# Python用GRU模型预测比特币价格时间序列数据2案例可视化|附代码数据

门控循环单元（GRU）是一种循环神经网络（RNN）类型，旨在有效地捕获序列数据中的长期依赖关系。它是传统RNN的扩展，与长短期记忆（LSTM）网络具有相似性。

我们将简要了解GRU模型以及如何在PyTorch中使用GRU实现序列数据预测。

**GRU简介**

GRU背后的核心思想是解决梯度消失问题，并提高RNN在长序列中保留信息的能力。GRU通过引入门控机制来实现这一点，这些机制调节了网络内部信息的流动。

典型的GRU单元包括以下组件：

* **更新门**：确定要保留多少过去的信息以及允许多少新信息通过。它基于当前时间步的输入和先前的隐藏状态进行计算。
* **重置门**：控制应该忽略哪些过去的隐藏状态部分。它的计算方式与更新门类似。
* **候选激活**：在考虑重置门的情况下，根据当前输入和先前的隐藏状态计算新的候选激活。
* **隐藏状态**：将候选激活与更新门结合，以产生当前的隐藏状态。

更新门和重置门允许模型选择性地更新或忽略来自先前时间步的信息，从而解决了梯度消失问题，并有助于捕获长期依赖关系。

# 案例1：序列数据预测

**数据准备**

在PyTorch中，我们将使用GRU模型实现序列数据预测。首先，我们加载本教程所需的必要库。

python复制代码

import torch

import torch.nn as nn

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

接下来，我们将使用简单的序列数据。下面的代码展示了如何生成序列数据并在图上可视化它。在此，我们使用720个样本作为训练数据，130个样本用于预测。

# 定义参数

python复制代码

step\_size = 4 # 步长大小

N = 850 # 总样本数

forecast\_start = 720 # 预测开始的索引位置

通过定义这些参数，我们可以根据需求生成相应的序列数据，并在后续步骤中用于训练和预测。

# 生成数据

python复制代码

# 创建一个时间数组 t

t = np.arange(0, N)

# 生成带有正弦波和随机噪声的序列数据 x

x = np.sin(0.03 \* t) + 1.2 \* np.random.rand(N) + t/300

# 将数据转换为Pandas DataFrame格式

df = pd.DataFrame(x, columns=['Value'])

# 为了方便后续操作，可以将时间 t 也加入到 DataFrame 中

df['Time'] = t

# 如果只需要原始数据 x 而不需要时间 t，则上面的 'Time' 列可以省略

# 绘制数据

python复制代码

# 绘制整个数据序列

plt.plot(df['Time'], df['Value'])

# 在预测开始点处绘制一条垂直线

plt.axvline(x=df['Time'][forecast\_start], c="r", label="预测开始点")

# 添加图例

plt.legend()

# 显示图形

plt.show()

注意：在上面的代码中，我添加了 'Time' 列到 DataFrame df 中，以便后续可以更方便地根据时间索引来划分训练集和测试集。同时，在绘制垂直线时，我使用了 x=df['Time'][forecast\_start] 来确保垂直线准确地绘制在预测开始的时间点上。如果你不需要 'Time' 列，你可以忽略它，并在绘制垂直线时直接使用 forecast\_start（但这样绘制出来的线可能不会非常精确，因为它基于的是索引而不是实际的时间值）。

图表

描述已自动生成 接下来，我们将数据转换为具有给定长度的序列和标签。以下函数用于为序列数据创建标签。

python复制代码

# 将数据转换为序列和标签，给定序列长度

def create\_labels(data, step):

X = np.array([data[i:i+step] for i in range(len(data) - step)])

y = np.array(data[step:])

return X, y

我们可以使用forecast\_start变量将数据分为训练集和测试集，然后生成序列数据及其标签。为了将数据转换为LSTM的输入格式，我们使用np.reshape()函数。训练集和测试集被转换为PyTorch张量，并使用这些张量创建DataLoader对象。

python复制代码

# 准备训练和测试数据

values = df['Value'].values # 只需要值列，因为我们已经有了时间t作为索引

train, test = values[:forecast\_start], values[forecast\_start:N]

# 生成序列数据

trainX, trainY = create\_labels(train, step\_size)

testX, testY = create\_labels(test, step\_size)

# 将数据重新整形以匹配LSTM的输入

trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))

testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

# 将数据转换为PyTorch张量

trainX\_tens = torch.tensor(trainX, dtype=torch.float32)

trainY\_tens = torch.tensor(trainY, dtype=torch.float32)

testX\_tens = torch.tensor(testX, dtype=torch.float32)

testY\_tens = torch.tensor(testY, dtype=torch.float32)

# 为训练集创建DataLoader

train\_dataset = torch.utils.data.TensorDataset(trainX\_tens, trainY\_tens)

train\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=64, shuffle=True) # 通常训练时会打乱数据

注意：在创建DataLoader时，我添加了shuffle=True参数，这在训练时通常是一个好习惯，因为它可以帮助模型更好地泛化。此外，请确保在调用create\_labels函数时，只传递了值列（df['Value'].values），因为我们只关心这些值作为序列数据。时间t在这里仅用于索引，不需要作为输入数据的一部分。

