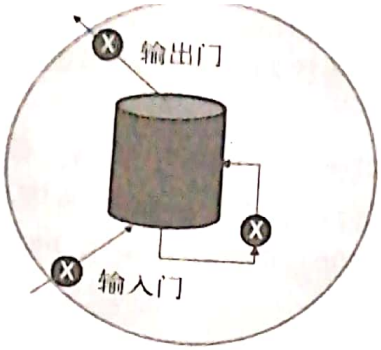

$$i_t = \sigma(W_{ii}x_t + W_{hi}h_t + b_i)$$
$$f_t = \sigma(W_{if}x_t + W_{hf}h_t + b_f)$$
$$o_t = \sigma(W_{io}x_t + W_{ho}h_t + b_o)$$
$$g_t = \tanh(W_{ig}x_{t-1} + W_{hc}h_{t-1} + b_g)$$
$$c_t = f_t * c_{t-1} + i_{t-1} * g_{t-1}$$

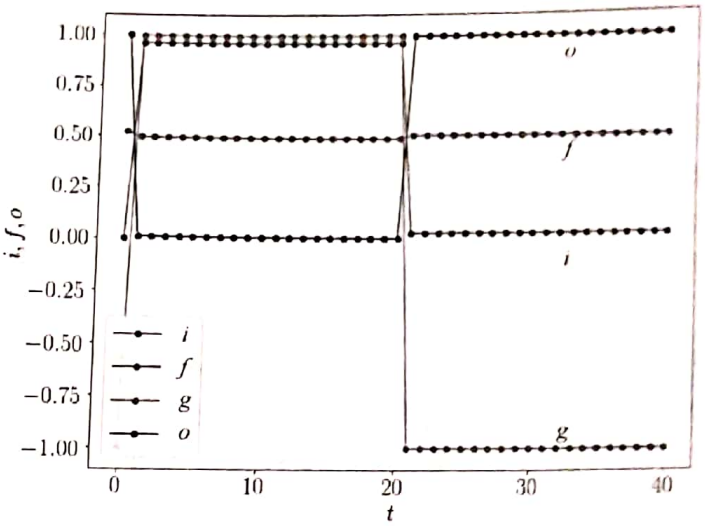

图 10.16　LSTM 单元内部的 3 个门的开启情况（另见彩插）

根据图 10.16 可知，输入门 i 和输入更新 g 都从一个高值掉到了低值，这种变化说明 LSTM 在初期一直打开了信息进入的通道，但到了一半时间之后，它开始关闭这些通道。相反，输出门 o 开始打开，同时遗忘门 f 也有了微小的变化，在一半时间以后也提高了激活值。输入门 o 的打开使 LSTM 单元的 h 变量开始输出，从而有机会影响输出层。遗忘门 f 数值的逐渐增加导致了单元 c 状态的逐渐衰减。

因此，当神经网络训练好了之后，这个 LSTM 单元就学会了通过组合自身可控制的各种门来调节内部和外部的状况，从而正确地给出输出预测。

10.4　LSTM 作曲机

在了解了 RNN 和 LSTM 的工作原理，并知道如何用它们解决序列的预测和生成问题之后，我们将进入实战，制作一个 LSTM 作曲机。

我们的思路是：首先，将一个 MIDI 文件拆解成一个特殊的序列；其次，用这个序列训练一个 LSTM 网络；最后，用这个神经网络持续不断地输出新的序列，这便是机器创作的音乐。

总的来说，大体可以分成 4 个步骤：数据准备，构建模型，训练模型，生成序列。

在深入 LSTM 作曲机的实现细节之前，我们需要先了解一下 MIDI 音乐文件。

10.4.1　MIDI 文件

MIDI（musical instrument digital interface，乐器的数字化接口）原本是个人电脑与外界音乐乐器设备的一种接口形式，电脑借助它就可以将来源于键盘乐器的声音信号转化为数字信息存储起来。

