mini_watson第二阶段实验报告

徐轶琦 何青蓉

实验进展

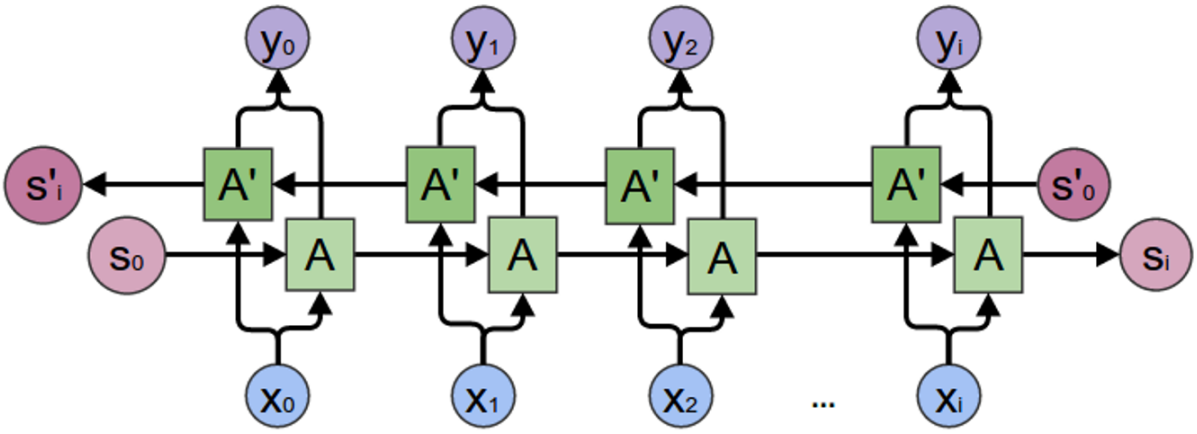
在第一阶段完成了淘宝客服聊天数据的预处理，将句子处理成句对的形式，并建立了词典。然后并通过阅读文献和相关代码对模型定义与训练有了初步了解，将Seq2Seq模型拆分成2个部分，逐步实现encoder和decoder模型。 在这一阶段完善了模型，完整实现了encoder，decoder以及Seq2Seq模型，并且定义了损失函数，利用处理过的数据对模型进行了训练，得到了一个模型，已经可以进行简单的对话。

实验方法

encoder模型

1. encoder模型定义

- 使用多层的Gated Recurrent Unit（GRU）作为Encoder，遍历每个词（Token），每个时刻的输入是上一个时刻的隐状态和输入，产生一个输出和新的隐状态，作为下一个时刻的输入隐状态。把最后一个时刻的隐状态作为Decoder的初始隐状态。最终返回所有时刻的输出和最后时刻的隐状态。



- 在接入RNN之前有一个embedding层，用来把每一个词（ID）映射成一个连续的稠密的向量，认为这个向量编码了一个词的语义。在模型里，把它的大小定义成和RNN的隐状态大小一样。有了embedding之后，模型会把相似的词编码成相似的向量（距离比较近）。

2. 实现encoder模型的步骤

- 把词的ID通过embedding层变成向量
- 把padding后的数据进行pack，利用torch.nn.utils.rnn.pack_padded_sequence和torch.nn.utils.rnn.pad_packed_sequence两个函数进行pack和unpack。
- 传入GRU进行Forward计算
- Unpack计算结果
- 把双向GRU的结果向量加起来
- 返回所有时刻的输出和最后时刻的隐状态
- 代码实现如下所示