**模型定义与训练**

我们使用PyTorch的nn.Module类定义了一个GRU模型。在初始化方法中，我们初始化了GRU模型的输入、隐藏和输出尺寸。nn.GRU()方法使用指定的输入和隐藏尺寸构建了GRU层，其中batch\_first=True表示输入和输出张量具有(batch\_size, sequence\_length, input\_size)的形状。此外，我们使用nn.Linear()方法定义了一个全连接线性层，该层将GRU的隐藏状态输出映射到所需的输出尺寸。

在forward方法中，我们实现了通过GRU层的前向传递，生成了输出张量'out'。然后，我们将全连接层应用于GRU最后一个时间步的输出（out[:, -1, :]），生成了模型的最终输出。

python复制代码

# 定义GRU模型

class GRUModel(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

super(GRUModel, self).\_\_init\_\_()

self.gru = nn.GRU(input\_size, hidden\_size, batch\_first=True)

self.fc = nn.Linear(hidden\_size, output\_size)

def forward(self, x):

out, \_ = self.gru(x)

out = self.fc(out[:, -1, :]) # 提取最后一个时间步的输出

return out

我们定义了模型的超参数，并使用上述定义的GRUModel类初始化了模型。我们采用MSELoss()作为损失函数，并使用Adam优化器进行训练。

# 超参数设置

python复制代码

input\_size = step\_size # 输入尺寸设置为步长大小

hidden\_size = 128 # 隐藏层尺寸为128

output\_size = 1 # 输出尺寸为1

epochs = 100 # 训练轮次为100

learning\_rate = 0.0001 # 学习率设置为0.0001

# 实例化GRU模型

model = GRUModel(input\_size=input\_size, hidden\_size=hidden\_size, output\_size=output\_size)

# 定义损失函数和优化器

criterion = nn.MSELoss() # 使用均方误差损失函数

optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate) # 使用Adam优化器，并设置学习率

接下来，我们通过迭代训练轮次来训练模型，并在每10个轮次后打印损失值。

# 训练模型

python复制代码

# 遍历所有训练轮次

for epoch in range(epochs):

model.train() # 设置模型为训练模式

# 遍历数据加载器中的每一个批次数据

for batch\_X, batch\_Y in train\_loader:

optimizer.zero\_grad() # 清空之前所有优化参数的梯度

output = model(batch\_X) # 前向传播，得到模型预测结果

# 计算当前小批量数据的模型预测值与真实标签之间的损失

loss = criterion(output, batch\_Y)

# 反向传播，计算损失关于模型参数的梯度

loss.backward()

# 使用指定的优化算法根据计算得到的梯度更新模型参数

optimizer.step()

# 每10个轮次打印一次损失值

if (epoch + 1) % 10 == 0:

print(f'轮次 [{epoch + 1}/{epochs}], 损失值: {loss.item():.4f}')

现在，我们可以开始训练模型了。

Epoch [10/100], Loss: 7.4051

Epoch [20/100], Loss: 4.0839

Epoch [30/100], Loss: 1.6807

Epoch [40/100], Loss: 0.5536

Epoch [50/100], Loss: 0.2236

Epoch [60/100], Loss: 0.1506

Epoch [70/100], Loss: 0.1338

Epoch [80/100], Loss: 0.1286

Epoch [90/100], Loss: 0.1256

Epoch [100/100], Loss: 0.1231

# ****预测****

训练完成后，我们可以使用训练好的模型对测试数据进行预测，并将其以图形的方式可视化。

python复制代码

# 评估

with torch.no\_grad():

model.eval() # 设置模型为评估模式

testPredict = model(testX\_tens) # 使用模型对测试数据进行预测

# 绘制结果

index = range(len(testY))

plt.plot(index, testY, label="真实值")

plt.plot(index, testPredict.numpy(), label="预测值")

plt.legend() # 显示图例

plt.show() # 显示图形

图表, 折线图

描述已自动生成

# 案例2：比特币价格（BTC/USDT）进行了预测

在本文中，我们利用了一个基于GRU（门控循环单元）的模型（在代码中称为gruMODEL）对3月27日的比特币价格（BTC/USDT）进行了预测，并计算了预测的准确率。

从tensorflow.keras.models导入了Sequential模型，从tensorflow.keras.optimizers导入了随机梯度下降（SGD）优化器，从tensorflow.keras.layers导入了激活函数（Activation）、全连接层（Dense）、丢弃层（Dropout）和门控循环单元（GRU）。此外，还导入了matplotlib.pyplot用于绘图，numpy用于数值计算，pandas用于数据处理，以及sklearn.metrics中的平均绝对误差（MAE）作为评估指标。

python复制代码

from tensorflow.keras.models import Sequential

from tensorflow.keras.optimizers import SGD

from tensorflow.keras.layers import Activation, Dense, Dropout, GRU

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

from sklearn.metrics import mean\_absolute\_error

# 读取CSV文件中的数据

data = pd.read\_csv("/content/BTC.csv")

