北京交通大学《时间序列数据分析挖掘》课程组

实验3 时序数据生成&神经网络

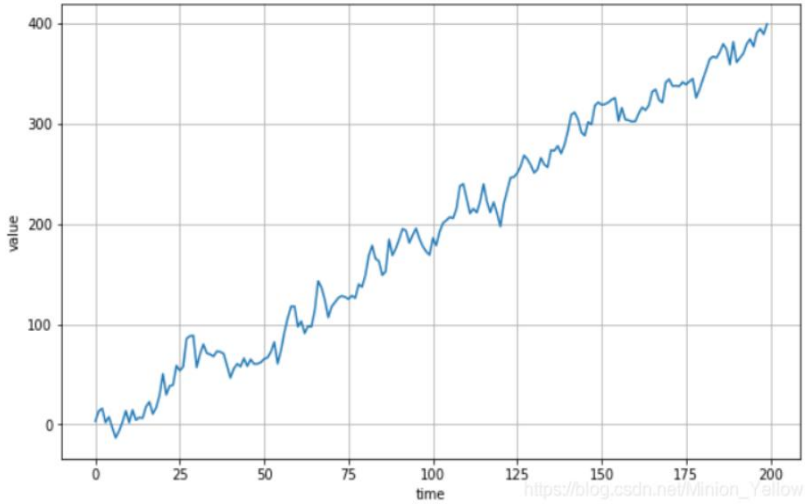
目录

时间序列数据模式

趋势

时间序列的走势在较长时期内沿着某一方向持续发展变化

受某种基本因素的影响，数据随时间变化时表现为一种确定倾向，按某种规则稳步地增长或下降



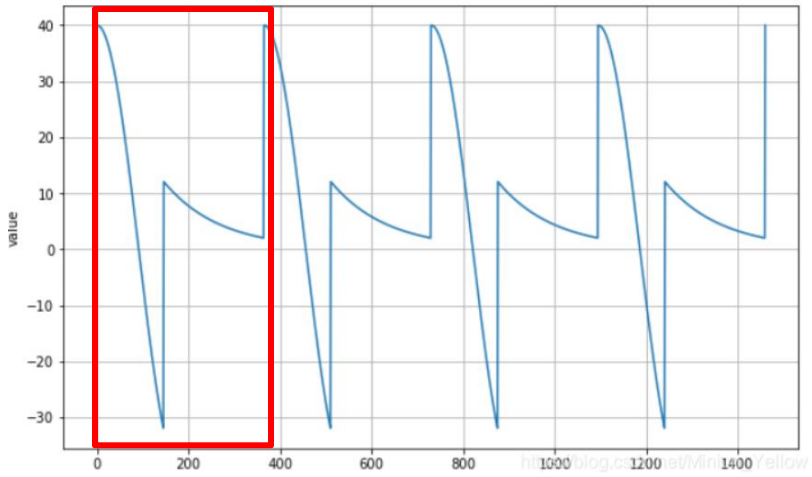
上升趋势

下降趋势

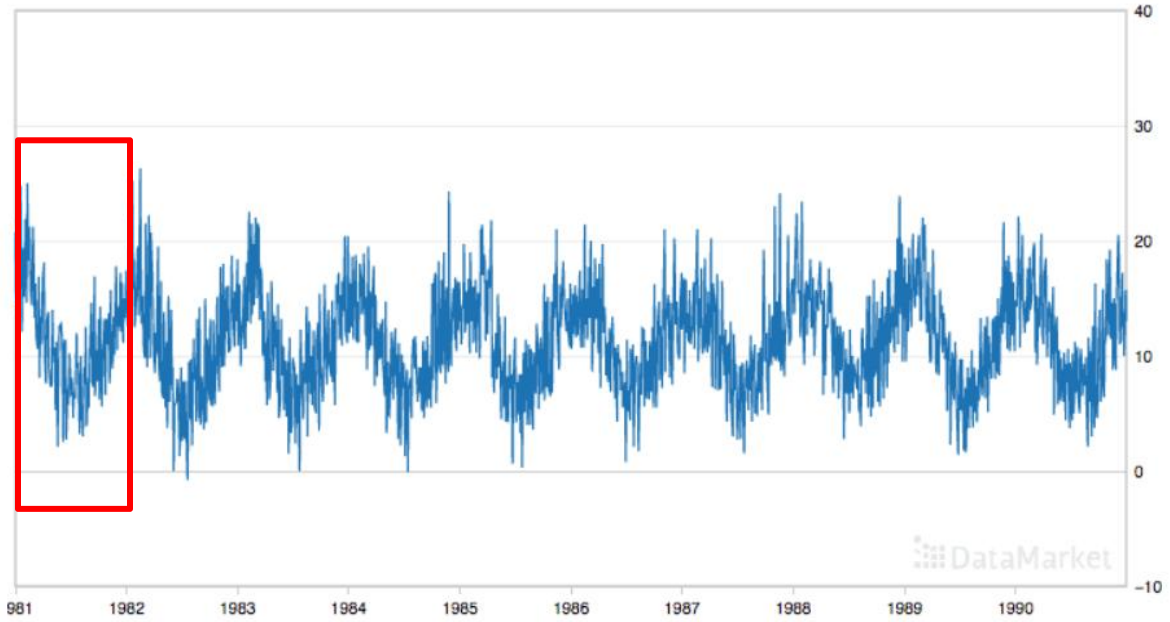
时间序列数据模式

季节性

时间序列数据呈现出按照一段时间（日、周、月、年）重复和循环的周期现象，这些数据往往与上个周期同期的观测数据存在很强的依赖



季节性



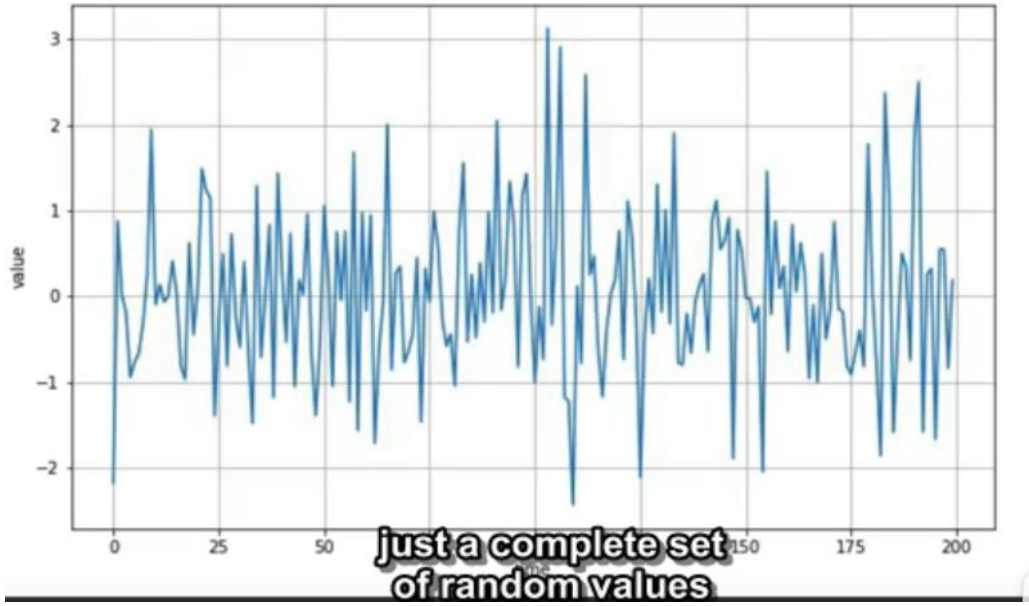
季节性

时间序列数据模式

白噪声

一系列独立分布的正态序列；

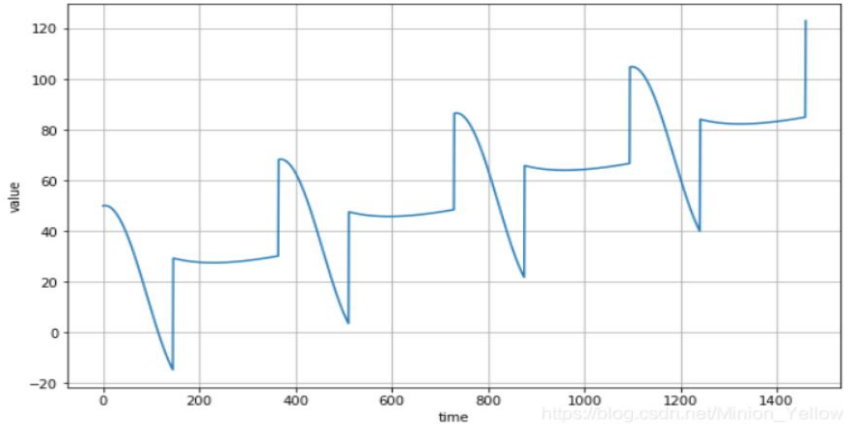
序列无相关性，无趋势性，有随机性，它服从均值为0，方差为σ^2的正态分布



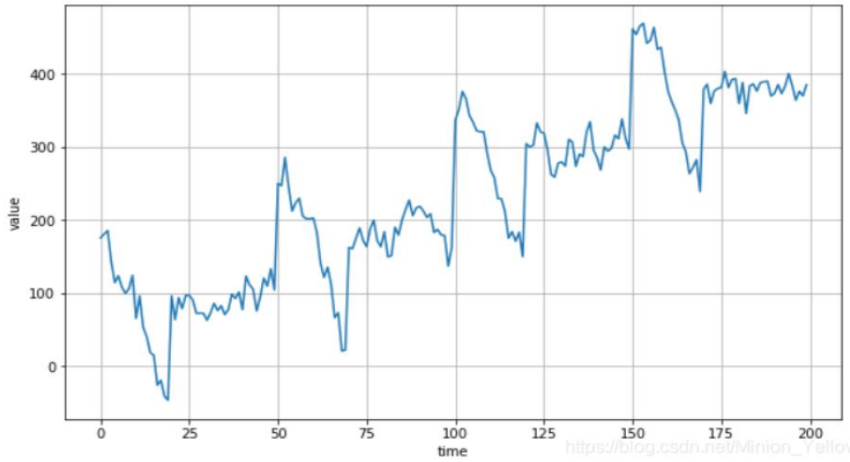
时间序列数据模式

模式叠加

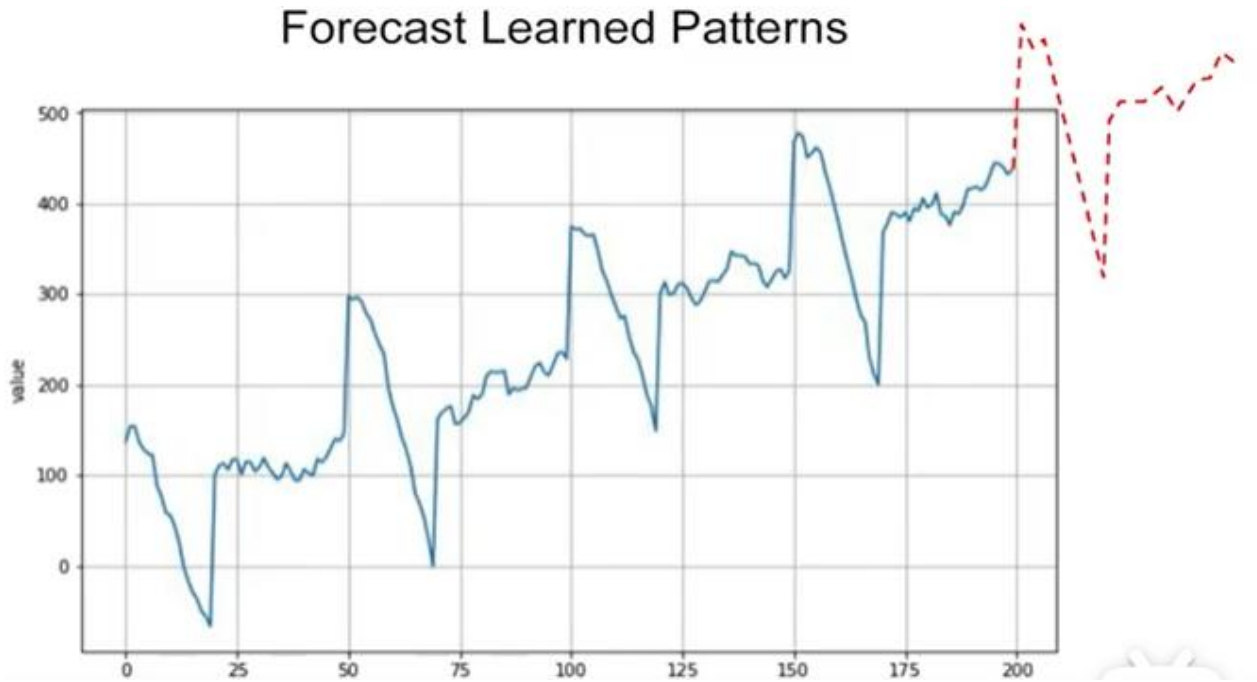
实际应用场景中的时序数据可能结合了多种模式（趋势、季节性、噪声）



