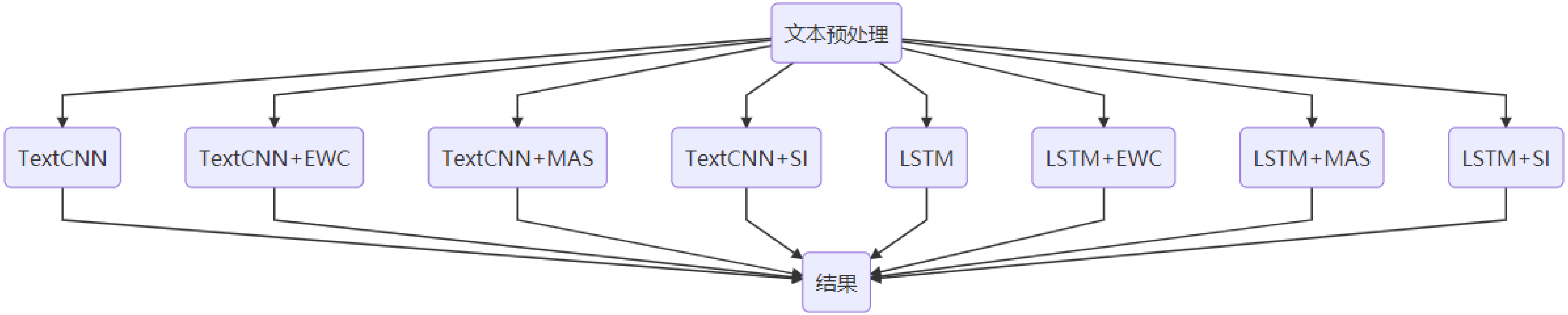
基于开放环境的文本分类任务

19335174 施天予
19335177 孙奥远

概述

- 开放环境学习范式主要有辅助数据、微调、知识蒸馏、参数重构、元学习等。本次项目我们小组使用参数重构的策略进行了基于开放环境的文本分类问题的求解。我们实现了TextCNN、LSTM两个深度学习模型,尝试了EWC、MAS、SI三种不同的参数重构方法。

流程图


文本预处理
TextCNN
TextCNN+EWC
TextCNN+MAS
TextCNN+SI
LSTM
LSTM+EWC
LSTM+MAS
LSTM+SI
结果

文本处理

- 本次项目的文本处理与期中项目大同小异。与之前相同的文本处理过程就不再详细介绍了，主要步骤如下：
- 脏数据清洗
- 分词
- 统一大小写
- 去除停用词

文本处理

- 斯坦福的glove.6B.300d.txt预训练词向量。
- 给训练集中的每个单词一个序号，构建一个word-id的词典。
- 将每个文档的单词转为序号。
- 统一每个文档的长度。通过对数据集的观察，选择80作为每个文档的长度进行padding。将过长的文档截断，过短的文档用0补齐。
- 使用词典和Glove预训练词向量构建一个权重矩阵，使每个id对应该单词的word vector。
- 最后，在CNN和RNN中使用nn.Embedding载入Glove预训练权重模型，就可以完成整个词嵌入的过程。

卷积神经网络

- 卷积神经网络用于文本分类的核心思想是抓取文本的局部特征，通过不同的卷积核尺寸来提取文本的N-gram信息，然后通过最大池化操作来突出各个卷积操作提取的最关键信息，拼接后通过全连接层对特征进行组合，最后通过交叉熵损失函数来训练模型。

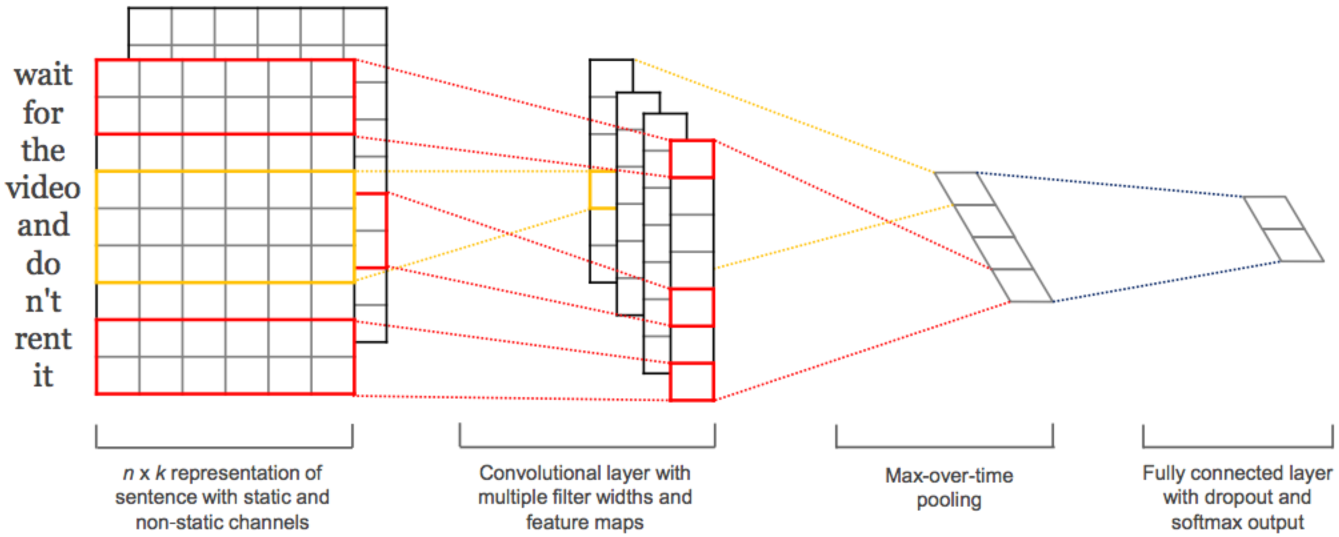


Figure 1: Model architecture with two channels for an example sentence.

