miniWatson项目总结报告

信息学院 徐轶琦 2018202197

项目分工

- 前期准备
  - 文献搜集：徐轶琦
  - 相关代码、实现框架的搜集和复现：何青蓉
- 数据预处理
  - 分词、构建词典、去除低频词和过长的句子：徐轶琦
- 模型的建立
  - encoder、decoder模型的建立：何青蓉
- 模型的训练
  - 解码算法的定义、loss函数的定义：徐轶琦
- 模型的优化
  - attention机制：何青蓉
  - teaching force：徐轶琦
- 交互界面
  - 交互界面设计：何青蓉

项目介绍

- Watson 是 IBM 自然语言处理领域的智能问答系统。本小组计划基于 Seq2Seq 模型，利用淘宝客服对话数据，实现一个 miniWatson 聊天机器人。
- 人机对话模型是人工智能研究领域的热门话题，聊天机器人有着广泛的应用，例如客 户服务应用和在线帮助。传统的聊天机器人主要是基于检索式的模型——提前建立一个问答库，根据用户输入的问题检索相应的答案，这样的模型输出的答案是固定不变的，且问答系统无法回答语料库中未出现的问题。随着深度神经网络研究的发展，基于深度学习的生成式模型让问答系统能够产生更加灵活的答案，回答语料库中未出现的问题，使人机交互更加灵活。
- 本项目实现 miniWatson 采用的是生成式模型——基于 RNN 的 Seq2Seq 序列到序列模型。Seq2Seq 属于 encoder-decoder 结构的一种，利用两个RNN，一个 RNN作为 encoder， 负责将输入的序列压缩成指定长度的向量，这个向量表征了序列的语义;另一个 RNN 作为decoder，负责根据语义向量生成指定的序列。
- 本次实验报告着重介绍我在小组工作中的贡献。

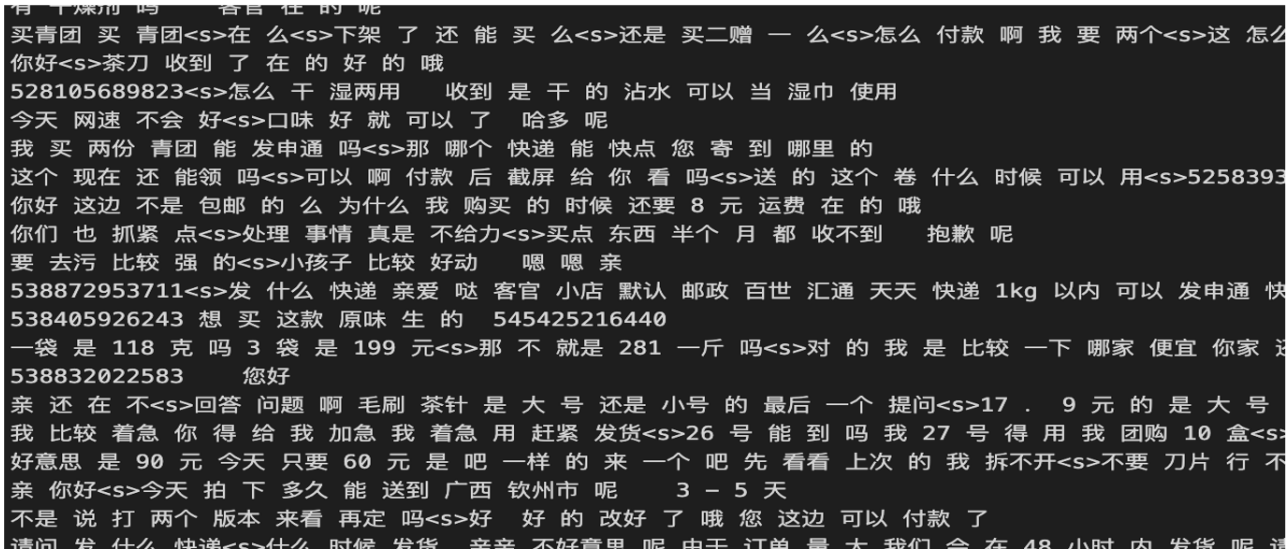
前期资料搜集

- 在确定选题之后，我开展了对聊天机器人现有成果的调研。我搜集了有关对话的相关论文，包括
  - Sequence to sequence learning with neural networks(NIPS) 介绍了seq2seq模型
  - Neural Responding Machine for Short-Text Conversation（ACL）介绍了seq2seq 应用于短文本对话
  - Learning to Respond with Deep Neural Networks for Retrieval-Based Human-Computer Conversation System（SIGIR）介绍了如何利用深度学习实现检索式对话
- 我了解到在当前的对话领域主要分为了2大方向，一种是生成式，一种是检索式，检索式存在的问题在于需要搜集大量已有回答作为候选，数据庞大且范围限定，而生成式能够解决这样的问题。在与小组成员讨论后我们确定实现一个生成式的问答系统，以对话领域的经典模型seq2seq模型为基础实现chatbot