趋势+季节性



趋势+季节性+噪声



学习时序数据中的模式用于预测未来值

时序数据生成

绘制序列

1. 导入所需要的包

```
import numpy as np
import torch.nn as nn
import torch
import matplotlib.pyplot as plt
import random
torch.set_default_tensor_type(torch.DoubleTensor)
```

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；
将tensor的默认类型设置为 torch.doubletensor，便于反向传播

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

时序数据生成

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

时序数据生成

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)
```

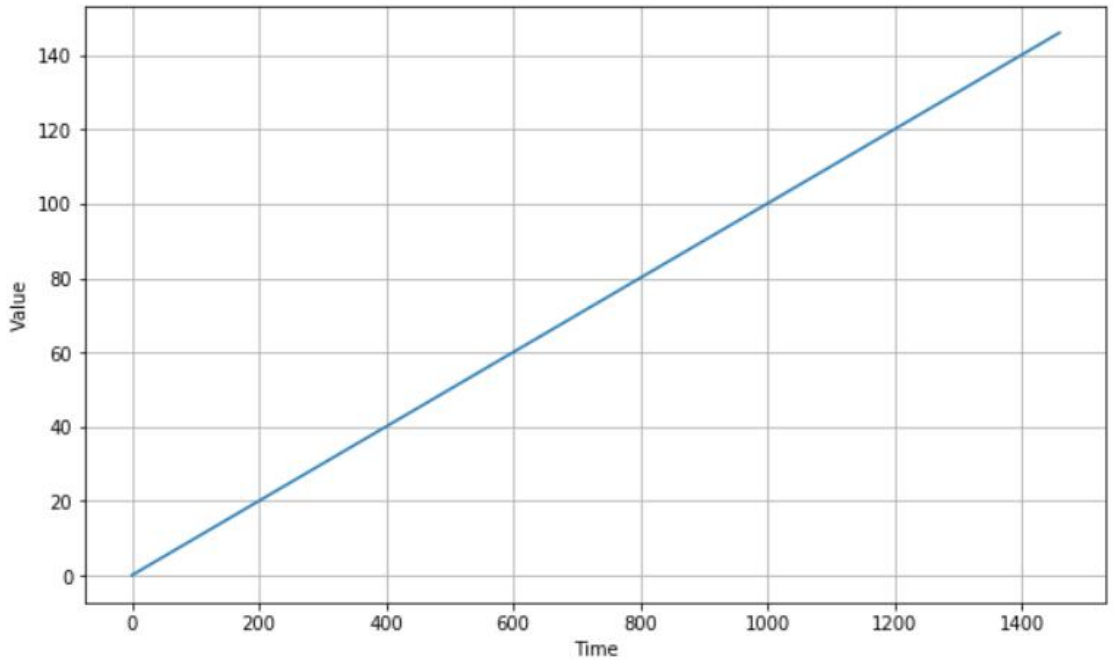
季节性的实现包括两步，第一步构造一个周期内的序列，第二步实现该序列的循环，用amplitude控制幅值大小

时序数据生成

上升趋势的生成

调用趋势生成函数，生成指定长度的上升趋势序列并绘制