而 MIDI 文件（扩展名.mdi）则是记录这种信号的数字文件格式，与 WAV 文件不同，它并非

直接对音频进行采样记录，而是将音乐的音符记录下来。当 MIDI 乐器演奏了一个音符的时候，它随之将音符转换成 MIDI 信息。在播放的时候，MIDI 文件的播放器实际上是根据这些记录下的音符重新在计算机中模拟各种乐器演奏整首乐曲的。

具体地，MIDI 文件中包含了大量的音轨，这些音轨都可以独立代表一个乐器。每一个音轨中都包含了一串 MIDI 消息（msg）组成的序列，每一个消息都包括音符（note）、速度（velocity，相当于演奏乐器的敲击力度）与时间（time，距离上一个音符的时间长度）这 3 种不同的重要信息。

这样只要把某一个音轨（如钢琴音轨）的消息全部提取出来，我们就得到了一个序列，只不过这个序列是由 3 个整数构成的。其中，音符序列的取值范围是 0~88，速度序列的取值范围是 0~127，时间序列是一个实数序列，它的数值取决于两个音符间的时间长短。

10.4.2　数据准备

为了获得统一的编码，我们将第 3 个序列也就是时间序列也进行了离散化处理。我们把所有可能的时间间隔长短均匀地划分成 10 个小区间，每个小区间对应一种类别；另外，我们将 0 单独作为一个类别，这是因为原始数据中包含大量的 0，因此把它单独分为一类。这样，时间这个序列的每一帧就包含了 11 种不同的可能情况。

于是，我们就得到了如图 10.17 所示的序列。

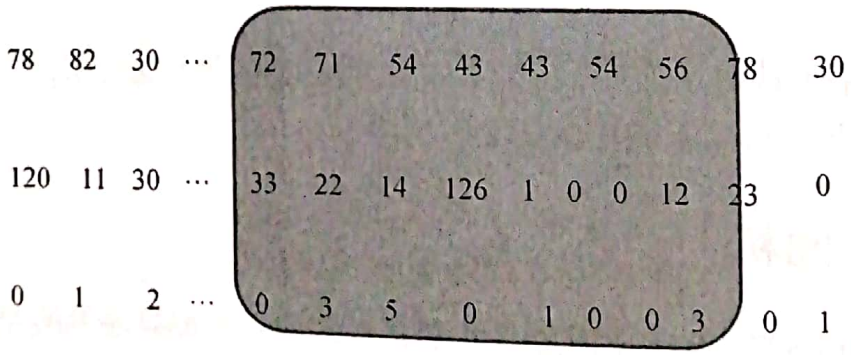

图 10.17　训练数据的片段

图中每一列就代表一个消息，它由 3 个不同的整数组成。在训练阶段，我们以每 31 个消息为一个窗口，其中前 30 个消息作为输入，最后一个消息作为输出，让 LSTM 利用前 30 个预测最后一个。这样，这个长度为 31 的窗口就会在整个序列上从左向右移动，从而形成训练数据。

10.4.3　模型结构

由于我们要预测的序列实际上分成了 3 个部分：音符、速度和间隔时间。三者彼此之间相对独立，联系性不强。因此，在构造模型的时候，我们将这 3 个序列的预测部分分离开来。具体的模型结构如图 10.18 所示。

