实验三 自动写诗

一、 实验目的

1. 理解和掌握循环神经网络概念及在深度学习框架中的实现。
2. 掌握使用深度学习框架进行文本生成任务的基本流程：如数据读取、构造网络、训练和预测等。

二、 实验要求

1. 基于 Python 语言和任意一种深度学习框架（实验指导书中使用 Pytorch 框架进行介绍），完成数据读取、网络设计、网络构建、模型训练和模型测试等过程，最终实现一个可以自动写诗的程序。网络结构设计要有自己的方案，不能与实验指导书完全相同。
2. 随意给出首句，如给定“湖光秋月两相和”，输出模型续写的诗句。也可以根据自己的兴趣，进一步实现写藏头诗（不做要求）。要求输出的诗句尽可能地满足汉语语法和表达习惯。实验提供预处理后的唐诗数据集，包含 57580 首唐诗（在课程网站下载），也可以使用其他唐诗数据集。
3. 按规定时间在课程网站提交实验报告、代码。

三、 实验原理

实验使用到的网络结构主要有 Embedding、LSTM 以及全连接层。在 pytorch 中分别定义如下：

1. Class torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2.0, scale_grad_by_freq=False, sparse=False, _weight=None)

参数介绍：

num_embeddings: 词汇表的大小。

embedding_dim: 每个嵌入向量的维度。

padding_idx: 如果提供的话，输出遇到此下标时用零填充。

max_norm: 如果提供的话，会对词嵌入进行归一化，使它们的范数小于提供的值。

norm_type: 对于 max_norm 选项计算 p 范数时的 p。

scale_grad_by_freq: 如果提供的话，会根据 mini-batch 中单词频率缩放梯度。

2. Class torch.nn.LSTM(input_size, hidden_size, num_layers=1, bias=True, batch_first=False, dropout=0, bidirectional=False)

参数介绍：

input_size: 输入的特征数目。

hidden_size: 隐状态的特征数目。

num_layers: LSTM 的层数。

bias:是否使用偏置。

batch_first: 是否将 batch 放在第一维。如果为 True，那么输入输出的 Tensor 形式都是(batch, seq_len, num_features)。

dropout: 如果非零，则除了最后一层，每层的输出都加个 Dropout 层。

bidirectional: 是否使用双向 LSTM。

输入：(注: 以下均考虑 batch_first 为 False)

intput, 形式为(seq_len, batch, input_size)。

h_0, 形式为(num_layers * num_directions, batch, hidden_size)。

c_0, 形式为(num_layers * num_directions, batch, hidden_size)。

输出：

output，形式为(seq_len, batch, num_directions * hidden_size)。

h_n, 形式为(num_layers * num_directions, batch, hidden_size)。

c_n, 形式为(num_layers * num_directions, batch, hidden_size)。

3. Class torch.nn.Linear(in_features, out_features, bias=True)。

参数：

in_features: 每个输入样本的大小。

out_features: 每个输出样本的大小。

bias: 默认值 True，若设置为 False，这层不会学习偏置。

四、 实验所用数据集及工具

实验提供预处理过的数据集，含有 57580 首唐诗，每首诗限定在 125 词，不足 125 词的以</s>填充。数据集以 npz 文件形式保存，包含三个部分：

（1） data: 诗词数据，将诗词中的字转化为其在字典中的序号表示。

（2） ix2word: 序号到字的映射

（3） word2ix: 字到序号的映射

除了此数据集，也可自由选择其他唐诗数据集进行训练，根据需要自行预处理，如：https://github.com/chinese-poetry/chinese-poetry。

五、实验步骤与方法

实验可简单地划分为数据准备、模型构建、模型训练和模型预测四个步骤。

1. 数据准备

```
def prepareData():
    datas = np.load("tang.npz")
    data = datas['data']
    ix2word = datas['ix2word'].item()
    word2ix = datas['word2ix'].item()
    data = torch.from_numpy(data)
    dataloader = DataLoader(data,
                            batch_size=16,
                            shuffle=True,
                            num_workers=2)
    return dataloader, ix2word, word2ix
```

2. 模型构建

模型构建时，需要创建一个继承自 nn.Module 的类，在这个类的_init_( )中定义网络结构，在 forward 中定义前向传播过程。

示例中的模型相当简单，只有 Embedding、单层 LSTM、全连接层三个网络层，同学们可以自行进行拓展，以获得更好的效果。

示例：

```
class PoetryModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(PoetryModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, self.hidden_dim, num_layers=1,
        batch_first=True)
        self.linear = nn.Linear(self.hidden_dim, vocab_size)

    def forward(self, input, hidden=None):
        embeds = self.embeddings(input)    # [batch, seq_len] => [batch, seq_len,
embed_dim]
        batch_size, seq_len = input.size()
        if hidden is None:
            h_0 = input.data.new(1, batch_size, self.hidden_dim).fill_(0).float()
            c_0 = input.data.new(1, batch_size, self.hidden_dim).fill_(0).float()
        else:
            h_0, c_0 = hidden
        output, hidden = self.lstm(embeds, (h_0, c_0))
        output = self.linear(output)
        output = output.reshape(batch_size * seq_len, -1)
        return output, hidden
```

3. 模型训练

模型训练包括定义模型、设置优化器和损失函数、获取模型输出、计算误差、误差反向传播等步骤。

```
def train(dataloader, word2ix):
    # 定义模型
```

```
    model = PoetryModel(len(word2ix),
                        embedding_dim=Config.embedding_dim,
                        hidden_dim=Config.hidden_dim)
    optimizer = optim.Adam(model.parameters(), lr=Config.lr)
    criterion = nn.CrossEntropyLoss()
    loss_meter = meter.AverageValueMeter()
```

4. 模型预测

```
def generate(model, start_words, ix2word, word2ix):
    results = list(start_words)
    start_words_len = len(start_words)
    # 第一个词语是<START>
    input = t.Tensor([word2ix['<START>']]).view(1, 1).long()
    hidden = None
    model.eval()
    with torch.no_grad():
        for i in range(Config.max_gen_len):
            output, hidden = model(input, hidden)
            # 如果在给定的句首中，input 为句首中的下一个字
            if i < start_words_len:
                w = results[i]
                input = input.data.new([word2ix[w]]).view(1, 1)
            # 否则将 output 作为下一个 input 进行
            else:
                top_index = output.data[0].topk(1)[1][0].item()
                w = ix2word[top_index]
                results.append(w)
                input = input.data.new([top_index]).view(1, 1)
```

```
            if w == '<EOP>':
                del results[-1]
                break
    return results
```