```
#时间
time = np.arange(4 * 365 + 1)
#只包含上升趋势的序列
series = trend(time, 0.1)
#设置画布大小
plt.figure(figsize=(10, 6))
#根据time series绘制序列
plot_series(time, series)
plt.show()
```



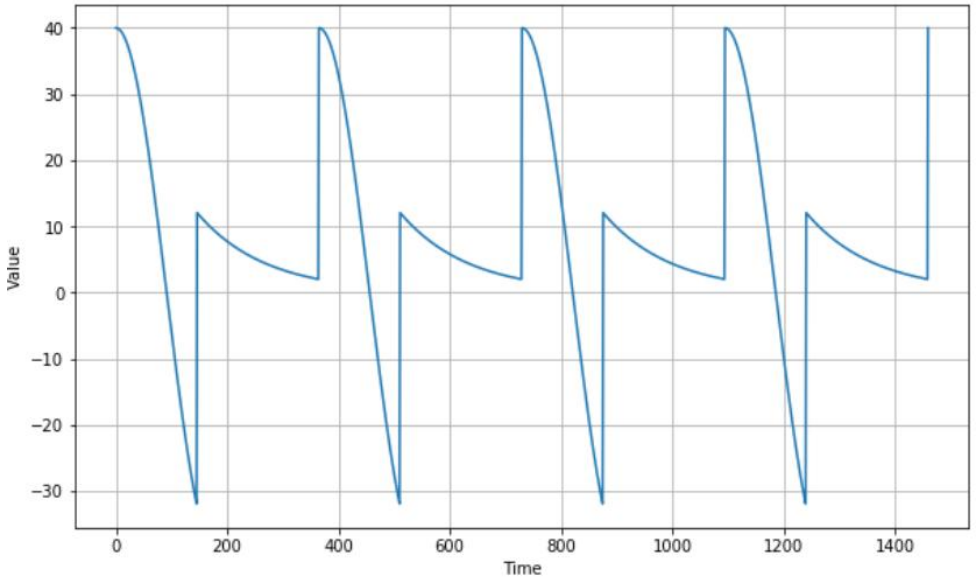
上升趋势

时序数据生成

季节性的生成

调用季节性生成函数、绘制序列函数

```
#设置季节性幅值
amplitude = 40
#生成季节性序列
series = seasonality(time, period=365, amplitude=amplitude)
#绘制序列
plt.figure(figsize=(10, 6))
plot_series(time, series)
plt.show()
```



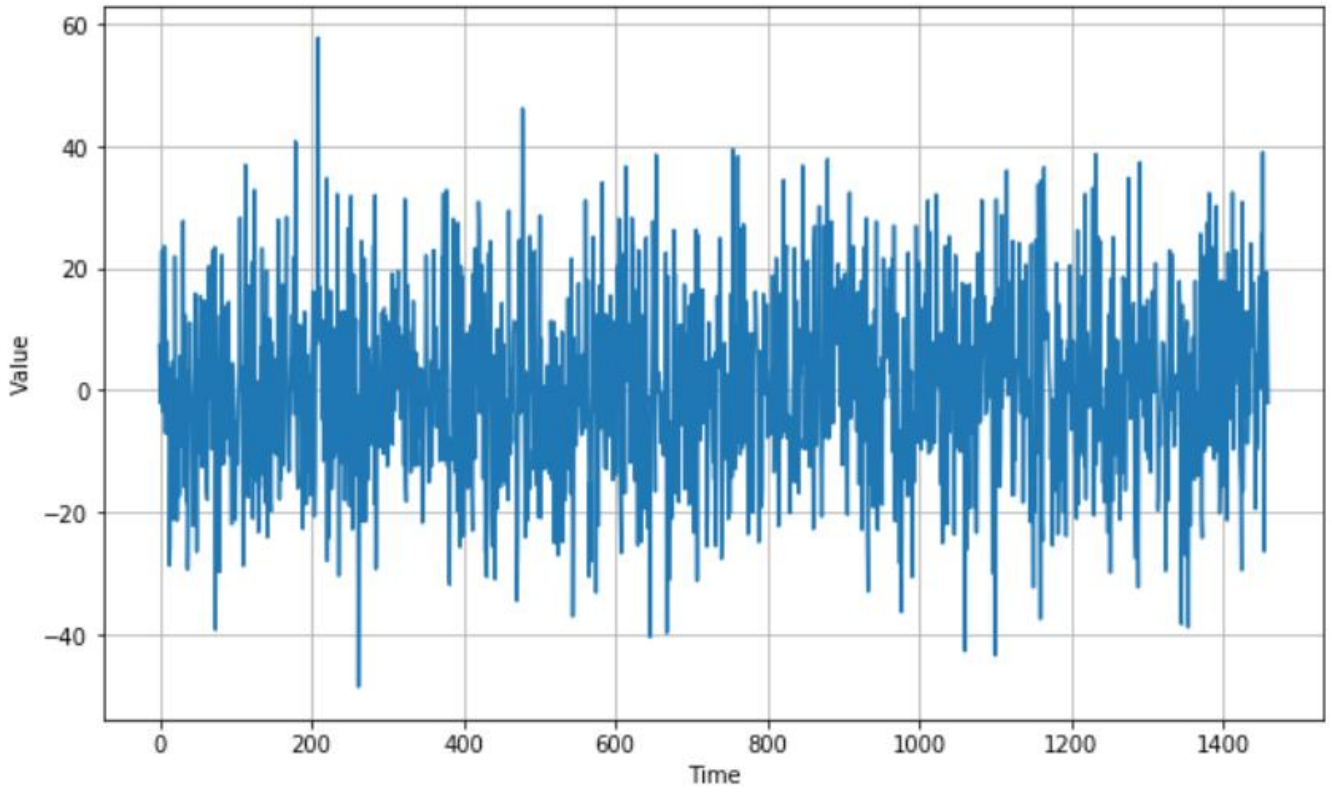
季节性

时序数据生成

白噪声的生成

调用噪声生成函数、绘制序列函数

