语音信息处理实验报告

课程名称：语音信息处理　　实验日期：2023/10/19

班级：人工智能 3 班　　姓名：实验 2-3 缺失　　学号：各位保重

实验一 GMM 的实现

一、实践要求

通过补全 GMM 中算法的核心代码，进行英语数字语音识别模型的训练及测试，从而熟悉并掌握 GMM 算法的原理。

二、实践内容

1. 补全计算对数似然函数代码。
2. 补全 EM 算法代码。
3. 理解并掌握 GMM 算法的原理。

三、实践结果与分析

1. 补全对数似然函数代码

```
def log_likelihood(self, feats):
    N = len(feats)
    # E step
    gamma = np.zeros((N, self.K))
    self.Estep(gamma, feats)
    llh = np.mean(np.log10(np.sum(gamma, axis=1)))
    return llh
```

```
def Estep(self, gamma:np.ndarray, feats: np.ndarray):
    """
    :param feats: N x D
    :param gamma: N x K
    pi: K
    mu: K x D
    sigma: K x D x D
    """
    N, K = gamma.shape
    for n in range(N):
        for k in range(K):
            gamma[n, k] = self.pi[k] * self.gaussian(feats[n], self.mu[k], self.sigma[k])
```

2. 补全 EM 算法代码

```
def Mstep(self, gamma: np.ndarray, feats: np.ndarray):
    K, N = gamma.shape
    Nsum = np.sum(gamma, axis=1)
    for i in range(K):
        mu = np.dot(gamma[i, :], feats) / Nsum[i]
        sigma = np.zeros((self.D,self.D))
        for j in range(N):
            sigma += gamma[i,j] * np.outer(feats[j,:] - mu, feats[j,:] - mu)
        sigma = sigma / Nsum[i]

        self.mu[i] = mu # update the normal with new parameters
        self.sigma[i] = sigma
        self.pi[i] = Nsum[i] / np.sum(Nsum) # normalize the new priors


def EM(self, feats: np.ndarray):
    """
    :param feats: N x D
    :return:
    """
    N = len(feats)
    # E step
    gamma = np.zeros((N, self.K))
    self.Estep(gamma, feats)
    gamma = gamma.T / np.sum(gamma, axis=1)
    # M step
    # 用最大似然的方法求出模型参数
    self.Mstep(gamma, feats)
    return self.log_likelihood(feats)
```

3. 理解并掌握 GMM 算法的原理

写点笔记在这里就行了，图片太大我没法上传

实验二 HMM 的实现

一、实践要求

通过补全 HMM 的前向算法、后向算法、维特比解码算法的核心代码，完成观察序列的路径的解码，从而理解并掌握 HMM 算法的原理。

二、实践内容

1. 补全前向算法代码。

2. 补全后向算法代码。

3. 补全维特比解码算法代码。

4. 理解并掌握 HMM 算法的原理。

三、实践结果与分析

1. 补全前向传播代码

```
def forward(self, O):
    r"""
    get P(O|M) by forward algorithm
        \alpha_{t+1}(j) =b_{j}(o_{t+1})\sum_{i=1}^M(\alpha_{t}(i)a_{ij})
        At+1(j) = ∑i{At(i) x aij} x bj(ot+1)
    :param O:
    :return:
    """
    #  definition forward probability matrix
    self.o_len = len(O) # 观测序列长度
    self.forward_prob_matrix = np.zeros((self.o_len, self.s_set_num)) # O x P


    # t=0
    # 初始化:令 α0(i)=πi×bi(o1),其中 πi 是初态概率。
    self.forward_prob_matrix[0,:] = self.pi * self.B[:, O[0]]
    # T=1:T, compute forward probability matrix
    # 递推:对 t=1 至 T-1 重复计算 αt+1(j)。
    """
        TODO
    """
    for j in range(1, self.o_len):
        for i in range(self.s_set_num):
            #前一时刻所有状态的概率乘以转移概率得到 i 状态概率
            #i 状态的概率*i 状态到 j 观测的概率
            temp = 0
            for k in range(self.s_set_num):
                temp = temp + self.forward_prob_matrix[j-1, k] * self.A[k, i]
            self.forward_prob_matrix[j, i] = temp * self.B[i, O[j]]
    # get the last time total forward probability
    # 终止:总概率 P(O|λ)=∑jαT(j)。
    forward_prob = np.sum(self.forward_prob_matrix[self.o_len-1, :])
    return forward_prob
```

