显卡是计算机的基础且重要的组成部分，将计算机系统的显示信息进行转换以驱动显示器运行，控制显示器的正确显示。显卡是人机交互的重要设备之一。

显卡的发展经历了ISA、VESA、PCI、AGP、PCI-Express五种接口类型，nVidia和AMD的PCI-E显卡还支持多显卡技术（nVidia的有桥/无桥SLI，AMD的CrossFire），进一步增强了计算机的计算性能，还流传出诸如“四路泰坦”（指用四块高端级别的NVIDIA泰坦独立显卡来组建SLI，以实现更加恐怖的超高级性能）的词汇。

基于强大的并行计算性能，显卡在生活中有着非常广泛的应用。从最基本的亮机、图像显示，到视频硬件解码，再到支持3D游戏运行。在生产力方面，进行图像处理、视频剪辑、建模渲染等。再到科学领域，进行科学计算、有限元模拟、深度学习。还包括区块链经济的比特币、以太坊等虚拟货币。上述内容无不需要强大的算力支持，也就需要显卡的出现。NVIDIA最新的旗舰级显卡RTX3090的算力已经可以达到120.9 MH/s，支持流畅运行原生8K游戏，拥有10496个CUDA核心，功耗也达到了惊人的350W。

不难看出，我们对于显卡的需求是客观存在的。但由于生产工艺的复杂，GPU核心的产能有限，也没有专门的生产GPU核心的生产线。除了核心，显存等附属芯片同样产能有限，有线的产能还要分配给内存、手机等其他领域。受限的产能造成了显卡价格始终居高不下，并且还时常波动。

自2020年末，显卡价格开始全球性的上涨。受2020年疫情影响，导致全球供应链紧张，显卡芯片产能不足，市场供不应求。部分显卡的价格，即使是在已经面世两年后，任然超过了首发售价。由于货源少的缘故，除了RTX30系列不断疯涨中，市场上16系列、20系列或RX5000系列等显卡开始缺货并疯狂涨价中，这是十分不常见的事。除此之外，NVIDIA的30系显卡的定价也引发了热烈讨论。足见人们对显卡的关注程度之深。因而，认为研究显卡价格具有实际意义，并能对显卡的消费行为提出一定的指导作用。

对显卡价格的分析，可以分为对显卡首发价格，即官方定价的分析，以及影响显卡价格波动的因素。对显卡官方定价可以从显卡本身的配置出发，而对价格波动则要结合当时实际情况。

对于显卡本身价格的分析可以表现出官方显卡本身的态度，一定程度上反映了显卡的制造成本、科技水平，同一世代不同价位的显卡表现出同一架构不同的周边配置和附件对价格的影响。通常情况下，同一世代的显卡的性能越好，显卡的价格越高。不同世代相近价格的显卡，世代较新的性能较高。性能相近的消费级显卡与专业级显卡，专业级显卡的价格更高。同一显卡在不同显卡制造商的不同型号上价格亦会有所浮动，可能通过减配降低价格，也可能用过更好的用料、散热、更高的核心频率甚至是灯效、联名等非性能因素而提升价格。

对于显卡实际价格的波动则是市场供求机制的简单体现。供不应求，则价格上升；供过于求则价格下降。一般来说，新显卡发售后，由于对新显卡的需求，以及部分商家的囤货行为，显卡价格会首先上升。对于官方指导价而言信价比最高的显卡一般涨幅较为明显。待到显卡热潮过去，一般是数个月之后，显卡价格就会逐渐回落到首发价附近。在下一世代的显卡发售之后，上时代的显卡，由于新显卡供不应求，又可能会出现一次较小幅度的价格上升。等到新显卡的供货稳定后，经销商出于清除库存的原因，显卡价格会再次下跌，常常会低于首发售价。

首先，对于显卡首发价格，即官方指导价格的分析，应当从显卡本身出发，分析显卡本身的各项硬件因素。由于非硬件因素而造成的显卡溢价视情况而言波动非常大，在这里暂不作讨论。这里只分析真正影响显卡跑分的因素。