卷积神经网络

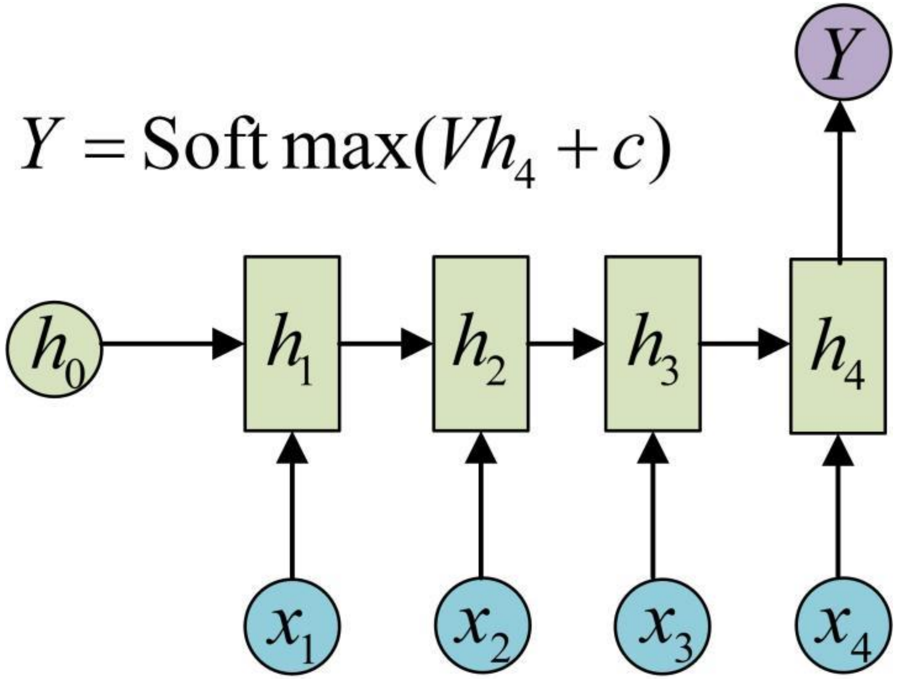
```python
class TextCNN(nn.Module):
    def __init__(self,
                 class_num=4,
                 kernel_num=128,
                 kernel_size_list=(3,4,5),#三个核
                 dropout=0.5,embed_dim=300):

        super(TextCNN, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(weight)
        self.embedding.weight.requires_grad = True

        self.conv1d_list = nn.ModuleList([
            nn.Conv1d(embed_dim, kernel_num, kernel_size)
                for kernel_size in kernel_size_list#卷积层
        ])
        self.linear = nn.Linear(kernel_num * len(kernel_size_list), class_num)#全连接层
        self.dropout = nn.Dropout(dropout)#防止过拟合
```

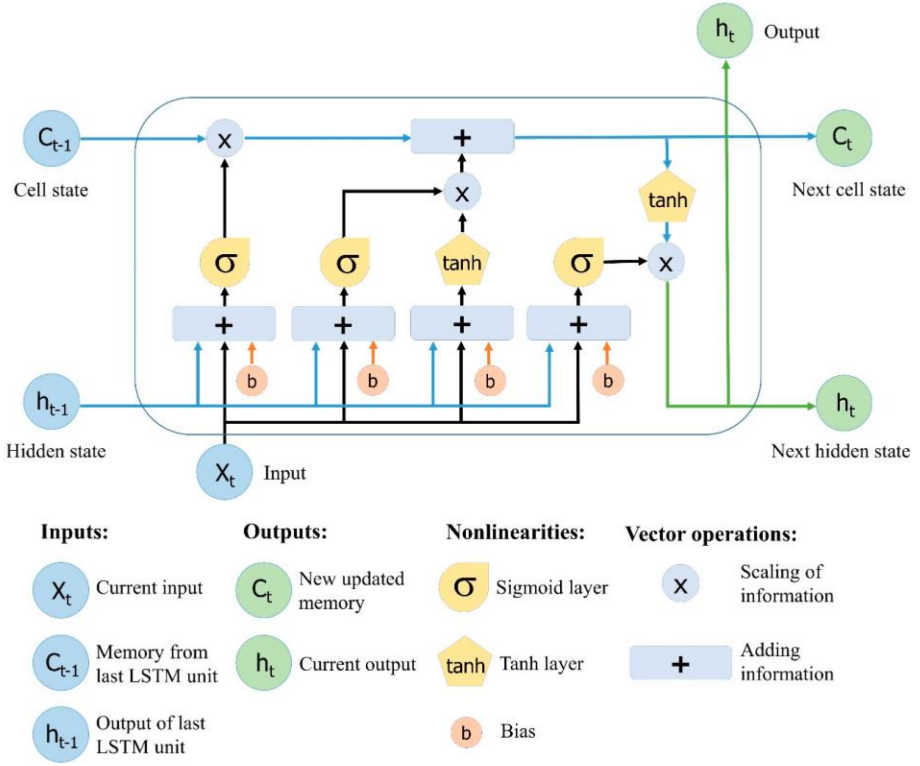
```
TextCNN(
  (embedding): Embedding(50107, 300)
  (conv1d_list): ModuleList(
    (0): Conv1d(300, 128, kernel_size=(3,), stride=(1,))
    (1): Conv1d(300, 128, kernel_size=(4,), stride=(1,))
    (2): Conv1d(300, 128, kernel_size=(5,), stride=(1,))
  )
  (linear): Linear(in_features=384, out_features=4, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

