实验5 循环神经网络实验

目 录

RNN预测模拟数据

回顾

- 循环神经网络能够处理**任意长度的时序数据**

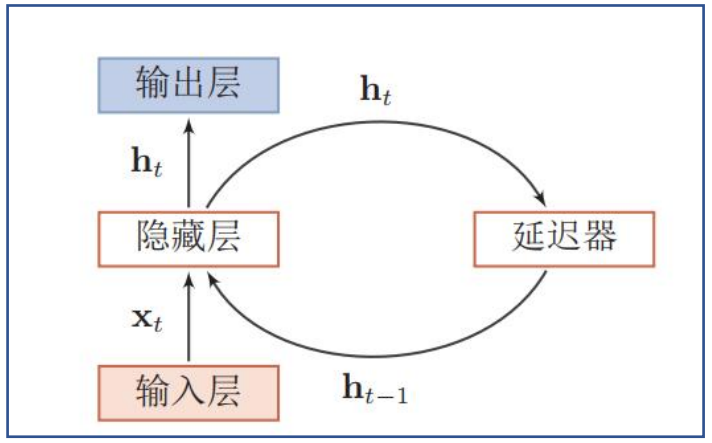
给定一个输入序列 ， $x_{1:T}=(x_1,x_2,\cdots,x_t,\cdots x_T)$ 其计算公式如下：

$$h_t=f(h_{t-1},x_t),$$
$$y_t=g(h_t)$$

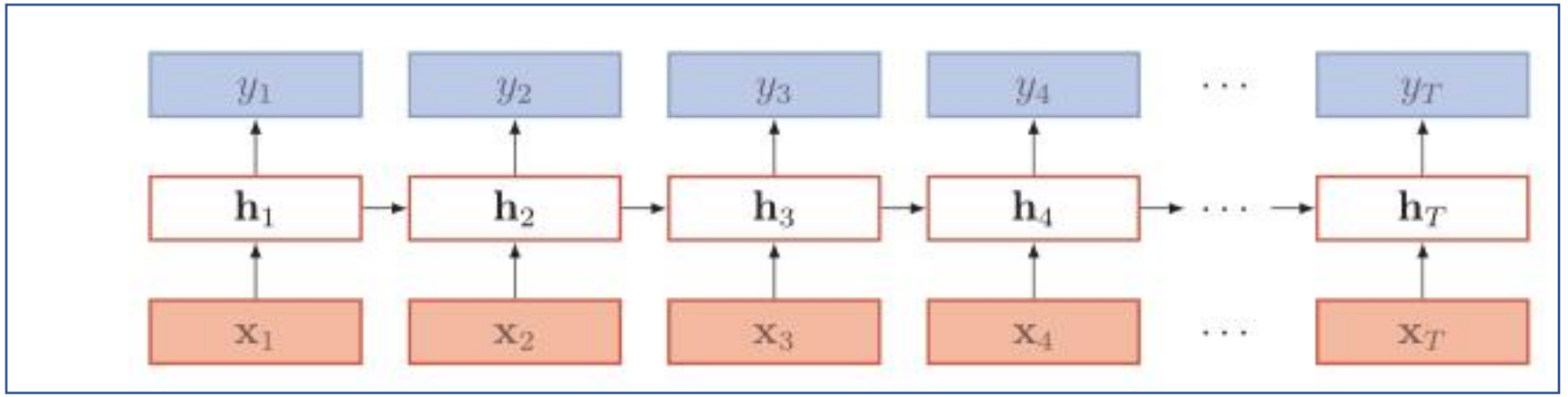
$f(\bullet),g(\bullet)$ 实例化

$$h_t=\sigma(W_h x_t+U_h h_{t-1}+b_h),$$
$$y_t=\sigma(W_y h_t+b_y)$$

激活函数可以替换，如tanh、Relu



画法一



画法二

RNN预测模拟数据

torch.nn.RNN

CLASS torch.nn.RNN(*args,* * kwargs)

参数说明:

- **input_size** – 输入x的特征数量。
- **hidden_size** – 隐层的特征数量。
- **num_layers** – RNN的层数。
- **nonlinearity** – 指定非线性函数使用tanh还是relu。默认是tanh。
- **bias** – 如果是False，那么RNN层就不会使用偏置权重 b_ih和b_hh,默认是True
- **batch_first** – 如果True的话，那么输入Tensor的shape应该是[batch_size, time_step, feature],输出也是这样。
- **dropout** – 如果值非零，那么除了最后一层外，其它层的输出都会套上一个dropout层。
- **bidirectional** – 如果True，将会变成一个双向RNN，默认为False。

RNN预测模拟数据

torch.nn.RNN

RNN的输入： (input, h_0)

- **input (seq_len, batch, input_size):** 保存输入序列特征的tensor。input可以是被填充的变长的序列。
- **h_0 (num_layers * num_directions, batch, hidden_size):** 保存着初始隐状态的tensor

RNN的输出： (output, h_n)

- **output (seq_len, batch, hidden_size * num_directions):** 保存着RNN最后一层的输出特征。如果输入是被填充过的序列，那么输出也是被填充的序列。
- **h_n (num_layers * num_directions, batch, hidden_size):** 保存着最后一个时刻隐状态。

RNN预测模拟数据

绘制序列

1. 导入所需要的包

```
import numpy as np
import torch
import matplotlib.pyplot as plt
import torch.utils.data as Data
import torch.nn as nn
import pandas as pd
```

numpy用于数据处理；
torch.nn用于构建网络；
torch用于对tensor进行处理；
引入画图库方便后续可视化；

2. 定义绘制序列函数

```
#绘制序列
def plot_series(time, series, format="-", start=0, end=None, label=None):
    #根据时间轴和对应数据列表绘制序列图像
    plt.plot(time[start:end], series[start:end], format, label=label)
    #设置横纵轴意义
    plt.xlabel("Time")
    plt.ylabel("Value")
    #设置图例说明字体大小
    if label:
        plt.legend(fontsize=14)
    #显示网格
    plt.grid(True)
```

以时间为横坐标、以序列值为纵坐标，设置曲线对应的label，将序列进行可视化；

设置横纵轴label；

画布显示网格，便于观察

RNN预测模拟数据

趋势、噪声的生成函数

3. 趋势模式的生成函数

```
#趋势模式
def trend(time, slope=0):
    #序列与时间呈线性关系
    return slope * time
```

使序列的值与时间坐标呈线性关系来构造趋势；具体用参数slope来控制上升还是下降，陡峭还是平缓

4. 噪声模式的生成函数

```
#白噪声
def white_noise(time, noise_level=1, seed=None):
    #生成正态分布的伪随机数序列
    rnd = np.random.RandomState(seed)
    #noise_level控制噪声幅值大小
    return rnd.randn(len(time)) * noise_level
```

生成满足正态分布的序列，并用noise_level控制噪声幅值的大小

RNN预测模拟数据

季节性的生成函数

5. 季节性模式的生成函数

```
#季节性（周期性）模式
def seasonal_pattern(season_time):
    """Just an arbitrary pattern, you can change it if you wish"""
    #分段函数(自变量取值[0,1])作为一个模式
    return np.where(season_time < 0.4,
                    np.cos(season_time * 2 * np.pi),
                    1 / np.exp(3 * season_time))

#将某个季节性（周期性）模式循环多次
def seasonality(time, period, amplitude=1, phase=0):
    """Repeats the same pattern at each period"""
    #将时间映射到0-1之间
    season_time = ((time + phase) % period) / period
    return amplitude * seasonal_pattern(season_time)
```

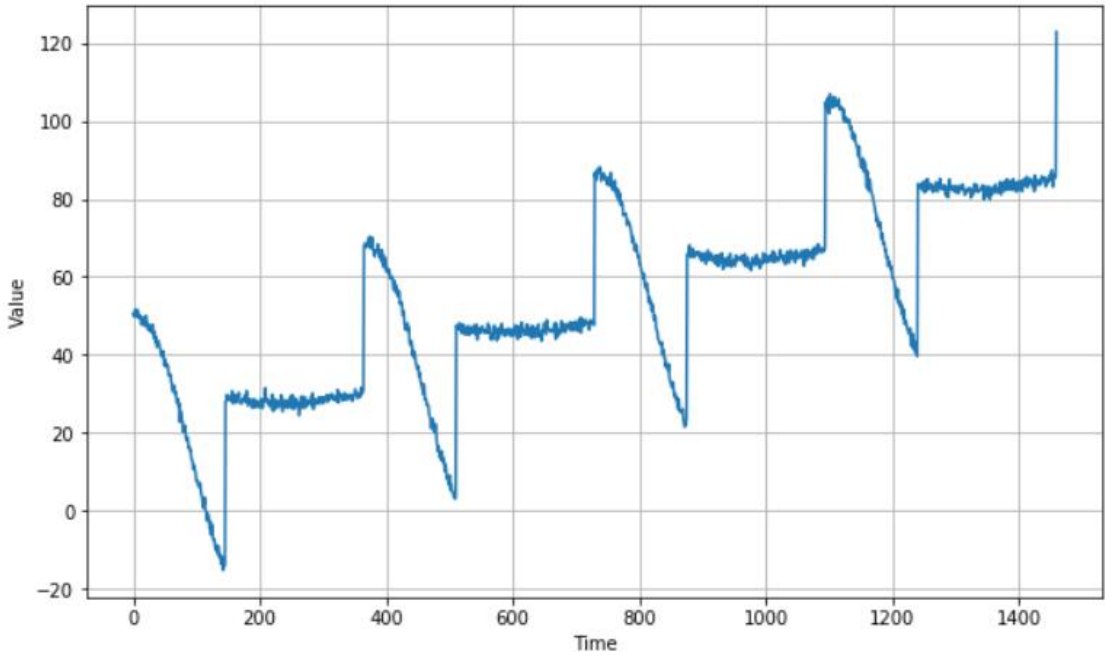
季节性的实现包括两步，第一步构造一个周期内的序列，第二步实现该序列的循环，用amplitude控制幅值大小

RNN预测模拟数据

- **生成混合模式的时序数据**

```
time=np.arange(4*365+1)
baseline=10
slope=0.05
amplitude=40
noise_level=1
series=baseline+trend(time,slope)\
       +seasonality(time,period=365,amplitude=amplitude)\
       +white_nosie(time,noise_level=noise_level,seed=42)

plt.figure(figsize=(10, 6))
plot_series(time, series)
plt.show()
```



RNN预测模拟数据

参数设置

```
# 设置超参数
input_size = 1
hidden_size = 256
output_size = 1
epochs = 200
lr = 0.05
batch_size = 128
time_step = 5
```