1. 显卡架构

架构其实是影响显卡性能的最重要指标，新一代架构的出现往往也就意味着产品的迭代升级，比如从Maxwell（麦克斯韦）→Pascal（帕斯卡）→Turing（图灵）→Ampere（安培）。

显卡架构是显示芯片各种处理单元的组成和工作模式，效率高的架构在同等参数下执行力更高，性能也更好。比如Nvida的开普勒架构，比费米架构的效率和功耗都进步了很多，同等参数的显卡比如费米架构的GTX550Ti和开普勒架构的GTX650，在性能上GTX650就稍强，功耗也更低。显卡的驱动也很重要，显卡使用最新的驱动往往比老版本的驱动表现的要好一些。

1. 核心数量

cuda是nvidia主推的并行计算架构。cuda core，或者称为sp，是主要的运算单元，内部有分别处理int和单精度float的部分，用于执行一些常用的运算指令。除了sp之外还有双精度单元用于科学计算以及特殊运算单元sfu来执行比较复杂的运算指令比如三角函数。一个特定计算量的任务，如果核心的数目越多，那么单位时间内执行的运算就越多，所以完成这个任务所花费的时间就越少，显然计算速度就越快。CUDA运算速度只和核心频率有关，而CUDA核心数量则决定了显卡的计算力的强弱。

1. 显卡频率
   1. 基准频率

是指显示核心的工作频率，其工作频率在一定程度上可以反映出显示核心的性能，但显卡的性能是由核心频率、显存、像素管线、像素填充率等等多方面的情况所决定的，因此在显示核心不同的情况下，核心频率高并不代表此显卡性能强劲。

* 1. 加速频率

加速频率表示在环境允许GPU进入超频模式时,可以达到的最高频率。

频率的数值并不是无休止的高才好，高频率必然带来高发热量高能耗，同时也容易出现花屏之类的问题。另外频率的高低在相同型号下才可进行比较，型号不同的显卡相互对比频率则没有任何意义。

1. 显存配置

显存，也被叫做帧缓存，它的作用是用来存储显卡芯片处理过或者即将提取的渲染数据。如同计算机的内存一样，显存是用来存储要处理的图形信息的部件。

* 1. 显存类型

1. SDDR4/GDDR4

DDR4相比DDR3最大的区别有三点：16bit预取机制（DDR3为8bit），同样内核频率下理论速度是DDR3的两倍；更可靠的传输规范，数据可靠性进一步提升；工作电压降为1.2V，更节能。

SDDR4/GDDR4即为DDR4内存的显存版本。

1. GDDR5

第五版图形用双倍数据传输率存储器（Graphics Double Data Rate, version 5，简称GDDR5），是一种高性能显卡使用的同步动态随机存取存储器，专为高带宽需求计算机应用所设计。由AMD、SK Hynix、三星电子、NVIDIA、JEDEC等联合制定，取代GDDR3以及GDDR4显示存储器。

1. GDDR5X

美光科技于2015年10月宣布成功开发出GDDR5X，比GDDR5更高、逼近现时HBM的带宽速度（于存储器总线维持256比特、等效时钟频率14GHz的条件下，至少可拥有448GB/s的带宽）。已知GDDR5X相较于GDDR5的改变，在于存储器预取从8n提升到16n、更低的运行电压（和DDR3L相同的1.35V）、其余部分与GDDR5的基本相同。

1. GDDR6

GDDR6英文全称为Graphics Double Data Rate, version 6，属于第六代版图形用双倍数据传输率存储器。简单来说，也就是显卡的缓存。GDDR6 是目前最新六代技术，相比目前主流的GDDR5更先进，频率更高，更有利于提升显卡性能。GDDR6显存采用双通道读写设计,显存速度高达16Gbps, 工作电压相比上一代GDDR5显存有所降低,最高缓存容量能够高达32Gb.