# 选取除“Price”和“Vol.”列外的前六列

data = data.iloc[:, 0:6]

# 将“Price”列作为目标变量y

y = data.loc[:, ['Price']]

# 从原始数据中删除“Price”和“Vol.”列

data = data.drop(['Price', 'Vol.'], axis='columns')

# 打印处理后的数据的前五行

print(data.head(5))

# 打印目标变量y的前五行

print(y.head(5))

输出数据的前五行显示了日期（Date）、开盘价（Open）、最高价（High）和最低价（Low）等信息，这些被用作特征变量。而目标变量y则显示了对应日期的收盘价（Price）。

表格

描述已自动生成

图片包含 表格

描述已自动生成

在 [6] 代码中：

python复制代码

data = data.set\_index('Date') # 将'Date'列设置为数据框的索引

data.index = pd.to\_datetime(data.index, unit='ns') # 将索引转换为datetime64[ns]类型

print(data.index) # 打印转换后的日期索引

输出结果表明，data 的索引已经成功转换为了以纳秒为单位的时间戳格式，并且按照从 '2022-03-25' 到 '2018-02-26' 的时间顺序排列。这个 DatetimeIndex 包含了 1489 个日期，没有固定的频率（freq=None）。

在代码中：

python复制代码

aim = 'Price' # 设定目标变量为'Price'

这行代码声明了一个变量 aim，并将其值设置为字符串 'Price'，这通常用于后续的代码中以标识目标变量。

在 代码中：

python复制代码

data.shape # 查询数据框的形状

输出 (1489, 3) 表明 data 数据框包含 1489 行和 3 列。

在代码中：

python复制代码

X\_train = data[300:] # 将数据框从第300行开始到末尾作为训练集特征

X\_test = data[:300] # 将数据框从开始到第300行（不包括第300行）作为测试集特征

y\_train = y[300:] # 将目标变量从第300行开始到末尾作为训练集标签

y\_test = y[:300] # 将目标变量从开始到第300行（不包括第300行）作为测试集标签

print(y\_test) # 打印测试集标签

这段代码将数据划分为训练集和测试集，其中训练集包含第300行之后的数据，测试集包含从开始到第300行（不包括第300行）的数据。并且打印出了测试集的目标变量值（即 BTC 的价格）。

接着定义了一个名为 line\_plot 的函数，用于绘制两条线的图形。这个函数接受两条线（line1 和 line2）、两个标签（label1 和 label2）、一个标题（title）和线宽（lw）作为参数。在图形中，两条线会被分别绘制出来，并且会添加 Y 轴标签、标题和图例。图例的位置由 loc='best' 自动选择最佳位置。

表格

描述已自动生成

在代码中：

python复制代码

line\_plot(y\_train[aim], y\_test[aim], 'training', 'test', title='')

该行代码调用了之前定义的 line\_plot 函数，用于绘制训练集和测试集的目标变量（在这里是比特币价格）随时间变化的折线图。图中的两条线分别代表了训练集（'training'）和测试集（'test'）的价格变化，标题为空（title=''）。

图表, 直方图

描述已自动生成

上面的图片链接显示了训练集和测试集价格随时间变化的折线图。

接下来，定义了两个数据归一化函数：

python复制代码

def normalise\_zero\_base(continuous): return continuous / continuous.iloc[0] - 1

def normalise\_min\_max(continuous): return (continuous - continuous.min()) / (data.max() - continuous.min())

* normalise\_zero\_base 函数将连续数据以第一行数据为基准进行归一化，即除以第一行数据然后减去1。
* normalise\_min\_max 函数将连续数据缩放到0到1之间，通过减去最小值并除以数据范围（最大值减去最小值）实现。但注意这里使用了全局的 data.max() 而不是 continuous.max()，这可能是一个错误，因为通常我们会希望针对 continuous 自身的最大值和最小值进行归一化。

在之后的代码中：

python复制代码

X\_train = normalise\_zero\_base(X\_train)

X\_test = normalise\_zero\_base(X\_test)

y\_train = normalise\_zero\_base(y\_train)

y\_test = normalise\_zero\_base(y\_test)

这四行代码分别对训练集和测试集的特征变量（X\_train 和 X\_test）以及目标变量（y\_train 和 y\_test）进行了基于第一行数据的归一化处理。

在代码中：

python复制代码

import numpy as np

X\_train = np.expand\_dims(X\_train, axis=1)

X\_test = np.expand\_dims(X\_test, axis=1)

这里使用 NumPy 的 expand\_dims 函数为 X\_train 和 X\_test 数据集增加了一个新的维度，即在第二个维度（索引为1）上增加了维度，使得它们的形状从 (样本数, 特征数) 变为 (样本数, 1, 特征数)。这通常是为了适应某些需要特定输入形状的模型，如循环神经网络（RNN）或卷积神经网络（CNN）。

