In [1]:
```
import torch
import torch.nn as nn
import numpy as np
import time
```

# RNN

## 定义本例中的RNN模型

In [2]:
```
class my_RNN(nn.Module):
    def __init__(self, word_num, embedding_size, hidden_size, output_size, num_layers=1): # 初始化
        super().__init__()
        self.embedding = nn.Embedding(word_num, embedding_size) # 将输入的每个时间步长下的数据映射到embedding_size的空间
        self.rnn = nn.RNN(embedding_size, hidden_size, num_layers, batch_first=True) # rnn层
        self.fc = nn.Linear(hidden_size, output_size) # 全连接层
        self.num_layers = num_layers
        self.hidden_size = hidden_size

    def forward(self, x, hidden_vector):
        # 输入X的初始形状为[batch_size, num_steps]
        x = self.embedding(x) # 输入形状变为[batch_size, num_steps, embedding_size]
        output, hidden_vector = self.rnn(x, hidden_vector) # output形状为[batch_size, num_steps, hidden_size]
        # hidden_vector形状为[batch_size, hidden_size]
        # 取出最后一个时刻的预测
        output = output[:, -1, :] # [batch_size, hidden_size]
        output = self.fc(output) # [batch_size, output_size]
        return output, hidden_vector

    def init_h(self, batch_size):
        # 初始h0为[num_layers, batch_size, hidden_size]的零向量
        return torch.zeros(self.num_layers, batch_size, self.hidden_size)
```

## 读取数据

In [3]:
```
f = open('../Pytorch_Book_ZhouRUC/dataset/poems_clean.txt', "r", encoding='utf-8')
poems = []
for line in f.readlines():
    title, poem = line.split(":") # 将题目和诗句分开
    poem = poem.replace(" ", "") # 将诗句连接在一起
    poem = poem.replace("\n", "") # 将换行符去掉
    if len(poem) > 0:
        poems.append(list(poem)) # 转化为存储一个个字符的列表保存起来
```

## 构造字符的字典

In [4]:
```
word_dict = {}
idx = 1
for poem in poems:
    for char in poem:
        if word_dict.get(char) == None:
            word_dict[char] = idx
            idx += 1
```

## 构造一个诗句到字符编号的映射

In [5]:
```
poems_idx = []
for poem in poems:
    poem_idx = []
    for char in poem:
        poem_idx.append(word_dict[char])
    poems_idx.append(poem_idx)
```

## 构造数据和标签

In [6]:
```
max_len = 50
X = []
Y = []
for poem_idx in poems_idx:
    idx_for_input = poem_idx[: -1]
    idx_for_pred = poem_idx[-1] # 标签为一首诗的最后一个字的编码
    Y.append(idx_for_pred)
    idx_for_input = idx_for_input + [0] * (max_len - len(idx_for_input)) # 数据为最后一个字之前的内容
    X.append(idx_for_input)
```

## 划分训练集，测试集

In [7]:
```
idx = np.random.permutation(range(len(X)))
X = [X[i] for i in idx]
Y = [Y[i] for i in idx]

# 1/5为测试集，4/5为训练集
X_test = X[:len(X) // 5]
X_test = torch.IntTensor(np.array(X_test, dtype = int))
X_train = X[len(X) // 5:]
X_train = torch.IntTensor(np.array(X_train, dtype = int))
Y_test = Y[:len(Y) // 5]
Y_test = torch.IntTensor(np.array(Y_test, dtype = int))
Y_train = Y[len(Y) // 5:]
Y_train = torch.IntTensor(np.array(Y_train, dtype = int))
```

## 构造训练集和验证集的Dataloader

In [8]:
```
batch_size = 64

train_dataset = torch.utils.data.TensorDataset(X_train, Y_train)
test_dataset = torch.utils.data.TensorDataset(X_test, Y_test)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size = batch_size, shuffle=False)
```

## 设定模型实例和超参数

In [64]:
```
word_num = output_size = len(word_dict) + 1
embedding_size = 64
hidden_size = 256
my_model = my_RNN(word_num, embedding_size, hidden_size, output_size).cuda() # 生成rnn模型的一个实例
lr = 1e-3
optimizer = torch.optim.Adam(my_model.parameters(), lr = lr) # 设定优化器
# 设定学习率的scheduler，每过50个epoch将学习率减半
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.5)
```

## 训练、测试模型并打印实验结果

In [33]:
```
def accuracy(outputs, labels):
    preds = torch.max(outputs, dim=1)[1]
    return torch.sum(preds == labels).item() / len(labels)

def test(model, test_loader):
    model.eval()
    test_loss = 0
    test_acc = 0
    for batch, data in enumerate(test_loader):
        x, y = data[0], data[1]
        h0 = model.init_h(len(x))
        x, y, h0 = x.cuda(), y.cuda(), h0.cuda()
        output, hidden_vector = model(x, h0)
        y = y.long()
        loss = nn.CrossEntropyLoss()(output, y)
        acc = accuracy(output, y)
        test_loss += loss.item()
        test_acc += acc

    test_loss /= len(test_loader)
    test_acc /= len(test_loader)
    return test_loss, test_acc

def printlog(epoch, train_time, train_loss, train_acc, test_loss, test_acc, epochs=50):
    print(f"Epoch [{epoch}/{epochs}], time: {train_time:.2f}s, loss: {train_loss:.4f}, acc: {train_acc:.4f}, test_loss: {tes
```

