarity)
index = np.argmax(similarity)
if max_similarity < 0.8:
    print('没有找到对应的问题，您想问的是不是：
       ', questions[index])
    continue

print(v_str + ' 最相似的问题：', questions[index])
print(v_str + ' 答案：
   ', answers[index][0:100], "...")

top_10_similarity = np.argsort(-similarity)[0:10]
top_10_question = questions[top_10_similarity]
esim_similarity = similarity_predict([text] * 10, top_10
   _question)
bert_similarity = bert_similarity_predict([text] * 10, t
   op_10_question)
index_dic = {}
print(v_str + ' 和 ESIM Bert Top10 候选集：')
df_top_10 = pd.DataFrame(columns=['question', v_str, 'ES
   IM', 'Bert'])
pd.set_option('colheader_justify', 'center')

for i, index in enumerate(top_10_similarity):
    df_top_10.loc[i] = [top_10_question[i], similarity[i
       ndex], esim_similarity[i], bert_similarity[i]]
    index_dic[i] = index

print(df_top_10)

esim_index = np.argsort(-esim_similarity)[0]
print('ESIM 最相似的问题：第' + str(esim_index) + '个',
    questions[index_dic[esim_index]])
print('ESIM 答
   案:', answers[index_dic[esim_index]][0:100], "...")

bert_index = np.argsort(-bert_similarity)[0]
print('Bert 最相似的问题：第' + str(bert_index) + '个',
    questions[index_dic[bert_index]])
print('Bert 答
   案:', answers[index_dic[bert_index]][0:100], "...")
```

项目整合运行的结果如下：

请输入您的问题：老年人怎么买保险
TextCNN 预测是闲聊的概率为： 0.0
Bert 预测是闲聊的概率为： 0.0
当前输入的问题为封闭域问题
请选择句向量编码方式：
1. Word2Vec
2. ElMo
3. Bert
4. sentence-transformers
1
正在将问题库中的所有问题转换为句向量...
Word2Vec 最大相似度： 0.9354390089955581
Word2Vec 最相似的问题： 18岁怎么买保险
Word2Vec 答案： 18岁是刚刚步入社会的年龄，事业没有稳定，也就没有稳定的经济来源，很多年轻人都是月光族或者学生，基本上谈不上存款，理财。因此养成强制性存款的习惯很重要，另外重疾和住院医疗之类的保障型险种也是必须要考虑 ...
Word2Vec 和 ESIM Bert Top10 候选集：

	question	Word2Vec	ESIM	Bert
0	18岁怎么买保险	0.935439	0.012047	0.990777
1	儿童大病保险怎么买？给小孩子怎么买医疗保险？	0.932112	0.008959	0.996257
2	未成年人要买保险吗	0.927549	0.005981	0.996361
3	平安福保险怎么样？给老年人买可以吗	0.922342	0.002192	0.983991
4	信用卡黑户可以买保险吗	0.916736	0.002022	0.997086
5	给小孩5岁买保险需要户口本吗	0.911541	0.003315	0.997526
6	买车保险怎么买	0.905734	0.526106	0.993165
7	16岁，买什么保险好？	0.905249	0.264077	0.987225
8	平安福给老年人买可以吗？	0.902203	0.007072	0.996949
9	有农村医疗保险还需要买疾病保险吗？	0.901213	0.002043	0.997678

ESIM最相似的问题：第6个 买车保险怎么买
ESIM答案：对于新手买车，保险方面要做好这些保险的投保：交强险、第三者责任险、车辆损失险、车上人员险、无过失责任险、不计免赔特约险等等。首先，交强险是首要购买的险种。其次，对于驾驶技能不熟练的新手，除了投保交强险 ...
Bert最相似的问题：第9个 有农村医疗保险还需要买疾病保险吗？
Bert答案：农村合作医疗是以社会平均化，起点不高，比例偏低，农村医疗保险有保而不包的缺陷，所以社保是基础，要先投社保再去投商业保险，在有农村医疗保险的基础上添加商业疾病保险是必要的。商业疾病保险能够弥补农村医疗保 ...

请输入您的问题：30岁的人买哪种保险好
TextCNN 预测是闲聊的概率为： 0.0
Bert 预测是闲聊的概率为： 0.0
当前输入的问题为封闭域问题
请选择句向量编码方式：
1. Word2Vec
2. ElMo
3. Bert
4. sentence-transformers
4
正在将问题库中的所有问题转换为句向量...
Batches: 100%|██████████| 60/60 [00:01<00:00, 46.73it/s]
sentence-transformers 最大相似度： 0.9308185
sentence-transformers 最相似的问题： 23岁买什么保险好啊？
sentence-transformers 答案： 根据您提供的信息，建议您购买一份综合意外保险。保障普通意外、意外医疗、交通意外、住院津贴等等。产品推荐：“快乐嘟嘟-吉祥人生”意外伤害保险：http://www.hzins.com/pr###ct/a ...
sentence-transformers 和 ESIM Bert Top10 候选集：

	question	sentence-transformers	ESIM	Bert
0	23岁买什么保险好啊？	0.930818	0.950781	0.987265
1	29岁人想买份保障比较全面的保险，买哪个险种合适	0.920261	0.011859	0.754948
2	39岁的人适合买新华保险的哪个险种	0.914711	0.013590	0.993451
3	今年23岁应该买什么类型的保险	0.913497	0.012260	0.973440
4	三十多岁的人买平安保险，月收入8000，适合买哪种险种	0.881692	0.066652	0.952664
5	商业保险可靠吗？27岁买什么商业保险好？	0.880111	0.005270	0.994478
6	26岁该如何选择一份合适的保险？	0.861960	0.005768	0.994934
7	在人寿保险中35岁的人适合买什么险中	0.853407	0.923426	0.543210
8	44岁的人上什么保险好？	0.841219	0.206861	0.992290
9	适合中老年的保险多不多，能买哪些保险？	0.823819	0.023488	0.994572

ESIM最相似的问题：第0个 23岁买什么保险好啊？
ESIM答案：根据您提供的信息，建议您购买一份综合意外保险。保障普通意外、意外医疗、交通意外、住院津贴等等。产品推荐：“快乐嘟嘟-吉祥人生”意外伤害保险：http://www.hzins.com/pr###ct/a ...
Bert最相似的问题：第6个 26岁该如何选择一份合适的保险？
Bert答案：26岁这个年龄适合购买的保险险种有很多，但考虑到工作上面刚稳定，还是选择一些保意外和医疗方面的基础产品，只有在足够保障的前提下，在去考虑其他类型的保险，各种保险不是一次性就能买足的，可以在人生的不同阶 ...