在代码中：

python复制代码

X\_train.shape

该行代码输出了归一化并扩展维度后的 X\_train 数据集的形状，结果是 (1189, 1, 3)，表示有1189个样本，每个样本是一个序列，包含3个特征。

从 TensorFlow 导入 Keras 模块后，下面的代码定义了一个基于 GRU（门控循环单元）架构的神经网络模型。以下是该代码的中文解释：

python复制代码

from tensorflow import keras # 从 TensorFlow 导入 Keras 模块

# 定义 GRU 模型

gruMODEL = keras.Sequential() # 创建一个顺序模型

# 添加第一个 GRU 层，并应用了 Dropout 正则化

gruMODEL.add(keras.layers.GRU(

units=1024, # 隐藏层单元数为 1024

input\_shape=(1, 3), # 输入形状为 (时间步长, 特征数)，这里时间步长为 1，特征数为 3

activation='PReLU', # 激活函数为 PReLU（参数化修正线性单元）

recurrent\_activation="sigmoid", # 循环层激活函数为 sigmoid

use\_bias=True, # 使用偏置项

kernel\_initializer="glorot\_uniform", # 权重初始化方法为 Glorot 均匀分布

recurrent\_initializer="orthogonal", # 循环层权重初始化方法为正交矩阵

bias\_initializer="zeros", # 偏置项初始化方法为 0

# 以下均为正则化和约束选项，此处未使用

kernel\_regularizer=None,

recurrent\_regularizer=None,

bias\_regularizer=None,

activity\_regularizer=None,

kernel\_constraint=None,

recurrent\_constraint=None,

bias\_constraint=None,

dropout=0.0, # 在输入层上丢弃的输入单元比例，此处未使用

recurrent\_dropout=0.0, # 在循环层上丢弃的单元比例，此处未使用

return\_sequences=False, # 是否返回完整序列的输出，这里只返回最后一个时间步的输出

return\_state=False, # 是否返回最后一个时间步的状态，此处不返回

go\_backwards=False, # 是否反向处理输入序列

stateful=False, # 如果为 True，则批次中的每个样本的索引 i 的状态将被用作下一个批次中索引 i 的样本的初始状态

unroll=False, # 如果为 True，则网络将展开，否则使用符号循环

time\_major=False, # 输入和输出的形状格式，False 表示 (batch, time, features)，True 表示 (time, batch, features)

reset\_after=True # 是否在每个时间步重置门控单元

))

# 添加 Dropout 层，丢弃率为 0.9

gruMODEL.add(keras.layers.Dropout(0.9))

以上代码定义了一个包含一层 GRU 和一层 Dropout 的神经网络模型，其中 GRU 层用于处理序列数据，Dropout 层用于防止过拟合。Dropout 层的丢弃率设置为 0.9，意味着在训练过程中，每个神经元有 90% 的概率被丢弃。

基于 TensorFlow 的 Keras 框架，我们构建了一个包含两层 GRU（门控循环单元）的模型，并在每层 GRU 之后添加了 Dropout 层以进行正则化，防止过拟合。以下是模型的构建过程：

python复制代码

from tensorflow import keras

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import GRU, Dropout

# 初始化 GRU 模型

gruMODEL = Sequential()

# 第一层 GRU，并添加 Dropout 正则化

gruMODEL.add(GRU(

units=1024, # 隐藏单元数量

input\_shape=(1, 3), # 输入数据的形状

activation='PReLU', # 激活函数

recurrent\_activation="sigmoid", # 循环层激活函数

use\_bias=True, # 是否使用偏置项

kernel\_initializer="glorot\_uniform", # 权重初始化方法

recurrent\_initializer="orthogonal", # 循环权重初始化方法

bias\_initializer="zeros", # 偏置项初始化方法

# 正则化、约束等参数均设置为 None，即不使用

dropout=0.0, # 在每个时间步的输入之间丢弃一部分单元

recurrent\_dropout=0.0, # 在每个时间步的循环层之间丢弃一部分单元

return\_sequences=False, # 是否返回输出序列的最后一个时间步

return\_state=False, # 是否返回最后一个时间步的状态

go\_backwards=False, # 是否反向处理输入序列

stateful=False, # 如果为 True，则批次中的每个样本将具有其自己的内部状态

unroll=False, # 如果为 True，则网络将展开其计算图，这可能会消耗更多的内存但会加速计算

time\_major=False, # 输入数据的形状格式，此处为（batch, time, features）

reset\_after=True # 在计算输出时，是否重置隐藏状态

))

gruMODEL.add(Dropout(0.9)) # 添加 Dropout 层，丢弃率为 0.9

# 第二层 GRU

gruMODEL.add(GRU(

units=2048, # 隐藏单元数量

activation='PReLU', # 激活函数

recurrent\_activation="sigmoid", # 循环层激活函数

use\_bias=True, # 是否使用偏置项

kernel\_initializer="glorot\_uniform", # 权重初始化方法

recurrent\_initializer="orthogonal", # 循环权重初始化方法

bias\_initializer="zeros", # 偏置项初始化方法

# 正则化、约束等参数均设置为 None，即不使用

dropout=0.0, # 在每个时间步的输入之间丢弃一部分单元

recurrent\_dropout=0.0, # 在每个时间步的循环层之间丢弃一部分单元

return\_sequences=False, # 是否返回输出序列的最后一个时间步

return\_state=False, # 是否返回最后一个时间步的状态

go\_backwards=False, # 是否反向处理输入序列

stateful=False, # 如果为 True，则批次中的每个样本将具有其自己的内部状态

unroll=False, # 如果为 True，则网络将展开其计算图

time\_major=False, # 输入数据的形状格式

reset\_after=True # 在计算输出时，是否重置隐藏状态

))