- **input_size**:输入数据的特征数量，时序数据为1
- **hidden_size**:隐藏层的特征个数
- **output_size**:输出的时间窗口的长度
- **epochs**：训练轮数
- **lr**:学习率
- **batch_size**：一个批量的大小
- **time_step**:输入的时间窗口的长度

RNN预测模拟数据

数据预处理

```
#训练集的比例
split_prop=0.7
#前70%的数据作为训练集
train_data = series[:int(split_prop * int(series.size))]
#剩下的数据作为测试集
test_data = series[int(split_prop * int(series.size)):]

# # 数据归一化
train_data_normalized = (train_data - train_data.min()) / (train_data.max() - train_data.min())
test_data_normalized = (test_data - train_data.min()) / (train_data.max() - train_data.min())
```

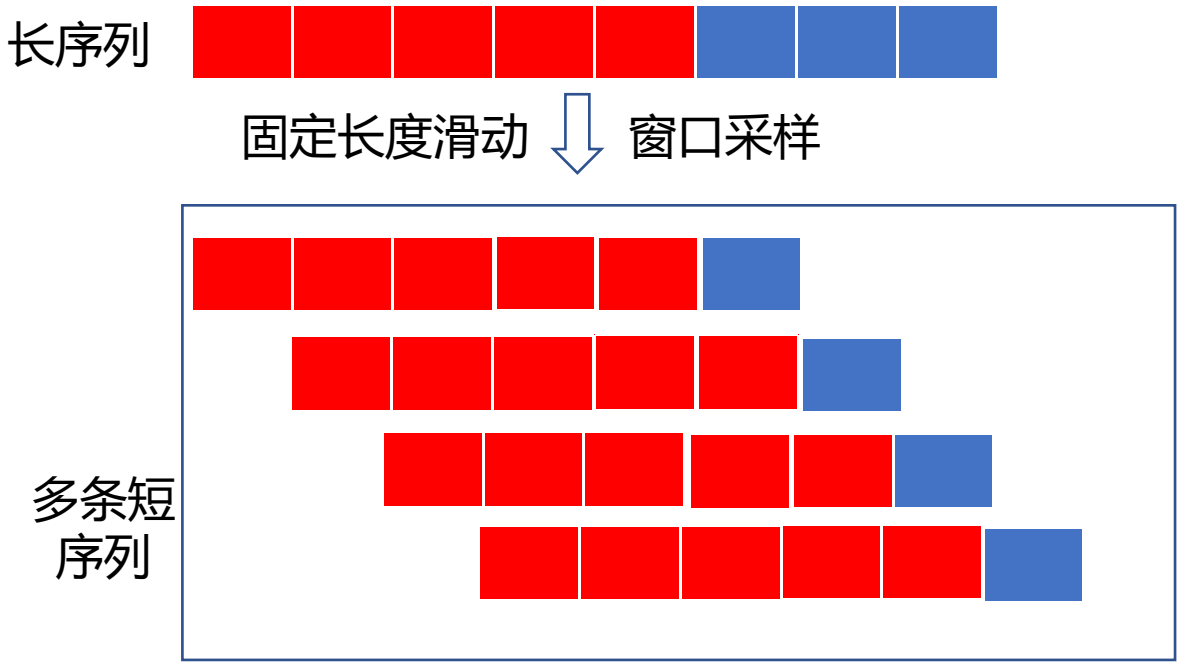
- 数据的标准化（normalization）是将数据按比例缩放，使之落入一个小的特定区间。
- 去除数据的单位限制，将其转化为无量纲的纯数值，便于不同单位或量级的指标能够进行比较和加权。
- 其中最典型的就是数据的归一化处理，即将数据统一映射到[0,1]区间上。

RNN预测模拟数据

滑动窗口采样

```
train_x = []
train_y = []
test_x = []
test_y = []
# 对训练数据采样
i = 0
while (i + time_step + output_size < len(train_data_normalized)):
    # 输入的序列
    train_x.append(train_data_normalized[i:i + time_step])
    # 输出的序列
    train_y.append(train_data_normalized[i + time_step:i + time_step + output_size])
    i += output_size

# 对测试数据采样
j = 0
while (j + time_step + output_size < len(test_data_normalized)):
    # 输入的序列
    test_x.append(test_data_normalized[j:j + time_step])
    # 输出的序列
    test_y.append(test_data_normalized[j + time_step:j + time_step + output_size])
    j += output_size
```



固定滑动窗口采样示意图

RNN预测模拟数据

装入数据

```
#将数据转换为tensor格式
train_x = torch.tensor(train_x,dtype=torch.float32)
train_y = torch.tensor(train_y,dtype=torch.float32)
test_x = torch.tensor(test_x,dtype=torch.float32)
test_y = torch.tensor(test_y,dtype=torch.float32)

# 将训练数据装入DataLoader
train_dataset = Data.TensorDataset(train_x, train_y)
train_loader = Data.DataLoader(dataset=train_dataset,
                               batch_size=batch_size,
                               shuffle=True, num_workers=0)
```

将训练集装入DataLoader 中，便于之后的训练过程取出数据

class torch.utils.data.TensorDataset(data_tensor, target_tensor)

- **data_tensor** (*Tensor*) – 包含样本数据
- **target_tensor** (*Tensor*) – 包含样本目标（标签）

class torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, num_workers=0)

- **dataset** (Dataset) – 加载数据的数据集
- **batch_size** (int, optional) – 每个batch加载多少个样本(默认: 1)
- **shuffle** (bool, optional) – 设置为True时会在每个epoch重新打乱数据(默认: False)

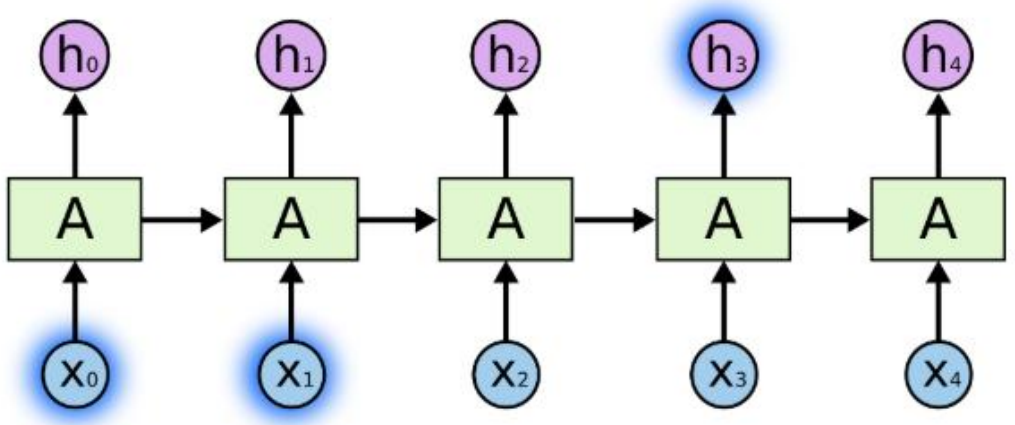
RNN预测模拟数据

构建RNN网络

```
# 构建RNN网络
class MYRNN(nn.Module):
    def __init__(self, input_size, hidden_size,output_size,time_step):
        super(MYRNN, self).__init__()
        self.input_size=input_size
        self.hidden_size = hidden_size
        self.output_size=output_size
        self.time_step=time_step

        # 创建RNN层和linear层，RNN层提取特征，linear层用作最后的预测
        self.rnn = nn.RNN(
            input_size=self.input_size,
            hidden_size=self.hidden_size,
            num_layers=1,
            batch_first=True,
        )
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, x):
        #获得RNN的计算结果，舍去h_n
        r_out, _ = self.rnn(x)
        # #按照RNN模型结构修改input_seq的形状，作为linear层的输入
        r_out = r_out.reshape(-1, self.hidden_size)
        out = self.out(r_out)
        # 将out恢复成(batch,seq_len,output_size)
        out = out.reshape(-1, self.time_step, self.output_size)
        # return所有batch的seq_len的最后一项
        return out[:, -1, :]
```



RNN预测模拟数据

模型参数初始化

```
# 实例化神经网络
net = MYRNN(input_size, hidden_size, output_size, time_step)
# 初始化网络参数
for param in net.parameters():
    nn.init.normal_(param, mean=0, std=0.01)
# 设置损失函数
loss = nn.MSELoss()
# 设置优化器
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
# 如果GPU可用，就用GPU运算;否则使用CPU运算
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# 将net复制到device（GPU或CPU）
net.to(device)
```

- 从**均值为0、标准差为0.01**的**正态分布**中取随机数，初始化网络模型参数
- 将损失函数设置为**均方误差**
- 优化器使用**随机梯度下降法**
- GPU可用则使用GPU运算，否则使用CPU运算

RNN预测模拟数据

开始训练

```
train_loss = []
test_loss = []
# 开始训练
for epoch in range(epochs):
    train_l = []
    test_l = 0
    for x, y in train_loader:
        # RNN输入应为input (seq_len, batch, input_size)，将x转化为三维数据
        x = torch.unsqueeze(x, dim=2)
        # 将x,y放入device中
        x = x.to(device)
        y = y.to(device)
        # 计算得到预测值y_predict
        y_predict = net(x)
        # 计算y_predict与真实y的loss
        l = loss(y_predict, y)
        # 清空所有被优化过的Variable的梯度.
        optimizer.zero_grad()
        # 反向传播，计算当前梯度
        l.backward()
        # 根据梯度更新网络参数
        optimizer.step()
        train_l.append(l.item())
    # 修改测试集的维度以便放入网络中
    test_x_temp = torch.unsqueeze(test_x, dim=2)
    # 测试集放入device中
    test_x_temp = test_x_temp.to(device)
    test_y_temp = test_y.to(device)
    # 得到测试集的预测结果
    test_predict = net(test_x_temp)
    # 计算测试集loss
    test_l = loss(test_predict, test_y_temp)
    # 打印
    print("Epoch%d:train loss=%.5f,test loss=%.5f" % (epoch + 1, np.array(train_l).mean(), test_l.item()))
    train_loss.append(np.array(train_l).mean())
    test_loss.append(test_l.item())
```