```python
    def forward(self, x):
        # x的形状为(batch, word_nums)
        # 经过嵌入层之后x的形状为(batch, word_nums, embed_dim)
        x = self.embedding(x)

        #因为conv1d的输入需要为: (batch, in_channels, in_length)
        # in_channels是embed_dim, in_length是word_nums
        # 需要将x转换为: (batch, embed_dim, word_nums)
        x = x.transpose(1, 2)

        # 经过conv1d之后, (batch, kernel_num, out_length)
        # out_length = word_nums - kernel_size + 1
        x = [F.relu(conv1d(x)) for conv1d in self.conv1d_list]

        # pooling作用在第3维, 窗口大小为第三维的大小
        # 在池化之后变为(batch, kernel_num, 1)
        # squeeze(2)删除第3维
        x = [F.max_pool1d(i, i.shape[2]).squeeze(2) for i in x]

        # shape: (batch, kernel_num * len(kernel_size_list))
        x = torch.cat(x, dim=1)
        x = self.dropout(x)
        # shape: (batch, class_num)
        x = self.linear(x)
        return x
```

循环神经网络

- 对于一个分类问题，我们使用的RNN循环神经网络是一种Sequence-to-Vector结构，即输入一个序列，输出一个单独的数据，通常在最后一个序列上进行输出变换。





循环神经网络

```python
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()
        self.word_embed = nn.Embedding.from_pretrained(weight)
        self.word_embed.weight.requires_grad = True
        self.rnn = nn.LSTM(
            input_size = EMBEDDING_DIM,
            hidden_size = HIDDEN_SIZE,
            num_layers = 1,
            batch_first = True,
        )
        self.dropout = nn.Dropout(0.5)
        self.out = nn.Linear(HIDDEN_SIZE, 4)

    def forward(self, x):
        r = self.word_embed(x.long())
        r, _ = self.rnn(r)
        r = self.dropout(r)
        out = self.out(r[:, -1, :])
        return out
```

```
RNN(
  (word_embed): Embedding(50107, 300)
  (rnn): LSTM(300, 64, batch_first=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (out): Linear(in_features=64, out_features=4, bias=True)
)
```

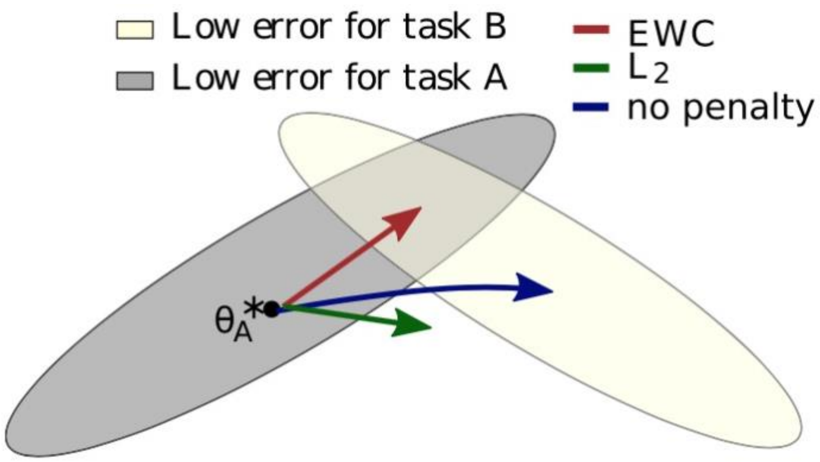
wordlist size	50107
embedding dim	300
hidden size	64
layer num	1
dropout	0.5

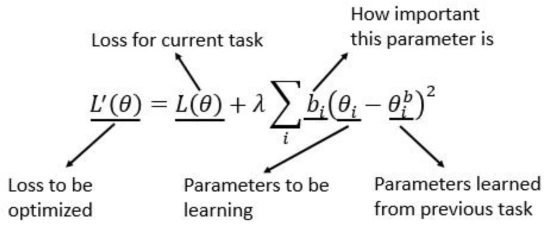
Baseline

- 设置 Baseline 的目的是为了给接下来的三种参数重构方法作为参考，观察我们的方法是否有效。Baseline 的方法也十分简单，只需将模型在一个训练集练完后立即就在下一个训练集上训练，不需要计算参数的重要程度，也不用改变 loss。这种方法得到的平均准确率应为最后模型的下限。

EWC

- EWC 利用 Fisher 矩阵计算模型参数的重要程度，当模型在一个训练集上训练完后，我们需要将模型在之前所有练过的数据集的验证集上进行测试，得到参数重要程度的 EWC 权重矩阵。
- 我们将 EWC 设置为一个 class，其共有两个重要的函数，calculate_importance 用于计算参数的重要程度，penalty 用于计算惩罚项。





EWC

- Fisher矩阵
- λ=100

```python
def _calculate_importance(self):
    precision_matrices = {}
    for n, p in self.params.items():
        precision_matrices[n] = p.clone().detach().fill_(0)
    self.model.train()
    for module in self.model.modules():
        if isinstance(module, nn.Dropout):
            module.training = False
    # self.model.eval()
    if self.dataloaders[0] is not None:
        number_data = sum([len(loader) for loader in self.dataloaders])
        for dataloader in self.dataloaders:
            for data in dataloader:
                self.model.zero_grad()
                input = data[0].to(self.device)
                output = self.model(input)
                label = data[1].to(self.device)
                loss = F.cross_entropy(output, label)
                loss.backward()
                for n, p in self.model.named_parameters():
                    precision_matrices[n].data += p.grad.data ** 2 / number_data
        precision_matrices = {n: p for n, p in precision_matrices.items()}
    return precision_matrices

def penalty(self, model: nn.Module):
    loss = 0
    for n, p in model.named_parameters():
        _loss = self._precision_matrices[n] * (p - self.p_old[n]) ** 2
        loss += _loss.sum()
    return loss
```

MAS

- Omega矩阵
- λ=0.1