1. HBM/HBM2

HBM比起GDDR5拥有更高的带宽和比特，比特部分每一颗HBM存储器就高达1024位，存储器时钟频率只有500左右，电压也比GDDR5小，还能缩小存储器布置空间，不过制造困难成本也高，所以供应量非常少。在HBM发布之后，HBM 2也成功开发出来，存储器比特提升至两倍。

目前在显存行业有两个方向，一是传统的GDDR继续演化，NVIDIA RTX 20系列已经用上最新的GDDR6，二就是高带宽的HBM，已经进化到第二代，NVIDIA、AMD的专业计算卡以及AMD的部分高端显卡都配备了它。

* 1. 数据位数

数据位数指的是在一个时钟周期之内能传送的bit数，它是决定显存带宽的重要因素，与显卡性能息息相关。当显存种类相同并且工作频率相同时，数据位数越大，它的性能就越高。

* 1. 显存容量

显存容量是显卡上本地显存的容量数，这是选择显卡的关键参数之一。显存容量的大小决定着显存临时存储数据的能力，在一定程度上也会影响显卡的性能。显存容量也是随着显卡的发展而逐步增大的，并且有越来越增大的趋势。显存容量从早期的512KB、1MB、2MB等极小容量，发展到64MB、128MB、256MB、512MB、768MB，一直到目前主流的2GB、4GB、6GB和高档显卡的8GB、16GB、32GB某些专业显卡甚至已经具有48GB的显存了。在显卡最大分辨率方面，最大分辨率在一定程度上跟显存有着直接关系，因为这些像素点的数据最初都要存储于显存内，因此显存容量会影响到最大分辨率。

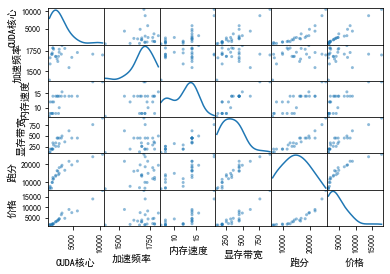
* 1. 显存带宽

显存带宽就是显示芯片与显存之间的桥梁，带宽越大，则显示芯片与显存之间的通讯就越快捷。为了标示这宽度，显存带宽的单位为：字节/秒。显存的带宽与显存的位宽及显存的速度有关。

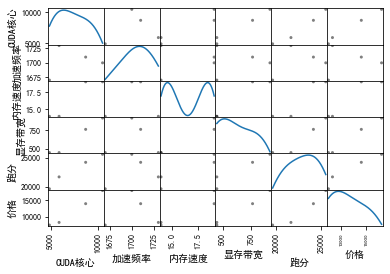
而后，对于显卡价格波动的因素，需要结合显卡价格波动进行综合分析。只选定一款显卡进行价格分析显然是不合理的，但对每一款显卡都进行分析又显得重复，工作量也过于庞大。因此，需要选择合适的标志显卡进行分析。

1. 新显卡发售对旧显卡的影响
2. 时间对显卡价格的影响
3. 数字货币对显卡价格的影响

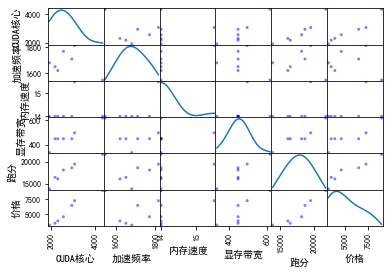
最后，应当分析造成2021年年初以来的显卡价格上涨原因。



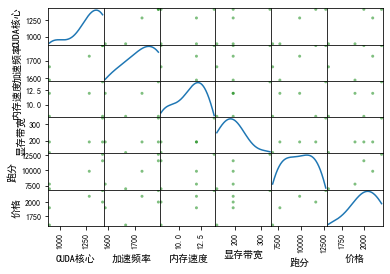
上图列举了部分采集到的影响显卡价格的硬件相关信息，以及他们互相之间的关系。从上图中可以粗略地得出：显卡的价格和显卡的CUDA核心数量、显卡加速频率、显存速度、显存带宽、显卡本身的跑分都有一定的关系，其中，CUDA核心数量、显存带宽、显卡跑分呈现正相关，显卡的加速频率以及显存带宽虽然不直接影响显卡价格，但是决定显卡价格的上界。