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # 初始化GRU，这里输入和hidden大小都是hidden_size，这里假设embedding层的输
出大小是hidden_size
        # 如果只有一层，那么不进行Dropout，否则使用传入的参数dropout进行GRU的
Dropout。
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers,
                          dropout=(0 if n_layers == 1 else dropout),
bidirectional=True)

    def forward(self, input_seq, input_lengths, hidden=None):
        # 输入是(max_length, batch)，Embedding之后变成(max_length, batch,
hidden_size)
        embedded = self.embedding(input_seq)
        # Pack padded batch of sequences for RNN module
        # 因为RNN(GRU)要知道实际长度，所以PyTorch提供了函数pack_padded_sequence把
输入向量和长度
        # pack到一个对象PackedSequence里，这样便于使用。
        packed = torch.nn.utils.rnn.pack_padded_sequence(embedded,
input_lengths)
        # 通过GRU进行forward计算，需要传入输入和隐变量
        # 如果传入的输入是一个Tensor (max_length, batch, hidden_size)
        # 那么输出outputs是(max_length, batch, hidden_size*num_directions)。
        # 第三维是hidden_size和num_directions的混合，它们实际排列顺序是
num_directions在前面，
        # 因此我们可以使用outputs.view(seq_len, batch, num_directions,
hidden_size)得到4维的向量。
        # 其中第三维是方向，第四位是隐状态。

        # 而如果输入是PackedSequence对象，那么输出outputs也是一个PackedSequence对
象，我们需要用
        # 函数pad_packed_sequence把它变成shape为(max_length, batch,
hidden*num_directions)的向量以及
        # 一个list，表示输出的长度，当然这个list和输入的input_lengths完全一样，因此
通常我们不需要它。
        outputs, hidden = self.gru(packed, hidden)
        # 参考前面的注释，我们得到outputs为(max_length, batch,
hidden*num_directions)
        outputs, _ = torch.nn.utils.rnn.pad_packed_sequence(outputs)
        # 我们需要把输出的num_directions双向的向量加起来
        # 因为outputs的第三维是先放前向的hidden_size个结果，然后再放后向的
hidden_size个结果
        # 所以outputs[:, :, :self.hidden_size]得到前向的结果
        # outputs[:, :, self.hidden_size:]是后向的结果
        # 注意，如果bidirectional是False，则outputs第三维的大小就是hidden_size，
        # 这时outputs[:, : ,self.hidden_size:]是不存在的，因此也不会加上去。
        # 对Python slicing不熟的读者可以看看下面的例子：
```

```
        # >>> a=[1,2,3]
        # >>> a[:3]
        # [1, 2, 3]
        # >>> a[3:]
        # []
        # >>> a[:3]+a[3:]
        # [1, 2, 3]

        # 这样就不用写下面的代码了:
        # if bidirectional:
        #     outputs = outputs[:, :, :self.hidden_size] + outputs[:, :
,self.hidden_size:]
        outputs = outputs[:, :, :self.hidden_size] + outputs[:, :
,self.hidden_size:]
        # 返回最终的输出和最后时刻的隐状态。
        return outputs, hidden
```

decoder模型

1. decoder模型定义

- 每个时刻的输入是上一个时刻的隐状态和上一个时刻的输出。初始隐状态是Encoder最后时刻的隐状态。利用RNN计算新的隐状态和输出的第一个词，接着用新的新状态和第一个词计算第二个词，以此类推直到遇到结束符。
- 利用Attention机制，在Decoder进行t时刻计算的时候，除了t-1时刻的隐状态、当前时刻的输入，还会参考Encoder所有时刻的输入。

2. attention机制

- 用当前时刻的GRU计算出的新的隐状态来计算注意力得分，首先它用一个score函数计算这个隐状态和Encoder的输出的相似度得分，得分越大，说明越应该注意这个词。然后再用softmax函数把score变成概率。
- Score函数计算方法

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_a [\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

3. decoder模型的实现

- 把词ID输入embedding层
- 使用单向的GRU继续Forward进行一个时刻的计算
- 使用新的隐状态计算注意力权重
- 用注意力权重得到context向量
- Context向量和GRU的输出拼接起来，经过一个全连接网络，使得输出大小仍然是hidden_size

- 使用一个投影矩阵把输出从hidden_size变成词典大小，然后用softmax变成概率
- 返回输出和新的隐状态