```
#设置噪声幅值
noise_level = 15
#生成噪声序列
noise = white_noise(time, noise_level, seed=42)
#绘制序列
plt.figure(figsize=(10, 6))
plot_series(time, noise)
plt.show()
```



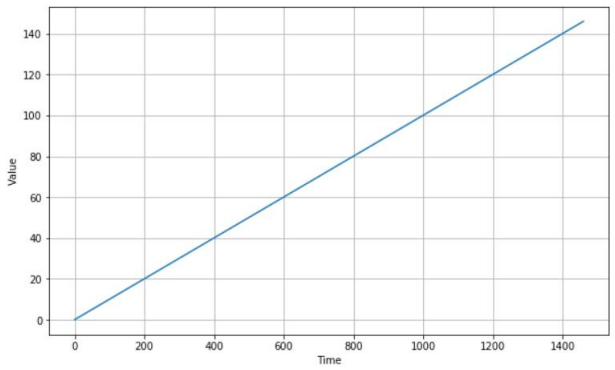
白噪声

时序数据生成

趋势+季节性的叠加

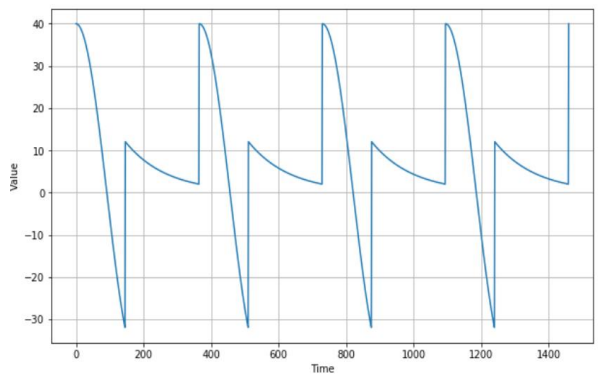
```
#基础序列+上升趋势+季节性
slope = 0.05
baseline=10
series = baseline + trend(time, slope) + seasonality(time, period=365, amplitude=amplitude)
#绘制序列
plt.figure(figsize=(10, 6))
plot_series(time, series)
plt.show()
```

生成单独的模式后，将不同模式的序列相加，实现模式的叠加，并调用绘制序列函数进行可视化



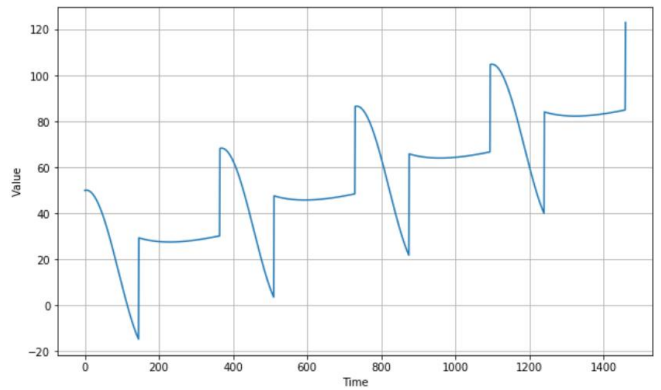
趋势

+



季节性

=



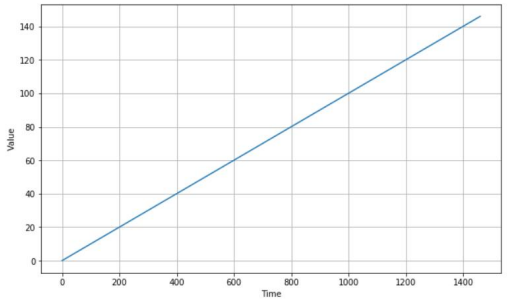
模式叠加

时序数据生成

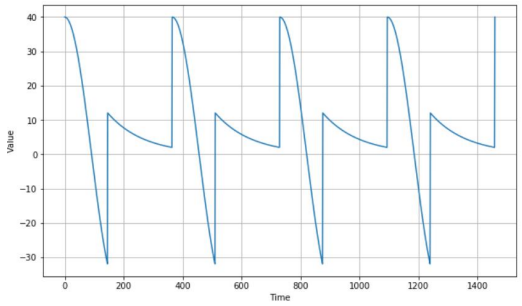
趋势、季节性和白噪声的叠加