In [70]:
```
def train(model, optimizer, scheduler, train_loader, test_loader, epochs=50):
    test_acc_list = []
    for epoch in range(epochs):
        model.train()
        train_loss = 0
        train_acc = 0
        start = time.time()
        for batch, data in enumerate(train_loader):
            optimizer.zero_grad()
            x, y = data[0], data[1]
            h0 = model.init_h(len(x))
            x, y, h0 = x.cuda(), y.cuda(), h0.cuda()
            output, hidden_vector = model(x, h0)
            y = y.long()
            loss = nn.CrossEntropyLoss()(output, y)
            acc = accuracy(output, y)
            train_loss += loss.item()
            train_acc += acc
            loss.backward()
            optimizer.step()
```

```
        scheduler.step()
        end = time.time()
        duration = end - start
        test_loss, test_acc = test(model, test_loader)
        train_loss /= len(train_loader)
        train_acc /= len(train_loader)
        test_acc_list.append(test_acc)
        printlog(epoch+1, duration, train_loss, train_acc, test_loss, test_acc, epochs)
    return test_acc_list
```

In [71]:
```
result = train(my_model, optimizer, scheduler, train_loader, test_loader, epochs=100)
```

```
Epoch [1/100], time: 2.04s, loss: 0.0403, acc: 0.9994, test_loss: 13.1785, test_acc: 0.0837
Epoch [2/100], time: 1.90s, loss: 0.0381, acc: 0.9996, test_loss: 13.1955, test_acc: 0.0837
Epoch [3/100], time: 1.93s, loss: 0.0369, acc: 0.9995, test_loss: 13.2059, test_acc: 0.0820
Epoch [4/100], time: 2.03s, loss: 0.0360, acc: 0.9995, test_loss: 13.2158, test_acc: 0.0826
Epoch [5/100], time: 1.93s, loss: 0.0351, acc: 0.9996, test_loss: 13.2384, test_acc: 0.0833
Epoch [6/100], time: 1.92s, loss: 0.0349, acc: 0.9997, test_loss: 13.2460, test_acc: 0.0826
Epoch [7/100], time: 1.91s, loss: 0.0340, acc: 0.9996, test_loss: 13.2687, test_acc: 0.0820
Epoch [8/100], time: 1.94s, loss: 0.0358, acc: 0.9996, test_loss: 13.2865, test_acc: 0.0818
Epoch [9/100], time: 1.93s, loss: 0.0363, acc: 0.9993, test_loss: 13.2928, test_acc: 0.0818
Epoch [10/100], time: 1.93s, loss: 0.0361, acc: 0.9993, test_loss: 13.3064, test_acc: 0.0824
Epoch [11/100], time: 1.98s, loss: 0.0370, acc: 0.9993, test_loss: 13.3432, test_acc: 0.0822
Epoch [12/100], time: 2.00s, loss: 0.0436, acc: 0.9987, test_loss: 13.3272, test_acc: 0.0826
Epoch [13/100], time: 1.90s, loss: 0.0441, acc: 0.9985, test_loss: 13.3465, test_acc: 0.0833
Epoch [14/100], time: 1.95s, loss: 0.0441, acc: 0.9989, test_loss: 13.3604, test_acc: 0.0822
Epoch [15/100], time: 1.93s, loss: 0.0340, acc: 0.9995, test_loss: 13.3615, test_acc: 0.0826
Epoch [16/100], time: 2.04s, loss: 0.0331, acc: 0.9993, test_loss: 13.3708, test_acc: 0.0835
Epoch [17/100], time: 1.90s, loss: 0.0306, acc: 0.9995, test_loss: 13.3862, test_acc: 0.0833
Epoch [18/100], time: 1.91s, loss: 0.0335, acc: 0.9996, test_loss: 13.4023, test_acc: 0.0824
Epoch [19/100], time: 1.99s, loss: 0.0305, acc: 0.9993, test_loss: 13.4022, test_acc: 0.0812
Epoch [20/100], time: 1.93s, loss: 0.0285, acc: 0.9994, test_loss: 13.4350, test_acc: 0.0800
Epoch [21/100], time: 1.96s, loss: 0.0274, acc: 0.9996, test_loss: 13.4405, test_acc: 0.0818
Epoch [22/100], time: 1.95s, loss: 0.0270, acc: 0.9995, test_loss: 13.4620, test_acc: 0.0820
Epoch [23/100], time: 1.95s, loss: 0.0280, acc: 0.9996, test_loss: 13.4717, test_acc: 0.0843
Epoch [24/100], time: 1.95s, loss: 0.0467, acc: 0.9981, test_loss: 13.5061, test_acc: 0.0839
Epoch [25/100], time: 1.91s, loss: 0.0413, acc: 0.9982, test_loss: 13.5013, test_acc: 0.0835