gruMODEL.add(Dropout(0.8)) # 添加 Dropout 层，丢弃率为 0.8

在这个模型中，我们首先定义了一个 Sequential 模型，然后添加了两个 GRU 层，每个 GRU 层后面都跟随了一个 Dropout 层。第一层 GRU 的隐藏单元数为 1024，输入数据的形状为 (1, 3)，表示每个样本有一个时间步长和三个特征。第二层 GRU 的隐藏单元数增加到了 2048，并且两个 Dropout 层的丢弃率分别设置为 0.9 和 0.8，用于在训练过程中随机丢弃一部分神经元的输出，以减少过拟合的风险。

# 第三层GRU层

gruMODEL.add(GRU(

units=4096, # 神经元数量

activation='PReLU', # 激活函数为PReLU

recurrent\_activation="sigmoid", # 递归激活函数为sigmoid

use\_bias=True, # 使用偏置项

kernel\_initializer="glorot\_uniform", # 权重初始化方法

recurrent\_initializer="orthogonal", # 递归权重初始化方法

bias\_initializer="zeros", # 偏置初始化方法

kernel\_regularizer=None, # 权重正则化方法

recurrent\_regularizer=None, # 递归权重正则化方法

bias\_regularizer=None, # 偏置正则化方法

activity\_regularizer=None, # 激活正则化方法

kernel\_constraint=None, # 权重约束

recurrent\_constraint=None, # 递归权重约束

bias\_constraint=None, # 偏置约束

dropout=0.0, # dropout层比例

recurrent\_dropout=0.0, # 递归dropout层比例

return\_sequences=False, # 是否返回输出序列的最后一个输出

return\_state=False, # 是否返回最后一个状态

go\_backwards=False, # 是否反向计算

stateful=False, # 是否是有状态的网络

unroll=False, # 是否展开网络

time\_major=False, # 输入张量的形状格式

reset\_after=True)) # 是否在每个时间步之后重置门控单元

gruMODEL.add(Dropout(0.09)) # 添加dropout层，防止过拟合

# 第四层GRU层

gruMODEL.add(GRU(

units=512, # 神经元数量

activation='PReLU', # 激活函数为PReLU

recurrent\_activation="sigmoid", # 递归激活函数为sigmoid

# ...（以下参数与第三层相同，省略）

reset\_after=True))

gruMODEL.add(Dropout(0.09)) # 添加dropout层，防止过拟合

# 输出层

gruMODEL.add(Dense(units=1)) # 添加全连接层，输出单元数为1

# 编译RNN

gruMODEL.compile(optimizer="sgd", # 使用随机梯度下降优化器

loss='mean\_squared\_error') # 使用均方误差作为损失函数

gruMODEL.summary() # 打印模型概览信息

该模型结构概览如下：

模型从输入层开始，通过一系列的门控循环单元（GRU）层和dropout层进行信息处理和特征提取，最终通过一个全连接层（Dense）输出预测结果。

* 第一层是gru\_21，这是一个GRU层，具有1024个神经元，其输出形状为(None, 1, 1024)，意味着对于任意长度的输入序列，该层会输出一个长度为1的序列，每个序列元素是1024维的特征向量。该层包含3,162,112个参数。
* 接着是dropout\_18层，这是一个dropout层，用于防止过拟合，其输出形状与gru\_21相同，但没有任何可训练的参数。
* 接下来是gru\_22层，同样是一个GRU层，但神经元数量增加至2048个，输出形状为(None, 1, 2048)，包含18,888,704个参数。
* 类似的，dropout\_19层作为dropout层，用于进一步减少过拟合的风险。
* gru\_23层是模型中的第三层GRU层，神经元数量增加到4096个，输出形状为(None, 1, 4096)，包含75,526,144个参数。
* dropout\_20层继续作为dropout层，减少过拟合。
* gru\_24层是模型中的最后一层GRU层，其输出形状变为(None, 512)，意味着该层将4096维的特征向量压缩至512维，包含7,080,960个参数。
* dropout\_21层是最后一个dropout层。
* 最后是dense\_3层，这是一个全连接层，只有一个神经元，用于输出预测结果，其输出形状为(None, 1)，包含513个参数。

整个模型包含总共104,658,433个可训练参数，没有不可训练的参数。这些参数在模型训练过程中会被优化器调整，以最小化损失函数。模型的架构反映了从输入到输出的信息处理和特征提取的逐步深化过程。