完成网络初始化和训练参数的设置后，开始训练。

在每一轮训练中，对每一小批数据计算梯度，反向传播更新参数。

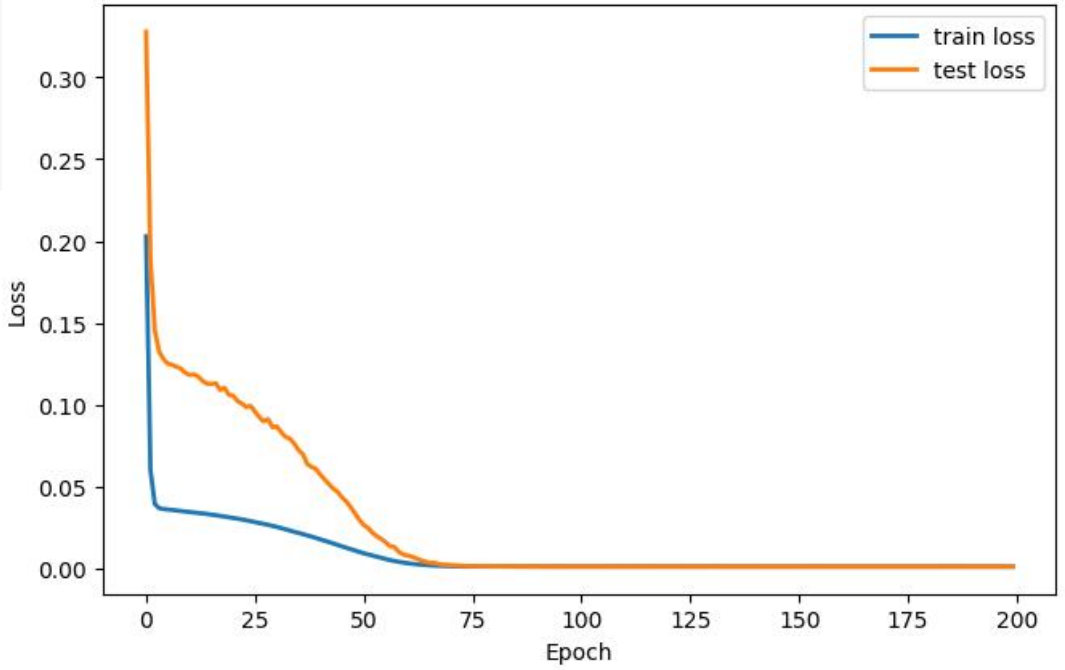
每一轮训练后，计算并输出当前模型在训练集和测试集上的损失。

RNN预测模拟数据

- 绘制loss曲线

```
# 画出Loss趋势图
plt.plot(range(epochs), train_loss, label="train loss", linewidth=2)
plt.plot(range(epochs), test_loss, label="test loss", linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

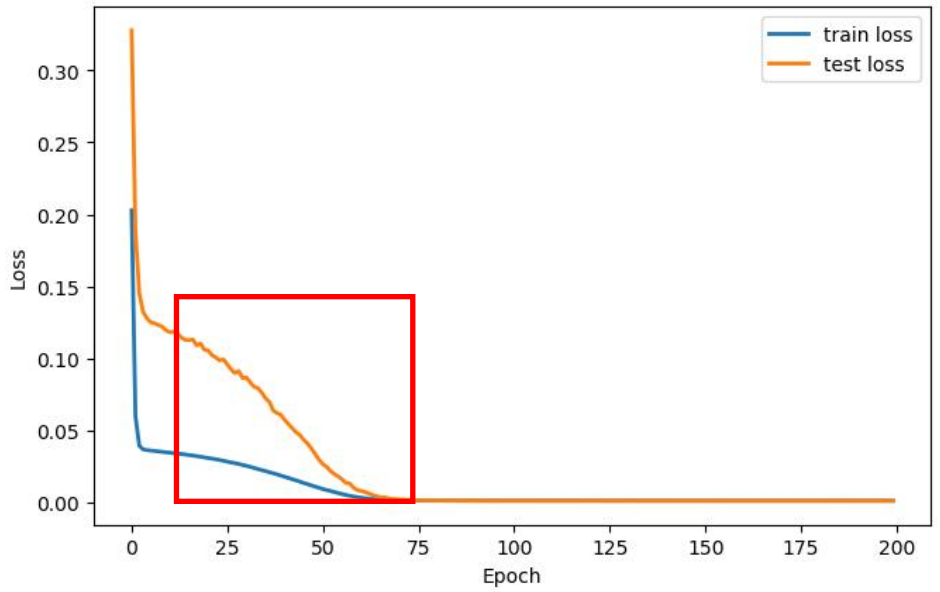
```
Epoch191:train loss=0.00106,test loss=0.00102
Epoch192:train loss=0.00106,test loss=0.00103
Epoch193:train loss=0.00105,test loss=0.00104
Epoch194:train loss=0.00107,test loss=0.00102
Epoch195:train loss=0.00105,test loss=0.00102
Epoch196:train loss=0.00105,test loss=0.00102
Epoch197:train loss=0.00108,test loss=0.00098
Epoch198:train loss=0.00106,test loss=0.00101
Epoch199:train loss=0.00105,test loss=0.00104
Epoch200:train loss=0.00106,test loss=0.00100
```

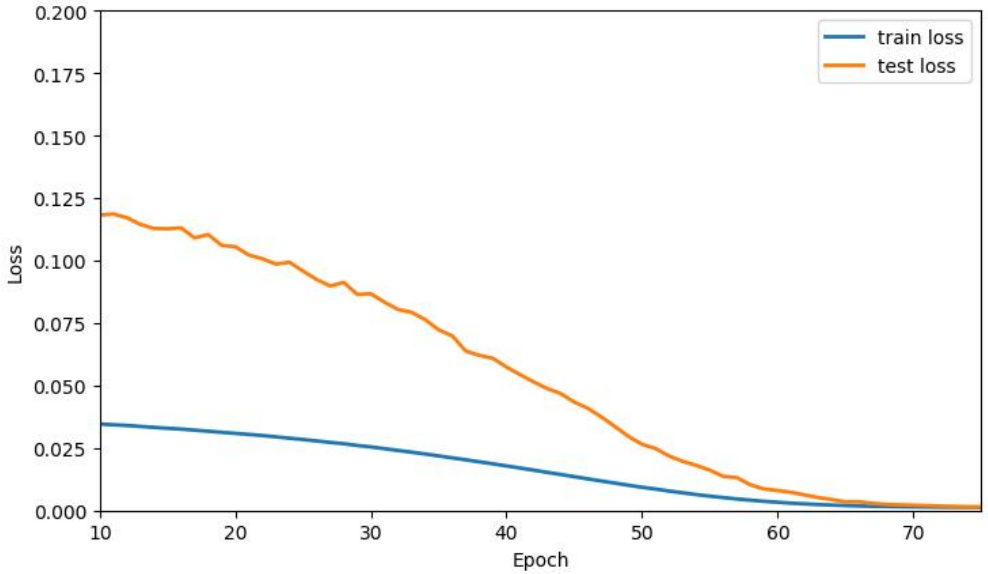


RNN预测模拟数据

- **绘制loss曲线**

```
# Loss局部图
plt.plot(range(epochs), train_loss, label="train loss", linewidth=2)
plt.plot(range(epochs), test_loss, label="test loss", linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.xlim(10, 75)
plt.ylim(0, 0.2)
plt.legend()
plt.show()
```





绘制局部曲线，观察变化情况

需要注意这个图像不同机器可能下降速度不同，但总体趋势是不断下降的。

RNN预测模拟数据

预测并与真实值对比

```
# 将测试集放入模型计算预测结果
test_x_temp = torch.unsqueeze(test_x, dim=2)
test_x_temp = test_x_temp.to(device)
predict = net(test_x_temp)
# 逆归一化
predict = predict.cpu().detach().numpy() * (train_data.max() - train_data.min()) + train_data.min()
test_y = np.array(test_y) * (train_data.max() - train_data.min()) + train_data.min()
# 将数据从[[ouyput_size]...]转换为[x1,x2,x3...]
predict_result = []
test_y_result = []
for item in predict:
    predict_result += list(item)
for item in test_y:
    test_y_result += list(item)
```

- 计算得到预测结果
- 测试集当前值域为[0,1]，需将其逆归一化，映射回原来的值域
- 将结果的维度转换为一维

RNN预测模拟数据

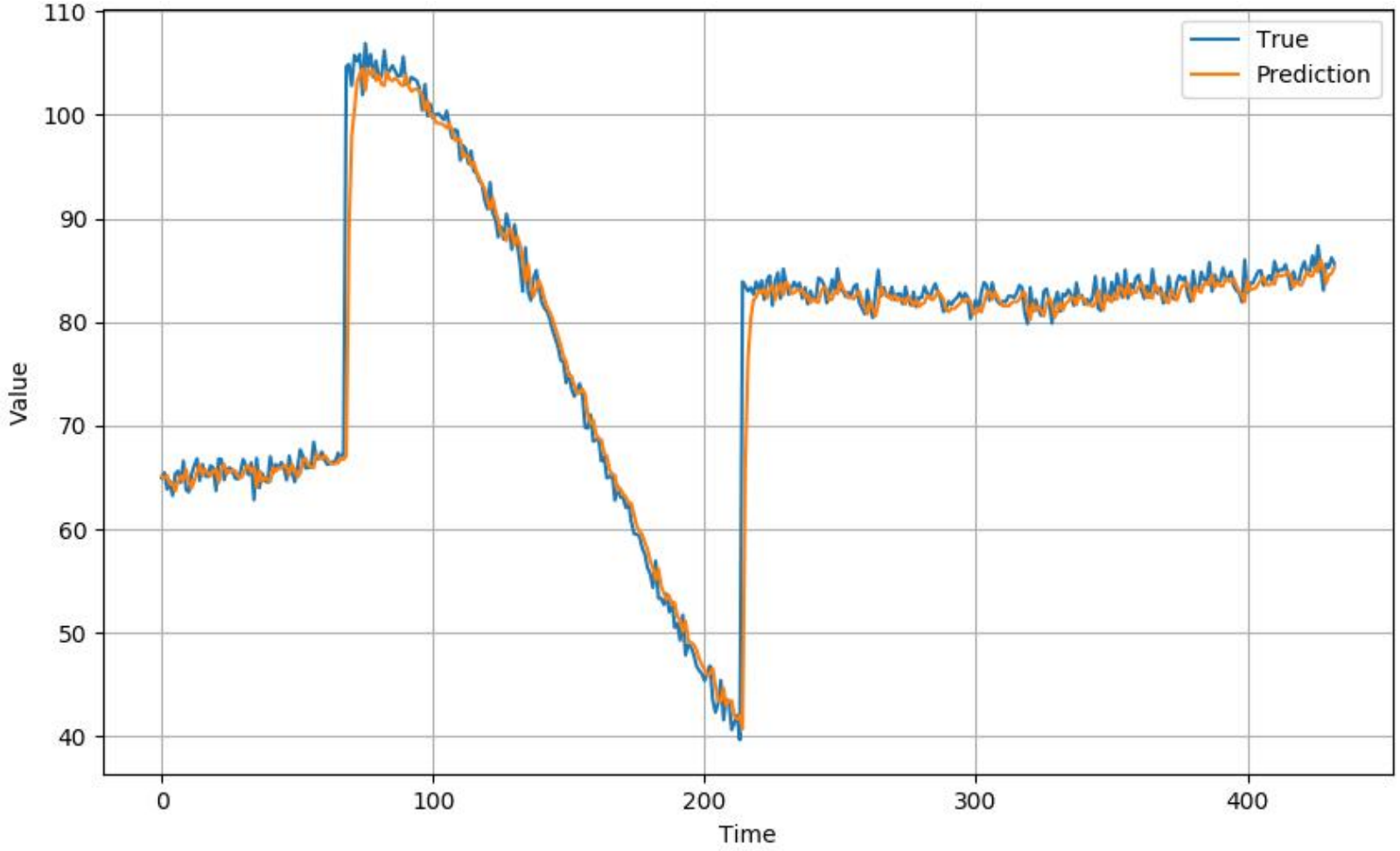
预测并与真实值对比