```python
def calculate_importance(self):
    precision_matrices = {}
    for n, p in self.params.items():
        precision_matrices[n] = p.clone().detach().fill_(0)
    self.model.train()
    for module in self.model.modules():
        if isinstance(module, nn.Dropout):
            module.training = False
    # self.model.eval()
    if self.dataloaders[0] is not None:
        num_data = sum([len(loader) for loader in self.dataloaders])
        for dataloader in self.dataloaders:
            for data in dataloader:
                self.model.zero_grad()
                output = self.model(data[0].to(self.device))
                output.pow_(2)
                loss = torch.sum(output,dim=1)  # 不需要label!
                loss = loss.mean()
                loss.backward()
                for n, p in self.model.named_parameters():
                    precision_matrices[n].data += p.grad.abs() / num_data
    precision_matrices = {n: p for n, p in precision_matrices.items()}
    return precision_matrices

def penalty(self, model: nn.Module):
    loss = 0
    for n, p in model.named_parameters():
        _loss = self._precision_matrices[n] * (p - self.p_old[n]) ** 2
        loss += _loss.sum()
    return loss
```

SI(突触智能)

神经网络由若干神经元以及和神经元相关的参数组成。若神经网络的参数在时间t有一个微小的变化，损失函数值的变化能用其对参数的梯度表示

$$g = \frac{\partial L}{\partial \theta}$$

$$L(\theta(t) + \sigma(t)) - L(\theta(t)) \approx \sum_k g_k(t) * \sigma_k(t)$$

从t0到t1，损失函数值的变化:

$$\int_C g(\theta(t))d\theta = \int_{t_0}^{t_1} g(\theta(t)) * \theta'(t)dt = L(\theta(t_1)) - L(\theta(t_0))$$

SI(突触智能)

训练任务u时,单个权值的变化对损失函数的影响可以表示如下

$$\int_{t^{u-1}}^{t^u} g(\theta(t)) * \theta'(t)dt = \sum_k \int_{t^{u-1}}^{t^u} g_k(\theta(t))\theta'_k(t)dt = -\sum_k \omega_k^u = L(\theta(t^u)) - L(\theta(t^{u-1}))$$

$$\omega_k^u = -\int_{t^{u-1}}^{t^u} g_k(\theta(t))\theta'_k(t)dt$$

SI(突触智能)

在训练任务u时，第k个参数的路径积分为 ω_k^u

在训练过程中，ω_k^u 越大，说明第k个参数在任务u的训练过程中调整越大，因此说明第k个参数对任务u越重要。

训练任务u时,第k个参数对于以往任务的重要性度量表示如下:

$$\Omega_k^u = \sum_{v<u} \frac{\omega_k}{(\Delta_k^v)^2 + \xi}$$

$$\Delta_k^v = \theta_k(t^v) - \theta_k(t^{v-1})$$

SI(突触智能)

训练任务u时，使用的改进的损失函数为:

$$L'_u = L_u + c\sum_k \Omega_k^u(\hat{\theta}_k - \theta_k)^2$$

SI(突触智能)

Calculate_importance计算参数的重要性

```python
def calculate_importance(self):
    n_p_prev = {}
    n_omega = {}
    if self.task_id != 0:
        for n, p in self.model.named_parameters():
            n = n.replace('.', '__')   #buffer name不能包含'.'
            if p.requires_grad:
                p_prev = getattr(self.model, '{}_p_prev_task'.format(n))
                W = getattr(self.model, '{}_W'.format(n))
                omega = getattr(self.model, '{}_omega'.format(n))
                p_current = p.detach().clone()
                omega_new = omega + W/((p_current - p_prev)**2 + self.eta)
                n_omega[n] = omega_new
                n_p_prev[n] = p_current
                self.model.register_buffer('{}_p_prev_task'.format(n), p_current)
                self.model.register_buffer('{}_omega'.format(n), omega_new)
    else:
        for n, p in self.model.named_parameters():
            n = n.replace('.', '__')
            if p.requires_grad:
                n_p_prev[n] = p.detach().clone()
                n_omega[n] = p.detach().clone().zero_()
                self.model.register_buffer('{}_p_prev_task'.format(n), p.detach().clone())
                self.model.register_buffer('{}_omega'.format(n),p.detach().clone().zero_())
    return n_p_prev, n_omega
```

SI(突触智能)

Penalty计算损失函数中的正则化项

Update更新参数的路径积分