Epoch [26/100], time: 2.02s, loss: 0.0322, acc: 0.9991, test_loss: 13.5099, test_acc: 0.0824
Epoch [27/100], time: 1.92s, loss: 0.0289, acc: 0.9992, test_loss: 13.5176, test_acc: 0.0831
Epoch [28/100], time: 1.90s, loss: 0.0259, acc: 0.9996, test_loss: 13.5325, test_acc: 0.0824
Epoch [29/100], time: 1.90s, loss: 0.0245, acc: 0.9995, test_loss: 13.5429, test_acc: 0.0824
Epoch [30/100], time: 1.99s, loss: 0.0236, acc: 0.9996, test_loss: 13.5556, test_acc: 0.0824
Epoch [31/100], time: 1.93s, loss: 0.0237, acc: 0.9996, test_loss: 13.5691, test_acc: 0.0824
Epoch [32/100], time: 1.93s, loss: 0.0230, acc: 0.9996, test_loss: 13.5761, test_acc: 0.0816
Epoch [33/100], time: 1.96s, loss: 0.0246, acc: 0.9995, test_loss: 13.6416, test_acc: 0.0822
Epoch [34/100], time: 1.97s, loss: 0.0556, acc: 0.9961, test_loss: 13.5980, test_acc: 0.0826
Epoch [35/100], time: 2.22s, loss: 0.0476, acc: 0.9964, test_loss: 13.6218, test_acc: 0.0841
Epoch [36/100], time: 2.01s, loss: 0.0403, acc: 0.9981, test_loss: 13.5924, test_acc: 0.0855
Epoch [37/100], time: 1.94s, loss: 0.0315, acc: 0.9990, test_loss: 13.6177, test_acc: 0.0845
Epoch [38/100], time: 1.94s, loss: 0.0265, acc: 0.9993, test_loss: 13.6117, test_acc: 0.0847
Epoch [39/100], time: 1.94s, loss: 0.0248, acc: 0.9993, test_loss: 13.6358, test_acc: 0.0855
Epoch [40/100], time: 1.95s, loss: 0.0227, acc: 0.9996, test_loss: 13.6409, test_acc: 0.0859
Epoch [41/100], time: 1.99s, loss: 0.0231, acc: 0.9994, test_loss: 13.6669, test_acc: 0.0849
Epoch [42/100], time: 1.93s, loss: 0.0240, acc: 0.9995, test_loss: 13.6540, test_acc: 0.0845
Epoch [43/100], time: 1.97s, loss: 0.0232, acc: 0.9995, test_loss: 13.6791, test_acc: 0.0843
Epoch [44/100], time: 2.02s, loss: 0.0233, acc: 0.9994, test_loss: 13.7036, test_acc: 0.0847
Epoch [45/100], time: 1.94s, loss: 0.0295, acc: 0.9993, test_loss: 13.6838, test_acc: 0.0839
Epoch [46/100], time: 1.94s, loss: 0.0312, acc: 0.9986, test_loss: 13.7156, test_acc: 0.0814
Epoch [47/100], time: 1.99s, loss: 0.0337, acc: 0.9986, test_loss: 13.7424, test_acc: 0.0855
Epoch [48/100], time: 2.09s, loss: 0.0347, acc: 0.9987, test_loss: 13.7602, test_acc: 0.0833
Epoch [49/100], time: 2.00s, loss: 0.0294, acc: 0.9989, test_loss: 13.7739, test_acc: 0.0845
Epoch [50/100], time: 1.95s, loss: 0.0299, acc: 0.9988, test_loss: 13.7851, test_acc: 0.0845
Epoch [51/100], time: 1.95s, loss: 0.0259, acc: 0.9990, test_loss: 13.8000, test_acc: 0.0835
Epoch [52/100], time: 2.03s, loss: 0.0222, acc: 0.9994, test_loss: 13.7970, test_acc: 0.0847
Epoch [53/100], time: 2.03s, loss: 0.0206, acc: 0.9994, test_loss: 13.8027, test_acc: 0.0839
Epoch [54/100], time: 2.13s, loss: 0.0202, acc: 0.9995, test_loss: 13.8120, test_acc: 0.0839
Epoch [55/100], time: 2.12s, loss: 0.0202, acc: 0.9995, test_loss: 13.8170, test_acc: 0.0845
Epoch [56/100], time: 1.94s, loss: 0.0191, acc: 0.9995, test_loss: 13.8204, test_acc: 0.0835
Epoch [57/100], time: 1.96s, loss: 0.0190, acc: 0.9996, test_loss: 13.8298, test_acc: 0.0847
Epoch [58/100], time: 1.96s, loss: 0.0202, acc: 0.9995, test_loss: 13.8317, test_acc: 0.0833
Epoch [59/100], time: 1.93s, loss: 0.0199, acc: 0.9995, test_loss: 13.8502, test_acc: 0.0843
Epoch [60/100], time: 1.92s, loss: 0.0182, acc: 0.9996, test_loss: 13.8531, test_acc: 0.0839
Epoch [61/100], time: 1.93s, loss: 0.0174, acc: 0.9996, test_loss: 13.8672, test_acc: 0.0843
Epoch [62/100], time: 2.00s, loss: 0.0170, acc: 0.9996, test_loss: 13.8706, test_acc: 0.0839