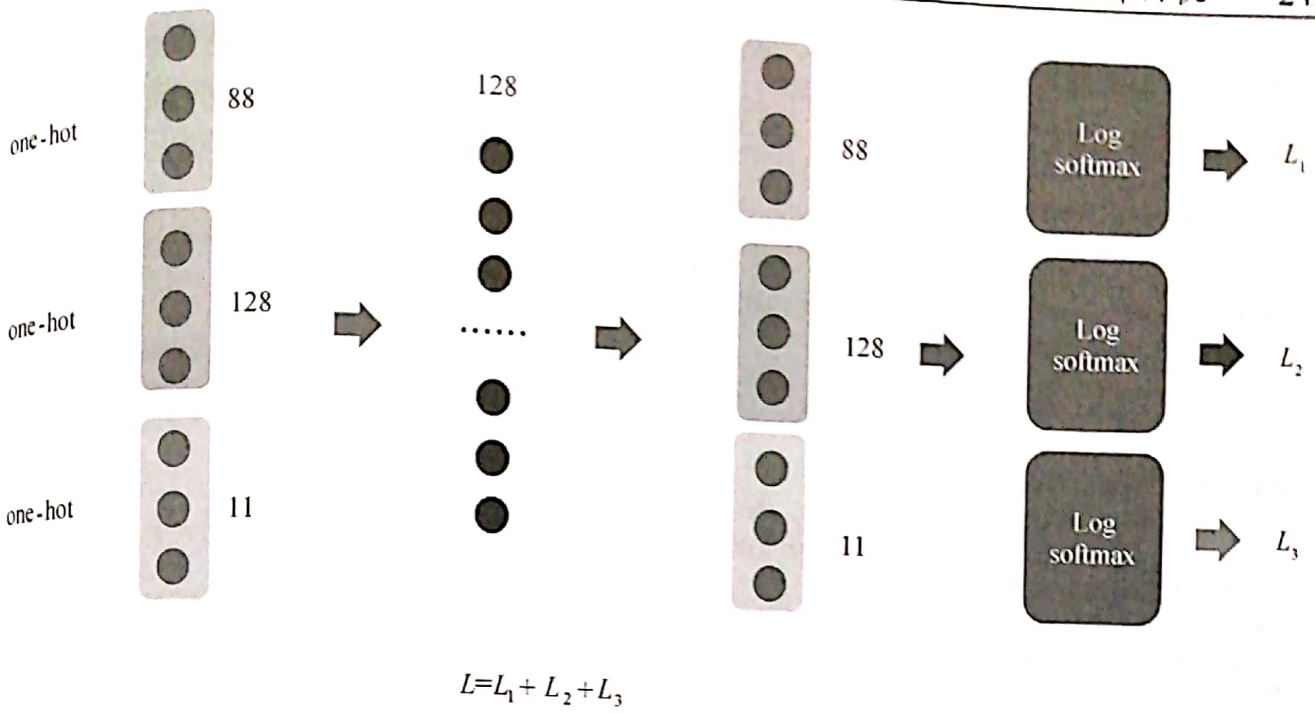

图 10.18 用于生成 MIDI 音乐的神经网络模型结构

可以看到，在输入层和输出层，我们将神经元都划分为了 3 组，每一组专门负责一种序列信息。3 种信号的每一个输入都是一个离散类型变量，因此都采用独热编码的方式输入神经网络，其中类型的种类数决定了输入神经元的个数。因此，在输入层分别有 88、128 和 11 个神经元，一共有 227 个神经元。同理，在输出层，我们也将神经元划分为了 3 组，分别对应 88、128 和 11 个神经元，每个神经元都会经历 softmax 函数，输出一个 0~1 区间中的数表示概率，并且同一组的神经元满足归一化条件。也就是说，第一组的 88 个输出神经元的输出值加起来等于 1，第二组的 128 个神经元的输出值加起来等于 1……

然而，在隐含层方面，我们不再分组，这就意味着 3 组不同的信号又混合到了一起，从而捕获不同序列之间的微弱关系。在隐含层，我们使用了 128 个 LSTM 单元，并且只有一层。

在损失函数方面，我们分别计算 3 组不同神经元输出的交叉熵，并将它们分别命名为 L_1、L_2 和 L_3。最后的总损失函数就是三者之和。

这里先将 3 组不同的向量通过各自独立的独热编码得到 3 组向量，再将它们进行拼接，然后将拼接后的向量放进 LSTM 模型中进行训练。最后在得到结果后，需要按照拼接的逆向操作将 3 组向量从中拆分出来。

10.4.4 代码实现

下面我们将展示详细的代码实现。

1. MIDI 文件的读取

MIDO 是一个非常方便好用的 Python 包，可以直接读入 MIDI 音乐，也可以输出 MIDI 音乐。我们只需要运行 `pip install mido` 就可以安装它了。