```
#基础序列+上升趋势+季节性+白噪声
series += noise
#绘制序列
plt.figure(figsize=(10, 6))
plot_series(time, series)
plt.show()
```

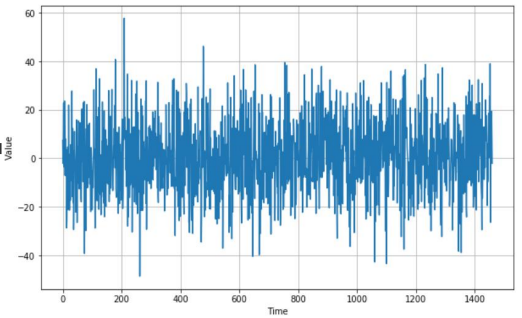
在上一个序列（趋势+季节性）的基础上加入噪声，通过调整noise_level来调整噪声幅值的大小



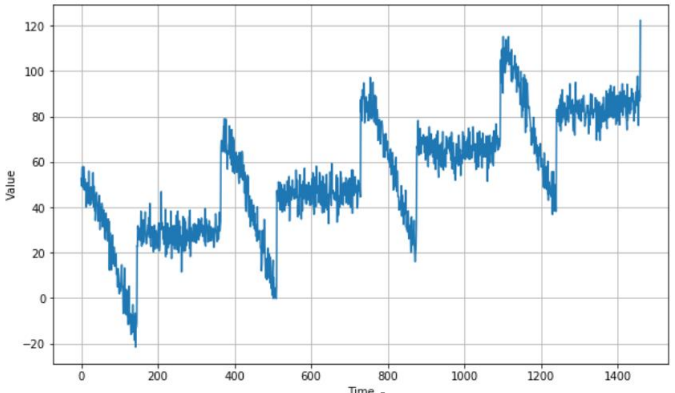
趋势



季节性



白噪声



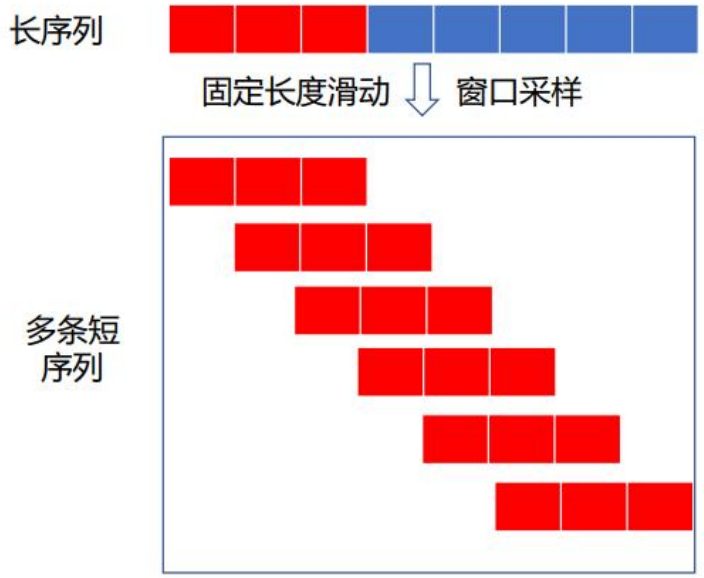
模式叠加

目 录

数据处理

滑窗、划分特征与标签

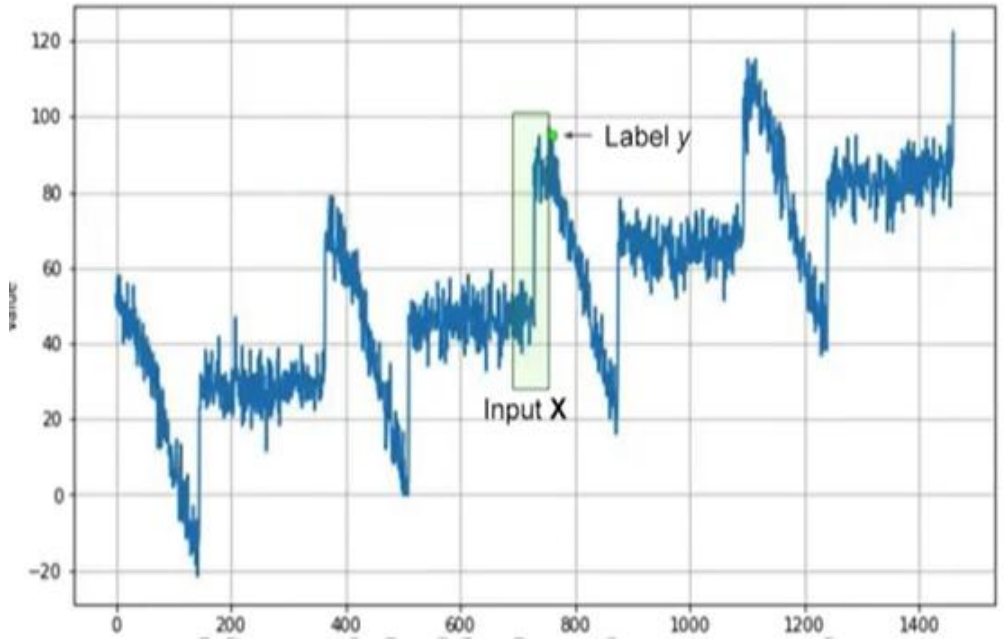
时序数据滑窗采样



固定滑动窗口采样示意图

滑窗大小为window_size，
将长序列采样为多条短序列

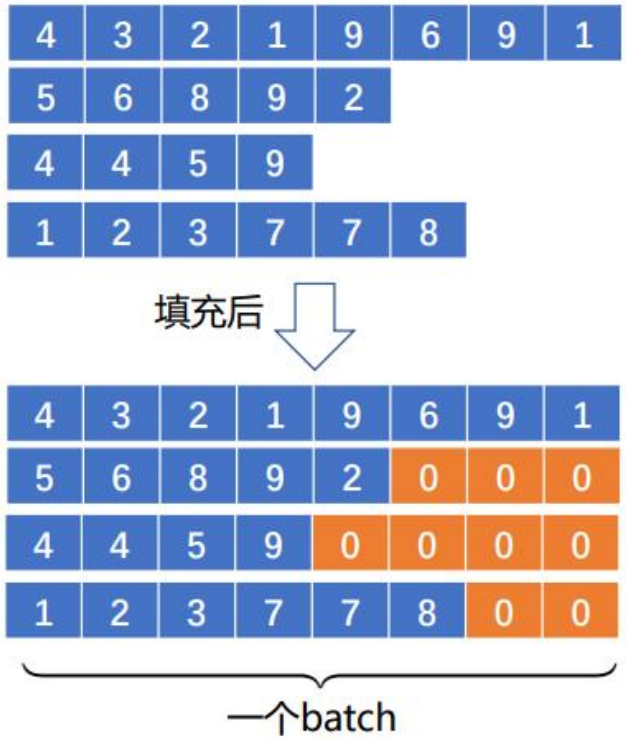
时序数据特征/标签划分



针对每条短序列，将前面的的(window_size-1)个值看作feature，将最后一个值看作label。

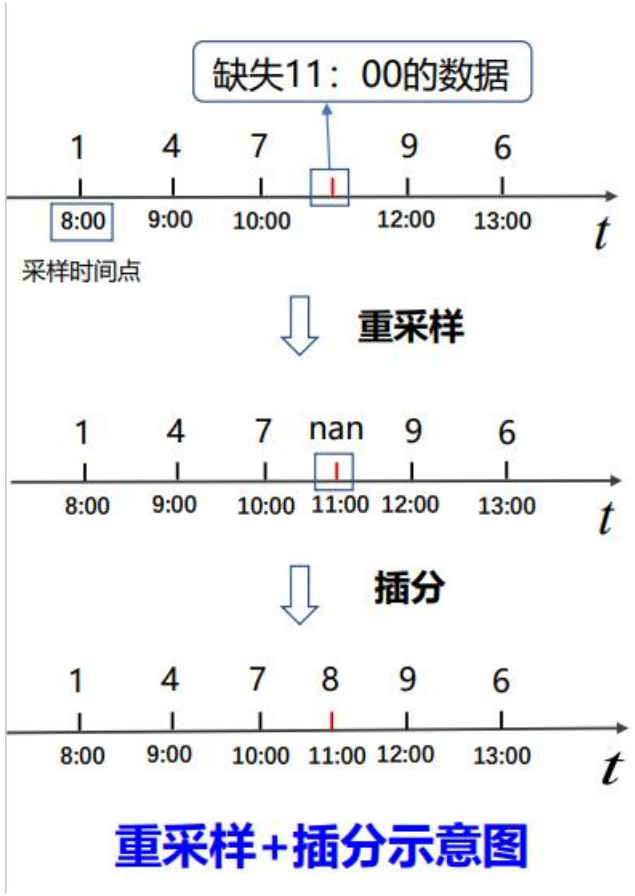
不等长序列与缺失值的处理

不等长序列



序列填充示意图

缺失值处理



重采样+插分示意图

训练集、验证集、测试集划分方法

先划分原始长序列，再采样短序列

梯度下降法

- **批量**梯度下降（Batch Gradient Descent，BGD）

  在每次迭代中，对**所有**样本求梯度，开销较大。

- **随机**梯度下降（Stochastic Gradient Descent，SGD）

  随机采样一个样本来更新参数，大大降低了计算开销。

- **小批量**梯度下降（mini-batch Gradient Descent）

SGD提高了计算效率，但是随机性比较大，下降过程比较曲折，效率也相应降低。所以mini-batch GD采取了一个**折中**的方法，每次选取一定数目(mini-batch)的样本组成一个小批量样本，用这个小批量来更新梯度，在减少计算成本的同时提高稳定性。

数据处理

Batch_size

- **批量**梯度下降（Batch Gradient Descent，BGD）

Batch_size为训练集样本总数；实际上未分批，基于整个数据集得到梯度，梯度准确，但数据量大时，计算非常耗时，同时神经网络常是非凸的，网络最终可能收敛到初始点附近的局部最优点。

- **随机**梯度下降（Stochastic Gradient Descent，SGD）

Batch_size=1；每次计算一个样本，梯度不准确，所以学习率要降低。

- **小批量**梯度下降（mini-batch Gradient Descent）

选择合适的Batch_size；一定程度上缓解了GD算法掉进初始点附近的局部最优值的问题，同时梯度准确了，学习率要加大

数据处理

1. 划分训练集、测试集函数

```
#指定比例 划分训练集和测试集
def train_test_split(series,split_prop):
    train=series[:int(split_prop*int(series.size))]
    test=series[int(split_prop*int(series.size)):]
    return train,test
```

2. 滑窗、划分特征/标签函数

```
def data_process(train,test,window_size):
    # 将数据转为tensor并滑窗，得到短序列
    train_tensor=torch.from_numpy(train)
    train_window_split=train_tensor.unfold(0,window_size,1)
    train_set=train_window_split.numpy()

    test_tensor=torch.from_numpy(test)
    test_window_split=test_tensor.unfold(0,window_size,1)
    test_set=test_window_split.numpy()

    # 将训练集短序列打乱
    train_temp1=train_set.tolist()
    random.shuffle(train_temp1)
    train_temp2=np.array(train_temp1)

    # 将短序列划分为feature和label
    train_feature_array=train_temp2[:,:window_size-1]
    train_label_array=train_temp2[:,window_size-1:]
    test_feature_array=test_set[:,:window_size-1]
    test_label_array=test_set[:,window_size-1:]

    # 将ndarray转为tensor
    train_feature=torch.from_numpy(train_feature_array)
    train_label_temp=torch.from_numpy(train_label_array)
    test_feature=torch.from_numpy(test_feature_array)
    test_label_temp=torch.from_numpy(test_label_array)