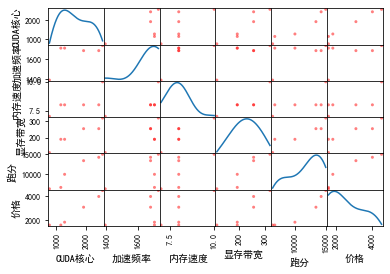
图：RTX30(Ampere)系列



图：RTX20(Turing)系列



图：GTX16(Turing)系列



图：GTX10(Pascla)系列

import pandas as pd

from pandas import DataFrame

import matplotlib.pyplot as plt

import numpy as np

from pylab import mpl

from sklearn.linear\_model import LinearRegression

from sklearn import metrics

from sklearn import model\_selection

mpl.rcParams['font.sans-serif'] = ['MicroSoft YaHei']

data = pd.read\_excel('C:/Users/Fraxinus/Desktop/BigData/GPU.xlsx', 'Sheet2', index\_col=0)

#data.dropna(inplace=True)

ndata = data[['CUDA核心', '加速频率', '内存速度', '显存带宽', '跑分', '价格']]

X = data.iloc[:, 3:13].values.astype(float)

plt.figure()

#pd.plotting.scatter\_matrix(ndata, diagonal='kde')

plt.xticks(fontsize=5)

plt.yticks(fontsize=5)

sdata = data[['Generation', 'CUDA核心']]

sdata.boxplot(by='Generation')

sdata = data[['Generation', '价格']]

sdata.boxplot(by='Generation')

sdata = data[['Generation', '显存带宽']]

sdata.boxplot(by='Generation')

sdata = data[['Generation', '跑分']]

sdata.boxplot(by='Generation')

sdata = data[['Architecture', 'CUDA核心']]

sdata.boxplot(by='Architecture')

sdata = data[['Architecture', '价格']]

sdata.boxplot(by='Architecture')

sdata = data[['Architecture', '显存带宽']]

sdata.boxplot(by='Architecture')

sdata = data[['Architecture', '跑分']]

sdata.boxplot(by='Architecture')

plt.show()

#y = data.iloc[:, 14].values.astype(float)

y = data.iloc[:, 15].values.astype(float)

X\_train, X\_test, y\_train, y\_test = model\_selection.train\_test\_split(X, y, test\_size=0.35, random\_state=4)

linregTr = LinearRegression()

linregTr.fit(X\_train, y\_train)

print(linregTr.intercept\_, linregTr.coef\_)

y\_train\_pred = linregTr.predict(X\_train)

y\_test\_pred = linregTr.predict(X\_test)

train\_err = metrics.mean\_squared\_error(y\_train, y\_train\_pred)

test\_err = metrics.mean\_squared\_error(y\_test, y\_test\_pred)

print("train err = " + str(train\_err), "\ntest err = " + str(test\_err))

pred\_score = linregTr.score(X\_test, y\_test)

print("Predict score is " + str(pred\_score))

随机森林（Random Forests）

随机森林是一种重要的基于Bagging的集成学习方法，可以用来做分类、回归等问题。

随机森林由LeoBreiman（2001）提出，从原始训练样本集N中有放回地重复随机抽取k个样本生成新的训练样本集合，然后根据自助样本集生成k个分类树组成随机森林，新数据的分类结果按分类树投票多少形成的分数而定。其实质是对决策树算法的一种改进，将多个决策树合并在一起，每棵树的建立依赖于一个独立抽取的样品，森林中的每棵树具有相同的分布，分类误差取决于每一棵树的分类能力和它们之间的相关性。特征选择采用随机的方法去分裂每一个节点，然后比较不同情况下产生的误差。能够检测到的内在估计误差、分类能力和相关性决定选择特征的数目。单棵树的分类能力可能很小，但在随机产生大量的决策树后，一个测试样品可以通过每一棵树的分类结果经统计后选择最可能的分类。