首先，导入所需要的 Python 包：

```
#导入必需的依赖包

#与 PyTorch 相关的包
import torch
import torch.utils.data as DataSet
import torch.nn as nn
from torch.autograd import Variable
import torch.optim as optim


#导入 MIDI 音乐处理的包
from mido import MidiFile, MidiTrack, Message

#导入计算与绘图必需的包
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

然后，读入 MIDI 文件，并从消息序列中抽取出音符、速度和间隔时间 3 个序列：

```
#从硬盘中读入 MIDI 音乐文件
mid = MidiFile('./music/krebs.mid') #a Mozart piece

notes = []

time = float(0)
prev = float(0)

original = [] #original 记载了原始的 message 数据，以便后面进行比较

#对 MIDI 文件中所有的消息进行循环
for msg in mid:
    #时间的单位是秒，而不是帧
    time += msg.time

    #如果当前消息不是描述信息
    if not msg.is_meta:
        #仅提炼第一个 channel 的音符
        if msg.channel == 0:
            #如果当前音符为打开的
            if msg.type == 'note_on':
                #获得消息中的信息（编码在字节中）
                note = msg.bytes()
                #我们仅对音符信息感兴趣。音符信息按如下形式记录 [type, note, velocity]
                note = note[1:3] #操作完这一步后，note[0]存音符，note[1]存速度(力度)
                #note[2]存距上一个 message 的时间间隔
                note.append(time - prev)
                prev = time
                #将音符添加到列表 notes 中
                notes.append(note)
                #在原始列表中保留这些音符
                original.append([i for i in note])
```

2. 数据集的准备

最后转化出的 3 个序列数据都存放到了 notes 列表中。接下来，将这个原始的序列转化为我们想要的格式，即将数据离散化为类型变量。

```
#note 和 velocity 都可以看作类型变量
#time 为 float 类型，按照区间将其转化成离散的类型变量
#首先，找到 time 变量的取值区间并进行划分。由于大量 message 的 time 为 0，因此把 0 归为一个特别的类
intervals = 10
values = np.array([i[2] for i in notes])
max_t = np.amax(values) #区间中的最大值
min_t = np.amin(values[values > 0]) #区间中的最小值
interval = 1.0 * (max_t - min_t) / intervals

#接下来，将每一个 message 编码成 3 个独热向量，将这 3 个向量合并到一起就构成了 slot 向量
dataset = []
for note in notes:
    slot = np.zeros(89 + 128 + 12)

    #由于 note 介于 24~112 之间，因此减 24
    ind1 = note[0]-24
    ind2 = note[1]
    #由于 message 中有大量 time=0 的情况，因此将 0 归为单独的一类，其他的一律按照区间划分
    ind3 = int((note[2] - min_t) / interval + 1) if note[2] > 0 else 0
    slot[ind1] = 1
    slot[89 + ind2] = 1
    slot[89 + 128 + ind3] = 1
    #将处理后得到的 slot 数组加入 dataset 中
    dataset.append(slot)
```

最终形成了我们想要的总的序列 dataset。之后，将这个序列沿着时间窗口切分成标准的训练数据对，并把总的数据集切分成训练集和校验集。

```
#生成训练集和校验集
X = []
Y = []
#首先，按照预测的模式，将原始数据生成一对一对的训练数据
n_prev = 30 #滑动窗口长度为 30

#对数据中的所有数据进行循环
for i in range(len(dataset)-n_prev):
    #往后取 n_prev 个 note 作为输入属性
    x = dataset[i:i+n_prev]
    #将第 n_prev+1 个 note（编码前）作为目标属性
    y = notes[i+n_prev]
    #注意 time 要转化成类别的形式
    ind3 = int((y[2] - min_t) / interval + 1) if y[2] > 0 else 0
    y[2] = ind3

    #将 X 和 Y 加入数据集中
    X.append(x)
  Y.append(y)

#将数据集中的前 n_prev 个音符作为种子，用于生成音乐
```

```
seed = dataset[0:n_prev]

#将所有数据顺序打乱重排
idx = np.random.permutation(range(len(X)))
#形成训练与校验数据集列表
X = [X[i] for i in idx]
Y = [Y[i] for i in idx]

