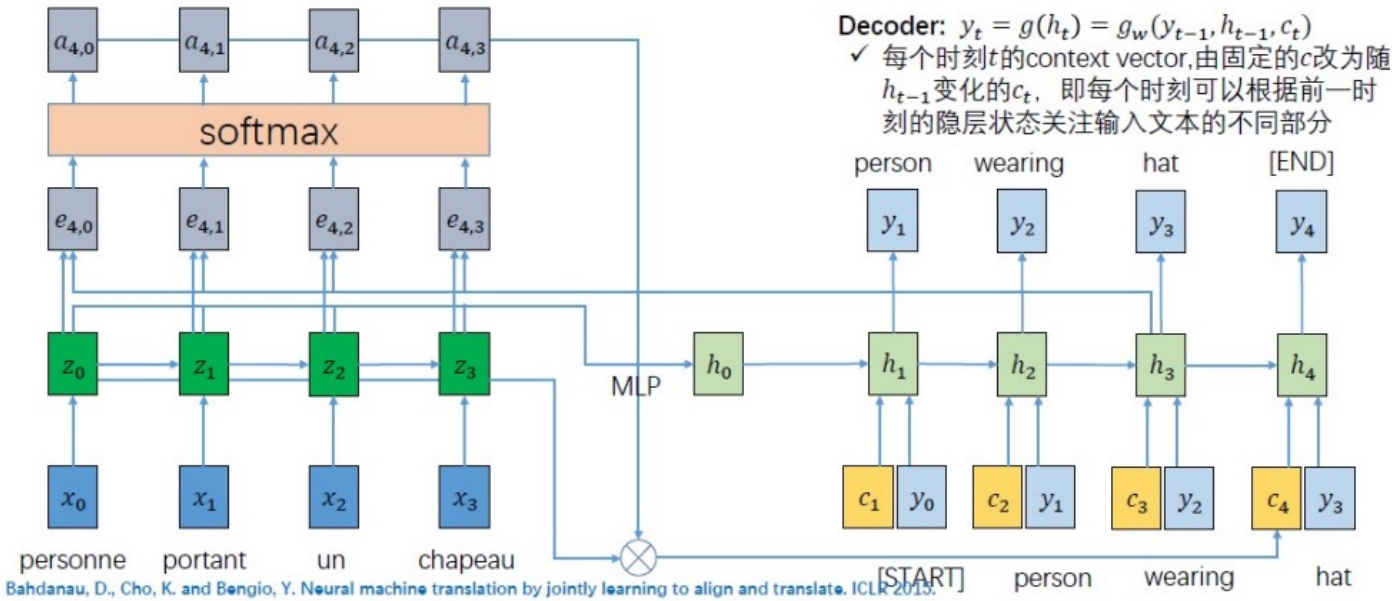
# Lab9_attention mechanism作业情况




Input: $x = x_1, x_2, .., x_T$
Output: $y = y_1, y_2, .., y_T$

Encoder: $z_t = RNN(x_t, z_{t-1})$
$z_t$为编码器的隐藏状态

$$h_0 = \frac{1}{n}\sum_{i=1}^{T}\{MLP_1([z_1, z_1, \dots, z_t])\}$$

**或者 $h_0 = z_T$ (请从上述两种方式中选择一种有效的方式)**

Decoder:
$h_t = RNN(h_{t-1}, [c_t \oplus y_{t-1}])$
$h_t$为解码器的隐藏状态

$c_t = attn(Z, h_{t-1})$
其中$Z = [z_1, z_2, \dots, z_T]$

$$\alpha_i = \frac{exp^{z_i^T h_{t-1}}}{\sum_{j=1}^{T} exp^{z_j^T h_{t-1}}}$$

$$c_t = \sum_{i=1}^{T} \alpha_i z_i$$

Output： $\hat{y_t} = MLP_2(h_t)$
$\hat{y_t}$为预测的字符

# Lab9_attention mechanism参考答案 1

```python
def decode(self, dec_y, enc_hidden, state, is_train = True):
    dec_embs = self.emb_layer(dec_y)  # 进行词嵌入
    trg_len = dec_y.shape[1]          # 获取了目标序列的长度
    outputs = []
    state = state.transpose(1, 0)
    for t in range(trg_len):
        scores = torch.bmm(state, enc_hidden.transpose(2, 1))                # 计算解码器当前时间步的注意力分数
        alpha = self.softmax(scores)                                         # 转换为注意力权重(batch_size, 1, seq_len)
        cont_vec = torch.bmm(alpha, enc_hidden).squeeze(1)                   # 根据注意力权重，计算当前时间步的上下文向量
        input_vec = torch.cat([cont_vec, dec_embs[:, t, :]], 1).unsqueeze(1) # 将上下文向量和解码器输入序列的词嵌入向量拼接在一起
        sent_hidden, state = self.decoder(input_vec, state.transpose(0, 1))  # 将当前时间步的输入向量输入到解码器中
        state = state.transpose(0, 1)              # 再次转置解码器的状态，以恢复原始的维度顺序。
        pred = self.linear(sent_hidden)            # 输出预测字符
        outputs += [pred]
    sent_outputs = torch.cat(outputs, dim = 1)     # 将所有时间步的预测结果连接起来，形成最终的输出序列
    return sent_outputs, state
```

# Lab9_attention mechanism参考答案 2

```python
def encode(self, enc_x, state):
    enc_emb = self.emb_layer(enc_x)
    enc_hidden, state = self.encoder(enc_emb, state)
    # 对编码器的隐藏状态 enc_hidden
    # 应用一个多层感知机（MLP），然后进行平均池化操作