```
# 指定figure的宽和高
fig_size = plt.rcParams['figure.figsize']
fig_size[0] = 10
fig_size[1] = 6
plt.rcParams['figure.figsize'] = fig_size
# 画出实际和预测的对比图
plt.plot(range(len(test_y_result)), test_y_result, label='True')
plt.plot(range(len(predict_result)), predict_result, label='Prediction')
plt.xlabel("Time")
plt.ylabel("Value")
plt.grid(True)
plt.legend()
plt.show()
# 与整体数据进行比较
plt.plot(range(len(series)), series, label='True')
plt.plot(range(len(series) - len(predict_result), len(series)), predict_result, label='Prediction')
plt.xlabel("Time")
plt.ylabel("Value")
plt.grid(True)
plt.legend()
plt.show()
```

分别在局部和整体对比预测结果与实际值

RNN预测模拟数据

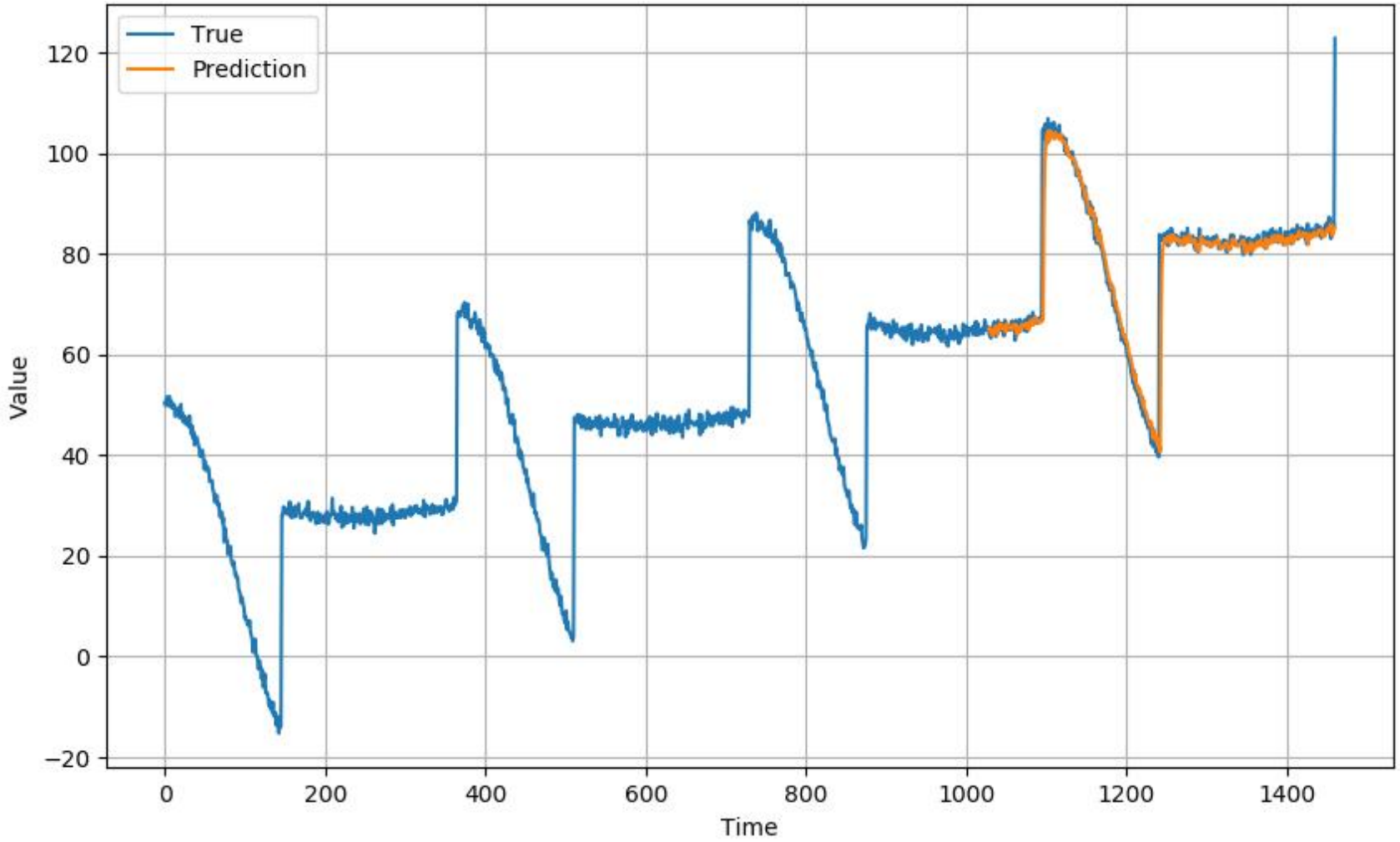
- **预测并与真实值对比**



预测值与真实值对比（局部）

RNN预测模拟数据

- **预测并与真实值对比**



预测值与真实值对比（全局）

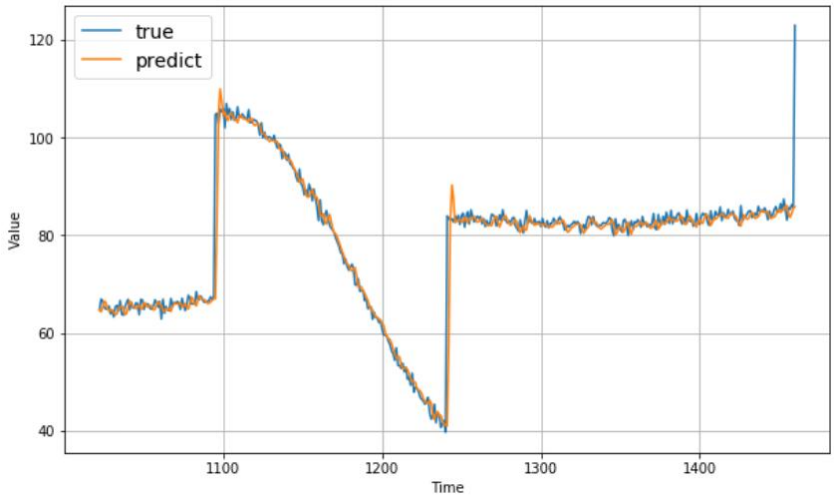
RNN预测模拟数据

与CNN对比

两层卷积神经网络（平均池化）

```
from sklearn.metrics import mean_absolute_error as mae
mae_nn=mae(test_true, test_predict)
print(mae_nn)
```

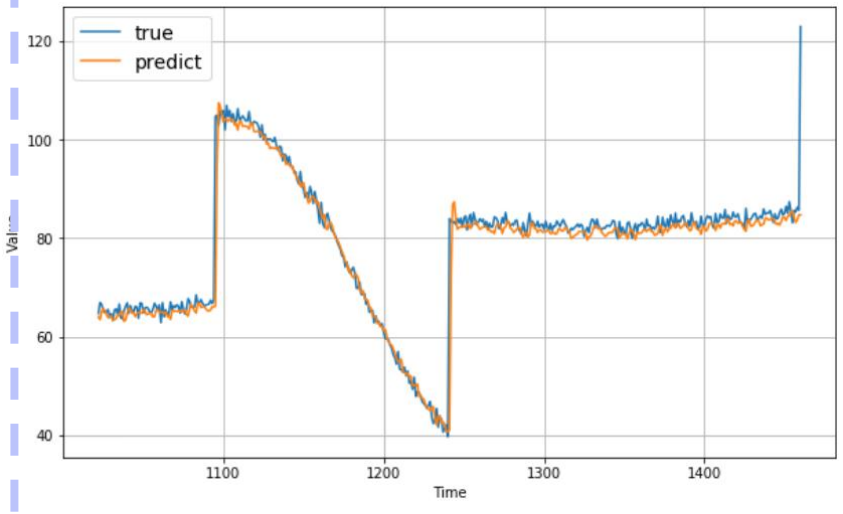
```
1.4410519066307985
```



两层卷积神经网络（最大池化）

```
from sklearn.metrics import mean_absolute_error as mae
mae_nn=mae(test_true, test_predict)
print(mae_nn)
```

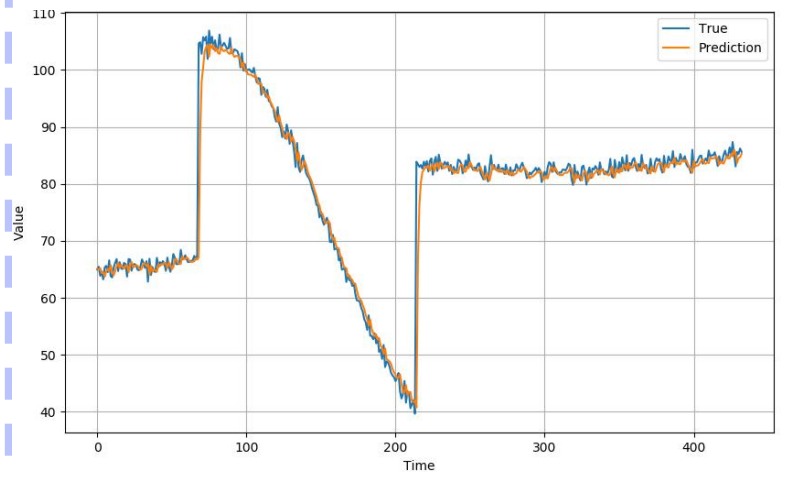
```
1.4921763508994446
```



循环神经网络（单向，tanh）