表格

描述已自动生成

文本

中度可信度描述已自动生成

在构建网络结构图时，我们定义了一个包含多个GRU层的模型。以下是基于提供代码生成的网络结构图的描述：

首先，我们定义了网络中的层数和各层的神经元数量，它们分别是：[3, 15, 30, 45, 90, 1]。随后，我们为这些层定义了对应的文本标签和颜色属性。

网络结构图使用Graphviz的DOT语言描述，并通过Python代码生成。整个图从左至右（LR）排列，各层之间通过直线（splines=line）连接。节点间的间距（nodesep）和层次间的间距（ranksep）分别设定为0.08和1，以便于清晰展示。

对于图中的每个节点（即每个神经元），我们设定了固定的尺寸（fixedsize=true），并使用了填充颜色（fillcolor）和形状（shape=circle）。输入层（Input）和输出层（Output）分别用黑色填充，而中间的GRU层用灰色填充。节点的标签被隐藏，仅显示填充颜色。

接下来，代码创建了多个子图（subgraph），每个子图代表网络中的一个层。每个子图中的节点数量与该层的神经元数量相对应。例如，输入层（Input）包含3个节点，第一个GRU层包含15个节点，依此类推。

在子图中，我们定义了节点的样式（style）、颜色（color）、边框宽度（penwidth）和填充颜色（fillcolor）。对于输入层和输出层，我们使用了黑色填充；对于中间的GRU层，我们使用了灰色填充。

最后，代码生成了连接各层节点的边。例如，“l310 -> l441”表示第三层的第一个节点连接到第四层的第一个节点。通过这种方式，我们构建了整个网络结构图。

以下是部分生成的连接关系：

在以下的代码中，我们定义了一个网络结构图，其基于Graphviz的DOT语言格式进行描述。网络结构由多个层组成，包括输入层、多个GRU（门控循环单元）层和输出层。以下是代码的详细解释和翻译：

python复制代码

# 定义网络层数和每层神经元数量

layers = [3,15,30,45,90,1]

# 定义各层的文本标签

layers\_str = ["Input"] + ["GRU"] \* (len(layers) - 2) + ["Output"]

# 定义各层的颜色属性（这里实际未使用）

layers\_col = ["none"] + ["none"] \* (len(layers) - 2) + ["none"]

# 定义各层的填充颜色

layers\_fill = ["black"] + ["gray"] \* (len(layers) - 2) + ["black"]

# 节点边框宽度

penwidth = 15

# 字体设置

font = "Hilda 10"

# 开始打印DOT语言描述

print("digraph G {")

print("\tfontname = "{}"".format(font))

print("\trankdir=LR") # 设置图的布局方向为从左到右

print("\tsplines=line") # 设置边为直线

print("\tnodesep=.08;") # 设置节点之间的间距

print("\tranksep=1;") # 设置层次之间的间距

print("\tedge [color=black, arrowsize=.5];") # 设置边的颜色和箭头大小

print("\tnode [fixedsize=true,label="",style=filled," + \

"color=none,fillcolor=gray,shape=circle]\n") # 设置节点属性，如固定大小、填充样式和颜色等

# 绘制各层（子图）

for i in range(0, len(layers)):

print("\tsubgraph cluster\_{} {{".format(i))

print("\t\tcolor={};".format(layers\_col[i])) # 设置子图颜色（这里实际未使用）

print("\t\tnode [style=filled, color=white, penwidth={},"

"fillcolor={} shape=circle];".format(penwidth, layers\_fill[i])) # 设置子图中节点的属性

print("\t\t", end=' ')

for a in range(layers[i]):

print("l{}{} ".format(i + 1, a), end=' ') # 打印节点标签

print(";")

print("\t\tlabel = {};".format(layers\_str[i])) # 设置子图的标签

print("\t}\n")

# 绘制节点间的边

for i in range(1, len(layers)):

for a in range(layers[i - 1]):

for b in range(layers[i]):

print("\tl{}{} -> l{}{}".format(i, a, i + 1, b)) # 绘制从第i层的第a个节点到第i+1层的第b个节点的边

print("}") # 结束DOT语言描述

这段代码将生成一个描述网络结构的DOT文件，其中包含从输入层开始，通过多层GRU网络，最后到达输出层的连接关系。每个层由一组节点组成，这些节点之间通过边进行连接。通过Graphviz软件可以将此DOT文件渲染成网络结构图。

表格

描述已自动生成

这些连接关系展示了网络中信息流动的方向，即输入层的数据经过多层GRU处理后，最终到达输出层。

python复制代码

# 使用'w'模式打开名为'model.txt'的文件，准备写入网络层结构信息

with open('model.txt', 'w') as layers\_file:

# 定义网络各层的神经元数量

layers = [3,5,10,15,20,1]

# 定义各层的文本标签

layers\_str = ["Input"] + ["GRU"] \* (len(layers) - 2) + ["Output"]

# 定义各层的颜色属性（此处未实际使用）

layers\_col = ["none"] + ["none"] \* (len(layers) - 2) + ["none"]

# 定义各层的填充颜色

layers\_fill = ["black"] + ["gray"] \* (len(layers) - 2) + ["black"]

# 节点边框宽度

penwidth = 15

# 字体设置

font = "Hilda 10"

