

成 绩

****

深度学习与自然语言处理

第五次大作业

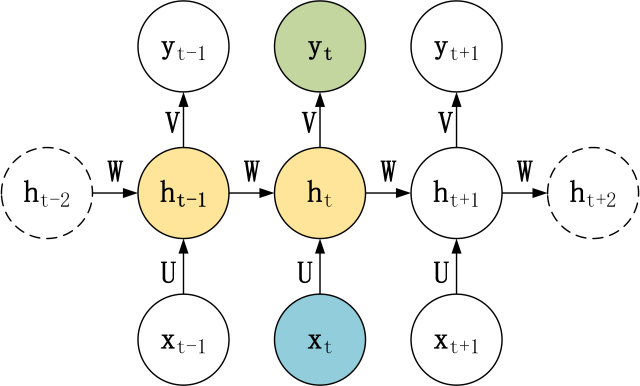
|  |  |
| --- | --- |
| 院（系）名称 | 自动化科学与电气工程学院 |
| 专业名称 | 电子信息 |
| 学号 | ZY2103309 |
| 姓名 | 熊方书 |
| 指导教师 | 秦曾昌 |

2022年5月19日

1. **任务描述**

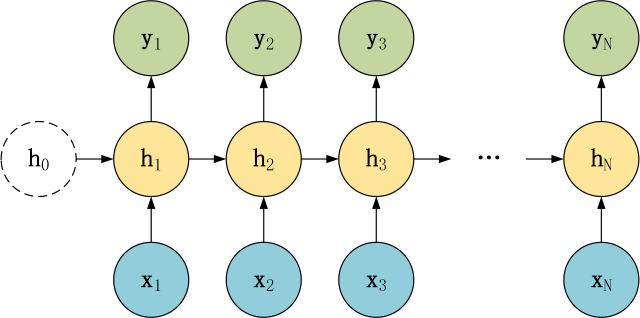
基于 Seq2Seq 模型来实现文本生成的模型，输入可以为一段已知的金庸小说段落，来生成新的段落并做分析。

1. **实验原理**
2. **RNN结构**

****

**图1 RNN结构**

RNN 基本的模型如上图所示，每个神经元接受的输入包括：前一个神经元的隐藏层状态 h(用于记忆) 和当前的输入 x (当前信息)。神经元得到输入之后，会计算出新的隐藏状态 h 和输出 y，然后再传递到下一个神经元。因为隐藏状态 h 的存在，使得 RNN 具有一定的记忆功能。针对不同任务，通常要对 RNN 模型结构进行少量的调整，根据输入和输出的数量，分为三种比较常见的结构：N vs N、1 vs N、N vs 1。



**图2 N vs N**

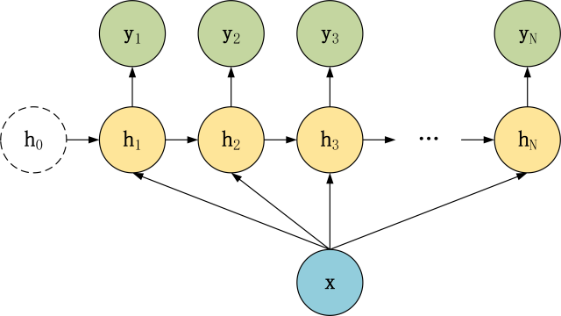
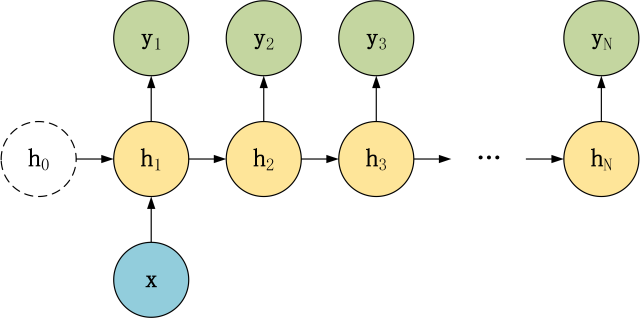
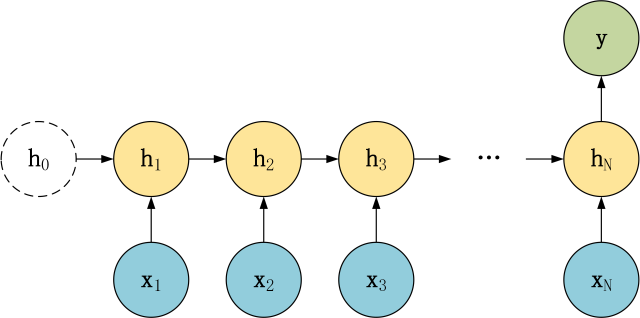