随机森林有许多优点：

具有极高的准确率,随机性的引入，使得随机森林不容易过拟合,随机性的引入，使得随机森林有很好的抗噪声能力,能处理很高维度的数据，并且不用做特征选择,既能处理离散型数据，也能处理连续型数据，数据集无需规范化,训练速度快，可以得到变量重要性排序,容易实现并行化

随机森林的缺点：当随机森林中的决策树个数很多时，训练时需要的空间和时间会较大,随机森林模型还有许多不好解释的地方，是个黑盒模型。

极端随机森林

极端随机森林同样是一种多棵决策树集成的分类器，与随机森林分类器比较，主要有两点不同：对于每个决策树的训练集，RF采用的是随机采样bootstrap来选择采样集作为每个决策树的训练集，而extra tree一般不采用随机采样，即每个决策树采用原始训练集。RandomForest应用的是Bagging模型，ExtraTree使用的所有的样本，只是特征是随机选取的，因为分裂是随机的，所以在某种程度上比随机森林得到的结果更加好。在选定了划分特征后，RF的决策树会基于信息增益，基尼系数，均方差之类的原则，选择一个最优的特征值划分点，这和传统的决策树相同。但是Extra tree比较的激进，会随机的选择一个特征值来划分决策树。从第二点可以看出，由于随机选择了特征值的划分点位，而不是最优点位，这样会导致生成的决策树的规模一般会大于RF所生成的决策树。也就是说，模型的方差相对于RF进一步减少，但是bias相对于RF进一步增大。在某些时候，Extra tree的泛化能力比RF更好。

from sklearn import model\_selection

from sklearn.preprocessing import StandardScaler

from sklearn.ensemble import RandomForestRegressor, ExtraTreesRegressor, GradientBoostingRegressor

from sklearn.metrics import r2\_score, mean\_squared\_error, mean\_absolute\_error

import numpy as np

import pandas as pd

'''

随机森林回归

极端随机森林回归

梯度提升回归

通常集成模型能够取得非常好的表现

'''

# 1 准备数据

# 读取GPU信息

data = pd.read\_excel('C:/Users/Fraxinus/Desktop/BigData/GPU.xlsx', 'Sheet2', index\_col=0)

data = data.iloc[0:29, 0:16]

data.dropna(inplace=True)

# 查看数据描述

X = data.iloc[:, 3:13].values.astype(float)

y = data.iloc[:, 14].values.astype(float)

#y = data.iloc[:, 15].values.astype(float)

y = np.ravel(y)

# 2 分割训练数据和测试数据

# 随机采样25%作为测试 75%作为训练

X\_train, X\_test, y\_train, y\_test = model\_selection.train\_test\_split(X, y, test\_size=0.25, random\_state=4)

# 3 训练数据和测试数据进行标准化处理

ss\_X = StandardScaler()

X\_train = ss\_X.fit\_transform(X\_train)

X\_test = ss\_X.transform(X\_test)

ss\_y = StandardScaler()

y\_train = ss\_y.fit\_transform(y\_train.reshape(-1, 1))

y\_test = ss\_y.transform(y\_test.reshape(-1, 1))

# 4 三种集成回归模型进行训练和预测

# 随机森林回归

rfr = RandomForestRegressor()

# 训练

rfr.fit(X\_train, y\_train.ravel())

# 预测 保存预测结果

rfr\_y\_predict = rfr.predict(X\_test)