# 写入DOT语言描述，用于描述网络结构图

layers\_file.write("digraph G {\n") # 开始定义有向图G

layers\_file.write("\tfontname = "{}"\n".format(font)) # 设置字体名称为Hilda 10

layers\_file.write("\trankdir=LR\n") # 设置图的布局方向为从左到右

layers\_file.write("\tsplines=line\n") # 设置边为直线

layers\_file.write("\tnodesep=.08;\n") # 设置节点之间的间距

layers\_file.write("\tranksep=1;\n") # 设置层次之间的间距

layers\_file.write("\tedge [color=black, arrowsize=.5];\n") # 设置边的颜色和箭头大小

layers\_file.write("\tnode [fixedsize=true,label="",style=filled,color=none,fillcolor=gray,shape=circle]\n\n") # 设置节点属性

# 绘制各层（子图）

for i in range(0, len(layers)):

layers\_file.write("\tsubgraph cluster\_{} {{\n".format(i)) # 开始定义第i个子图

layers\_file.write("\t\tcolor={};\n".format(layers\_col[i])) # 设置子图颜色（此处未实际使用）

layers\_file.write("\t\tnode [style=filled, color=white, penwidth={},fillcolor={} shape=circle];\n".format(penwidth, layers\_fill[i])) # 设置子图中节点的属性

layers\_file.write("\t\t")

for a in range(layers[i]):

layers\_file.write("l{}{} ".format(i + 1, a)) # 写入节点标签

layers\_file.write(";\n")

layers\_file.write("\t\tlabel = {};\n".format(layers\_str[i])) # 设置子图的标签

layers\_file.write("\t}\n\n") # 结束定义第i个子图

# 绘制节点间的边

for i in range(1, len(layers)):

for a in range(layers[i - 1]):

for b in range(layers[i]):

layers\_file.write("\tl{}{} -> l{}{}\n".format(i, a, i + 1, b)) # 绘制从第i层的第a个节点到第i+1层的第b个节点的边

layers\_file.write("}\n") # 结束DOT语言描述

上述代码段使用Python的with语句打开了一个名为model.txt的文件，用于写入一个Graphviz DOT格式的文本文件。该文件描述了一个由多层组成的网络结构，其中包含了输入层、多个GRU层和输出层。代码首先定义了网络各层的神经元数量、文本标签、颜色属性（尽管实际并未使用）和填充颜色。接着，它使用循环语句为每一层生成了相应的子图（subgraph）描述，并为这些子图中的每一个节点分配了标签。最后，代码通过嵌套循环为每一对相邻层之间的节点绘制了边，表示它们之间的连接关系。整个DOT文件描述了整个网络的结构，并可以被Graphviz软件读取以生成对应的网络结构图。

digraph G {

fontname = "Hilda 10"

rankdir=LR

splines=line

nodesep=.08;

ranksep=1;

edge [color=black, arrowsize=.5];

node [fixedsize=true,label="",style=filled,color=none,fillcolor=gray,shape=circle]

subgraph cluster\_0 {

color=none;

node [style=filled, color=white, penwidth=15,fillcolor=black shape=circle];

l10 l11 l12 ;

label = Input;

}

subgraph cluster\_1 {

color=none;

node [style=filled, color=white, penwidth=15,fillcolor=gray shape=circle];

l20 l21 l22 l23 l24 ;

label = GRU;

}

subgraph cluster\_2 {

color=none;

node [style=filled, color=white, penwidth=15,fillcolor=gray shape=circle];

l30 l31 l32 l33 l34 l35 l36 l37 l38 l39 ;

label = GRU;

}

subgraph cluster\_3 {

color=none;

node [style=filled, color=white, penwidth=15,fillcolor=gray shape=circle];

l40 l41 l42 l43 l44 l45 l46 l47 l48 l49 l410 l411 l412 l413 l414 ;

label = GRU;

}

subgraph cluster\_4 {

color=none;

node [style=filled, color=white, penwidth=15,fillcolor=gray shape=circle];

l50 l51 l52 l53 l54 l55 l56 l57 l58 l59 l510 l511 l512 l513 l514 l515 l516 l517 l518 l519 ;

label = GRU;

}

subgraph cluster\_5 {

color=none;

node [style=filled, color=white, penwidth=15,fillcolor=black shape=circle];

l60 ;

label = Output;

}

l10 -> l20

# ****模型训练与评估****

**模型训练**

使用gruMODEL.fit方法对GRU神经网络模型进行训练。训练数据为X\_train和y\_train，共进行32个epoch的迭代，每个batch的大小设置为250。同时，使用X\_test和y\_test作为验证数据集进行模型性能验证。此外，通过keras.callbacks.ModelCheckpoint回调函数，在每个epoch结束后保存模型权重到/content/model/model\_{epoch}.h5文件中，文件名中包含epoch数。

从训练输出可以看出：

* 第一个epoch的训练损失为0.0444，验证损失为0.0120。
* 第二个epoch的训练损失下降到0.0118，验证损失也下降到0.0074。

这表明模型在训练过程中正在学习，并且逐渐优化损失函数。

# ****模型损失可视化****

通过matplotlib库绘制了训练损失和验证损失随epoch变化的曲线图。图中红色曲线代表训练损失，绿色曲线代表验证损失。从图中可以观察到，随着epoch的增加，两者均呈现下降趋势，说明模型正在逐渐优化。