    state = self.mlp(enc_hidden).mean(1).unsqueeze(0)
    return enc_hidden, state
```

```python
def decode(self, dec_y, enc_hidden, state, is_train = True):
    dec_embs = self.emb_layer(dec_y)  # 进行词嵌入
    trg_len = dec_y.shape[1]          # 获取解码器输入序列的长度
    outputs = []
    state = state.transpose(1, 0)

    for t in range(trg_len):
        scores = torch.bmm(state, enc_hidden.transpose(2, 1))
        alpha = self.softmax(scores)  # (batch_size, 1, seq_len)
        cont_vec = torch.bmm(alpha, enc_hidden).squeeze(1)
        input_vec = torch.cat([cont_vec, dec_embs[:, t, :]], 1).unsqueeze(1)
        sent_hidden, state = self.decoder(input_vec, state.transpose(0, 1))
        state = state.transpose(0, 1)
        # 输出预测字符
        pred = self.linear(sent_hidden)
        outputs += [pred]

    sent_outputs = torch.cat(outputs, dim = 1)
    return sent_outputs, state
```

# Lab9_attention mechanism 失分点

## （1）注意力机制计算错误

```
def decode(self, dec_y, enc_hidden, state):
    dec_embs = self.emb_layer(dec_y)

    batch_size, dec_seq_len, _ = dec_embs.size()
    attention_scores = torch.bmm(dec_embs, enc_hidden.permute(0, 2, 1))
    attention_weights = self.softmax(attention_scores)
    context_vector = torch.bmm(attention_weights, enc_hidden)
    concat_vector = torch.cat((dec_embs, context_vector), dim=2)
    dec_outputs, state = self.decoder(concat_vector, state)
    sent_outputs = self.mlp(dec_outputs)

    return sent_outputs, state
```

# Lab9_attention mechanism 失分点

## （2）注意力权重计算错误

```python
def decode(self, dec_y, enc_hidden, state):
    dec_embs = self.emb_layer(dec_y)

    batch_size = enc_hidden.shape[0]
    seq_len = enc_hidden.shape[1]

    attention = self.mlp(enc_hidden)
    attention_output = self.softmax(attention)

    rnn_input = torch.cat((dec_embs, attention_output), dim = 2)
    dec_output,state= self.decoder(rnn_input)
    sent_outputs = self.linear(dec_output)

    return sent_outputs, state