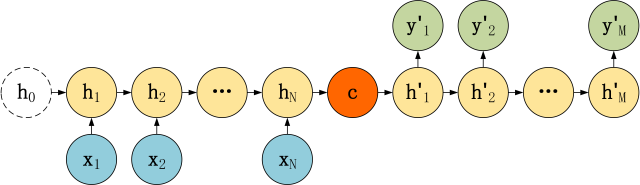
图3 1 vs N

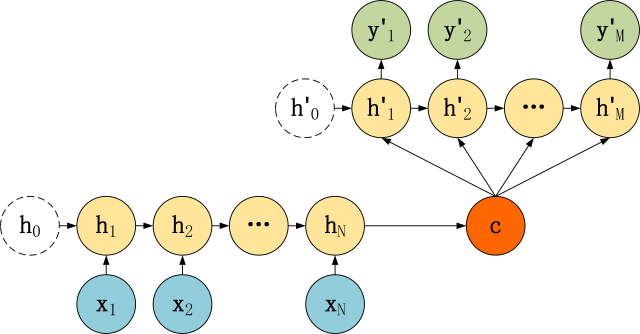


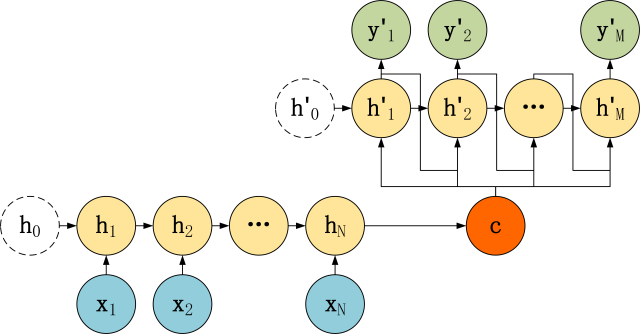
图四 Nvs1

1. **Seq2Seq工作原理**

Seq2Seq 是一种重要的 RNN 模型，也称为 Encoder-Decoder 模型，可以理解为一种 N×M的模型。模型包含两个部分：Encoder 用于编码序列的信息，将任意长度的序列信息编码到一个向量 c 里。而 Decoder 是解码器，解码器得到上下文信息向量 c 之后可以将信息解码，并输出为序列。Seq2Seq 模型结构有很多种，下面是几种比较常见的：

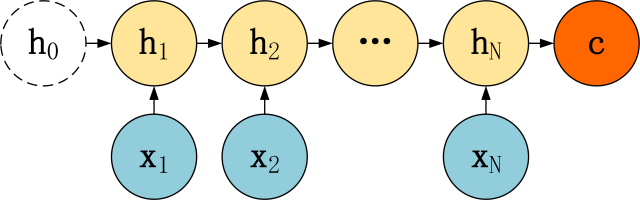






**2.1 编码器 Encoder**

这三种 Seq2Seq 模型的主要区别在于 Decoder，他们的 Encoder 都是一样的。下图是 Encoder 部分，Encoder 的 RNN 接受输入 x，最终输出一个编码所有信息的上下文向量 c，中间的神经元没有输出。Decoder 主要传入的是上下文向量 c，然后解码出需要的信息。