    # 将label张量降维
    train_label=train_label_temp.squeeze()
    test_label=test_label_temp.squeeze()

    return train_feature, train_label, test_feature, test_label
```

3. 划分batch函数

```
#将数据划分为指定大小的batch
def data_iter(batch_size, features, labels):
    num_examples= len(features)
    indices = list(range(num_examples))
    for i in range(0, num_examples, batch_size):
        # 最后一次可能不足一 个batch
        j = torch.LongTensor(indices[i: min(i+ batch_size, num_examples)])
        yield  features.index_select(0, j), labels.index_select(0, j)
```

数据处理

- **4. 设置划分比例进行划分，设置滑窗大小将长序列变为短序列、划分特征与标签等**

```
split_prop=0.7
train, test=train_test_split(series, split_prop)
window_size=6
train_feature, train_label, test_feature, test_label=data_process(train, test, window_size)
print(train_feature.shape)
print(train_label.shape)
print(test_feature.shape)
print(test_label.shape)
```

```
torch.Size([1017, 5])
torch.Size([1017])
torch.Size([434, 5])
torch.Size([434])
```

目录

一般流程

- 数据预处理，训练集测试集划分；
- 设计损失函数；
- 构建神经网络，参数初始化；
- 在训练集上进行训练，反向传播，利用梯度下降法等更新网络参数，达到最小化损失函数的目的；
- 在测试集上进行评估网络性能

神经网络应用于时序回归

1. 生成混合模式的时序数据

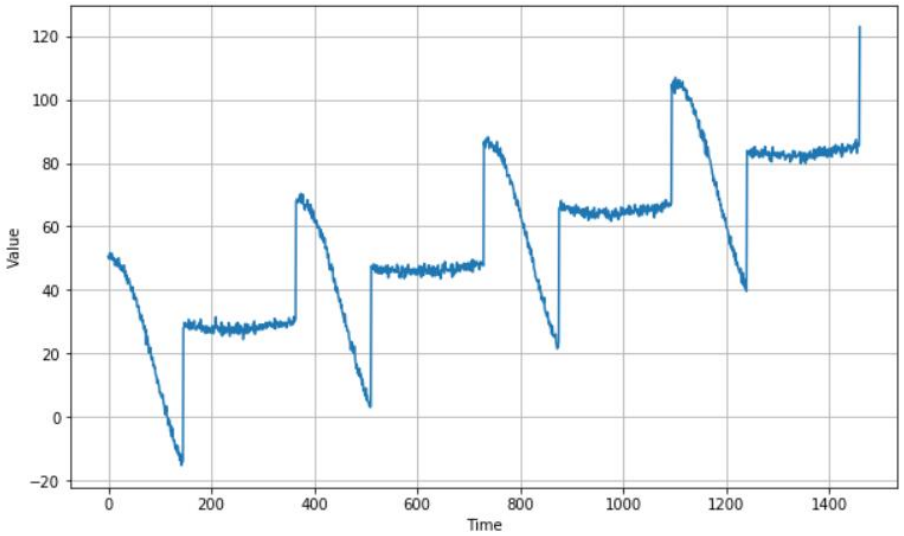
```
#生成序列并绘制图像
time = np.arange(4 * 365 + 1)
baseline = 10
slope = 0.05
amplitude = 40
noise_level = 1
series = baseline + trend(time, slope) + seasonality(time, period=365, amplitude=amplitude)+white_noise(time, noise_level, seed=42)

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

2. 数据处理

```
split_prop=0.7
train,test=train_test_split(series,split_prop)
window_size=6
train_feature,train_label,test_feature,test_label=data_process(train,test,window_size)
print(train_feature.shape)
print(train_label.shape)
print(test_feature.shape)
print(test_label.shape)
```

```
torch.Size([1017, 5])
torch.Size([1017])
torch.Size([434, 5])
torch.Size([434])
```



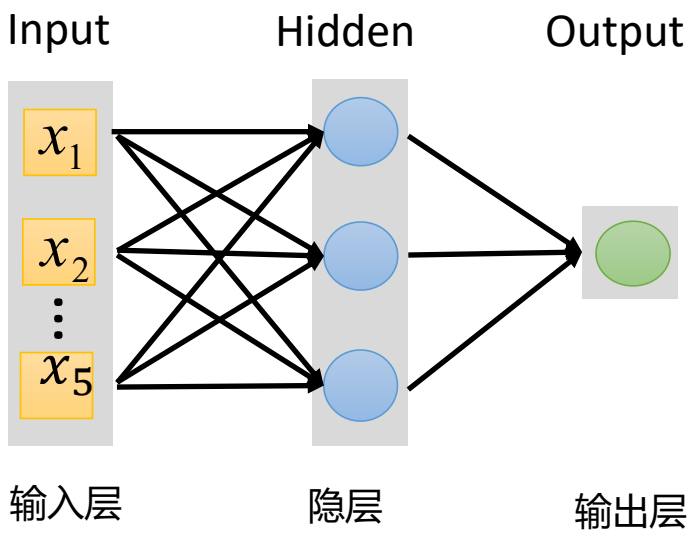
神经网络应用于时序回归

3. 构建三层前馈神经网络

构建网络

```
#实现FlattenLayer层
class FlattenLayer(torch.nn.Module):
    def __init__(self):
        super(FlattenLayer,self).__init__()
    def forward(self,x):
        return x.view(x.shape[0],-1)
#定义模型输入、输出、隐藏层参数
num_inputs,num_outputs,num_hiddens=5,1,3

#定义模型
net=nn.Sequential(
    FlattenLayer(),
    nn.Linear(num_inputs,num_hiddens),
    nn.ReLU(),
    nn.Linear(num_hiddens,num_outputs),
    )
```



3. 构建三层前馈神经网络

损失函数-均方误差

```
#损失函数
def squared_loss(y_hat, y):
    return(y_hat-y.view(y_hat.size()))**2/(y.shape[0])
```

$$loss = (\hat{y} - y)^2$$

参数初始化

```
#参数初始化
for params in net.parameters():
    torch.nn.init.normal_(params,mean=0,std=0.01)

lr= 0.01  #学习率
num_epochs= 200  #训练轮数
batch_size=128  #batch_size大小
loss =squared_loss  #损失函数
optimizer=torch.optim.Adam(net.parameters(),lr)  #设置优化器
```

4. 训练模型

```
train_loss=[]
test_loss=[]

#模型训练
for epoch in range(num_epochs): #外循环训练一轮
    train_l,test_l=0.0,0.0
    for X,y in data_iter(batch_size,train_feature, train_label): #内循环训练一个batch
        y_hat=net(X)  #计算模型输出
        l=loss(y_hat,y).sum()  #计算模型输出与真实数据之间的差距

        #梯度清零
        if optimizer is not None:
            optimizer.zero_grad()
        elif params is not None and params[0].grad is not None:
            for param in params:
                param.grad.data.zero_()
        #反向传播
        l.backward()
        optimizer.step()
    #该轮训练结束后 目前网络在训练集、测试集上的损失 并输出
    train_l= loss(net(train_feature), train_label)
    test_l= loss(net(test_feature), test_label)
    train_loss.append(train_l.mean().item())
    test_loss.append(test_l.mean().item())
    print('epoch %d, train loss %f, test loss %f' % (epoch + 1, train_l.mean().item(),test_l.mean().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

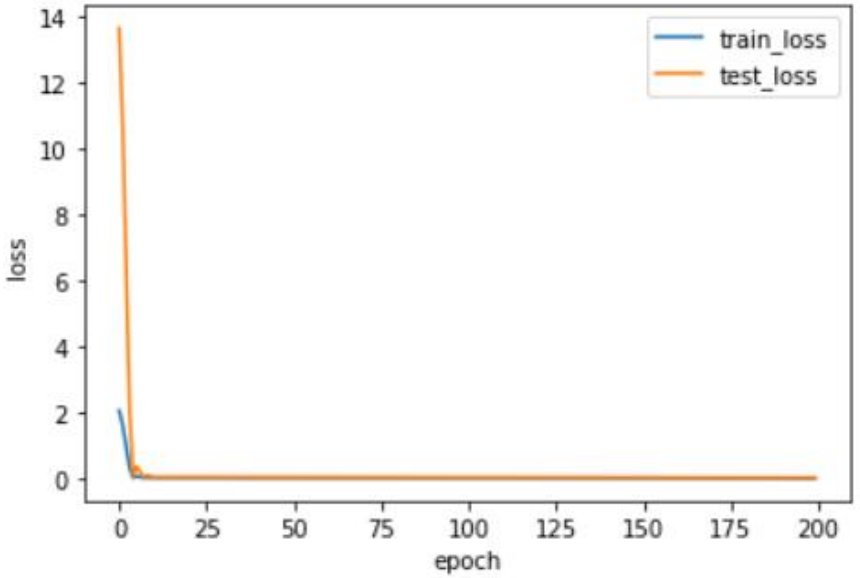
每一轮训练后，计算并输出当前模型在训练集和测试集上的损失

5. 绘制损失函数(loss)曲线