```

# Lab9_attention mechanism 失分点

## （3）没有计算注意力

```python
def decode(self, dec_y, enc_hidden, state):
    dec_embs = self.emb_layer(dec_y)
    repeated_enc_hidden = enc_hidden[:, -1].unsqueeze(1).repeat(1, dec_embs.size(1), 1)
    dec_input = torch.cat((dec_embs, repeated_enc_hidden), dim=2)
    dec_output, state = self.decoder(dec_input, state)
    output_flat = dec_output.reshape(-1, dec_output.size(2))
    output_flat = self.mlp(output_flat)
    output = self.linear(output_flat)
    output = output.view(dec_output.size(0), dec_output.size(1), -1)
    return output, state
```

# 深度学习
# Lab11-Retrieval task

兰韵诗

**这是期末大作业，请在6月14号结束之前提交！**

# Document Retrieval

- 在现代信息时代，人们面临着海量的信息，如何高效地从中检索到与自己需求最相关的信息成为了一个重要挑战。
- 为了提高信息检索系统的能力，研究人员和工程师们不断尝试各种方法，其中包括改进检索算法、优化搜索引擎等。其中，通过高效的检索方法来召回与用户提出的问题或需求最相关的文档是一个重要的研究方向。

# Document Retrieval

- 在给定问题的情况下，通过高效的检索方法，召回与提问最相关的3个文档，提高检索能力。
- 评估指标：Recall（召回率）是指检索系统能够检索到相关文档的能力，而MRR（Mean Reciprocal Rank，平均倒数排名）则是一种衡量检索系统排名效果的指标，它考虑了检索结果中相关文档的排名情况。

- 输入： document.json
- 输出： result.json

2

# Dataset

| 数据集 | 介绍 | 条数 |
| --- | --- | --- |
| document.json | 文档 数据集 | 26599条 |
| query_trainset.json | 问题 训练集 | 2044条 |
| query_testset.json | 问题 测试集 | 512条 |

ECNU

# 数据集介绍——document.json

```
[
    {
        "title_input_list" : xxx,          # 标题 经过分词后得到的 token IDs
        "title_embedding":xxx,             # 标题 embedding向量，大小为(N,)，N为embedding的维度。
        "fact_input_list":xxx,             # 文档 经过分词后得到的 token IDs
        "facts_embedding":xxx,             # 文档 embedding向量，大小为(N,)，N为embedding的维度
        "source":xxx,                      # 数据来源
        "category":xxx,                    # 数据类型
        "published_at":xxx                 # 发布时间
    },
    .....
]
```

# 数据集介绍——query_trainset.json

```
[
  {
    "query_input_list": xxx,          # 问题 经过分词后得到的 token IDs
    "query_embedding":xxx,            # 问题 embedding向量，大小为(N,)，N为embedding的维度。
    "evidence_list": [
      {
        "fact_input_list":xxx,        # 文档 经过分词后得到的 token IDs
        "fact_embedding":xxx          # 文档 embedding向量
      },
      {
        "fact_input_list":xxx,        # 文档 经过分词后得到的 token IDs
        "fact_embedding":xxx          # 文档 embedding向量
      },
      ....
    ]
  },
  ....
]
```

# 数据集介绍——query_testset.json

```
[
    {
        "query_input_list":xxx,          # 问题 经过分词后得到的 token IDs
        "query_embedding":xxx,           # 问题 embedding向量，大小为(N,)，N为embedding的维度
        "evidence_list": []              # 空列表
    },
    .....
]
```

# 返回结果——result.json：

```
[
    {
        "query_input_list" : xxx,              # 问题经过分词后得到的 token IDs
        "evidence_list": [
            {  "fact_input_list" : xxx  },     # 召回的第一个文档经过分词后得到的 token IDs
            {  "fact_input_list" : xxx  },     # 召回的第二个文档经过分词后得到的 token IDs
            {  "fact_input_list" : xxx  }      # 召回的第三个文档经过分词后得到的 token IDs
        ]
     },
    ....
]
```

# 3 Baseline Model

ECNU

# Cosine_similarity

1. 读取问题和文档 embeddings向量

2. 对每个问题进行检索：
   - 遍历每个问题，计算其与文档 embeddings 的相似度，并获取前 3 个最相关的文档索引。然后构建结果字典，包括 问题输入列表 和 相关证据列表（包括问题最相关的文档信息），并将结果添加到结果列表中。