#从中切分出 1/10 的数据放入校验集
validX = X[: len(X) // 10]
X = X[len(X) // 10 :]
validY = Y[: len(Y) // 10]
Y = Y[len(Y) // 10 :]

'''将列表再转化为 dataset，并用 dataloader 来加载数据。dataloader 是 PyTorch 开发采用的一套管理数据的方法。通常数据的存储放在 dataset 中，而对数据的调用则是通过 dataloader 完成的。同时，在进行预处理时，系统已经自动将数据打包成批（batch），每次调用都提取出一批（包含多条记录）。从 dataloader 中输出的每一个元素都是一个(x,y)元组，其中 x 为输入的张量，y 为标签。x 和 y 的第一个维度都是 batch_size 大小。'''

'''一批包含 30 个数据记录。这个数字越大，系统在训练的时候，每一个周期要处理的数据就越多，处理就越快，但总的数据量会减少。'''
batch_size = 30

#形成训练集
train_ds = DataSet.TensorDataset(torch.FloatTensor(np.array(X, dtype = float)),
torch.LongTensor(np.array(Y)))
#形成数据加载器
train_loader = DataSet.DataLoader(train_ds, batch_size = batch_size, shuffle = True, num_workers=4)


#校验数据
valid_ds = DataSet.TensorDataset(torch.FloatTensor(np.array(validX, dtype = float)),
torch.LongTensor(np.array(validY)))
valid_loader = DataSet.DataLoader(valid_ds, batch_size = batch_size, shuffle = True, num_workers=4)
```

3. 神经网络的建立

接下来，通过下面的代码构建 LSTM 网络。

```
class LSTMNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, out_size, n_layers=1):
        super(LSTMNetwork, self).__init__()
        self.n_layers = n_layers

        self.hidden_size = hidden_size
        self.out_size = out_size
        #一层 LSTM 单元
        self.lstm = nn.LSTM(input_size, hidden_size, n_layers, batch_first = True)
        #一个 Dropout 部件，以 0.2 的概率 dropout
        self.dropout = nn.Dropout(0.2)
        #一个全连接层
        self.fc = nn.Linear(hidden_size, out_size)
        #对数 Softmax 层
        self.softmax = nn.LogSoftmax(dim=1)
```

```
    def forward(self, input, hidden=None):
        #神经网络的每一步运算

        hhh1 = hidden[0] #读入隐含层的初始信息

        #完成一步 LSTM 运算
        #input 的尺寸为：batch_size, time_step, input_size
        output, hhh1 = self.lstm(input, hhh1) #input:batchsize*timestep*3
        #对神经元输出的结果进行 dropout
        output = self.dropout(output)
        #取出最后一个时刻的隐含层输出值
        #output 的尺寸为：batch_size, time_step, hidden_size
        output = output[:, -1, ...]
        #此时，output 的尺寸为：batch_size, hidden_size
        #输入一个全连接层
        out = self.fc(output)
        #out 的尺寸为：batch_size, output_size

        #将 out 的最后一个维度分割成 3 份 x，y，z，分别对应了对 note，velocity 以及 time 的预测

        x = self.softmax(out[:, :89])
        y = self.softmax(out[:, 89: 89 + 128])
        z = self.softmax(out[:, 89 + 128:])

        #x 的尺寸为 batch_size, 89
        #y 的尺寸为 batch_size, 128
        #z 的尺寸为 batch_size, 11
        #返回 x,y,z
        return (x,y,z)

    def initHidden(self, batch_size):
        #将隐含层单元变量全部初始化为 0
        #注意尺寸是：layer_size, batch_size, hidden_size
        out = []
        hidden1=Variable(torch.zeros(1, batch_size, self.hidden_size))
        cell1=Variable(torch.zeros(1, batch_size, self.hidden_size))
        out.append((hidden1, cell1))
        return out
```

我们定义了两个函数，一个用来计算特殊的损失函数，另一个计算一批预测数据的预测准确率。在这个项目中，由于损失函数是 3 部分损失函数之和，因此我们特别定义了自己的损失函数。