```
def penalty(self, model: nn.Module):
    Loss = 0.0
    for n, p in model.named_parameters():
        n = n.replace('.', '__')
        if p.requires_grad:
            prev_values = self._n_p_prev[n]
            omega = self._n_omega[n]
            loss = omega * (p - prev_values) ** 2
            Loss += loss.sum()
    return Loss

def update(self, model):
    for n, p in model.named_parameters():
        n = n.replace('.', '__')
        if p.requires_grad:
            if p.grad is not None:
                self.W[n].add_(-p.grad * (p.detach() - self.p_old[n]))
                self.model.register_buffer('{}_W'.format(n), self.W[n])
            self.p_old[n] = p.detach().clone()
    return
```

模型训练

- 损失函数交叉熵
- 优化函数Adam

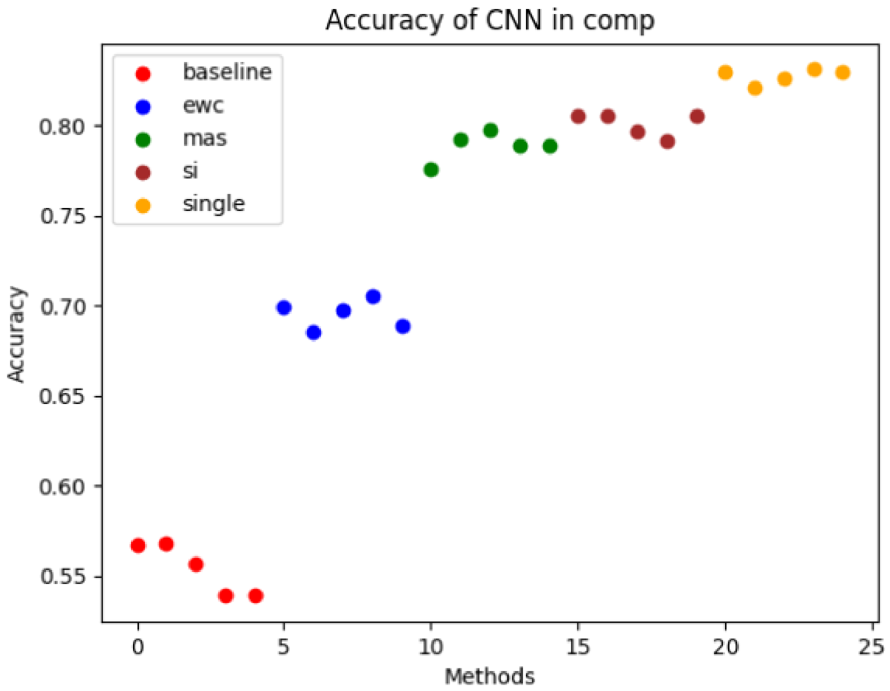
batch size	256
learning rate	0.001
epochs	35

```
for epoch in bar:
    # train
    model.train()
    for module in model.modules():
        if isinstance(module, nn.Dropout):
            module.training = True
    for x, y in tqdm.auto.tqdm(dataloader, leave=False):
        x, y = x.to(device), y.to(device)
        outputs = model(x)
        loss = objective(outputs, y)
        total_loss = loss
        lll_loss = lll_object.penalty(model)
        total_loss += lll_lambda * lll_loss
        lll_object.update(model)
        optimizer.zero_grad()
        total_loss.backward()
        optimizer.step()
```

模型评估

```python
def evaluate(model, test_dataloader, device):
    model.eval()
    correct_cnt = 0
    total = 0
    for data, labels in test_dataloader:
        data, labels = data.to(device), labels.to(device)
        outputs = model(data)
        _, pred_label = torch.max(outputs.data, 1)