```
# Luong 注意力layer
class Attn(torch.nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        if self.method not in ['dot', 'general', 'concat']:
            raise ValueError(self.method, "is not an appropriate attention
method.")
        self.hidden_size = hidden_size
        if self.method == 'general':
            self.attn = torch.nn.Linear(self.hidden_size, hidden_size)
        elif self.method == 'concat':
            self.attn = torch.nn.Linear(self.hidden_size * 2, hidden_size)
            self.v = torch.nn.Parameter(torch.FloatTensor(hidden_size))

    def dot_score(self, hidden, encoder_output):
        # 输入hidden的shape是(1, batch=64, hidden_size=500)
        # encoder_outputs的shape是(input_lengths=10, batch=64,
hidden_size=500)
        # hidden * encoder_output得到的shape是(10, 64, 500)，然后对第3维求和就
可以计算出score。
        return torch.sum(hidden * encoder_output, dim=2)

    def general_score(self, hidden, encoder_output):
        energy = self.attn(encoder_output)
        return torch.sum(hidden * energy, dim=2)

    def concat_score(self, hidden, encoder_output):
        energy =
self.attn(torch.cat((hidden.expand(encoder_output.size(0), -1, -1),
                  encoder_output), 2)).tanh()
        return torch.sum(self.v * energy, dim=2)

    # 输入是上一个时刻的隐状态hidden和所有时刻的Encoder的输出encoder_outputs
    # 输出是注意力的概率，也就是长度为input_lengths的向量，它的和加起来是1。
    def forward(self, hidden, encoder_outputs):
        # 计算注意力的score，输入hidden的shape是(1, batch=64,
hidden_size=500),
        # 表示t时刻batch数据的隐状态
        # encoder_outputs的shape是(input_lengths=10, batch=64,
hidden_size=500)
        if self.method == 'general':
            attn_energies = self.general_score(hidden, encoder_outputs)
        elif self.method == 'concat':
            attn_energies = self.concat_score(hidden, encoder_outputs)
        elif self.method == 'dot':
            # 计算内积，参考dot_score函数
            attn_energies = self.dot_score(hidden, encoder_outputs)
```

```
        # Transpose max_length and batch_size dimensions
        # 把attn_energies从(max_length=10, batch=64)转置成(64, 10)
        attn_energies = attn_energies.t()

        # 使用softmax函数把score变成概率，shape仍然是(64, 10)，然后用
unsqueeze(1)变成
        # (64, 1, 10)
        return F.softmax(attn_energies, dim=1).unsqueeze(1)


class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size,
n_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # 保存到self里，attn_model就是前面定义的Attn类的对象。
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # 定义Decoder的layers
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0
if n_layers == 1 else dropout))
        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        # 注意：decoder每一步只能处理一个时刻的数据，因为t时刻计算完了才能计算t+1时
刻。
        # input_step的shape是(1, 64)，64是batch，1是当前输入的词ID(来自上一个时刻
的输出)
        # 通过embedding层变成(1, 64, 500)，然后进行dropout，shape不变。
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)
        # 把embedded传入GRU进行forward计算
        # 得到rnn_output的shape是(1, 64, 500)
        # hidden是(2, 64, 500)，因为是两层的GRU，所以第一维是2。
        rnn_output, hidden = self.gru(embedded, last_hidden)
        # 计算注意力权重， 根据前面的分析，attn_weights的shape是(64, 1, 10)
        attn_weights = self.attn(rnn_output, encoder_outputs)

        # encoder_outputs是(10, 64, 500)
        # encoder_outputs.transpose(0, 1)后的shape是(64, 10, 500)
        # attn_weights.bmm后是(64, 1, 500)

        # bmm是批量的矩阵乘法，第一维是batch，我们可以把attn_weights看成64个(1,10)
的矩阵
        # 把encoder_outputs.transpose(0, 1)看成64个(10, 500)的矩阵
```

```
        # 那么bmm就是64个(1, 10)矩阵 x (10, 500)矩阵，最终得到(64, 1, 500)
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1))
        # 把context向量和GRU的输出拼接起来
        # rnn_output从(1, 64, 500)变成(64, 500)
        rnn_output = rnn_output.squeeze(0)
        # context从(64, 1, 500)变成(64, 500)
        context = context.squeeze(1)
        # 拼接得到(64, 1000)
        concat_input = torch.cat((rnn_output, context), 1)
        # self.concat是一个矩阵(1000, 500)，
        # self.concat(concat_input)的输出是(64, 500)
        # 然后用tanh把输出返回变成(-1,1)，concat_output的shape是(64, 500)
        concat_output = torch.tanh(self.concat(concat_input))