```
def criterion(outputs, target):
    #为本模型自定义的损失函数，由 3 部分组成，每部分都是一个交叉熵损失函数
    #分别对应 note、velocity 和 time 的交叉熵
    x, y, z = outputs
    loss_f = nn.NLLLoss()
    loss1 = loss_f(x, target[:, 0])
    loss2 = loss_f(y, target[:, 1])
    loss3 = loss_f(z, target[:, 2])
    return loss1 + loss2 + loss3
```

```
def rightness(predictions, labels):
    '''计算预测错误率的函数，其中 predictions 是模型给出的一组预测结果，batch_size 行 num_classes 列的
矩阵，labels 是数据中的正确答案'''
    #对于任意一行（一个样本）的输出值的第 1 个维度求最大，得到每一行最大元素的下标
    pred = torch.max(predictions.data, 1)[1]
    #将下标与 labels 中包含的类别进行比较，并累计得到比较正确的数量
    rights = pred.eq(labels.data).sum()
    return rights, len(labels) #返回正确的数量和这次一共比较了多少元素
```

4. 训练网络

定义好神经网络之后，我们便可以用下面的代码来训练这个网络了。

```
#定义一个 LSTM，其中输入层和输出层的单元个数取决于每个变量的类型取值范围
lstm = LSTMNetwork(89 + 128 + 12, 128, 89 + 128 + 12)
optimizer = optim.Adam(lstm.parameters(), lr=0.001)
num_epochs = 100
train_losses = []
valid_losses = []
records = []

#开始训练循环
for epoch in range(num_epochs):
    train_loss = []
    #开始遍历加载器中的数据
    for batch, data in enumerate(train_loader):
        #batch 为数字，表示已经进行了第几个 batch
        #data 为一个二元组，分别存储了一条数据记录的输入和标签
        #每个数据的第一个维度都是 batch_size = 30 的数组

        lstm.train() #标志 LSTM 当前处于训练阶段，Dropout 开始起作用
        init_hidden = lstm.initHidden(len(data[0])) #初始化 LSTM 的隐含单元变量
        optimizer.zero_grad()
        x, y = Variable(data[0]), Variable(data[1]) #从数据中提炼出输入和输出对
        outputs = lstm(x, init_hidden) #输入 LSTM，产生输出 outputs
        loss = criterion(outputs, y) #代入损失函数并产生 loss
        train_loss.append(loss.data.numpy()) #记录 loss
        loss.backward() #反向传播
        optimizer.step() #梯度更新
    if 0 == 0:
        #在校验集上运行一遍，并计算在校验集上的分类准确率
        valid_loss = []
        lstm.eval() #将模型标志为测试状态，关闭 dropout 的作用
        rights = []
        #遍历加载器加载进来的每一个元素
        for batch, data in enumerate(valid_loader):
            init_hidden = lstm.initHidden(len(data[0]))
            #完成 LSTM 的计算
            x, y = Variable(data[0]), Variable(data[1])
            #x 的尺寸：batch_size, length_sequence, input_size
            #y 的尺寸：batch_size, (data_dimension1=89+ data_dimension2=128+ data_dimension3=12)
            outputs = lstm(x, init_hidden)
            #outputs: (batch_size*89, batch_size*128, batch_size*11)
            loss = criterion(outputs, y)
            valid_loss.append(loss.data.numpy())
```

```
        #计算每个指标的分类准确度
        right1 = rightness(outputs[0], y[:, 0])
        right2 = rightness(outputs[1], y[:, 1])
        right3 = rightness(outputs[2], y[:, 2])
        rights.append((right1[0] + right2[0] + right3[0]) * 1.0 / (right1[1] + right2[1] + right3[1]))
    #打印结果
    print('第{}轮, 训练 Loss: {:.2f}, 校验 Loss: {:.2f}, 校验准确度: {:.2f}'.format(epoch,
                                                        np.mean(train_loss),
                                                        np.mean(valid_loss),
                                                        np.mean(rights)
                                                       ))
    records.append([np.mean(train_loss), np.mean(valid_loss), np.mean(rights)])
```

接下来，我们便可以打印 Loss 和准确度随着训练周期增长的曲线。