        correct_cnt += (pred_label == labels.data).sum().item()
        total += torch.ones_like(labels.data).sum().item()
    return correct_cnt / total
```

实验结果分析



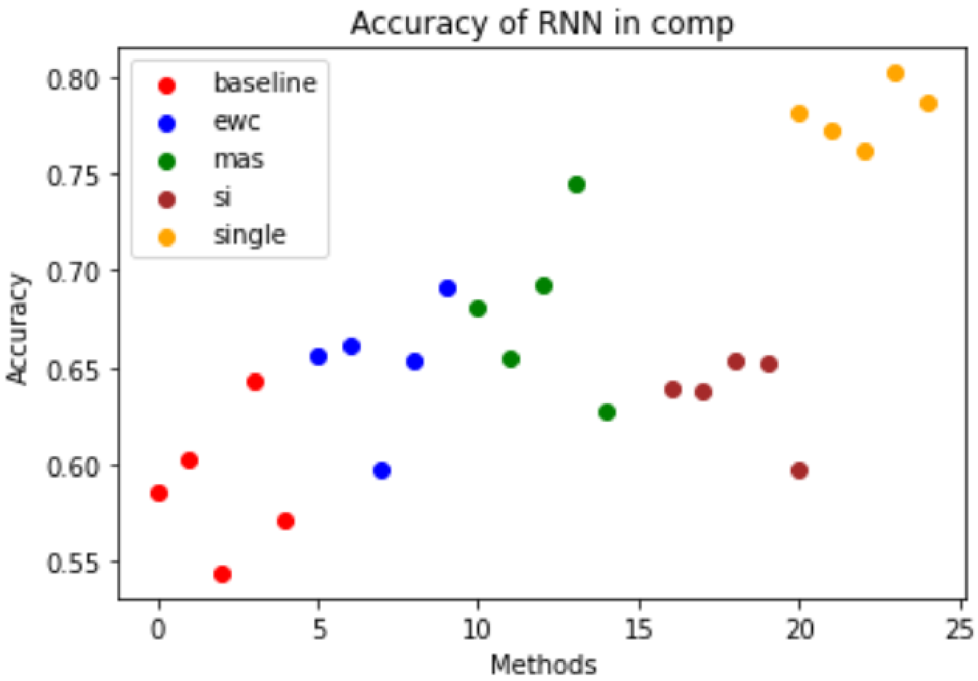
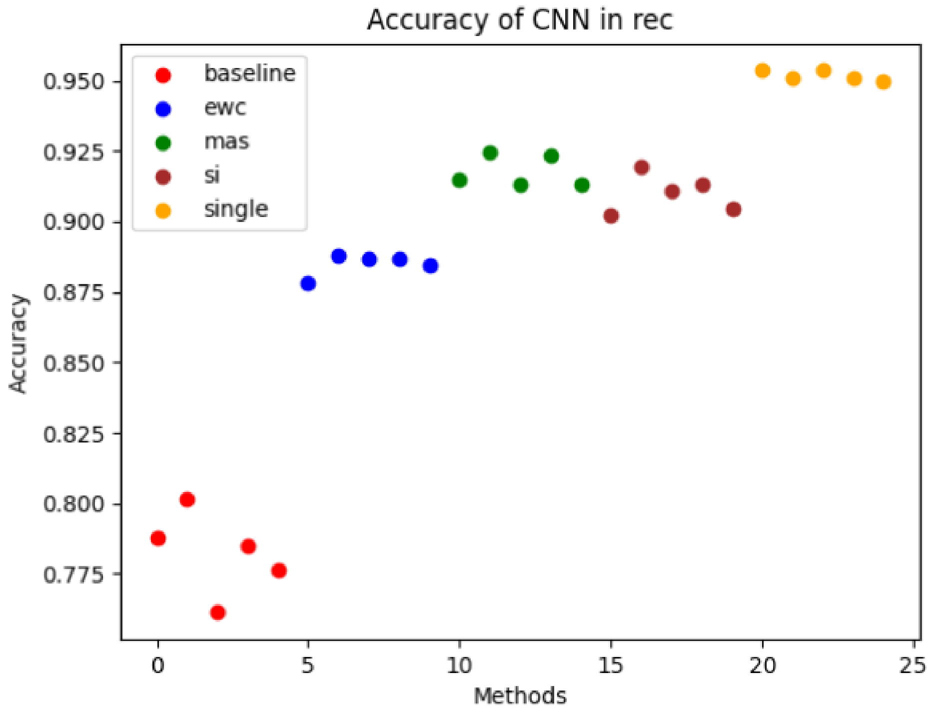


图 10: 所有范式在 comp 任务的结果

实验结果分析



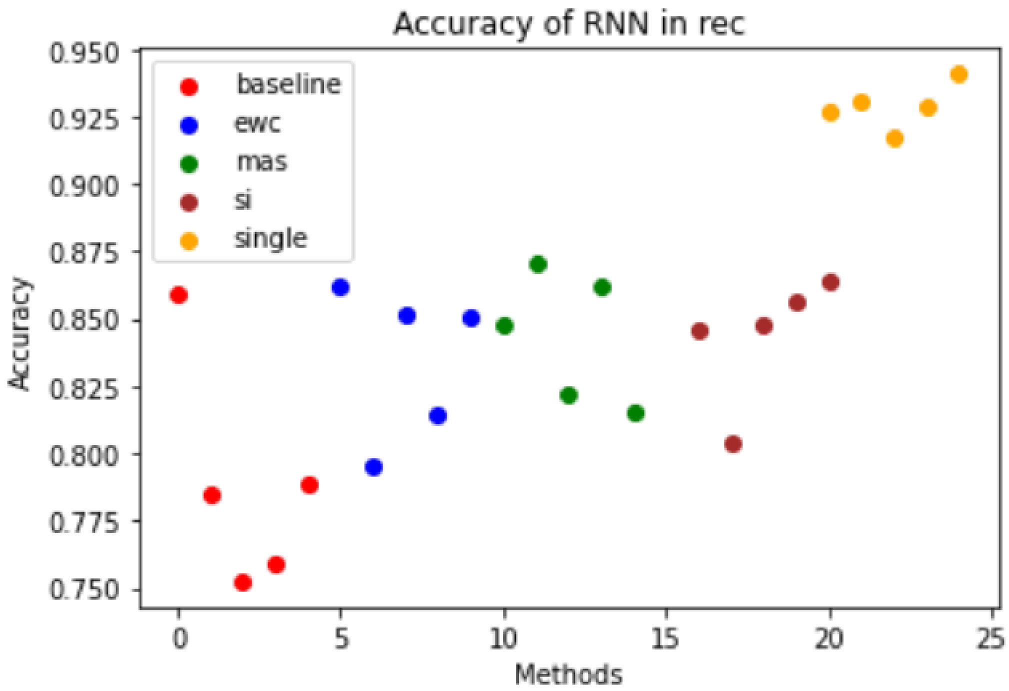


图 11: 所有范式在 rec 任务的结果

实验结果分析

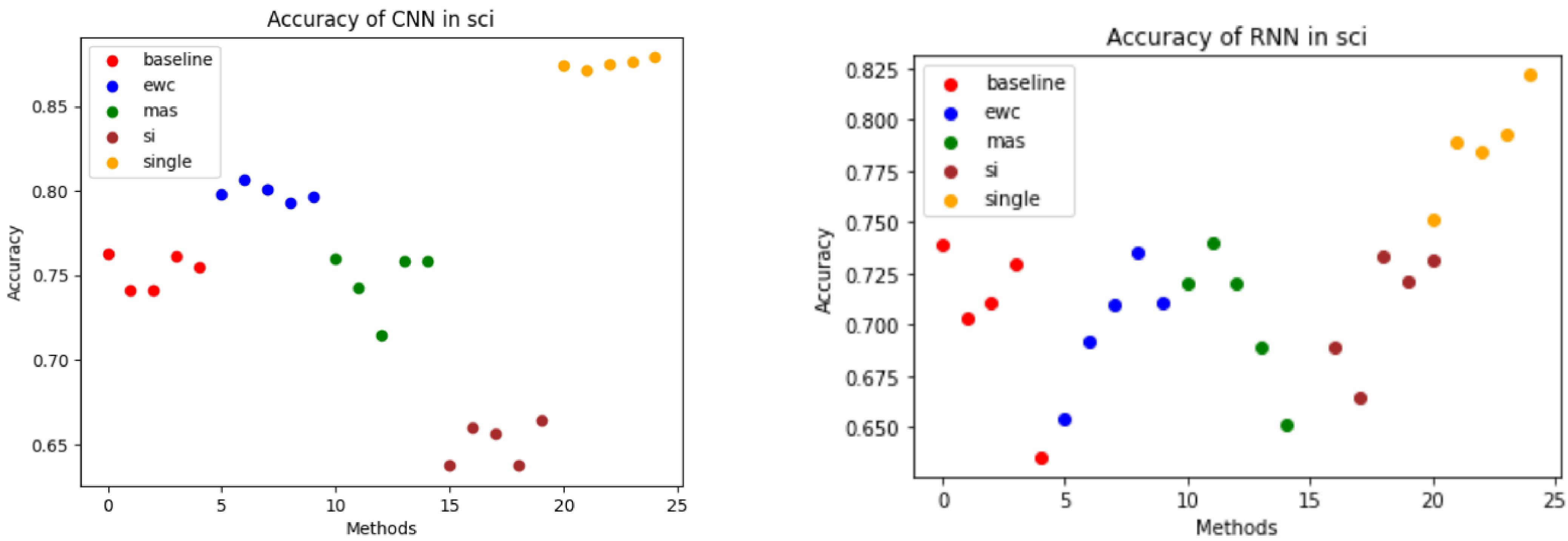
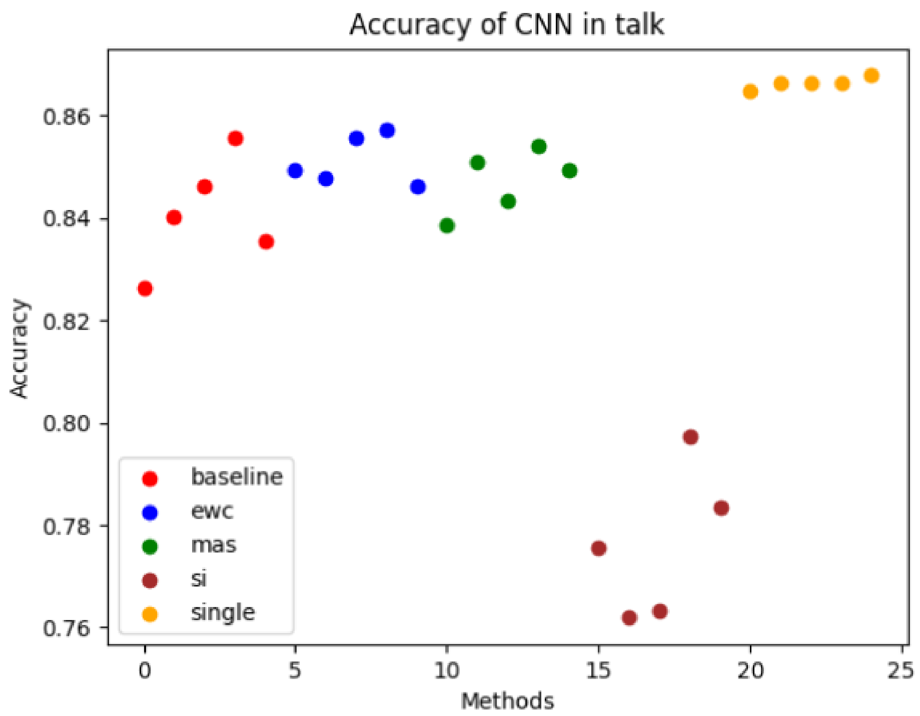


图 12: 所有范式在 sci 任务的结果

实验结果分析



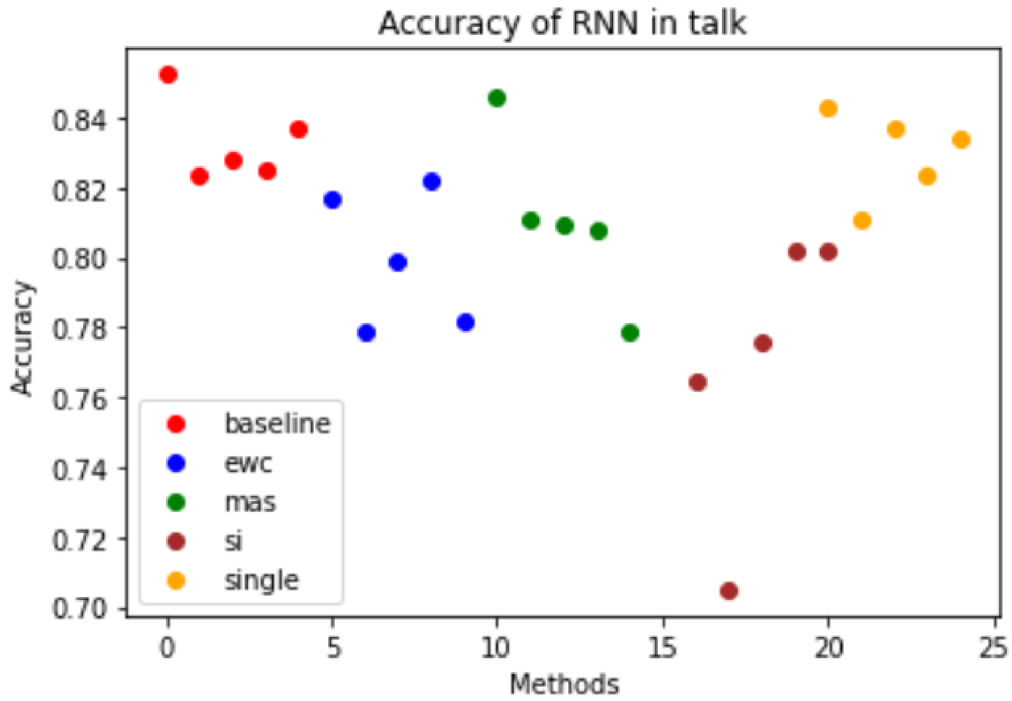