**模型加载与预测**

使用tensorflow.keras.models.load\_model方法加载了训练好的模型（第32个epoch的模型），并使用gruMODEL.predict方法对测试集X\_test进行预测。通过squeeze方法去除预测结果中可能存在的单一维度，然后使用mean\_absolute\_error计算了预测值与真实值之间的平均绝对误差（MAE），结果为0.02262356696266076。

**均方误差（MSE）计算**

进一步，使用sklearn.metrics.mean\_squared\_error计算了预测值与真实值之间的均方误差（MSE），并将其赋值给SCORE\_MSE变量。从输出结果可以看到，MSE的值为SCORE\_MSE，但在给出的代码片段中并未直接显示该值的具体数值。不过，根据上下文推测，它应该是一个数值结果，代表模型在测试集上的均方误差性能。

# ****结果解释与输出****



上述结果表示在测试集X\_test上，模型gruMODEL的预测结果preds与真实值y\_test之间的均方误差（MSE）为0.00082，这是一个非常小的值，表明模型的预测性能很好。

python复制代码

from sklearn.metrics import r2\_score

r2\_score = r2\_score(y\_test, preds)

r2\_score \* 100



这里计算了模型预测结果preds与真实值y\_test之间的决定系数（R²），并将其转换为百分比形式。R²值为98.52%，接近100%，说明模型的预测结果非常接近真实值，模型的拟合效果非常好。

python复制代码

y\_test.ravel().shape

图片包含 文本

描述已自动生成

这表示测试集y\_test经过ravel方法展平后是一个包含300个元素的一维数组，即测试集包含300个样本。

python复制代码

y\_testt = scaling.inverse\_transform(y\_test)

print(type(y\_testt))

这里使用scaling.inverse\_transform方法将经过缩放处理的测试集目标值y\_test转换回原始尺度，并打印其类型。输出结果显示y\_testt的类型是numpy.ndarray，即一个NumPy数组，表示转换成功。

python复制代码

preds = scaling.inverse\_transform(gruMODEL.predict(X\_test))

同样地，这里使用scaling.inverse\_transform方法将模型对测试集X\_test的预测结果preds（通常是在缩放后的尺度上）转换回原始尺度，以便与原始尺度的真实值进行比较或可视化。

python复制代码

line\_plot(y\_testt, preds, 'test', 'prediction', title='')

这行代码调用了一个名为line\_plot的函数（该函数在提供的代码段中未定义，但根据命名和参数可以推测其功能），用于绘制测试集的真实值y\_testt与模型预测值preds的折线图。图表的标题为空，x轴标签为'test'（可能表示测试集样本），y轴标签为'prediction'（表示预测值）。通过这条命令，我们可以直观地比较真实值与预测值之间的差异，进一步评估模型的性能。

图表, 折线图

描述已自动生成

对3月27日的比特币价格（BTC/USDT）进行了预测，并计算了预测结果的准确率。

首先，我们给出了3月27日的预测价格和实际价格：

* **预测价格**：基于GRU模型的预测结果显示，3月27日的比特币价格预测为**46351.2 BTC/USDT**。
* **实际价格**：在伊斯坦布尔时间3月27日凌晨2点22分，比特币的实际交易价格为**46564.00 BTC/USDT**。

接下来，我们详细描述了预测过程的代码实现：

python复制代码

In [222]:

prediction = np.array([[44331,44818,44090]]) # 这里假设的预测数据，用于后续计算

X\_testt = scaling.inverse\_transform(X\_test[0]) # 对测试数据进行逆缩放处理

prediction\_new = np.array([[(X\_test[0][0]/X\_testt[0]\*prediction[0])]]) # 基于缩放后的数据重新计算预测值

predictions = gruMODEL.predict(prediction\_new)[0][0] # 使用GRU模型进行预测

predictions = np.array([[predictions]]) \* prediction[0][0] / prediction\_new[0][0][0] # 对预测结果进行比例调整

f"27 March Prediction is {predictions[0][0]} BTC/USDT" # 输出预测结果

Out[222]:

文本

低可信度描述已自动生成

以上代码首先定义了一个假设的预测数据prediction，然后对该数据进行了一系列处理，包括逆缩放、重新计算预测值以及比例调整，最终得到了3月27日的预测价格。

接下来，我们计算了预测准确率：

python复制代码

In [223]:

print("27 March Accuracy: ")

real = 46564 # 实际价格

predict = predictions[0][0] # 预测价格

accuracy = 1- (real - predict) / real # 计算准确率

print("Accuracy: {}".format(accuracy))

文本

低可信度描述已自动生成

计算结果显示，3月27日的预测准确率为**0.99**，这表示该模型在预测比特币价格时具有较高的准确性。通过比较实际价格与预测价格，我们可以得出模型在价格预测方面的性能表现。

# ****结论****

GRU通过将遗忘门和输入门合并为一个单一的更新门，简化了传统LSTM网络的结构，使得其在捕捉时间依赖关系时计算效率更高。