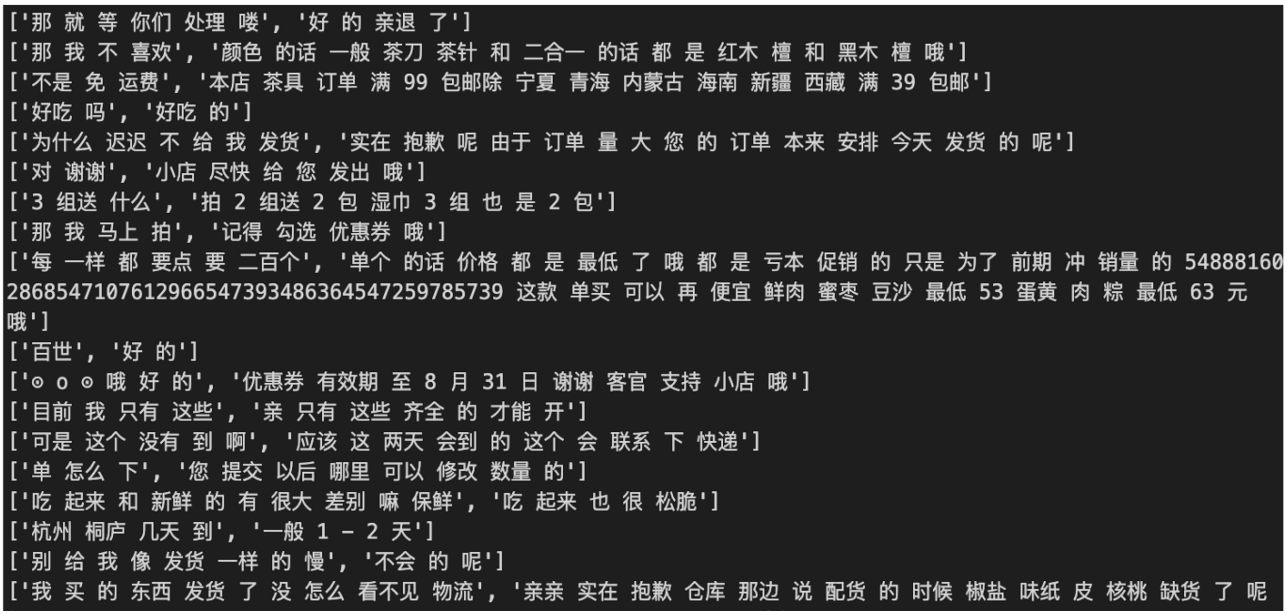
数据的预处理

在确定使用的模型和方法后，我开始搜索训练数据，项目要求的是针对特定领域的对话机器人，而不是open domain，这降低了一定的难度。我选定了淘宝客服与买家的对话数据，一共100000组对话作为我们小组的训练数据，并对其进行了数据预处理，便于后续构建tensor进行神经网络的训练。

1. 先对拿到的context_respond.txt数据进行切词，如下图所示



2. 去除特殊字符，然后将整段对话切分为<question，answer>的句对形式，如下所示



3. 读取句对中的词语，去除低频词，构建词典类。保持词语到索引index的映射，同时保持index到word的映射，记录词语出现的频次。词语总数为21390，实现的代码如下

```
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3  # Count SOS, EOS, PAD

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.num_words
            self.word2count[word] = 1
            self.index2word[self.num_words] = word
            self.num_words += 1
        else:
            self.word2count[word] += 1

    # Remove words below a certain count threshold
    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True

        keep_words = []
```