2. 补全后向传播代码

```
def backward(self, O):
    r"""
```

```
        get P(O|M) by backward algorithm
            \beta_{t}(i) = \sum_{j=1}^M(\beta_{t+1}(j)a_{ij}b_{j}(o_{t+1}))
            βt(i) = Σj{βt+1(j) x aji x bj(ot+1)}
        :param O:
        :return:
        """
        # definition backward probability matrix
        self.o_len = len(O)
        self.backward_prob_matrix = np.zeros((self.o_len, self.s_set_num))
        # t = T-1
        # 初始化:令 βT(i)=1,即最后一个时间节点的后向概率为 1。
        self.backward_prob_matrix[self.o_len - 1, :] = 1
        # t=T-2:-1, compute backward probability matrix
        # 递推:对 t=T-1 至 0 重复计算 βt(i)。
        """
            TODO
        """
        for j in range(self.o_len - 2, -1, -1):
            for i in range(self.s_set_num):
                for k in range(self.s_set_num):
                    self.backward_prob_matrix[j, i] += self.A[i, k] * self.B[k, O[j+1]] *
self.backward_prob_matrix[j+1, k]
        # get the first time total backward probability
        # 终止:得到初态 πi 的后向概率 η = ∑i{πi*bi(o1)*β1(i)}
        backward_prob = np.sum(self.pi*self.B[:,O[0]]*self.backward_prob_matrix[0])
        return backward_prob
```

3. 补全维特比解码算法代码

```
    def decoding(self,O):
        """ Viterbi Decoding
        get argmax P(Q|O,M)
            \delta_{t+1}(j) = max\limits_{1<=i<=M}(\delta_{t}(i)a_{ij}))b_{j}(o_{t+1}
        :return:
            best_path:
            best_prob:
        """

        #  definition the best probability matrix and last node maxtrix
        o_len = len(O)
        best_prob_matrix = np.zeros((o_len,self.s_set_num))
        last_node_matrix = np.zeros((o_len,self.s_set_num),dtype=int)
        # t=0
        best_prob_matrix[0,:] = self.pi * self.B[:,O[0]]
        last_node_matrix[0,:] = -1
        # T=1:T, compute best probability matrix and last node matrix
```

```
        for t in range(1, o_len):
            for j in range(self.s_set_num):
                max_prob = -float('inf')
                max_i = -1
                for i in range(self.s_set_num):
                    prob = best_prob_matrix[t-1, i] * self.A[i, j] * self.B[j, O[t]]
                    if prob > max_prob:
                        max_prob = prob
                        max_i = i
                best_prob_matrix[t, j] = max_prob
                last_node_matrix[t, j] = max_i
        # t=T-1, get best_prob and the last node
        last_node =  np.argmax(best_prob_matrix[o_len-1,:])
        best_prob = best_prob_matrix[o_len - 1,last_node]
        best_path = [last_node]
        # t=T-1:0,backtrack the path
        for t in range(o_len-1,0,-1):
            last_node = last_node_matrix[t, last_node]
            best_path.append(last_node)
        # reverse to the noraml path
        best_path = best_path[::-1]
        return best_path, best_prob
```

4. 理解并掌握 HMM 算法的原理

写点笔记在这里就行了，图片太大我没法上传

实验三 基于 Wenet 的连续语音识别系统实现（2 学时）

一、实践要求

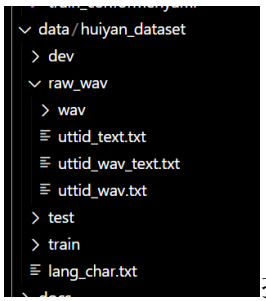
通过对开源语音识别工具包 Wenet 的学习，掌握完整的端到端语音识别流程，通过主动调整模型训练参数并观察最终的模型性能表现，与其他学生的结果进行对比，分析影响模型训练的重要影响因素，自行平衡模型性能与训练成本之间的关系。

二、实践内容

阅读实验手册，学习语音识别模型训练数据生成和语音识别模型的训练流程，并按照学习手册中的提示，逐步执行 run. sh 脚本进行相关实验，观察每个步骤生成的文件以及对应的结果，掌握每个步骤对应的功能，主动调整模型训练参数并观察最终的模型性能表现。