```
#绘制loss曲线
x=np.arange(num_epochs)
plt.plot(x, train_loss, label="train_loss", linewidth=1.5)
plt.plot(x, test_loss, label="test_loss", linewidth=1.5)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.show()
```

训练结束后，绘制训练集、测试集上损失函数随训练轮数的变化情况

```
epoch 195, train loss 0.010497, test loss 0.031537
epoch 196, train loss 0.010481, test loss 0.031512
epoch 197, train loss 0.010466, test loss 0.031488
epoch 198, train loss 0.010451, test loss 0.031465
epoch 199, train loss 0.010436, test loss 0.031443
epoch 200, train loss 0.010423, test loss 0.031421
```

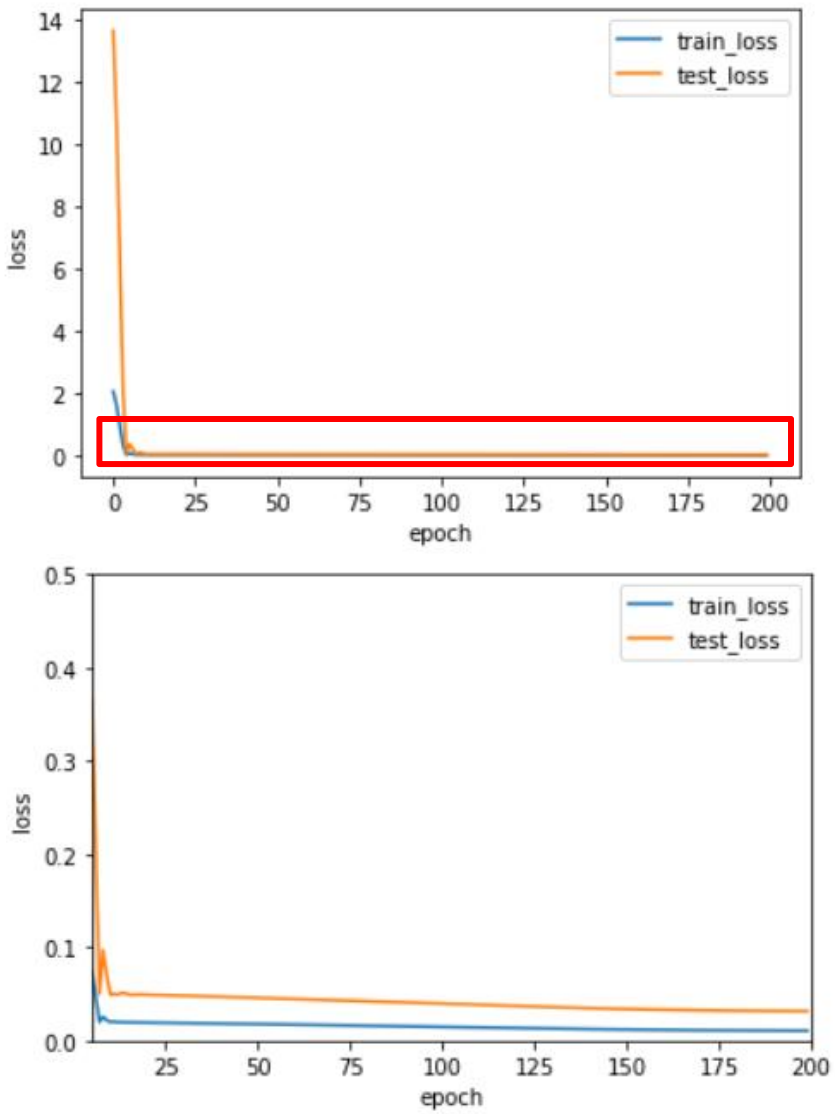


5. 绘制损失函数(loss)曲线

```
#绘制局部loss曲线
x=np.arange(num_epochs)
plt.plot(x, train_loss, label="train_loss", linewidth=1.5)
plt.plot(x, test_loss, label="test_loss", linewidth=1.5)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.xlim(5, num_epochs)
plt.ylim(0, 0.5)
plt.show()
```

- 该范围根据自己跑的值设定

绘制局部曲线，观察变化情况



5.预测并对比

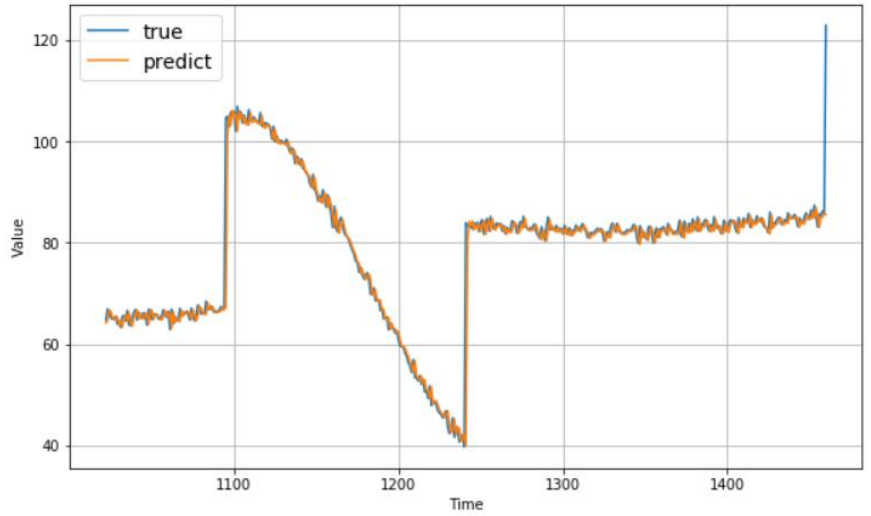
```
#预测函数 利用训练好的网络在测试集上进行测试、评估
def predict(x):
    temp=torch.from_numpy(x)
    x_tensor=temp.reshape(1,1,window_size-1)
    return net(x_tensor)
```

用训练好的模型在测试集上用预测函数实现单值预测；

想要预测某时刻的值，将该时刻前面(window_size-1)个真实值作为网络输入，网络输出即为预测值；

将真实序列与预测序列用不同颜色绘制在同一张图里，评价网络性能