从上图可以看到，Encoder 与一般的 RNN 区别不大，只是中间神经元没有输出。其中的上下文向量 c可以采用多种方式进行计算。

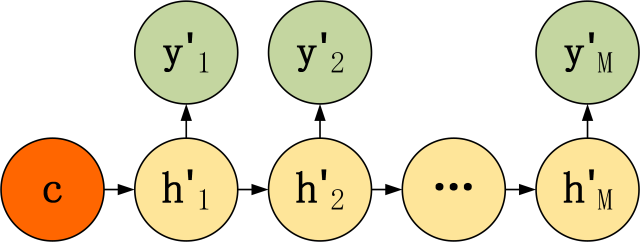


从公式可以看到，c可以直接使用最后一个神经元的隐藏状态 hN 表示；也可以在最后一个神经元的隐藏状态上进行某种变换 hN 而得到，q 函数表示某种变换；也可以使用所有神经元的隐藏状态 h1, h2, ..., hN 计算得到。得到上下文向量 c 之后，需要传递到 Decoder。

**2.2 解码器 Decoder**

Decoder 有多种不同的结构，这里主要介绍三种。

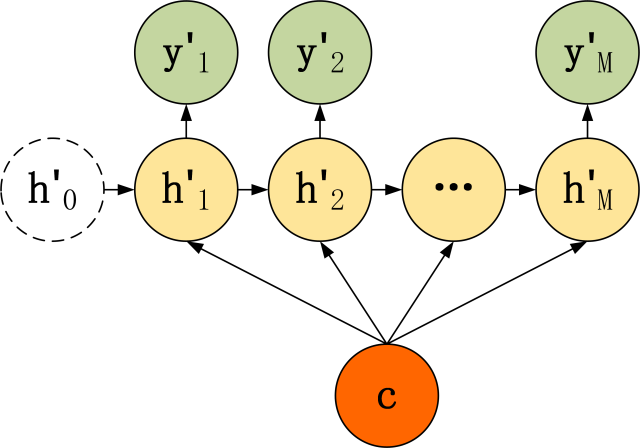
第一种：

****

第一种 Decoder 结构比较简单，将上下文向量 c当成是 RNN 的初始隐藏状态，输入到 RNN 中，后续只接受上一个神经元的隐藏层状态 h' 而不接收其他的输入 x。第一种 Decoder 结构的隐藏层及输出的计算公式：



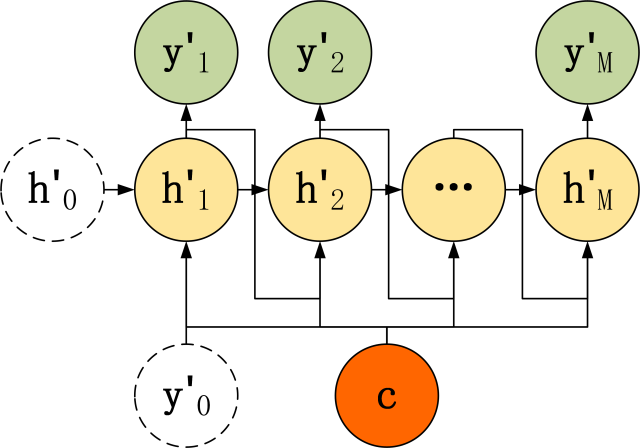
第二种：



第二种 Decoder 结构有了自己的初始隐藏层状态 h'0，不再把上下文向量 c当成是 RNN 的初始隐藏状态，而是当成 RNN 每一个神经元的输入。可以看到在 Decoder 的每一个神经元都拥有相同的输入 c，这种 Decoder 的隐藏层及输出计算公式：

Word2vec是基于统计语言模型而构建的，统计语言模型是用于统计一串字符序列成立的概率大小的概率模型。假定 代表一个字符串，组成成分为 ，记为 ，则由 的联合概率可计算出出现的概率：