三、实践结果与分析

按照 0-5 六个 stage 的顺序进行分析：



Stage = 0 时，拆分数据集为 dev，train，test，wav 放着源文件，uttid_text 是 id-label 映射表，uttid_wav 是 id-wav 是 id-path 映射表， 还有一个三者合并的集合，这就是第 0 步。

Stage = 1 时，对 label 进行处理，去除空格，提取 cmvn 声学特征。

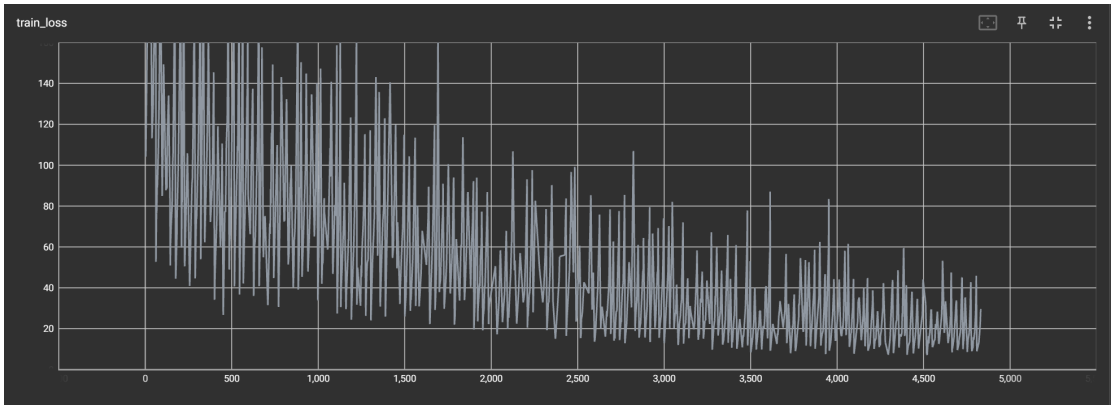
Stage = 2 时，生成词表字典，把单词映射成数字，text to token。

Stage = 3 时，把数据索引表整理成 json 格式

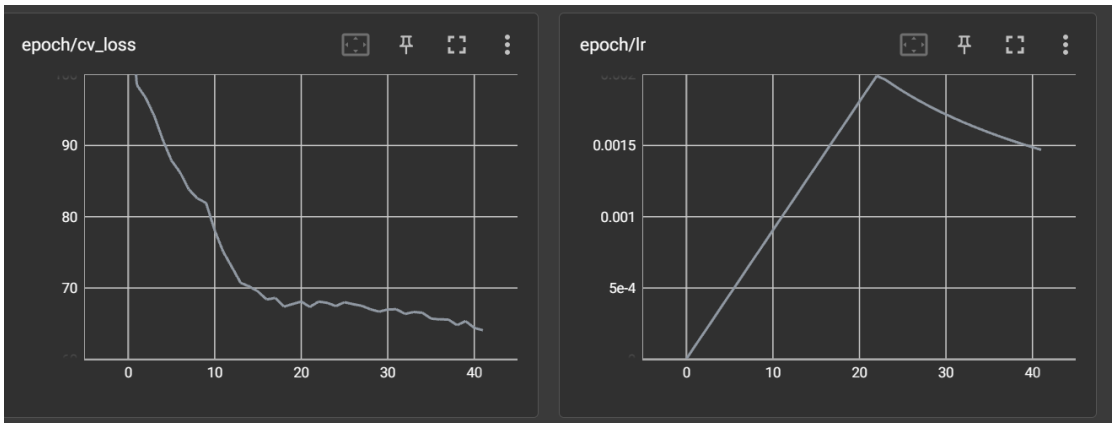
Stage = 4 时进入训练步骤，run.sh 中配置模型的 yaml 文件位于 conf/train_conformer.yaml

使用默认配置跑一遍，于 epoch42 时 earlystop，查看 tensorboard

the number of model params: 3560204



可见训练 loss 已经降到 40%及以下，趋于平稳，但还有下降空间



再来看 epoch 相关信息，可见 loss 已经趋于平稳，学习率缓慢下降，学习率的初始值是 2e-3，可见这是从 0 开始生长的 lr，到达峰值后缓慢下降，这是 warmup 策略导致的

```
optim: adam
optim_conf:
    lr: 0.002
scheduler: warmuplr
scheduler_conf:
    warmup_steps: 2500
```