```
from sklearn.metrics import mean_absolute_error as mae
mae_nn=mae(test_true, test_predict)
print(mae_nn)
```

```
1.3555869
```



目录

LSTM预测气温

回顾

LSTM：Long Short Term Memory networks

- 一种特殊形式的RNN
- 解决长程依赖问题

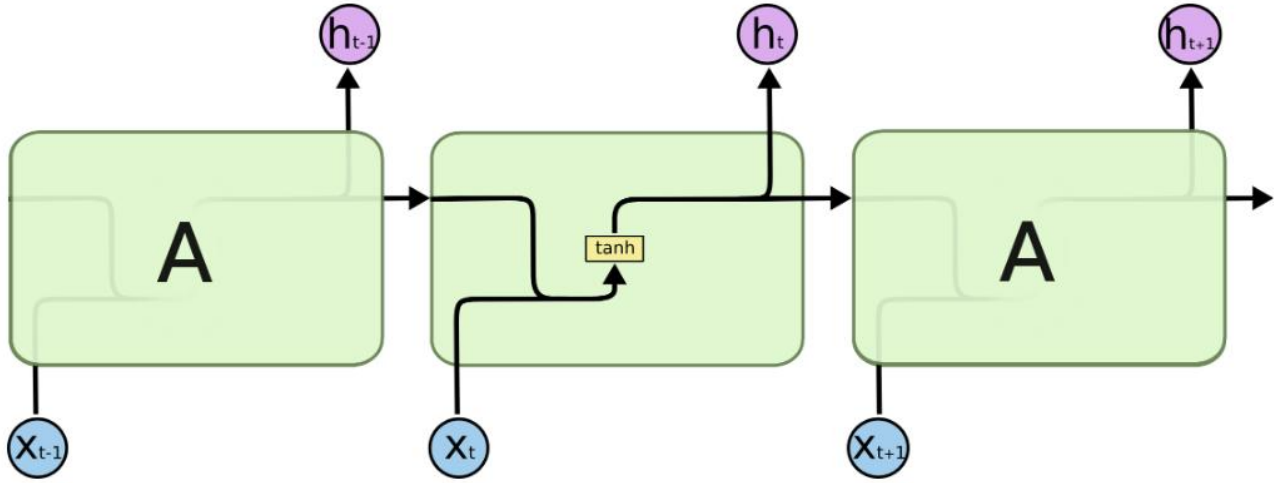
短程依赖，普通的RNN能够解决

长程依赖，借助LSTM解决

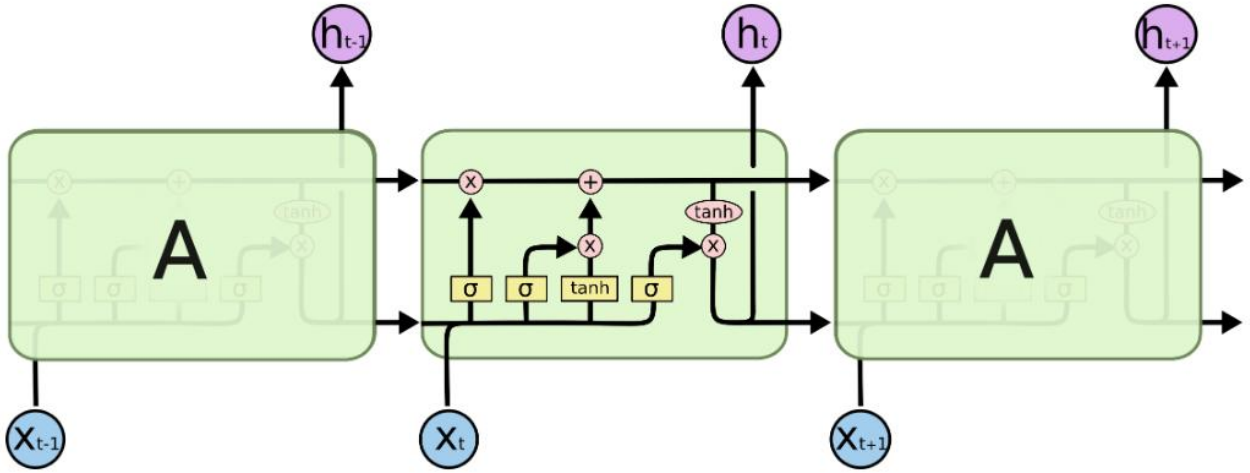
LSTM预测气温

- **回顾**

标准RNN的重复模块



LSTM 的重复模块



除了h在随时间流动，单元状态c也在随时间流动，单元状态c就代表着长期记忆。

LSTM预测气温

torch.nn.LSTM

CLASS torch.nn.LSTM(args, * kwargs)

参数说明:

- **input_size** – 输入x的特征数量。
- **hidden_size** – 隐层的特征数量。
- **num_layers** – LSTM的层数。
- **nonlinearity** – 指定非线性函数使用tanh还是relu。默认是tanh。
- **bias** – 如果是False，那么LSTM层就不会使用偏置权重 b_ih和b_hh,默认是True
- **batch_first** – 如果True的话，那么输入Tensor的shape应该是[batch_size, time_step, feature], 输出也是这样。
- **dropout** – 如果值非零，那么除了最后一层外，其它层的输出都会套上一个dropout层。
- **bidirectional** – 如果True，将会变成一个双向LSTM，默认为False。

LSTM预测气温

torch.nn.LSTM

LSTM输入: input, (h_0, c_0)

- **input (seq_len, batch, input_size):** 包含输入序列特征的Tensor。
- **h_0 (num_layers * num_directions, batch, hidden_size):**保存着batch中每个元素的初始化隐状态的Tensor
- **c_0 (num_layers * num_directions, batch, hidden_size):** 保存着batch中每个元素的初始化细胞状态的Tensor

LSTM输出 output, (h_n, c_n)

- **output (seq_len, batch, hidden_size * num_directions):** 保存RNN最后一层的输出的Tensor
- **h_n (num_layers * num_directions, batch, hidden_size):** Tensor， 保存着RNN最后一个时间步的隐状态。
- **c_n (num_layers * num_directions, batch, hidden_size):** Tensor， 保存着RNN最后一个时间步的细胞状态。

LSTM预测气温

数据处理

从1956年至今的虹桥机场的气象数据，包括**气温**、干球温度、风速风向等等。

STATION	DATE	SOURCE	REPORT_T	CALL_SIG	QUALITY_	AA1	AA2	AA3	AG1	AJ1	AL1	AY1	AY2	CALL_SIG	CIG	DEW	ED1	EQD	GA1	GA2	GA3	GA4	GE1
5.8E+10	1956-08-20T00:00:00	4	FM-12	99999	V020	06,0005,9	24,0050,9,1					8,1,99,9		99999	03600,1,C	+0250,1							
5.8E+10	1956-08-20T03:00:00	4	FM-12	99999	V020							6,1,99,9		99999	01230,1,C	+0250,1							
5.8E+10	1956-08-20T06:00:00	4	FM-12	99999	V020	06,0030,9,1						6,1,99,9		99999	00450,1,C	+0239,1							
5.8E+10	1956-08-20T09:00:00	4	FM-12	99999	V020							2,1,99,9		99999	00150,1,C	+0239,1							
5.8E+10	1956-08-20T12:00:00	4	FM-12	99999	V020	06,0006,9,1						4,1,99,9		99999	00270,1,C	+0228,1		Q01+000	08,1,+00270,9,07,9				
5.8E+10	1956-08-20T18:00:00	4	FM-12	99999	V020	06,0006,9,1						5,1,99,9		99999	22000,1,C	+0222,1		Q01+000002SCOTLCQ02+000042APCTENQ03+000042APC					
5.8E+10	1956-08-20T21:00:00	4	FM-12	99999	V020							2,1,99,9		99999	22000,1,C	+0222,1		Q01+000002SCOTLCQ02+000012APCTEN					
5.8E+10	1956-08-21T00:00:00	4	FM-12	99999	V020	24,0030,9,1						1,1,99,9		99999	22000,1,C	+0222,1		Q01+000042APCTENQ02+000152APC3					
5.8E+10	1956-08-21T03:00:00	4	FM-12	99999	V020							1,1,99,9		99999	00450,1,C	+0222,1							
5.8E+10	1956-08-21T06:00:00	4	FM-12	99999	V020							1,1,99,9		99999	00450,1,C	+0222,1							
5.8E+10	1956-08-21T09:00:00	4	FM-12	99999	V020							1,1,99,9		99999	22000,1,C	+0211,1							
5.8E+10	1956-08-21T12:00:00	4	FM-12	99999	V020							1,1,99,9		99999	22000,1,C	+0228,1		Q01+000042APCTENQ02+000112APC3					

GF1	HL1	IA1	IA2	KA1	KA2	MA1	MD1	ME1	MW1	MW2	MW3	OA1	OC1	QUALITY_REM	REPORT_T	SA1	SLP	SOURCE	TMP	UA1	UG1	VIS	WG1	WND
06,99,1,02,1,01,1,00450,1,04,1,02,1							6,1,005,1,+999,9		01,1					V020	FM-12		10052,1	4	+0278,1			014000,1,N,9		200,1,N,0051,1
08,99,1,08,1,08,1,01250,1,00,1,00,1							0,1,003,1,+999,9		60,1					V020	FM-12		10056,1	4	+0261,1			009000,1,N,9		250,1,N,0041,1
07,99,1,07,1,02,1,00450,1,07,1,07,1				999,N,+0250,1			9,9,015,1,+999,9		21,1					V020	FM-12		10040,1	4	+0272,1			020000,1,N,9		250,1,N,0051,1
07,99,1,07,1,08,1,00150,1,03,1,07,1							6,1,002,1,+999,9		02,1					V020	FM-12		10038,1	4	+0272,1			020000,1,N,9		270,1,N,0062,1
08,99,1,08,1,06,1,00250,1,00,1,00,1							2,1,024,1,+999,9		50,1					V020	FM-12		10062,1	4	+0239,1			014000,1,N,9		270,1,N,0051,1
07,99,1,99,9,00,1,99999,9,05,1,00,1				999,M,+0278,1					03,1					V020	FM-12		10075,1	4	+0228,1			014000,1,N,9		290,1,N,0031,1
07,99,1,99,9,00,1,99999,9,07,1,00,1							4,1,000,1,+999,9		02,1					V020	FM-12		10075,1	4	+0228,1			016000,1,N,9		320,1,N,0031,1
02,99,1,01,1,08,1,00450,1,04,1,00,1							2,1,015,1,+999,9		03,1					V020	FM-12		10090,1	4	+0250,1			020000,1,N,9		320,1,N,0031,1
05,99,1,05,1,01,1,00450,1,04,1,00,1							0,1,004,1,+999,9		02,1					V020	FM-12		10093,1	4	+0278,1			020000,1,N,9		290,1,N,0041,1
06,99,1,06,1,02,1,00450,1,00,1,00,1				999,N,+0228,1			6,1,011,1,+999,9		02,1					V020	FM-12		10082,1	4	+0289,1			020000,1,N,9		270,1,N,0051,1
02,99,1,02,1,04,1,00450,1,00,1,00,1							6,1,002,1,+999,9		01,1					V020	FM-12		10080,1	4	+0289,1			020000,1,N,9		320,1,N,0021,1
00,99,1,00,1,00,1,99999,9,00,1,00,1							2,1,012,1,+999,9		04,1					V020	FM-12		10092,1	4	+0250,1			009000,1,N,9		090,1,N,0010,1

LSTM预测气温

数据处理