        # out是(500, 词典大小=7826)
        output = self.out(concat_output)
        # 用softmax变成概率，表示当前时刻输出每个词的概率。
        output = F.softmax(output, dim=1)
        # 返回 output和新的隐状态
        return output, hidden
```

定义训练过程

1. 损失函数 由于判断语义相近存在一定的困难，因此在判定答案的相近性时，限制decoder的输出长度，使用交叉熵损失函数

```
def maskNLLLoss(inp, target, mask):
    # 计算实际的词的个数，因为padding是0，非padding是1，因此sum就可以得到词的个数
    nTotal = mask.sum()

    crossEntropy = -torch.log(torch.gather(inp, 1, target.view(-1,
1)).squeeze(1))
    loss = crossEntropy.masked_select(mask).mean()
    loss = loss.to(device)
    return loss, nTotal.item()
```

2. batch数据的训练

- 在decoder训练过程中，利用teacher forcing，无论模型在t-1时刻做什么预测都把t-1时刻的正确答案作为t时刻的输入，为了保证正确性，利用teacher_forcing_ratio参数随机的来确定本次训练是否teacher forcing
- 梯度裁剪(gradient clipping) 。这个技巧通常是为了防止梯度爆炸(exploding gradient)，它把参数限制在一个范围之内，从而可以避免梯度的梯度过大或者出现NaN等问题。注意：虽然它的名字叫梯度裁剪，但实际它是对模型的参数进行裁剪，它把整个参数看成一个向量，如果这个向量的模大于max_norm，那么就把这个向量除以一个值使得模等于max_norm
- 操作过程：把整个batch的输入传入encoder；把decoder的输入设置为特殊的，初始隐状态设置为encoder最后时刻的隐状态；decoder每次处理一个时刻的forward计算；如果是teacher forcing，把上个时刻的"正确的"词作为当前输入，否则用上一个时刻的输出作为当前时刻的输入；计算loss ； 反向计算梯度 ； 对梯度进行裁剪 ； 更新模型(包括encoder和decoder)参数

```
def train(input_variable, lengths, target_variable, mask, max_target_len,
encoder, decoder, embedding,
          encoder_optimizer, decoder_optimizer, batch_size, clip,
max_length=MAX_LENGTH):

    # 梯度清空
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # 设置device，从而支持GPU，当然如果没有GPU也能工作。
    input_variable = input_variable.to(device)
    lengths = lengths.to(device)
    target_variable = target_variable.to(device)
    mask = mask.to(device)

    # 初始化变量
    loss = 0
    print_losses = []
    n_totals = 0

    # encoder的Forward计算
    encoder_outputs, encoder_hidden = encoder(input_variable, lengths)

    # Decoder的初始输入是SOS，我们需要构造(1, batch)的输入，表示第一个时刻batch个输
入。
    decoder_input = torch.LongTensor([[SOS_token for _ in
range(batch_size)]])
    decoder_input = decoder_input.to(device)

    # 注意：Encoder是双向的，而Decoder是单向的，因此从下往上取n_layers个
    decoder_hidden = encoder_hidden[:decoder.n_layers]

    # 确定是否teacher forcing
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio
else False

    # 一次处理一个时刻
    if use_teacher_forcing:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )
            # Teacher forcing: 下一个时刻的输入是当前正确答案
            decoder_input = target_variable[t].view(1, -1)
            # 计算累计的loss
            mask_loss, nTotal = maskNLLLoss(decoder_output,
target_variable[t], mask[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal
    else:
        for t in range(max_target_len):
```

```
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )
            # 不是teacher forcing: 下一个时刻的输入是当前模型预测概率最高的值
            _, topi = decoder_output.topk(1)
            decoder_input = torch.LongTensor([[topi[i][0] for i in
range(batch_size)]])
            decoder_input = decoder_input.to(device)
            # 计算累计的loss
            mask_loss, nTotal = maskNLLLoss(decoder_output,
target_variable[t], mask[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal

    # 反向计算
    loss.backward()

    # 对encoder和decoder进行梯度裁剪
    _ = torch.nn.utils.clip_grad_norm_(encoder.parameters(), clip)
    _ = torch.nn.utils.clip_grad_norm_(decoder.parameters(), clip)

    # 更新参数
    encoder_optimizer.step()
    decoder_optimizer.step()

    return sum(print_losses) / n_totals
```

贪心解码Greedy decoding

- decoder训练完成后需要进行解码，查看效果，这里使用简单的贪心算法，即每次都选择概率最高的那个词，然后把这个词作为下一个时刻的输入，直到遇到EOS结束解码或者达到一个最大长度。
- 贪心算法也存在一定问题，不一定能得到最优解，因为某个答案可能开始的几个词的概率并不太高，但是后来概率会很大。之后也会考虑采用Beam-Search算法等算法比较效果。
- 操作过程：把输入传给Encoder，得到所有时刻的输出和最后一个时刻的隐状态；把Encoder最后时刻的隐状态作为Decoder的初始状态；Decoder的第一输入初始化为SOS；定义保存解码结果的tensor；循环直到最大解码长度

```
class GreedySearchDecoder(nn.Module):
    def __init__(self, encoder, decoder):
        super(GreedySearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, input_seq, input_length, max_length):
        # Encoder的Forward计算
        encoder_outputs, encoder_hidden = self.encoder(input_seq,
input_length)
        # 把Encoder最后时刻的隐状态作为Decoder的初始值
```

```
        decoder_hidden = encoder_hidden[:decoder.n_layers]
        # 因为我们的函数都是要求(time,batch)，因此即使只有一个数据，也要做出二维的。
        # Decoder的初始输入是SOS
        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long)
* SOS_token
        # 用于保存解码结果的tensor
        all_tokens = torch.zeros([0], device=device, dtype=torch.long)
        all_scores = torch.zeros([0], device=device)
        # 循环，这里只使用长度限制，后面处理的时候把EOS去掉了。
        for _ in range(max_length):
            # Decoder forward一步
            decoder_output, decoder_hidden = self.decoder(decoder_input,
decoder_hidden,
                                encoder_outputs)
            # decoder_outputs是(batch=1, vob_size)
            # 使用max返回概率最大的词和得分
            decoder_scores, decoder_input = torch.max(decoder_output,
dim=1)
            # 把解码结果保存到all_tokens和all_scores里
            all_tokens = torch.cat((all_tokens, decoder_input), dim=0)
            all_scores = torch.cat((all_scores, decoder_scores), dim=0)
            # decoder_input是当前时刻输出的词的ID，这是个一维的向量，因为max会减少
一维。
            # 但是decoder要求有一个batch维度，因此用unsqueeze增加batch维度。
            decoder_input = torch.unsqueeze(decoder_input, 0)
        # 返回所有的词和得分。
        return all_tokens, all_scores
```

第二阶段成果展示

将训练轮数定为2000轮，与训练模型好的模型进行了一系列对话，效果如下所示

```
你好
Bot: 您好
买的东西什么时候能到啊
Bot: 一般1到3天的时间
能不能便宜一点
Bot: 您拍下的订单价格已经改好咯您
可以开发票吗
Bot: 可以的
你们店没有优惠活动吗
Bot: 亲可以关注下
那快递可以发顺丰吗，不想要圆通
Bot: 可以的哦
赠品是什么
Bot: 是的哦
发什么快递
Bot: 亲爱哒客官小店默认邮政百世汇通中通快递
```

模型分析：目前对于一些简单的对话，机器人已经可以做简短的回答。但是如果问的问题中存在词典中未出现的词，机器人将无法回答问题，这也是下一步需要改进的地方。

后续任务

- 改进模型，在解码过程改进算法，提高问答准确度和合理性
- 尝试每次分析时，考虑上下文的联系，将问题与前文的对答联系起来，给出合理的回答。
- 为问答机器人设计UI界面，提高交互性