Epoch [63/100], time: 2.04s, loss: 0.0168, acc: 0.9996, test_loss: 13.8818, test_acc: 0.0847
Epoch [64/100], time: 2.01s, loss: 0.0165, acc: 0.9996, test_loss: 13.8892, test_acc: 0.0843
Epoch [65/100], time: 1.93s, loss: 0.0168, acc: 0.9996, test_loss: 13.8957, test_acc: 0.0847
Epoch [66/100], time: 1.95s, loss: 0.0166, acc: 0.9995, test_loss: 13.9082, test_acc: 0.0837
Epoch [67/100], time: 1.97s, loss: 0.0163, acc: 0.9996, test_loss: 13.9131, test_acc: 0.0843
Epoch [68/100], time: 1.99s, loss: 0.0159, acc: 0.9997, test_loss: 13.9168, test_acc: 0.0845
Epoch [69/100], time: 2.02s, loss: 0.0166, acc: 0.9996, test_loss: 13.9298, test_acc: 0.0841
Epoch [70/100], time: 2.05s, loss: 0.0162, acc: 0.9997, test_loss: 13.9294, test_acc: 0.0835
Epoch [71/100], time: 2.10s, loss: 0.0165, acc: 0.9996, test_loss: 13.9325, test_acc: 0.0835
Epoch [72/100], time: 2.09s, loss: 0.0173, acc: 0.9995, test_loss: 13.9546, test_acc: 0.0837
Epoch [73/100], time: 2.14s, loss: 0.0164, acc: 0.9997, test_loss: 13.9659, test_acc: 0.0833
Epoch [74/100], time: 1.91s, loss: 0.0154, acc: 0.9996, test_loss: 13.9808, test_acc: 0.0841
Epoch [75/100], time: 2.07s, loss: 0.0155, acc: 0.9996, test_loss: 13.9814, test_acc: 0.0831
Epoch [76/100], time: 2.19s, loss: 0.0151, acc: 0.9995, test_loss: 13.9867, test_acc: 0.0833
Epoch [77/100], time: 2.09s, loss: 0.0153, acc: 0.9996, test_loss: 13.9984, test_acc: 0.0837
Epoch [78/100], time: 1.95s, loss: 0.0149, acc: 0.9996, test_loss: 14.0006, test_acc: 0.0835
Epoch [79/100], time: 1.90s, loss: 0.0149, acc: 0.9996, test_loss: 14.0149, test_acc: 0.0835
Epoch [80/100], time: 1.93s, loss: 0.0141, acc: 0.9996, test_loss: 14.0298, test_acc: 0.0835
Epoch [81/100], time: 1.92s, loss: 0.0137, acc: 0.9996, test_loss: 14.0305, test_acc: 0.0816
Epoch [82/100], time: 1.92s, loss: 0.0135, acc: 0.9996, test_loss: 14.0478, test_acc: 0.0826
Epoch [83/100], time: 1.95s, loss: 0.0135, acc: 0.9996, test_loss: 14.0571, test_acc: 0.0826
Epoch [84/100], time: 2.05s, loss: 0.0141, acc: 0.9995, test_loss: 14.0669, test_acc: 0.0833
Epoch [85/100], time: 1.96s, loss: 0.0146, acc: 0.9997, test_loss: 14.0584, test_acc: 0.0822
Epoch [86/100], time: 1.92s, loss: 0.0154, acc: 0.9995, test_loss: 14.0723, test_acc: 0.0831
Epoch [87/100], time: 1.95s, loss: 0.0162, acc: 0.9994, test_loss: 14.0830, test_acc: 0.0798
Epoch [88/100], time: 1.97s, loss: 0.0163, acc: 0.9995, test_loss: 14.0965, test_acc: 0.0816
Epoch [89/100], time: 2.01s, loss: 0.0145, acc: 0.9995, test_loss: 14.0994, test_acc: 0.0835
```

```
Epoch [90/100], time: 2.02s, loss: 0.0135, acc: 0.9995, test_loss: 14.1039, test_acc: 0.0824
Epoch [91/100], time: 2.13s, loss: 0.0141, acc: 0.9996, test_loss: 14.1159, test_acc: 0.0822
Epoch [92/100], time: 1.98s, loss: 0.0135, acc: 0.9995, test_loss: 14.1184, test_acc: 0.0820
Epoch [93/100], time: 2.20s, loss: 0.0125, acc: 0.9995, test_loss: 14.1297, test_acc: 0.0824
Epoch [94/100], time: 2.03s, loss: 0.0120, acc: 0.9997, test_loss: 14.1396, test_acc: 0.0824
Epoch [95/100], time: 2.07s, loss: 0.0118, acc: 0.9997, test_loss: 14.1479, test_acc: 0.0824
Epoch [96/100], time: 2.00s, loss: 0.0116, acc: 0.9997, test_loss: 14.1543, test_acc: 0.0824
Epoch [97/100], time: 2.02s, loss: 0.0115, acc: 0.9997, test_loss: 14.1694, test_acc: 0.0824
Epoch [98/100], time: 2.11s, loss: 0.0114, acc: 0.9996, test_loss: 14.1796, test_acc: 0.0818
Epoch [99/100], time: 2.06s, loss: 0.0113, acc: 0.9997, test_loss: 14.1931, test_acc: 0.0820
Epoch [100/100], time: 1.98s, loss: 0.0113, acc: 0.9997, test_loss: 14.2003, test_acc: 0.0824
```