```
#绘制训练过程中的 Loss 曲线
a = [i[0] for i in records]
b = [i[1] for i in records]
c = [i[2] * 10 for i in records]
plt.plot(a, '-', label = 'Train Loss')
plt.plot(b, '-', label = 'Validation Loss')
plt.plot(c, '-', label = '10 * Accuracy')
plt.legend()
```

得到的训练曲线如图 10.19 所示，可以看到预测准确度在持续提升。

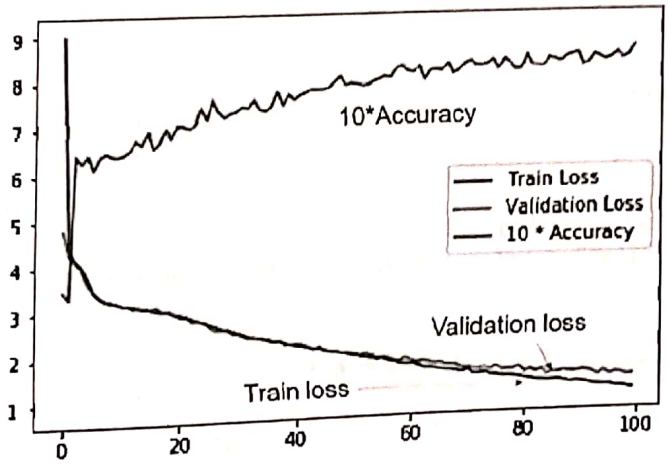

图 10.19 LSTM 的损失曲线与准确度曲线

5. 序列生成

在训练完成之后，就要使用训练好的模型来生成音乐了。生成音乐的第一步是给我们的模型一个生成种子，这个种子可以作为生成序列的起始点。简单起见，我们就用训练数据乐曲的前30个音符作为种子，这样可以让我们的乐曲更像音乐，另外也可以跟原始序列进行比较。

在生成阶段，我们将输出部分（输出层）按照随机的方式来采样生成序列，而不是按照最大概率的方式，这样做的好处是能够保持输出乐曲的多样性，让它听起来更像音乐，如图 10.20 所示。

图 10.20　模型生成音乐时采用的是随机采样

最后，代码如下：

```
#生成 3000 步
predict_steps = 3000

#初始时刻，将 seed（一段种子音符，即开始读入的音乐文件）赋给 x
x = seed
#将数据扩充为合适的形式
x = np.expand_dims(x, axis = 0)
#现在 x 的尺寸为：batch=1, time_step =30, data_dim = 229

lstm.eval()
initi = lstm.initHidden(1)
predictions = []
#开始每一步的迭代
for i in range(predict_steps):
    #根据前 n_prev 预测后面的一个音符
    xx = Variable(torch.FloatTensor(np.array(x, dtype = float)))
    preds = lstm(xx, initi)

    #返回预测的 note, velocity, time 的模型预测概率对数
    a,b,c = preds
    #a 的尺寸为：batch=1*data_dim=89, b 为 1*128, c 为 1*11

    #将概率对数转化为随机的选择
    ind1 = torch.multinomial(a.view(-1).exp(),  89 )
    ind2 = torch.multinomial(b.view(-1).exp(),  128)
    ind3 = torch.multinomial(c.view(-1).exp(),  11 )

    ind1 = ind1.data.numpy()[0] #0-89 中的整数
    ind2 = ind2.data.numpy()[0] #0-128 中的整数
    ind3 = ind3.data.numpy()[0] #0-11 中的整数

    #将选择转换为正确的音符等数值，注意 time 分为 11 类，第一类为 0 这个特殊的类，其余按照区间分类
    note = [ind1 + 24, ind2, 0 if ind3 ==0 else ind3 * interval + min_t]

    #将预测的内容进行存储
    predictions.append(note)
```

```
#将新的预测内容再次转变为输入数据，准备输入LSTM
slot = np.zeros(89 + 128 + 12, dtype = int)
slot[ind1] = 1
slot[89 + ind2] = 1
slot[89 + 128 + ind3] = 1
slot1 = np.expand_dims(slot, axis = 0)
slot1 = np.expand_dims(slot1, axis = 0)