3. 将结果写入 JSON 文件：
   - 将结果列表写入到结果文件 output/result.json 中。

# 4 Evaluation

ECNU

# 1. 评分指标

（1）Recall@k

$$recall = \frac{\text{算法结果中相关的}item\text{数量}}{\text{所有相关的}item\text{数量}}$$

表示检索结果中正确答案在前k个返回结果中的比例。

（2）MRR@k

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

表示平均倒数排名

- 对于每个查询，找到正确答案在返回结果中的排名（如果正确答案存在）。
- 如果正确答案在第m个位置，则MRR@k等于1/m。
- 对所有查询的MRR@k进行平均，即可得到MRR@k。

# 2. 评分指标

```python
def calculate_recall_at_k(result, gold, k):
    result = read_json(result)
    gold = read_json(gold)
    total_queries = len(result)
    recall = 0
    for r, g in zip(result, gold):
        correct_queries = 0
        # 去除开始，结尾字符
        relevant_facts = [e['fact_input_list'][1:-1] for e in g['evidence_list']]
        top_k_facts = [e['fact_input_list'][1:-1] for e in r['evidence_list'][:k]]
        # 转换成字符串
        relevant_facts = [' '.join(map(str, sublist)) for sublist in relevant_facts]
        top_k_facts = [' '.join(map(str, sublist)) for sublist in top_k_facts]

        if relevant_facts:  # 检查relevant_facts是否为空
            for answer in relevant_facts:
                if any(answer in fact for fact in top_k_facts):
                    correct_queries += 1
            recall += correct_queries / len(relevant_facts)  # 使用relevant_facts的长度来归一化

    recall_at_k = recall / total_queries
    return recall_at_k
```

```python
def calculate_mrr_at_k(result, gold, k):
    result = read_json(result)
    gold = read_json(gold)
    total_queries = len(result)
    mrr = 0
    mrr_at_k = 0

    for r, g in zip(result, gold):
        reciprocal_ranks = []
        relevant_facts = [e['fact_input_list'][1:-1] for e in g['evidence_list']]
        top_k_facts = [e['fact_input_list'][1:-1] for e in r['evidence_list'][:k]]
        relevant_facts = [' '.join(map(str, sublist)) for sublist in relevant_facts]
        top_k_facts = [' '.join(map(str, sublist)) for sublist in top_k_facts]

        if relevant_facts:  # 检查relevant_facts是否为空
            for answer in relevant_facts:
                for i, fact in enumerate(top_k_facts):
                    if answer in fact:
                        reciprocal_ranks.append(1 / (i + 1))  # 计算倒数排名并添加到列表中
                        break  # 找到第一个匹配的句子后停止搜索
            mrr += sum(reciprocal_ranks)/len(relevant_facts)
    mrr_at_k = mrr / total_queries
    return mrr_at_k
```

- 标准答案里面的fact_input_list只有一句话，只要你检索到的文档里面包含这句话即可。
- evidence_list可能包含0（query无法检索到相关的文档），2，3，4个
- 榜单的分值会作为我们主要打分依据，但我们还会考虑代码的**规范程度**、**方案创新程度**、**代码工作量**对分数(总分100)进行-10～+10分的调整。

# 3. baseline结果

Phase 1

Phase description
Homework phase1

**Max submissions per day:** 100

**Max submissions total:** 100

Download CSV | Download all submissions on leaderboard

Results

| # | User | Entries | Time of the Submission | Recall_at_3 ▲ | MRR_at_3 ▲ |
|---|---|---|---|---|---|
| 1 | 51265903028汪小曼 | 1 | 2024/05/08 15:17:45 | 0.2121 (1) | 0.1584 (1) |

# 5 Improvement

ECNU

# 可以考虑的改进

- 检索后rerank
- 检索算法
- .....

不限于上述，只是提供参考，大家可以自行选择高效的方法，提高检索的结果

# 注意事项

- 最多召回 三个 最相似的文档
- 只能使用我们分好的document.json
- 可随意选择模型进行建模、可随意选择python依存包进行建模
- 可选择适合自己的平台跑实验，最后在天梯提交预测数据和代码
- 提交的代码一定要是完整的、可复现的
- 不可以利用额外的数据集，不可以手动标注测试集