### 最大验证集预测精度

In [72]:
```
max(result)
```

Out[72]:
```
0.0859375
```

### 预测，生成藏头诗

In [100…
```
target = "助****教****老****师****都****辛****苦****了****"
generated_index = []
generated_txt = ""
for char in target:
    if char != "*":
        idx = word_dict[char]
        generated_index.append(idx)
        generated_txt += char
    else:
        inp = generated_index + [0] * (max_len - 1 - len(generated_index))
        inp = torch.IntTensor(np.array(inp, dtype=int)) # [49]
        inp = inp.unsqueeze(0) # [1, 49]
        h0 = my_model.init_h(1) # [1, 1, 256]
        inp, h0 = inp.cuda(), h0.cuda()
        pred, h = my_model(inp, h0) # pred shape [1, 5546]
        pred_idx = torch.max(pred, dim=1)[1].item() # 提取预测的最大值的index
        pred_char = [k for k, v in word_dict.items() if v == pred_idx][0] # 提取对应index的字
        generated_index.append(pred_idx)
        generated_txt += pred_char
```

In [106…
```
for i in range(0, len(generated_txt), 5):
    print(generated_txt[i:i+5])
```

```
助年格绿狗
教知阕也栽
老录右式天
师军岩宿行
都归志津为
辛公扬成携
苦邕人身何
了归庐知知
```

# LSTM

### 定义本例中的LSTM模型

In [111…
```
class my_LSTM(nn.Module):
    def __init__(self, word_num, embedding_size, hidden_size, output_size, num_layers=1):
        super().__init__()
        self.embedding = nn.Embedding(word_num, embedding_size)
        self.lstm = nn.LSTM(embedding_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.num_layers = num_layers
        self.hidden_size = hidden_size

    def forward(self, x, hidden):
        x = self.embedding(x) # --> [batch_size, num_steps, embedding_size]
        output, h = self.lstm(x, hidden) # output: [batch_size, num_steps, hidden_size]
        output = output[:, -1, :] # --> [batch_size, hidden_size]
        output = self.fc(output) # --> [batch_size, output_size]
        return output, h

    def init_hc(self, batch_size):
        h0 = torch.zeros(self.num_layers, batch_size, self.hidden_size)
        c0 = torch.zeros(self.num_layers, batch_size, self.hidden_size)
        return h0, c0
```

### 设定模型实例

In [108…
```
my_lstm_model = my_LSTM(word_num, embedding_size, hidden_size, output_size).cuda()
optimizer1 = torch.optim.Adam(my_lstm_model.parameters(), lr=lr)
scheduler1 = torch.optim.lr_scheduler.StepLR(optimizer1, step_size=50, gamma=0.5)
```

## 训练与验证并展示实验结果

In [117…

```
def test1(model, test_loader):
    model.eval()
    test_loss = 0
    test_acc = 0
    for batch, data in enumerate(test_loader):
        x, y = data[0], data[1]
        h0, c0 = model.init_hc(len(x))
        x, y, h0, c0 = x.cuda(), y.cuda(), h0.cuda(), c0.cuda()
        output, (hn, cn) = model(x, (h0, c0))
        y = y.long()
        loss = nn.CrossEntropyLoss()(output, y)
        acc = accuracy(output, y)
        test_loss += loss.item()
        test_acc += acc

    test_loss /= len(test_loader)
    test_acc /= len(test_loader)
    return test_loss, test_acc
```

In [115…

```
def train1(model, optimizer, scheduler, train_loader, test_loader, epochs=50):
    test_acc_list = []
    for epoch in range(epochs):
        model.train()
        train_loss = 0
        train_acc = 0
        start = time.time()
        for batch, data in enumerate(train_loader):
            optimizer.zero_grad()
            x, y = data[0], data[1]
            h0, c0 = model.init_hc(len(x))
            x, y, h0, c0 = x.cuda(), y.cuda(), h0.cuda(), c0.cuda()
            output, (hn, cn) = model(x, (h0, c0))
            y = y.long()
            loss = nn.CrossEntropyLoss()(output, y)
            acc = accuracy(output, y)
            train_loss += loss.item()
            train_acc += acc
            loss.backward()
            optimizer.step()

        scheduler.step()
        end = time.time()
        duration = end - start
        test_loss, test_acc = test1(model, test_loader)
        train_loss /= len(train_loader)
        train_acc /= len(train_loader)
        test_acc_list.append(test_acc)
        printlog(epoch+1, duration, train_loss, train_acc, test_loss, test_acc, epochs)
    return test_acc_list
```