```
data = pd.read_csv(path + '\气温.csv', index_col='DATE', na_values='+9999,9')
data = data['TMP']
data.index = pd.to_datetime(data.index)
start_time = pd.to_datetime('2019-01-01 00:00:00')
end_time = pd.to_datetime('2019-06-30 23:00:00')
data = data[start_time:end_time]
data=data.dropna()
data = data.str.split(",", expand=True)[0]
data = data.astype('int')/10
index=pd.date_range(start=start_time,end=end_time, freq='H')
data = data.reindex(index)
data = data.interpolate()
```

1. 获取气温数据；

2. 将index设置为时间格式，截取数据片段

3. 丢弃TMP的NaN数据；

4. 获取气温数值；

5. 补全DATE，设置间隔；

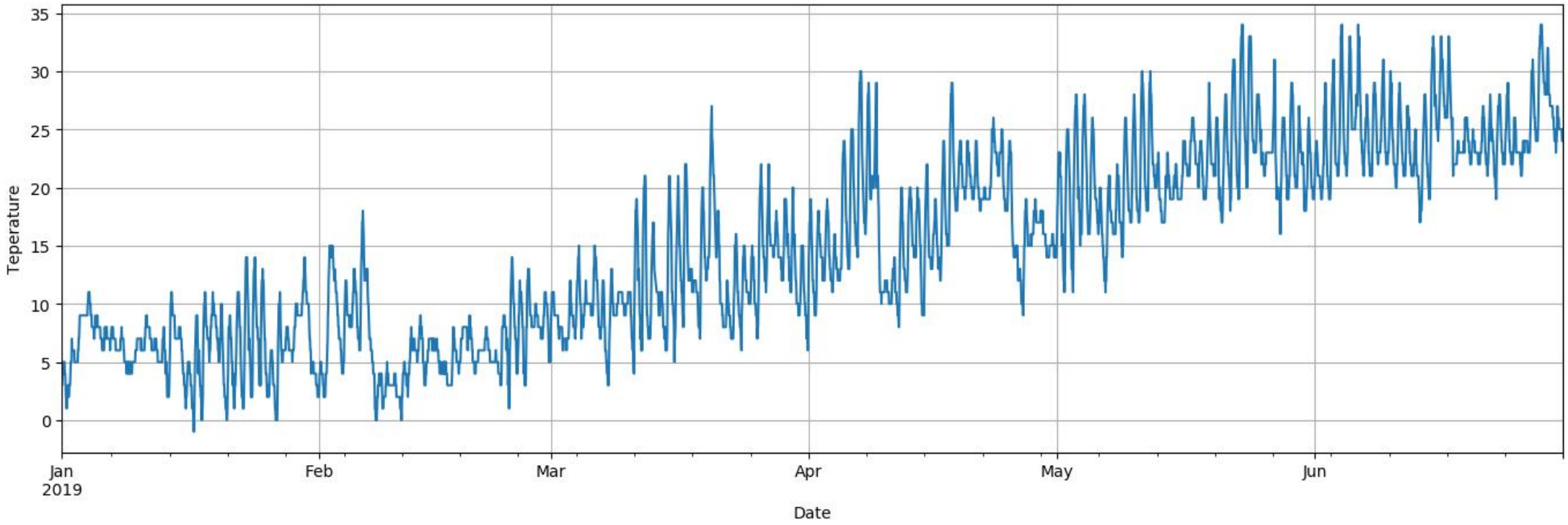
6. 对NaN插值；

LSTM预测气温

显示数据

```
data.plot()
plt.xlabel("Date")
plt.ylabel("Teperature")
plt.grid(True)
plt.show()
```

画出气温数据



LSTM预测气温

参数设置

```
# 设置超参数
input_size = 1
hidden_size = 128
output_size = 1
epochs = 100
lr = 0.05
batch_size = 20
time_step = 12
```

- **input_size**:输入数据的特征数量，时序数据为1
- **hidden_size**:隐藏层的特征个数
- **output_size**:输出的时间窗口的长度
- **epochs**：训练轮数
- **lr**:学习率
- **batch_size**：一个批量的大小
- **time_step**:输入的时间窗口的长度

LSTM预测气温

划分数据集

```
# 前140天用作训练
train_data = data[0:140 * 24]
# 剩下的时间用作测试
test_data = data[140 * 24:]
# # 数据归一化
train_data_normalized = (train_data - train_data.min()) / (train_data.max() - train_data.min())
test_data_normalized = (test_data - train_data.min()) / (train_data.max() - train_data.min())
```

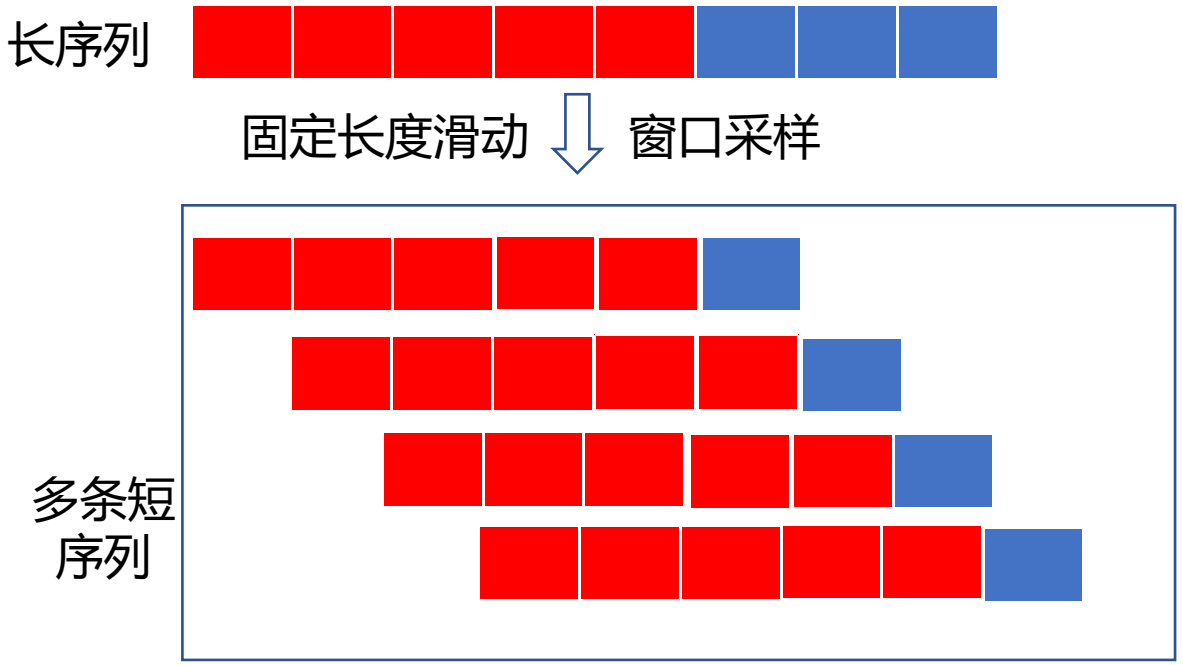
- 一小时记录一次，每天24条记录，故前140天共140*24条记录作为训练集，其余作为测试集。
- 与之前相同，对数据进行归一化

LSTM预测气温

滑动窗口采样

```
train_x = []
train_y = []
test_x = []
test_y = []
# 对训练数据采样
i = 0
while (i + time_step + output_size < len(train_data_normalized)):
    # 输入的序列
    train_x.append(train_data_normalized[i:i + time_step])
    # 输出的序列
    train_y.append(train_data_normalized[i + time_step:i + time_step + output_size])
    i += output_size

# 对测试数据采样
j = 0
while (j + time_step + output_size < len(test_data_normalized)):
    # 输入的序列
    test_x.append(test_data_normalized[j:j + time_step])
    # 输出的序列
    test_y.append(test_data_normalized[j + time_step:j + time_step + output_size])
    j += output_size
```



固定滑动窗口采样示意图

LSTM预测气温

装入数据

```
#将数据转换为tensor格式
train_x = torch.tensor(train_x,dtype=torch.float32)
train_y = torch.tensor(train_y,dtype=torch.float32)
test_x = torch.tensor(test_x,dtype=torch.float32)
test_y = torch.tensor(test_y,dtype=torch.float32)

# 将训练数据装入DataLoader
train_dataset = Data.TensorDataset(train_x, train_y)
train_loader = Data.DataLoader(dataset=train_dataset,
                               batch_size=batch_size,
                               shuffle=True, num_workers=0)
```

将训练集装入DataLoader 中，便于之后的训练过程取出数据

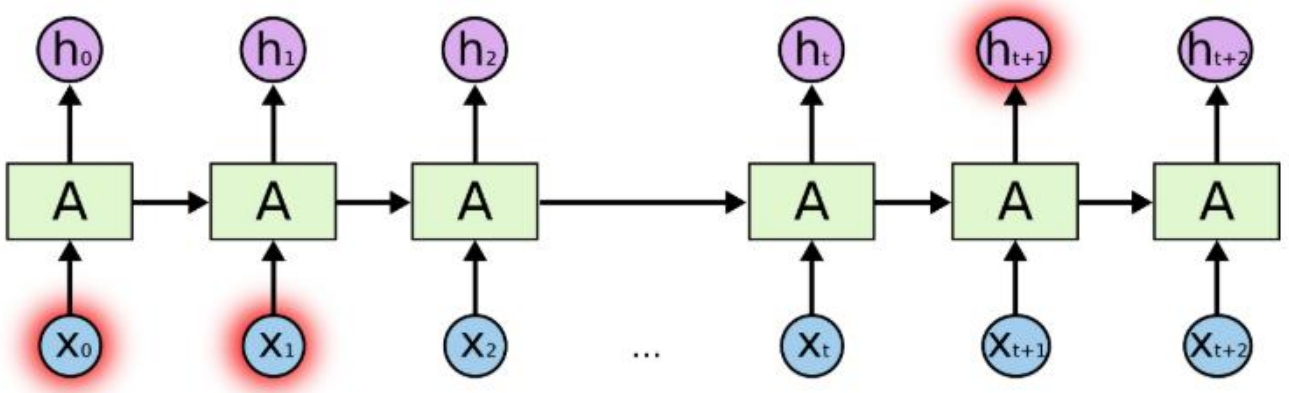
LSTM预测气温

构建LSTM网络