#slot1的数据格式为：batch=1*time=1*data_dim=229

#x拼接上新的数据
x = np.concatenate((x, slot1), 1)
#现在x的尺寸为：batch_size = 1 * time_step = 31 * data_dim =229

#滑动窗口往前平移一次
x = x[:, 1:, :]
#现在x的尺寸为：batch_size = 1 * time_step = 30 * data_dim = 229
```

6. 最终结果

等待若干时间步之后，我们就得到了新的生成序列。不过，这个新序列仍然需要进行一些变换，最终变成原来的MIDI消息，再将MIDI消息串拼接成MIDI音乐格式。

```
#将生成的序列转化为MIDI的消息，并保存MIDI音乐
mid = MidiFile()
track = MidiTrack()
mid.tracks.append(track)

for i, note in enumerate(predictions):
    #在note一开始插入一个147，表示打开note_on
    note = np.insert(note, 0, 147)
    #将整数转化为字节
    bytes = note.astype(int)
    #创建一个message
    msg = Message.from_bytes(bytes[0:3])
    #0.001025为任意取值，可以调节音乐的速度
    #由于生成的time都是一系列的间隔时间，转化为msg后时间尺度过小，因此需要调节放大
    time = int(note[3]/0.001025)
    msg.time = time
    #将message添加到音轨中
    track.append(msg)

#保存文件
mid.save('music/new_song.mid')
########################################
```

我们将生成的MIDI文件保存到了music/new_song.mid中。读者可以播放试试。

10.5 小结

本章我们学习了用深度学习的方式生成序列的方法，并重点讲解了RNN和LSTM的工作原理，最后用这种序列生成方法生成了一段MIDI音乐。

本章的重点就是 RNN 和 LSTM 的工作原理。RNN 与我们以前接触的前馈神经网络相比，增加了隐含层内部的连接，这些连接可以赋予 RNN 记忆能力。但是，由于 RNN 非线性激活函数的存在，在长时间的运行中，信号会不断衰减，所以一般 RNN 的记忆不会很长，这就导致了我们无法学习和记忆存在于数据之中的长时间模式。

为了解决这个问题，人们提出了 RNN 的一个改进版本 LSTM。LSTM 对 RNN 的改进就体现为每一个隐含层神经元多出了很多内部结构，即 3 个控制门和 1 个内部的“蓄水池”。这些机制可以使 LSTM 具有长期的记忆能力。

为了进一步理解 RNN 和 LSTM 的工作原理，我们通过一个简单的任务（正则文法的学习和生成）进一步剖析了 RNN 和 LSTM 的运作。我们发现，RNN 是通过动力系统的方式来存储信息的，并将信息对应到动力系统的吸引子上面；而单个的 LSTM 单元可以通过内部的“蓄水池”（即细胞的 c 状态）来存储信息，并通过精心调控输入、输出和遗忘门来做到精准的信息记忆。

最后，我们将所学的内容应用到了学习一个音乐的 MIDI 序列上。我们将 MIDI 音乐视作 3 个相互近似独立、弱相关的序列，从而将 MIDI 音乐生成的问题转化为一个序列预测问题，并通过让 LSTM 读取 MIDI 文件而达到训练的目的。最后，通过运行训练好的 LSTM，一首还算动听的 MIDI 乐曲便生成了。

10.6　Q&A

Q：网络输入可不可以取复数？

A：可以取复数，而且在取了复数以后，它还会具有很多优秀的性质。

Q：在 RNN“动力系统”这部分，如果特征值大于 1，动力系统不会发散吗？

A：是会发散的，所以从吸引子往外跳了。

10.7　扩展阅读

[1] 关于 RNN 如何进行序列生成：http://karpathy.github.io/2015/05/21/rnn-effectiveness/。

[2] 关于 LSTM 如何工作：http://colah.github.io/posts/2015-08-Understanding-LSTMs/。

[3] 关于用 RNN 识别上下文无关文法：Rodriguez P, et al. A Recurrent Neural Network that Learns to Count. Connection Science. Vol. 11, No1, 1999: 5-40。