In [118…

```
result1 = train1(my_lstm_model, optimizer1, scheduler1, train_loader, test_loader, epochs=100)
```

```
Epoch [1/100], time: 3.73s, loss: 6.3538, acc: 0.0324, test_loss: 6.4176, test_acc: 0.0354
Epoch [2/100], time: 3.64s, loss: 6.3417, acc: 0.0323, test_loss: 6.4345, test_acc: 0.0354
Epoch [3/100], time: 3.60s, loss: 6.3324, acc: 0.0323, test_loss: 6.4336, test_acc: 0.0354
Epoch [4/100], time: 3.56s, loss: 6.3284, acc: 0.0324, test_loss: 6.4262, test_acc: 0.0354
Epoch [5/100], time: 3.57s, loss: 6.2985, acc: 0.0323, test_loss: 6.4065, test_acc: 0.0331
Epoch [6/100], time: 3.72s, loss: 6.2592, acc: 0.0328, test_loss: 6.3836, test_acc: 0.0340
Epoch [7/100], time: 3.62s, loss: 6.2210, acc: 0.0339, test_loss: 6.3794, test_acc: 0.0370
Epoch [8/100], time: 3.60s, loss: 6.1705, acc: 0.0345, test_loss: 6.3525, test_acc: 0.0360
Epoch [9/100], time: 3.63s, loss: 6.1046, acc: 0.0355, test_loss: 6.3386, test_acc: 0.0401
Epoch [10/100], time: 3.68s, loss: 6.0345, acc: 0.0379, test_loss: 6.3171, test_acc: 0.0403
Epoch [11/100], time: 3.61s, loss: 5.9592, acc: 0.0407, test_loss: 6.3108, test_acc: 0.0401
Epoch [12/100], time: 3.60s, loss: 5.8700, acc: 0.0437, test_loss: 6.2906, test_acc: 0.0409
Epoch [13/100], time: 3.60s, loss: 5.7758, acc: 0.0477, test_loss: 6.2998, test_acc: 0.0420
Epoch [14/100], time: 3.65s, loss: 5.6554, acc: 0.0523, test_loss: 6.3094, test_acc: 0.0438
Epoch [15/100], time: 3.61s, loss: 5.5262, acc: 0.0567, test_loss: 6.3034, test_acc: 0.0465
Epoch [16/100], time: 3.64s, loss: 5.3774, acc: 0.0650, test_loss: 6.3828, test_acc: 0.0436
Epoch [17/100], time: 3.71s, loss: 5.2197, acc: 0.0751, test_loss: 6.3487, test_acc: 0.0508
Epoch [18/100], time: 3.72s, loss: 5.0520, acc: 0.0857, test_loss: 6.4420, test_acc: 0.0457
Epoch [19/100], time: 3.63s, loss: 4.8788, acc: 0.0977, test_loss: 6.4341, test_acc: 0.0594
Epoch [20/100], time: 3.63s, loss: 4.6917, acc: 0.1113, test_loss: 6.4812, test_acc: 0.0598
Epoch [21/100], time: 3.59s, loss: 4.5056, acc: 0.1269, test_loss: 6.5925, test_acc: 0.0574
Epoch [22/100], time: 3.61s, loss: 4.3076, acc: 0.1480, test_loss: 6.6928, test_acc: 0.0584
Epoch [23/100], time: 3.59s, loss: 4.1319, acc: 0.1657, test_loss: 6.7362, test_acc: 0.0633
Epoch [24/100], time: 3.54s, loss: 3.9283, acc: 0.1864, test_loss: 6.8496, test_acc: 0.0611
Epoch [25/100], time: 3.59s, loss: 3.7476, acc: 0.2144, test_loss: 6.9000, test_acc: 0.0691
Epoch [26/100], time: 3.61s, loss: 3.5529, acc: 0.2403, test_loss: 7.0090, test_acc: 0.0703
Epoch [27/100], time: 3.60s, loss: 3.3758, acc: 0.2679, test_loss: 7.0851, test_acc: 0.0730
Epoch [28/100], time: 3.56s, loss: 3.1846, acc: 0.3023, test_loss: 7.2129, test_acc: 0.0755
Epoch [29/100], time: 3.56s, loss: 2.9976, acc: 0.3337, test_loss: 7.3397, test_acc: 0.0788
Epoch [30/100], time: 3.68s, loss: 2.8050, acc: 0.3721, test_loss: 7.4354, test_acc: 0.0839
Epoch [31/100], time: 3.54s, loss: 2.6487, acc: 0.4004, test_loss: 7.5577, test_acc: 0.0806
Epoch [32/100], time: 3.55s, loss: 2.4778, acc: 0.4367, test_loss: 7.6453, test_acc: 0.0874
Epoch [33/100], time: 3.55s, loss: 2.3269, acc: 0.4716, test_loss: 7.7479, test_acc: 0.0888
Epoch [34/100], time: 3.59s, loss: 2.1766, acc: 0.5027, test_loss: 7.8357, test_acc: 0.0925
Epoch [35/100], time: 3.60s, loss: 2.0479, acc: 0.5325, test_loss: 7.9064, test_acc: 0.0925
Epoch [36/100], time: 3.61s, loss: 1.9085, acc: 0.5619, test_loss: 8.1171, test_acc: 0.0858
Epoch [37/100], time: 3.56s, loss: 1.7824, acc: 0.5884, test_loss: 8.2046, test_acc: 0.0866
Epoch [38/100], time: 3.71s, loss: 1.6484, acc: 0.6217, test_loss: 8.2350, test_acc: 0.0981
Epoch [39/100], time: 3.58s, loss: 1.5252, acc: 0.6486, test_loss: 8.4103, test_acc: 0.0911
Epoch [40/100], time: 3.61s, loss: 1.4259, acc: 0.6759, test_loss: 8.5469, test_acc: 0.0907
Epoch [41/100], time: 3.59s, loss: 1.3319, acc: 0.6959, test_loss: 8.6097, test_acc: 0.0932
Epoch [42/100], time: 3.66s, loss: 1.2436, acc: 0.7195, test_loss: 8.7579, test_acc: 0.0977
Epoch [43/100], time: 3.56s, loss: 1.1370, acc: 0.7460, test_loss: 8.7550, test_acc: 0.0971
Epoch [44/100], time: 3.55s, loss: 1.0636, acc: 0.7654, test_loss: 8.8992, test_acc: 0.1024
Epoch [45/100], time: 3.57s, loss: 1.0060, acc: 0.7773, test_loss: 9.0377, test_acc: 0.0971
Epoch [46/100], time: 3.63s, loss: 0.8989, acc: 0.8058, test_loss: 9.1264, test_acc: 0.1030
Epoch [47/100], time: 3.58s, loss: 0.8188, acc: 0.8283, test_loss: 9.3062, test_acc: 0.0979
Epoch [48/100], time: 3.62s, loss: 0.7736, acc: 0.8367, test_loss: 9.3710, test_acc: 0.0997
Epoch [49/100], time: 3.57s, loss: 0.7271, acc: 0.8464, test_loss: 9.4624, test_acc: 0.1020
Epoch [50/100], time: 3.72s, loss: 0.5527, acc: 0.8982, test_loss: 9.5026, test_acc: 0.1059