```python
# 构建LSTM网络
class MYLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, time_step):
        super(MYLSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.time_step = time_step

        # 创建LSTM层和linear层，LSTM层提取特征，linear层用作最后的预测
        self.lstm = nn.LSTM(
            input_size=self.input_size,
            hidden_size=self.hidden_size,
            num_layers=1,
            batch_first=True,
            bidirectional=True
        )
        self.out = nn.Linear(self.hidden_size * 2, self.output_size)

    def forward(self, x):
        # 获得LSTM的计算结果，舍去h_n
        r_out, _ = self.lstm(x)
        # #按照LSTM模型结构修改input_seq的形状，作为linear层的输入
        r_out = r_out.reshape(-1, self.hidden_size * 2)
        out = self.out(r_out)
        # 将out恢复成(batch,seq_len,output_size)
        out = out.reshape(-1, self.time_step, self.output_size)
        # return所有batch的seq_len的最后一项
        return out[:, -1, :]
```



LSTM预测气温

模型参数初始化

```
# 实例化神经网络
net = MYLSTM(input_size, hidden_size, output_size, time_step)
# 初始化网络参数
for param in net.parameters():
    nn.init.normal_(param, mean=0, std=0.01)
# 设置损失函数
loss = nn.MSELoss()
# 设置优化器
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
# 如果GPU可用，就用GPU运算;否则使用CPU运算
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# 将net复制到device（GPU或CPU）
net.to(device)
```

- 从**均值为0、标准差为0.01**的**正态分布**中取随机数，初始化网络模型参数
- 将损失函数设置为**均方误差**
- 优化器使用**随机梯度下降法**
- GPU可用则使用GPU运算，否则使用CPU运算

LSTM预测气温

开始训练

```
train_loss = []
test_loss = []
# 开始训练
for epoch in range(epochs):
    train_l = []
    test_l = 0
    for x, y in train_loader:
        # RNN输入应为input (seq_len, batch, input_size)，将x转化为三维数据
        x = torch.unsqueeze(x, dim=2)
        # 将x,y放入device中
        x = x.to(device)
        y = y.to(device)
        # 计算得到预测值y_predict
        y_predict = net(x)
        # 计算y_predict与真实y的loss
        l = loss(y_predict, y)
        # 清空所有被优化过的Variable的梯度.
        optimizer.zero_grad()
        # 反向传播，计算当前梯度
        l.backward()
        # 根据梯度更新网络参数
        optimizer.step()
        train_l.append(l.item())
    # 修改测试集的维度以便放入网络中
    test_x_temp = torch.unsqueeze(test_x, dim=2)
    # 测试集放入device中
    test_x_temp = test_x_temp.to(device)
    test_y_temp = test_y.to(device)
    # 得到测试集的预测结果
    test_predict = net(test_x_temp)
    # 计算测试集loss
    test_l = loss(test_predict, test_y_temp)
    # 打印
    print("Epoch%d:train loss=%.5f,test loss=%.5f" % (epoch + 1, np.array(train_l).mean(), test_l.item()))
    train_loss.append(np.array(train_l).mean())
    test_loss.append(test_l.item())
```

完成网络初始化和训练参数的设置后，开始训练。

在每一轮训练中，对每一小批数据计算梯度，反向传播更新参数。

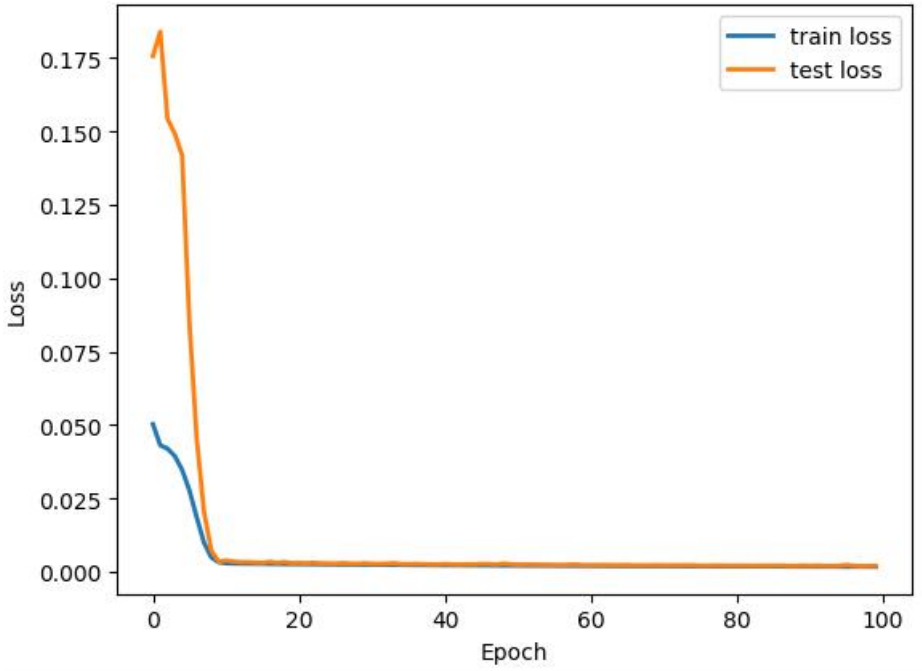
每一轮训练后，计算并输出当前模型在训练集和测试集上的损失。

LSTM预测气温

- 绘制loss曲线

```
# 画出Loss趋势图
plt.plot(range(epochs), train_loss, label="train loss", linewidth=2)
plt.plot(range(epochs), test_loss, label="test loss", linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

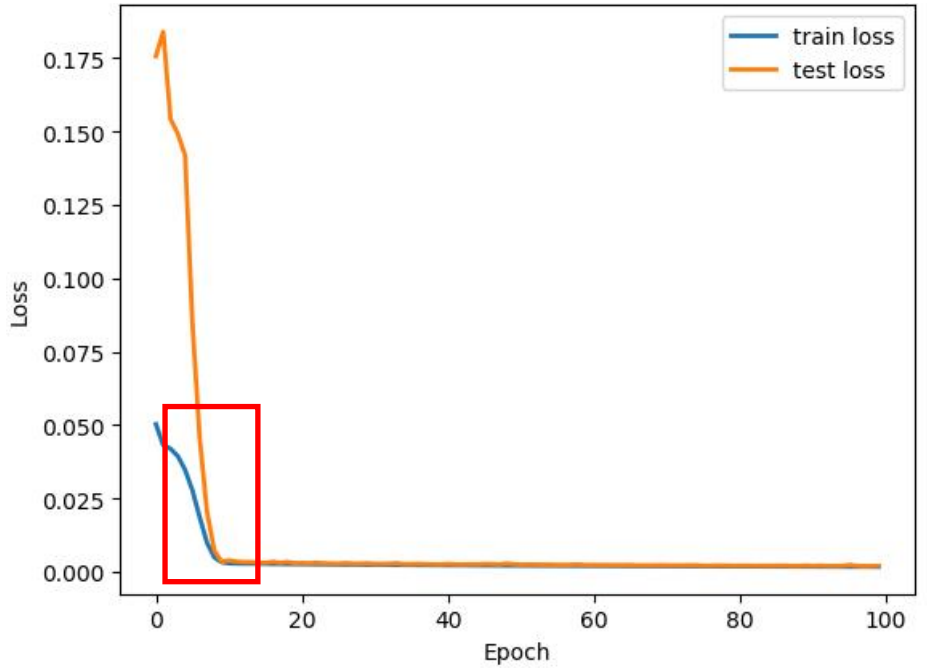
```
Epoch90:train loss=0.00170, test loss=0.00195
Epoch91:train loss=0.00169, test loss=0.00190
Epoch92:train loss=0.00168, test loss=0.00200
Epoch93:train loss=0.00167, test loss=0.00188
Epoch94:train loss=0.00167, test loss=0.00200
Epoch95:train loss=0.00167, test loss=0.00185
Epoch96:train loss=0.00167, test loss=0.00215
Epoch97:train loss=0.00165, test loss=0.00187
Epoch98:train loss=0.00165, test loss=0.00187
Epoch99:train loss=0.00164, test loss=0.00186
Epoch100:train loss=0.00165, test loss=0.00223
```



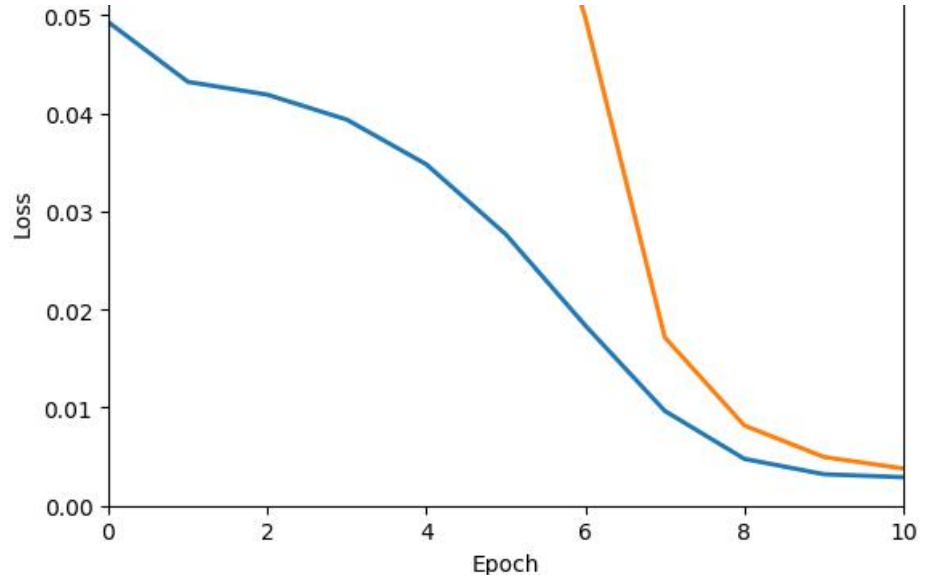
LSTM预测气温

绘制loss曲线

```
# Loss局部图
plt.plot(range(epochs), train_loss, label="train loss", linewidth=2)
plt.plot(range(epochs), test_loss, label="test loss", linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.xlim(0, 10)
plt.ylim(0, 0.06)
plt.legend()
plt.show()
```