4. 构建词典后，就可以将句子里面的词语用索引编号表示，接下来就是构建模型输入需要的tensor矩阵。为了提高gpu的运行效率，选择让多个句对共同参与训练，我每一次训练都会从句对中随机选出batch个句对，组成（batch，length）大小的tensor，length为句对中的最大长度，长度不足的都补零，为了提高读取效率，会将（batch，length）大小tensor转置为（length，batch）大小的tensor。部分结果如下图所示。
    1. 输入的问句tensor

```
input_variable: tensor([[ 204,  276,  255,   43,   43],
        [ 593,   44,  460,    6,    2],
        [ 255,  185,   28,  665,    0],
        [ 276,   30, 1159,  519,    0],
        [ 185,  242, 7828, 2283,    0],
        [ 150,   81,   28,  144,    0],
        [ 597,  270,  908,    2,    0],
        [  30, 1131,  356,    0,    0],
        [  22,  986, 3188,    0,    0],
        [ 142,  347,    2,    0,    0],
        [ 111,    2,    0,    0,    0],
        [ 215,    0,    0,    0,    0],
        [ 490,    0,    0,    0,    0],
        [ 276,    0,    0,    0,    0],
        [  16,    0,    0,    0,    0],
        [   2,    0,    0,    0,    0]])
lengths: tensor([16, 11, 10,  7,  2])
```

2. 目标回答tensor

```
target_variable: tensor([[ 204,  369,   27, 3504,  204],
        [ 204,   16,   28,  366,  204],
        [  27,   17,  136,   17,  136],
        [  28,  593,    2,    2,    2],
        [   2,   72,    0,    0,    0],
        [   0,  436,    0,    0,    0],
        [   0,   16,    0,    0,    0],
        [   0,  180,    0,    0,    0],
        [   0,  998,    0,    0,    0],
        [   0,   65,    0,    0,    0],
        [   0,   72,    0,    0,    0],
        [   0,  346,    0,    0,    0],
        [   0,   99,    0,    0,    0],
        [   0,   72,    0,    0,    0],
        [   0,  438,    0,    0,    0],
        [   0,   72,    0,    0,    0],
        [   0,  358,    0,    0,    0],
        [   0, 2878,    0,    0,    0],
        [   0,  142,    0,    0,    0],
        [   0,  874,    0,    0,    0],
        [   0,  150,    0,    0,    0],
        [   0,   12,    0,    0,    0],
        [   0,  339,    0,    0,    0],
        [   0,   72,    0,    0,    0],
        [   0,  360,    0,    0,    0],
        [   0,  110,    0,    0,    0],
        [   0,   72,    0,    0,    0],
        [   0,  438,    0,    0,    0],
        [   0,  110,    0,    0,    0],
        [   0,   62,    0,    0,    0],
        [   0,    2,    0,    0,    0]])
mask: tensor([[ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
        [False,  True, False, False, False],
```

模型的优化和训练

在模型的基本框架搭建完成后，对模型进行了部分优化以提高模型的准确性和收敛速度；然后利用贪心解码进行了效果测试

训练过程的优化

定义损失函数

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()
```

teaching force

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，但decoder并不能纠正错误，因此利用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

一次训练的操作过程

1. 把整个batch的输入传入encoder；
2. 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；
3. decoder每次处理一个时刻的forward计算；
4. 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；
5. 计算loss；
6. 反向计算梯度；
7. 对梯度进行裁剪；
8. 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。以后也会考虑采用Beam-Search算法等算法比较效果。

操作过程

1. 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；
2. 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；
3. 定义保存解码结果的tensor；
4. 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
```

miniwatson效果展示

之前尝试将模型训练1000轮，但是效果很差，chatbot基本只会几句问候语，然后尝试将模型训练了2000轮，得到良好的效果。最终将项目部署到django，为此设计了图形化页面，最终部分对话效果如下所示


我 2020/12/16 16:57
能送一些别的东西试吃吗
miniWatson 2020/12/16 16:57
送的哦
我 2020/12/16 16:58
亲送什么快递
miniWatson 2020/12/16 16:58
亲爱哒客官小店默认邮政百世汇通中通快递


我 2020/12/16 16:58
可以免运费吗
miniWatson 2020/12/16 16:58
亲可以关注下我们店铺的优惠券哦
我 2020/12/16 16:58
那快递能今天发吗
miniWatson 2020/12/16 16:58
可以的哦

项目体会

本次实验利用了seq2seq模型来实现chatbot，遗憾的是，由于能力有限在模型中并没有考虑上下文的影响，每一次训练只是针对一组对话，在之后的学习过程中会尝试加入上文，实现个性化的对话。 总之，通过这次项目的实现，对深度学习相关的应用，尤其是在对话领域的应用，有了初步的了解；对于pytorch编程也有了一定的掌握。这是一次非常好的体验。