```
请输入您的问题：吃饭了吗
TextCNN 预测是闲聊的概率为： 1.0
Bert 预测是闲聊的概率为： 1.0
当前输入的问题为闲聊
闲聊回答： 没有
请输入您的问题：在干嘛
TextCNN 预测是闲聊的概率为： 1.0
Bert 预测是闲聊的概率为： 1.0
当前输入的问题为闲聊
闲聊回答： 上班
请输入您的问题：你多大了
TextCNN 预测是闲聊的概率为： 1.0
Bert 预测是闲聊的概率为： 1.0
当前输入的问题为闲聊
闲聊回答： 高三
请输入您的问题： 你在上学吗
TextCNN 预测是闲聊的概率为： 1.0
Bert 预测是闲聊的概率为： 1.0
当前输入的问题为闲聊
闲聊回答： 我是高二
```

五、 项目总结

这个智能问答项目把所学的NLP知识全部串连起来了，是一个非常好的NLP项目，通过把老师讲的知识重新进行梳理，把所有的功能自己从头至尾实现一遍，我的收获非常大，不但加深了对理论知识的理解，也极大地提升了自己的编码和代码调试能力，使我对NLP领域有了新的认知。

在这里我要对集训营的所有老师衷心地说句谢谢，今后我将在NLP领域持续学习新的知识，不断提升自己的竞争力。