Warmup 策略的好处是，可以平滑地开始训练，模型不会过早的吸收数据的特征，防止过拟合。

回过头来看其他配置

```
encoder: conformer
encoder_conf:
    output_size: 128
    attention_heads: 4
    linear_units: 256
    num_blocks: 4
```

其中 output_size 是注意力层的输出大小，linear_units 则是前向传播网络的神经元数量，此处使用了 4 块 encoder。输出结果如下：

（源文档的表格感觉像 markdown，所以我就用了 markdown）

解码方式	解码结果CER
attention	Overall -> 88.79 % N=1642 C=235 S=688 D=719 I=51
	Mandarin -> 88.74 % N=1634 C=235 S=683 D=716 I=51
	Other -> 100.00 % N=1 C=0 S=1 D=0 I=0
	English -> 100.00 % N=7 C=0 S=4 D=3 I=0
ctc_greedy_search	Overall -> 43.79 % N=1642 C=936 S=648 D=58 I=13
	Mandarin -> 43.70 % N=1634 C=933 S=644 D=57 I=13
	Other -> 100.00 % N=1 C=0 S=1 D=0 I=0
	English -> 57.14 % N=7 C=3 S=3 D=1 I=0
ctc_prefix_beam_search	Overall -> 43.54 % N=1642 C=938 S=648 D=56 I=11
	Mandarin -> 43.45 % N=1634 C=935 S=644 D=55 I=11
	Other -> 100.00 % N=1 C=0 S=1 D=0 I=0
	English -> 57.14 % N=7 C=3 S=3 D=1 I=0
attention_rescoring	Overall -> 42.39 % N=1642 C=953 S=629 D=60 I=7
	Mandarin -> 42.29 % N=1634 C=950 S=625 D=59 I=7
	Other -> 100.00 % N=1 C=0 S=1 D=0 I=0
	English -> 57.14 % N=7 C=3 S=3 D=1 I=0

可以看到 attention 解码不如后三者，但加入 rescoring 重新判分机制后能力暴增，使用额外的自注意力和上下文注意力模块，对第一个序列中的每个 token 都计算一个新的注意力分数，新的注意力分数考虑了该 token 与其他 token 以及上下文语句的相关性，根据新的注意力分数对第一个生成序列进行“重分数”，优先考虑新的分数高的部分保留，分数低的部分可能替换，这样可以修正第一个生成序列中的语法或逻辑错误，同时保留前面正确部分不变，提升文本生成的质量与性能。

CTC(连接性时序分类)解码器是一种通过计算可能性概率来解码的方案，不同于 attention 注意力模型，此外 ctc_prefix_beam_search 和 ctc_greedy_search 的不同点是，前者使用了前束搜索模式，它舍弃每一步可能路径中权重较低的前

缀,只保留权重前 K 大的前缀,是一种求 K 大的方法,后者则使用了贪心法求解。但二者的分数是没有大的差距的

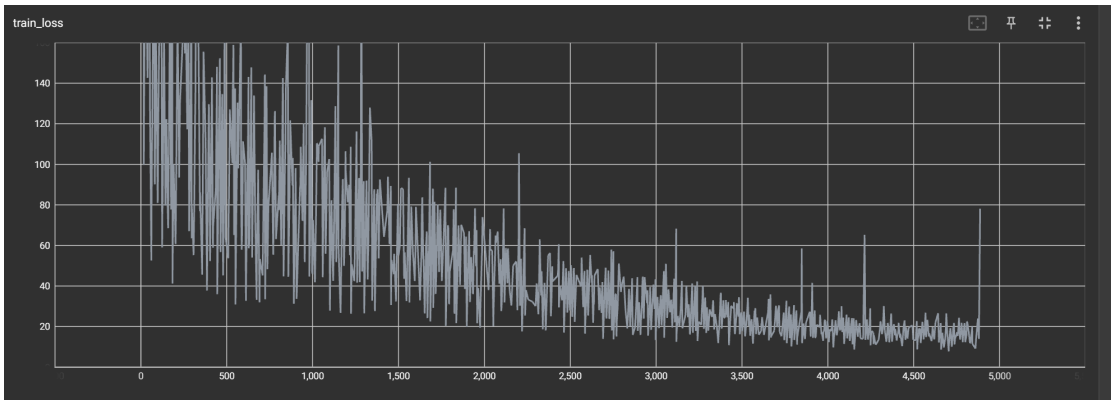
之前有一篇论文《On the Expressive Power of Deep Neural Networks》讲述了,宽而浅的神经网络,精确性要劣于窄而深的神经网络,这里我稍微尝试一下,使用 8 块 conformer，64units，重新进行实验，参数量维持在 3400000 上下，没有发生太大变化。