第三种：



第三种 Decoder 结构和第二种类似，但是在输入的部分多了上一个神经元的输出 y'。即每一个神经元的输入包括：上一个神经元的隐藏层向量 h'，上一个神经元的输出 y'，当前的输入 c(Encoder 编码的上下文向量)。对于第一个神经元的输入 y'0，通常是句子其实标志位的 embedding 向量。第三种 Decoder 的隐藏层及输出计算公式：



1. **实验内容与结果**

先对数据进行预处理，删除非中文字符和标点符号。

**输入：**左子穆脸色微微发紫，若不是大敌在外，早已发作，当强忍怒气，道：“不吃！”段誉插口道：“你这是什么瓜子？桂花？玫瑰？还是松子味的？”那少女道：“啊哟！瓜子还有许多讲究么？我可不知道了。

**输出**：我这瓜子是……妈妈……用蛇胆炒的，常吃眼目明亮，你试试看。”说着抓了一把，塞在段誉手中，又道：“吃不惯的人，觉得有点儿苦，其实很好吃的。”段誉不便拂她之意，拿了一粒瓜子送入口中，入口果觉辛涩，但略加辨味，便似谏果回甘，舌底生津，当下接连吃了起来。他将吃过的瓜子壳一片片的放在梁上，那少女却肆无忌惮，顺口便往下吐出。瓜子壳在众人头顶上乱飞，许多人都皱眉避开。左子穆又问：“姑娘……在道上听到什么消息，若能见告，在下……在下感激不尽。”他为了探听消息，言语只得十分客气。那少女道：“我听神农帮的说什么