Epoch [51/100], time: 3.64s, loss: 0.4038, acc: 0.9373, test_loss: 9.6256, test_acc: 0.1032
Epoch [52/100], time: 3.60s, loss: 0.3493, acc: 0.9508, test_loss: 9.7049, test_acc: 0.1055
Epoch [53/100], time: 3.66s, loss: 0.2951, acc: 0.9624, test_loss: 9.8158, test_acc: 0.1061
Epoch [54/100], time: 3.63s, loss: 0.2617, acc: 0.9683, test_loss: 9.8975, test_acc: 0.1053
Epoch [55/100], time: 3.61s, loss: 0.2422, acc: 0.9726, test_loss: 9.9779, test_acc: 0.1092
Epoch [56/100], time: 3.57s, loss: 0.2372, acc: 0.9738, test_loss: 10.0541, test_acc: 0.1047
Epoch [57/100], time: 3.59s, loss: 0.2352, acc: 0.9741, test_loss: 10.1198, test_acc: 0.1047
Epoch [58/100], time: 3.50s, loss: 0.2229, acc: 0.9753, test_loss: 10.1811, test_acc: 0.1057
Epoch [59/100], time: 3.61s, loss: 0.2145, acc: 0.9759, test_loss: 10.2799, test_acc: 0.1073
Epoch [60/100], time: 3.65s, loss: 0.1949, acc: 0.9798, test_loss: 10.3446, test_acc: 0.1051
Epoch [61/100], time: 3.58s, loss: 0.2020, acc: 0.9794, test_loss: 10.3998, test_acc: 0.1057
Epoch [62/100], time: 3.60s, loss: 0.1960, acc: 0.9795, test_loss: 10.4716, test_acc: 0.1057
Epoch [63/100], time: 3.60s, loss: 0.1774, acc: 0.9815, test_loss: 10.5458, test_acc: 0.1084
Epoch [64/100], time: 3.57s, loss: 0.1606, acc: 0.9839, test_loss: 10.5917, test_acc: 0.1102
Epoch [65/100], time: 4.01s, loss: 0.1455, acc: 0.9868, test_loss: 10.6606, test_acc: 0.1075
Epoch [66/100], time: 3.95s, loss: 0.1555, acc: 0.9838, test_loss: 10.7568, test_acc: 0.1059
Epoch [67/100], time: 3.64s, loss: 0.1386, acc: 0.9875, test_loss: 10.8005, test_acc: 0.1098
Epoch [68/100], time: 3.58s, loss: 0.1151, acc: 0.9909, test_loss: 10.8801, test_acc: 0.1086
Epoch [69/100], time: 3.76s, loss: 0.1093, acc: 0.9914, test_loss: 10.9084, test_acc: 0.1082
Epoch [70/100], time: 3.63s, loss: 0.1246, acc: 0.9884, test_loss: 11.0367, test_acc: 0.1065
Epoch [71/100], time: 3.58s, loss: 0.1194, acc: 0.9898, test_loss: 11.0651, test_acc: 0.1073
Epoch [72/100], time: 3.50s, loss: 0.0983, acc: 0.9918, test_loss: 11.1664, test_acc: 0.1061
Epoch [73/100], time: 3.67s, loss: 0.0903, acc: 0.9927, test_loss: 11.2155, test_acc: 0.1059
Epoch [74/100], time: 3.61s, loss: 0.1346, acc: 0.9841, test_loss: 11.1630, test_acc: 0.1084
Epoch [75/100], time: 3.61s, loss: 0.1450, acc: 0.9810, test_loss: 11.2432, test_acc: 0.1073
Epoch [76/100], time: 3.59s, loss: 0.1122, acc: 0.9877, test_loss: 11.3159, test_acc: 0.1065
Epoch [77/100], time: 3.71s, loss: 0.0914, acc: 0.9919, test_loss: 11.3880, test_acc: 0.1067
Epoch [78/100], time: 3.61s, loss: 0.0645, acc: 0.9960, test_loss: 11.4833, test_acc: 0.1077
Epoch [79/100], time: 3.60s, loss: 0.0523, acc: 0.9969, test_loss: 11.5316, test_acc: 0.1063
Epoch [80/100], time: 3.58s, loss: 0.0465, acc: 0.9973, test_loss: 11.5825, test_acc: 0.1079
Epoch [81/100], time: 3.59s, loss: 0.0357, acc: 0.9981, test_loss: 11.6665, test_acc: 0.1067
Epoch [82/100], time: 3.57s, loss: 0.0301, acc: 0.9990, test_loss: 11.7685, test_acc: 0.1086
Epoch [83/100], time: 3.55s, loss: 0.0379, acc: 0.9981, test_loss: 11.7846, test_acc: 0.1067
Epoch [84/100], time: 3.58s, loss: 0.0902, acc: 0.9861, test_loss: 11.7074, test_acc: 0.1084
Epoch [85/100], time: 3.67s, loss: 0.3219, acc: 0.9269, test_loss: 11.5574, test_acc: 0.1067
Epoch [86/100], time: 3.57s, loss: 0.2215, acc: 0.9589, test_loss: 11.5219, test_acc: 0.1072
Epoch [87/100], time: 3.64s, loss: 0.1280, acc: 0.9818, test_loss: 11.6139, test_acc: 0.1084
Epoch [88/100], time: 3.53s, loss: 0.0758, acc: 0.9928, test_loss: 11.6244, test_acc: 0.1098
Epoch [89/100], time: 3.57s, loss: 0.0508, acc: 0.9965, test_loss: 11.7311, test_acc: 0.1092
```

```
Epoch [90/100], time: 3.57s, loss: 0.0391, acc: 0.9975, test_loss: 11.8011, test_acc: 0.1084
Epoch [91/100], time: 3.56s, loss: 0.0323, acc: 0.9982, test_loss: 11.8463, test_acc: 0.1106
Epoch [92/100], time: 3.56s, loss: 0.0243, acc: 0.9989, test_loss: 11.9153, test_acc: 0.1102
Epoch [93/100], time: 3.58s, loss: 0.0193, acc: 0.9992, test_loss: 11.9581, test_acc: 0.1094
Epoch [94/100], time: 3.56s, loss: 0.0175, acc: 0.9993, test_loss: 12.0272, test_acc: 0.1096
Epoch [95/100], time: 3.57s, loss: 0.0230, acc: 0.9986, test_loss: 12.0457, test_acc: 0.1051
Epoch [96/100], time: 3.63s, loss: 0.1115, acc: 0.9799, test_loss: 11.9206, test_acc: 0.1061
Epoch [97/100], time: 3.61s, loss: 0.2612, acc: 0.9411, test_loss: 11.8518, test_acc: 0.1073
Epoch [98/100], time: 3.60s, loss: 0.2039, acc: 0.9591, test_loss: 11.7908, test_acc: 0.1121
Epoch [99/100], time: 3.70s, loss: 0.1135, acc: 0.9809, test_loss: 11.8949, test_acc: 0.1100
Epoch [100/100], time: 3.84s, loss: 0.0606, acc: 0.9934, test_loss: 11.9251, test_acc: 0.1139
```