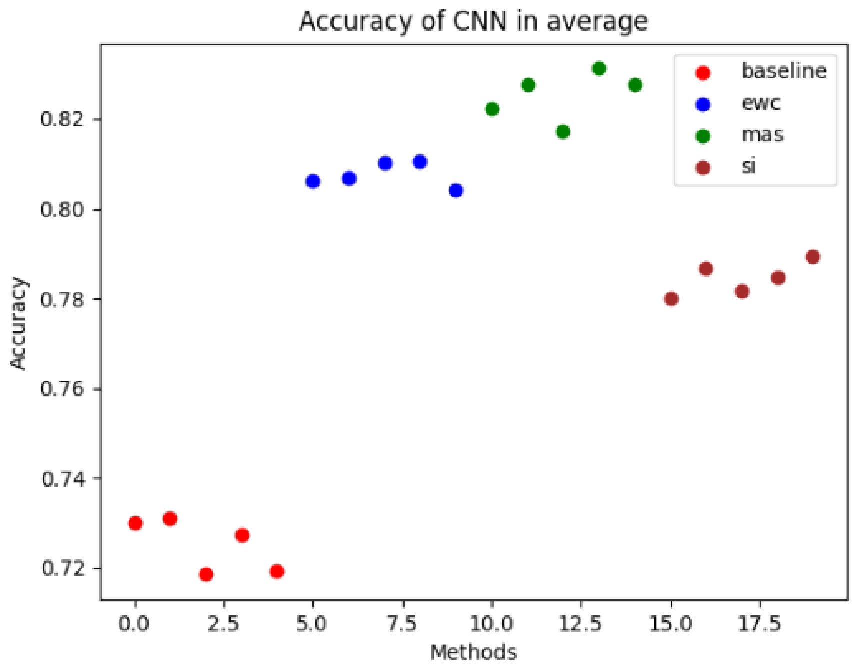


图 13: 所有范式在 talk 任务的结果

实验结果分析



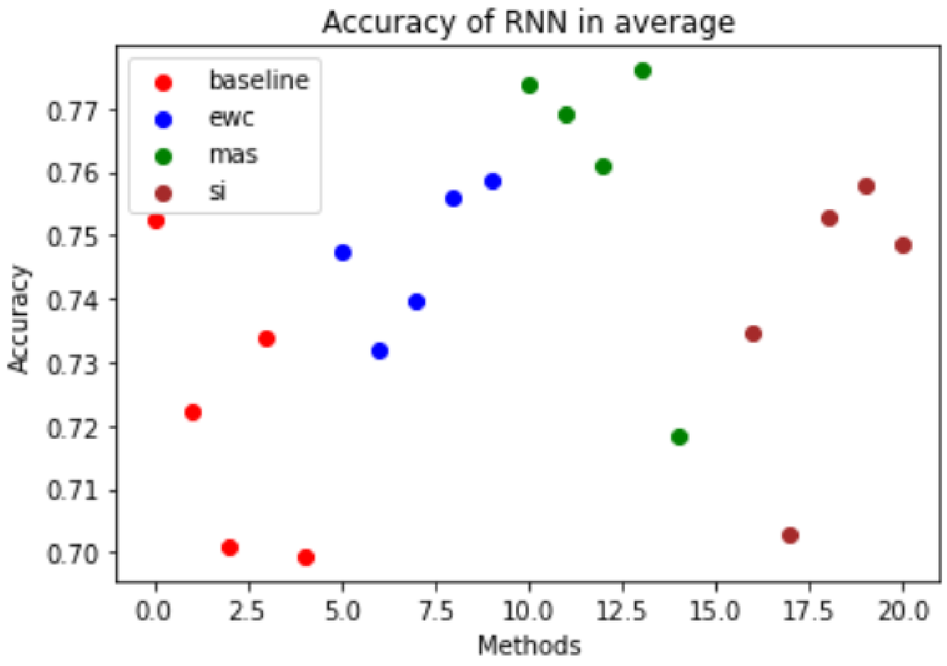


图 14: 所有范式的平均正确率结果

参考资料

- 李宏毅机器学习, https://speech.ee.ntu.edu.tw/~hylee/ml/2021-spring.html,2021.
- James Kirkpatrick and Razvan Pascanu and Neil Rabinowitz and Joel Veness and Guillaume Desjardins and Andrei A. Rusu and Kieran Milan and John Quan and Tiago Ramalho and Agnieszka Grabska-Barwinska and Demis Hassabis and Claudia Clopath and Dharshan Kumaran and Raia Hadsell. Overcoming catastrophic forgetting in neuralnetworks, CoRR, 2017.
- Rahaf Aljundi and Francesca Babiloni and Mohamed Elhoseiny and Marcus Rohrbach and Tinne Tuytelaars. Memory Aware Synapses: Learning what (not) to forget, ECCV, 2018.
- Soheil Kolouri and Nicholas A. Ketz and Andrea Soltoggio and Praveen K. Pilly, Sliced Cramer Synaptic Consolidation for Preserving Deeply Learned Representations, ICLR, 2020.

分工

施天予：

- 文本处理
- 循环神经网络
- EWC 和 MAS

孙奥远：

- 卷积神经网络
- SI
- 实验结果分析