**附录：代码**

import osimport unidecodeimport stringimport randomimport reimport numpy as npimport torchimport torch.nn as nnfrom torch.autograd import Variablefrom torch.nn import functional as Fimport timeimport mathimport matplotlib.pyplot as pltimport matplotlib.ticker as tickeros.environ['KMP\_DUPLICATE\_LIB\_OK']='True'all\_characters = string.printablen\_characters = len(all\_characters)# 获取训练数据all\_files = ""for file in os.listdir('./data'):# all\_files +=unidecode.unidecode(open('./data/'+file,encoding='UTF-8').read()) +"\n" all\_files += (open('./data/' + file, encoding='ansi').read())+ "\n"file\_len = len(all\_files)print("文本字符总长度:", file\_len)# 使用 CUDAuse\_cuda = Falseif torch.cuda.is\_available(): use\_cuda = Truechar\_set = set()char\_set.update(all\_files)all\_characters = dict(zip(char\_set, range(len(char\_set))))to\_character = {v: k for k, v in all\_characters.items()}n\_characters = len(all\_characters)print("总共的字符个数:", n\_characters)# 定义每次用于训练的字符串长度chunk\_len = 300# 随机挑选一段该长度 chunk\_len 的字符串def random\_chunk(chunk\_len): start\_index = random.randint(0, file\_len - chunk\_len) end\_index = start\_index + chunk\_len + 1 return all\_files[start\_index:end\_index]# main model class 模型定义class TextGenerate(nn.Module): def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, n\_layers=1, bi=True): super(TextGenerate, self).\_\_init\_\_() self.input\_size = input\_size self.hidden\_size = hidden\_size self.output\_size = output\_size self.n\_layers = n\_layers self.bi = bi self.encoder = nn.Embedding(input\_size, hidden\_size) self.lstm = nn.LSTM(hidden\_size, hidden\_size, n\_layers, bidirectional=self.bi) if self.bi: self.decoder = nn.Linear(hidden\_size\*2, output\_size) else: self.decoder = nn.Linear(hidden\_size, output\_size) self.out = nn.Linear(output\_size, output\_size) self.dropout = nn.Dropout(0.1) def forward(self, input, hidden, cell): # encoder input = self.encoder(input.view(1, -1)) input = self.dropout(input) output,states = self.lstm(input.view(1, 1, -1), (hidden, cell)) output = output.permute(1, 0, 2) # attention if self.bi: out1, out2 = output[:,:,:self.hidden\_size], output[:,:,self.hidden\_size:] h1, h2 = states[0][states[0].size()[0] - 2,:,:], states[0][states[0].size()[0] - 1,:,:] attn\_wts\_1 = F.softmax(torch.bmm(out1, h1.unsqueeze(2)).squeeze(2), 1) attn\_wts\_2 = F.softmax(torch.bmm(out2, h2.unsqueeze(2)).squeeze(2), 1) attn\_1 = torch.bmm(out1.transpose(1, 2), attn\_wts\_1.unsqueeze(2)).squeeze(2) attn\_2 = torch.bmm(out2.transpose(1, 2), attn\_wts\_2.unsqueeze(2)).squeeze(2) attn = torch.cat((attn\_1, attn\_2), 1) else: h = states.squeeze(0) attn\_wts = F.softmax(torch.bmm(output, h.unsqueeze(2)).squeeze(2), 1) attn = torch.bmm(output.transpose(1, 2), attn\_wts.unsqueeze(2)).squeeze(2) # decoder output = self.decoder(attn) output = self.dropout(output) output = self.out(output) return output, states def init\_hidden(self): if self.bi: return Variable(torch.zeros(self.n\_layers\*2, 1, self.hidden\_size)) else: return Variable(torch.zeros(self.n\_layers, 1, self.hidden\_size)) def init\_cell(self): if self.bi: return Variable(torch.zeros(self.n\_layers\*2, 1, self.hidden\_size)) else: return Variable(torch.zeros(self.n\_layers, 1, self.hidden\_size))# turn string into list of longsdef char\_tensor(string): tensor = torch.zeros(len(string)).long() for c in range(len(string)): # tensor[c] = all\_characters.index(string[c]) tensor[c] = all\_characters[string[c]] if use\_cuda: tensor = tensor.cuda() return Variable(tensor)# get random training datadef random\_training\_set(chunk\_len): chunk = random\_chunk(chunk\_len) inp = char\_tensor(chunk[:-1]) target = char\_tensor(chunk[1:]) return inp, target# evaluate modeldef evaluate(target\_str, prime\_str='A', predict\_len=100, temperature=0.8): model.load\_state\_dict(torch.load('./model\_generate.pt')) model.eval() hidden = model.init\_hidden() cell = model.init\_cell() if use\_cuda: hidden = hidden.cuda() cell = cell.cuda() prime\_input = char\_tensor(prime\_str) predicted = prime\_str + "\n-------->\n" # use priming string to "build up" hidden state for p in range(len(prime\_str) - 1): output,states = model(prime\_input[p], hidden, cell) if use\_cuda: hidden, cell = states[0].cuda(), states[1].cuda() else: hidden, cell = states[0], states[1] inp = prime\_input[-1] loss = 0. for p in range(predict\_len): output, states = model(inp, hidden, cell) if use\_cuda: output = output.cuda() hidden, cell = states[0].cuda(), states[1].cuda() else: hidden, cell = states[0], states[1] target = char\_tensor(target\_str[p]) loss += criterion(output, target) # sample from the network as a multinomial distribution output\_dist = output.data.view(-1).div(temperature).exp() top\_i = torch.multinomial(output\_dist, 1)[0] # add predicted character to string and use as next input # predicted\_char = all\_characters[top\_i] predicted\_char = to\_character[top\_i.item()] predicted += predicted\_char inp = char\_tensor(predicted\_char) loss\_tot = total\_loss(loss, predict\_len) perplexity = perplexity\_score(loss\_tot) return predicted, loss\_tot, perplexity# get lossdef total\_loss(loss, predict\_len): loss\_tot = loss.cpu().item()/predict\_len return loss\_tot# get perplexitydef perplexity\_score(loss): perplexity = 2\*\*loss return perplexity# helper function for time elapseddef time\_since(since): s = time.time() - since m = math.floor(s / 60) s -= m \* 60 return '%dm %ds' % (m, s)# train modeldef train(inp, target): model.train() target.unsqueeze\_(-1) hidden = model.init\_hidden() cell = model.init\_cell() if use\_cuda: hidden = hidden.cuda() cell = cell.cuda() model.zero\_grad() loss = 0.0 for c in range(chunk\_len): output, states = model(inp[c], hidden, cell) if use\_cuda: output = output.cuda() hidden, cell = states[0].cuda(), states[1].cuda() else: hidden, cell = states[0], states[1] loss += criterion(output, target[c]) loss.backward() model\_optimizer.step() torch.save(model.state\_dict(), './model\_generate.pt') loss\_tot = total\_loss(loss, chunk\_len) perplexity = perplexity\_score(loss\_tot) return loss\_tot, perplexity# generate text given contextdef generate(prime\_str='A', predict\_len=100, temperature=0.8): model.load\_state\_dict(torch.load('./model\_generate.pt')) model.eval() hidden = model.init\_hidden() cell = model.init\_cell() if use\_cuda: hidden = hidden.cuda() cell = cell.cuda() prime\_input = char\_tensor(prime\_str) predicted = prime\_str + "\n--------->\n" # use priming string to "build up" hidden state for p in range(len(prime\_str) - 1): output,states = model(prime\_input[p], hidden, cell) if use\_cuda: hidden, cell = states[0].cuda(), states[1].cuda() else: hidden, cell = states[0], states[1] inp = prime\_input[-1] for p in range(predict\_len): output, states = model(inp, hidden, cell) if use\_cuda: output = output.cuda() hidden, cell = states[0].cuda(), states[1].cuda() else: hidden, cell = states[0], states[1] # sample from the network as a multinomial distribution output\_dist = output.data.view(-1).div(temperature).exp() top\_i = torch.multinomial(output\_dist, 1)[0] # add predicted character to string and use as next input # predicted\_char = all\_characters[top\_i] predicted\_char = to\_character[top\_i.item()] predicted += predicted\_char inp = char\_tensor(predicted\_char) return predicted# mainif \_\_name\_\_ == "\_\_main\_\_": n\_epochs = 100 print\_every = 100 plot\_every = 100 hidden\_size = 200 n\_layers = 2 lr = 0.001 bi = True # define model model = TextGenerate(n\_characters, hidden\_size, n\_characters, n\_layers, bi) if use\_cuda: model = model.cuda() model\_optimizer = torch.optim.Adam(model.parameters(), lr=lr) criterion = nn.CrossEntropyLoss() # train the model start = time.time() all\_losses = [] all\_perplexities = [] loss\_avg = 0. perplexity\_avg = 0. for epoch in range(1, n\_epochs + 1): loss, perplexity = train(\*random\_training\_set(chunk\_len)) loss\_avg += loss perplexity\_avg += perplexity if epoch % print\_every == 0: print('[%s taken (%d epochs %d%% trained) Loss: %.4f Perplexity: %.4f]' % (time\_since(start), epoch, epoch / n\_epochs \* 100, loss, perplexity)) if epoch % plot\_every == 0: all\_losses.append(loss\_avg / plot\_every) all\_perplexities.append(perplexity\_avg / plot\_every) loss\_avg = 0. perplexity\_avg = 0. print("epoch:", epoch) plt.figure() plt.title("loss") plt.plot(all\_losses) plt.show() plt.figure() plt.title("perplexities") plt.plot(all\_perplexities) plt.show() # evaluation chunk = random\_chunk(500) prime\_str, target\_str = chunk[:251], chunk[251:] gen\_text, loss, perplexity = evaluate(target\_str, prime\_str, 250, temperature=0.8) print("\nLoss: ", loss, " Perplexity:" , perplexity, "\n") print("\n", gen\_text, "\n") # training evaluation # Pride and Prejudice - Jane Austen print(generate("左子穆脸色微微发紫，若不是大敌在外，早已发作，当强忍怒气，道：“不吃！”段誉插口道：“你这是什么瓜子？桂花？玫瑰？还是松子味的？”那少女道：“啊哟！瓜子还有许多讲究么？我可不知道了。", 300, temperature=0.8))