```
test_predict=[]
split_point=int(split_prop*int(series.size))
# 针对测试集中的数据，按滑窗大小取feature，利用训练好的网络进行预测
# 得到该feature对应的label
for i in range((split_point-(window_size-1)),(series.size-window_size+1)):
    x=series[i:i+window_size-1]
    y=predict(x)
    y=y.detach().numpy().squeeze()
    test_predict.append(y)
# 将测试集真实数据与网络预测得到的数据以不同颜色画在一张图里，便于对比
test_time=time[split_point:]
plt.figure(figsize=(10,6))
plot_series(test_time, test)
plot_series(test_time, test_predict)
plt.show()
```



目 录

实验要求：实验内容

作业1 时序数据生成&神经网络 复现

数据生成

实现时序各种模式（趋势、季节性、噪声）的函数；生成这三种模式叠加的序列并可视化

数据准备

将数据划分为训练集、测试集；划分为feature、label；划分batch

神经网络预测

构建神经网络，利用训练数据进行训练，输出训练过程中loss变化及曲线；用训练好的网络在测试集上进行测试，将测试结果与真实数据进行比照

实验要求：实验内容

作业2 机器学习用于真实时间序列预测

数据集： 乙醇数据（alcohol.csv，单变量数据）

数据划分

将数据预处理，然后划分为训练集、测试集（7：3，窗口大小为13，12预测1）；划分为feature、label；划分batch

神经网络预测

构建神经网络，利用训练数据进行训练，输出训练过程中loss变化及曲线；用训练好的网络在测试集上进行测试，将测试结果与真实数据进行比照

探寻不同学习率对预测性能的影响

设置学习率为**三种**不同的值（如0.1，0.01，0.001）。绘制预测曲线与真实值的对比图，计算mse指标。在实验分析中对测试结果进行对比和总结。

机器学习模型预测

构建一种机器学习模型（如决策树回归，支持向量回归 (SVR)），利用训练数据进行训练，用训练好的模型在测试集上进行测试，将测试结果与真实数据进行比照

实验要求：实验内容

作业2 机器学习用于真实时间序列预测

机器学习模型预测

构建一种机器学习模型（如决策树回归，支持向量回归 (SVR)），利用训练数据进行训练，用训练好的模型在测试集上进行测试，将测试结果与真实数据进行比照

机器学习库：sklearn

如：决策回归树模型

from sklearn.tree import DecisionTreeRegressor

```
# 创建决策树回归器模型
dt_regressor = DecisionTreeRegressor(random_state=2)

# 训练模型
dt_regressor.fit(train_feature, train_label)

# 在测试集上进行预测
y_pred = dt_regressor.predict(test_feature)
```

请注意，普通机器学习模型没有bacth一说

实验要求：实验内容

作业3 机器学习用于时间序列分类

数据集：SelfRegulationSCP1

6通道896序列长度的EEG(脑电图数据)时间序列

训练集： 268个样本，测试集293样本，二分类：积极与消极

提取每个样本的特征空间

对**每个样本**的**每个通道**的时间序列进行时域，频域，时频域特征的提取

时域特征：均值，标准差，变异系数，能量，一阶差分平均绝对误差，k阶中心矩，k阶原点矩等等

熵类特征：近似熵，样本熵，模糊熵等等

频域特征：EEG常见频带的能量，频谱熵，频率峰值等等

时频特征：小波变换 分量能量等，短时傅里叶，经验模态分解等等

每类特征至少包含1-2个，不局限于以上特征

采用机器学习算法分类

采用**神经网络（PPT）**与至少**两种机器学习模型**（如支持向量机，决策树，KNN等等）进行分类实验，分析与比较不同模型在相同特征下的分类结果，分析相同模型在不同特征组合下的分类结果

实验要求：实验内容

作业3 机器学习用于时间序列分类

数据集：SelfRegulationSCP1

6通道896序列长度的EEG(脑电图数据)时间序列

训练集： 268个样本，测试集293样本，二分类：积极与消极

采样频率为256hz,3.5秒时间：256*3.5=896

X_train.npy，y_train.npy (训练集的特征与训练集的标签)

X_test.npy，y_test.npy(测试集的特征与测试集的标签)

例如：

X_train为（268，896，6）

提取特征空间的X_train为（268，num_feature）

神经网络分类问题：

损失函数为二元交叉熵损失

loss改为：torch.nn.BCELoss

最后隐藏层输出大小：n_class

机器分类问题：

```
# 创建一个决策树分类器
clf = DecisionTreeClassifier()
# 使用训练数据拟合模型
clf.fit(X_train, y_train)
# 使用训练好的模型进行预测
y_pred = clf.predict(X_test)
# 计算准确率
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

实验问题

明确实验要求和题目数量

1.PPT复现代码依然需要提交，明确不同数据集的不同实验要求

2.不同作业分开写在不同的文件

预测后续10个步长进行预测值和真实值的比较：

1.有些同学没有预测未来10步

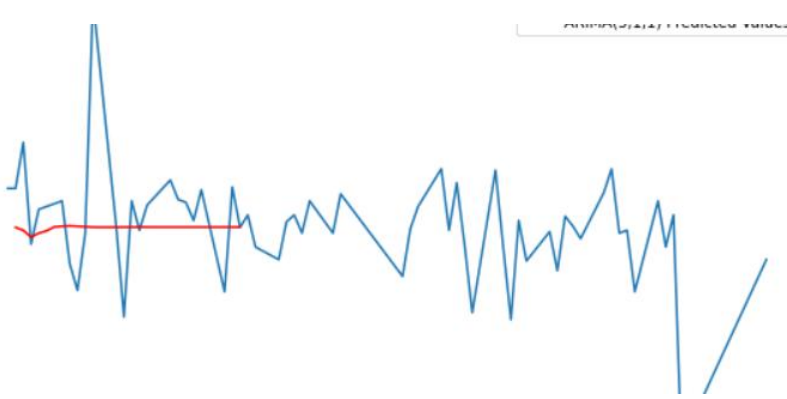
2.有些同学没有比较真实值（因为未来数据不清楚，可以将原始数据，分段，一部分进行拟合训练，未来一部分进行比较）

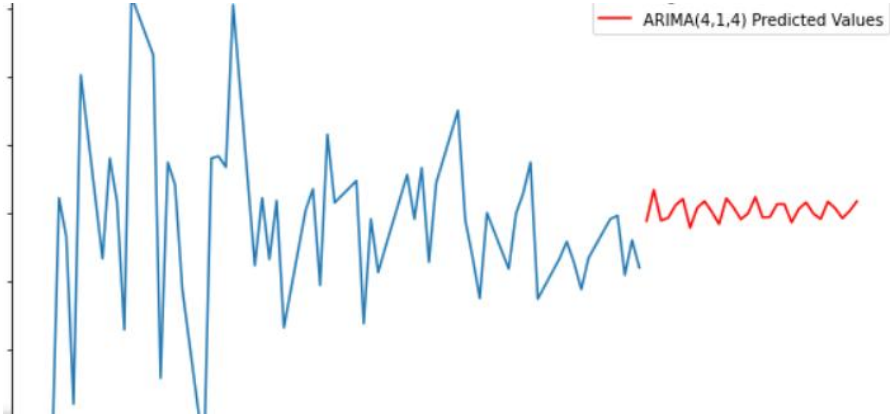
3.部分同学没有进行不同参数的比较：

实验问题

arima定阶

```
for ari in range (0, 3):
    for maj in range (0, 3):
        # 尝试不同p、q的ARMA模型，计算其AIC值print
        try:
            arma_obj=smtsa.ARIMA(zgpa_train_diff['close'].tolist(), order=(ari, 1, maj)).fit()
            print("p: ", ari)
            print("q: ", maj)
            print("AIC: ", arma_obj.aic, "\n")
            aicVal.append([ari, maj, arma_obj.aic])
        except Exception as e:
            print(e)
```





实验要求

- 完成作业1+作业2+作业3
- 作业截至时间：10月11日 16：00
- 作业提交内容：实验报告jupyter notebook格式，要求写注释（关键操作解释，实验结果对比分析等）
- 压缩包命名格式：实验3-学号-姓名