## 最大验证集预测精度

In [119…
```
max(result1)
```

Out[119]:
```
0.11391760651629072
```

## LSTM的泛化能力明显优于RNN

## 预测，生成藏头诗

In [122…
```
target = "助****教****老****师****都****辛****苦****了****"
generated_index = []
generated_txt = ""
for char in target:
    if char != "*":
        idx = word_dict[char]
        generated_index.append(idx)
        generated_txt += char
    else:
        inp = generated_index + [0] * (max_len - 1 - len(generated_index))
        inp = torch.IntTensor(np.array(inp, dtype=int)) # [49]
        inp = inp.unsqueeze(0) # [1, 49]
        h0, c0 = my_lstm_model.init_hc(1) # [1, 1, 256]
        inp, h0, c0 = inp.cuda(), h0.cuda(), c0.cuda()
        pred, (hn, cn) = my_lstm_model(inp, (h0, c0)) # pred shape [1, 5546]
        pred_idx = torch.max(pred, dim=1)[1].item() # 提取预测的最大值的index
        pred_char = [k for k, v in word_dict.items() if v == pred_idx][0] # 提取对应index的字
        generated_index.append(pred_idx)
        generated_txt += pred_char
```

In [123…
```
for i in range(0, len(generated_txt), 5):
    print(generated_txt[i:i+5])
```

```
助塘川格冷
教浆云子覆
老遒天路一
师春风水清
都落穷水中
辛弦曲遥平
苦天堂都堂
了节珂钟钟
```