绘制局部曲线，观察变化情况



预测并与真实值对比

```
# 将测试集放入模型计算预测结果
test_x_temp = torch.unsqueeze(test_x, dim=2)
test_x_temp = test_x_temp.to(device)
predict = net(test_x_temp)
# 逆归一化
predict = predict.cpu().detach().numpy() * (train_data.max() - train_data.min()) + train_data.min()
test_y = np.array(test_y) * (train_data.max() - train_data.min()) + train_data.min()
# 将数据从[[ouyput_size]...]转换为[x1,x2,x3...]
predict_result = []
test_y_result = []
for item in predict:
    predict_result += list(item)
for item in test_y:
    test_y_result += list(item)
```

- 计算得到预测结果
- 测试集当前值域为[0,1]，需将其逆归一化，映射回原来的值域
- 将结果的维度转换为一维

LSTM预测气温

预测并与真实值对比

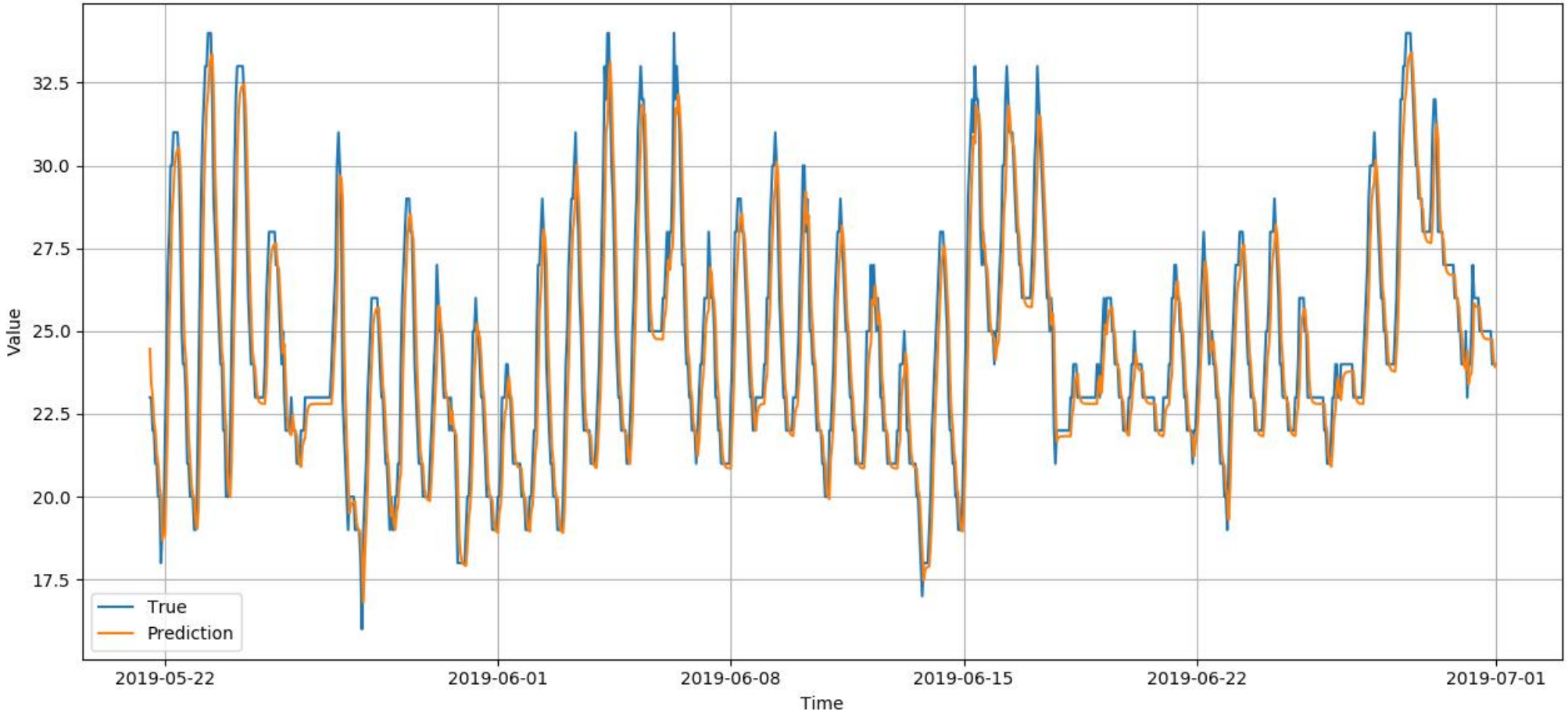
```
# 指定figure的宽和高
fig_size = plt.rcParams['figure.figsize']
fig_size[0] = 10
fig_size[1] = 6
plt.rcParams['figure.figsize'] = fig_size
# 画出实际和预测的对比图
plt.plot(data.index[len(data) - len(test_y_result):], test_y_result, label='True')
plt.plot(data.index[len(data) - len(predict_result):], predict_result, label='Prediction')
plt.xlabel("Time")
plt.ylabel("Value")
plt.grid(True)
plt.legend()
plt.show()

# 与整体数据进行比较
plt.plot(data.index, data, label='True')
plt.plot(data.index[len(data) - len(predict_result):], predict_result, label='Prediction')
plt.xlabel("Time")
plt.ylabel("Value")
plt.grid(True)
plt.legend()
plt.show()
```

分别在局部和整体对比预测结果与实际值

LSTM预测气温

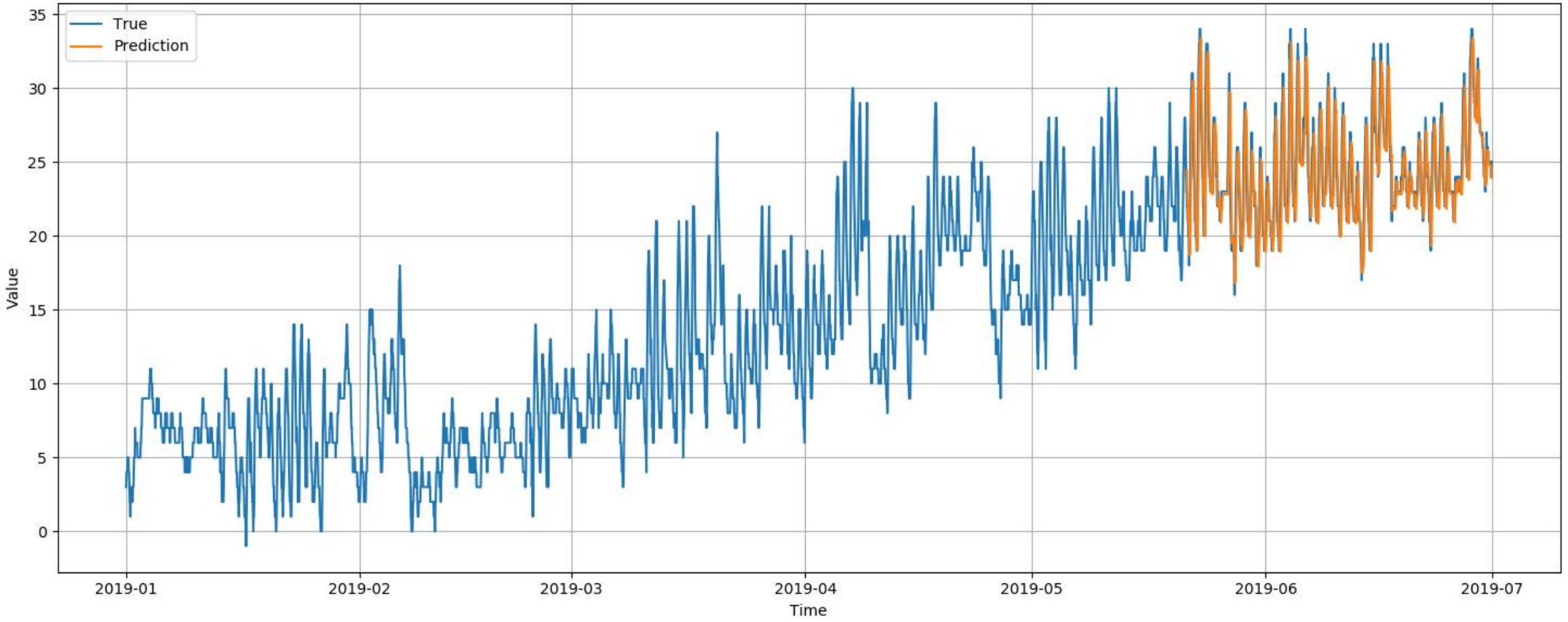
预测并与真实值对比



预测值与真实值对比（局部）

LSTM预测气温

预测并与真实值对比



预测值与真实值对比（全局）

目 录

实验一：循环神经网络实验

1. 生成模拟数据，对模拟数据构建RNN进行训练，绘制loss曲线；用训练好的RNN进行预测，并与真实值进行对比。
2. 针对气象数据，构建LSTM（**复现**PPT中网络）进行训练，绘制loss曲线；尝试使用**不同的预测长度**（例如：用过去12个小时的数据预测未来12个小时的气温）；

实验二：设计神经网络实验

1. 针对气象数据，设计一个与PPT中不同的网络进行预测（预测效果优于PPT中的网络可加分），对优化的思路进行总结和分析。(要求：使用pytorch)
2. 与上两次实验的结果进行对比，简要分析各个模型的优缺点。

作业要求

- 完成作业1+作业2
- 作业截至时间：10月26日16:00
- 作业提交内容：实验报告jupyter notebook格式，要求写注释（注释单独算分）

北京交通大学《时间序列数据分析挖掘》课程组

赵守国：shgzhao@bjtu.edu.cn, http://faculty.bjtu.edu.cn/7563/

王　晶：wj@bjtu.edu.cn, http://faculty.bjtu.edu.cn/9167/

夏佳楠：xiajn@bjtu.edu.cn , http://faculty.bjtu.edu.cn/9430/