the number of model params: 3425054

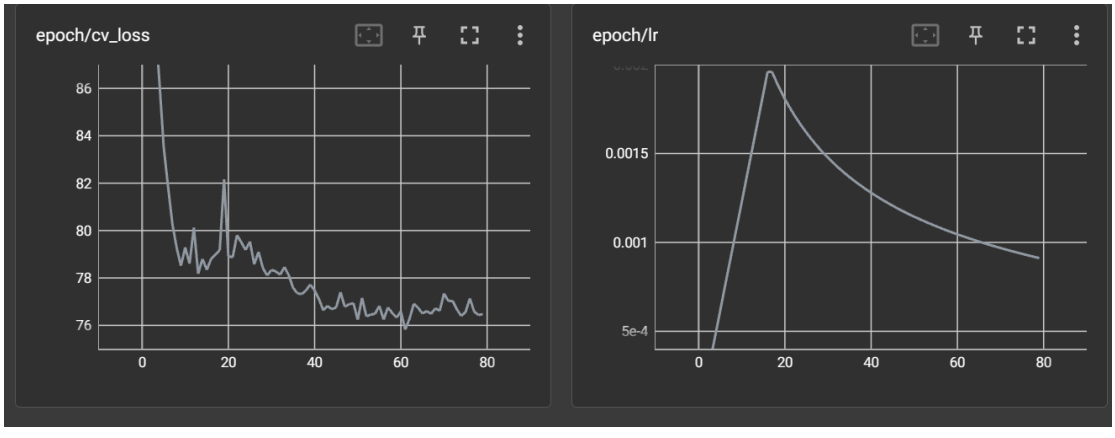
并且使 warmup 速度从 2500 步加快到 1000，主要是懒得等。

scheduler_conf:

warmup_steps: 1000



可见 loss 的下降速度要快于上次



中间经历了一次 loss 回升，应该是学习率导致的

解码方式	解码结果CER
attention	Overall -> 94.43 % N=1722 C=106 S=813 D=803 I=10
	Mandarin -> 94.41 % N=1718 C=106 S=811 D=801 I=10
	English -> 100.00 % N=4 C=0 S=2 D=2 I=0
ctc_greedy_search	Overall -> 70.33 % N=1722 C=535 S=974 D=213 I=24
	Mandarin -> 70.26 % N=1718 C=535 S=971 D=212 I=24
	English -> 100.00 % N=4 C=0 S=3 D=1 I=0
ctc_prefix_beam_search	Overall -> 70.15 % N=1722 C=534 S=970 D=218 I=20
	Mandarin -> 70.08 % N=1718 C=534 S=967 D=217 I=20
	English -> 100.00 % N=4 C=0 S=3 D=1 I=0
attention_rescoring	Overall -> 69.69 % N=1722 C=536 S=937 D=249 I=14
	Mandarin -> 69.62 % N=1718 C=536 S=934 D=248 I=14
	English -> 100.00 % N=4 C=0 S=3 D=1 I=0

很明显在新的训练中出现了严重的过拟合现象，得分甚至低于上一次的分数，神经网络经常出现此类情况，推测为是 warmup 过快和 conformer 深度过大导致。

实践心得

实操了一次语音识别，之前下载过 RVC，sovits 开源程序，自己制作了一些 ai 换声的音频，但一直没有深入到底层，这次算是一次先导，对 transformer 模块和注意力机制有了更深的了解，并且了解到了 conformer，warmup 策略等新概念，感觉非常不错。

建议：原项目 wenet 是纯 py 的，此处加入 pl 和 sh 两种新语言，虽然造成的困扰不多，但对实验环境的要求变得更加苛刻了，主要是没有对理解实验内容起到帮助，徒增烦恼，建议老师制作实验项目的时候也用纯 py，方便本地运行，而且咱学部的机器 cpu 只有四个核好可怜。