# 极端随机森林回归

…………………………

# 梯度提升回归

…………………………

# 5 模型评估

# 随机森林回归模型评估

print("随机森林回归的默认评估值为：", rfr.score(X\_test, y\_test))

print("随机森林回归的R\_squared值为：", r2\_score(y\_test, rfr\_y\_predict))

print("随机森林回归的均方误差为:", mean\_squared\_error(ss\_y.inverse\_transform(y\_test),

ss\_y.inverse\_transform(rfr\_y\_predict)))

print("随机森林回归的平均绝对误差为:", mean\_absolute\_error(ss\_y.inverse\_transform(y\_test),

ss\_y.inverse\_transform(rfr\_y\_predict)))

# 极端随机森林回归模型评估

……………………………

# 梯度提升回归模型评估

……………………………

随机森林回归的默认评估值为： 0.9383967247806242

随机森林回归的R\_squared值为： 0.9383967247806242

随机森林回归的均方误差为: 1211000.6066666667

随机森林回归的平均绝对误差为: 738.4999999999999

极端随机森林回归的默认评估值为： 0.978700394240416

极端随机森林回归的R\_squared值为： 0.9383967247806242

极端随机森林回归的均方误差为: 1211000.6066666667

极端随机森林回归的平均绝对误差为: 738.4999999999999

梯度提升回归回归的默认评估值为： 0.9767755934882469

梯度提升回归回归的R\_squared值为： 0.9383967247806242

梯度提升回归回归的均方误差为: 1211000.6066666667

梯度提升回归回归的平均绝对误差为: 738.4999999999999

TensorFlow™是一个基于数据流编程（dataflow programming）的符号数学系统，被广泛应用于各类机器学习（machine learning）算法的编程实现，其前身是谷歌的神经网络算法库DistBelief 。Tensorflow拥有多层级结构，可部署于各类服务器、PC终端和网页并支持GPU和TPU高性能数值计算，被广泛应用于谷歌内部的产品开发和各领域的科学研究。

Keras 是一个用 Python 编写的高级神经网络 API，它能够以 TensorFlow, CNTK, 或者 Theano 作为后端运行。Keras 的核心数据结构是 model，一种组织网络层的方式。最简单的模型是 Sequential 顺序模型，它由多个网络层线性堆叠。

# 全连接神经网络

model = Sequential()

model.add(Dense(units=8, input\_dim=1))

model.add(Activation('relu'))

# 隐藏层1024

model.add(Dense(1024))

model.add(Activation('relu'))

# Dropout层用于防止过拟合

model.add(Dropout(0.1))

model.add(Dense(1024))

model.add(Activation('relu'))

model.add(Dropout(0.1))

model.add(Dense(512))

model.add(Activation('relu'))

model.add(Dropout(0.1))

# 没有激活函数用于输出层，因为这是一个回归问题，我们希望直接预测数值，而不需要采用激活函数进行变换。

model.add(Dense(1))

# 使用高效的 ADAM 优化算法以及优化的最小均方误差损失函数

model.compile(loss='mean\_squared\_error', optimizer=Adam())

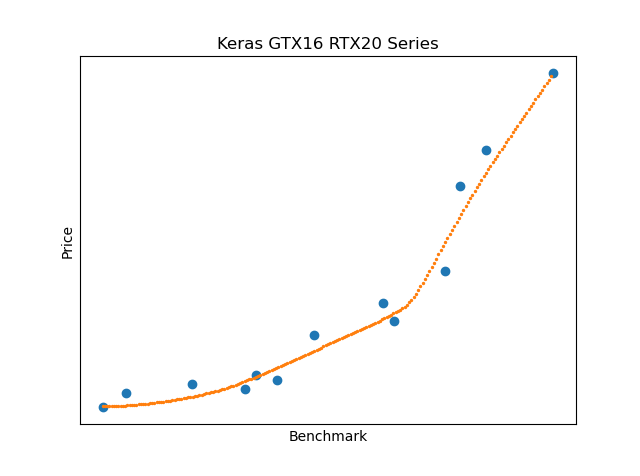
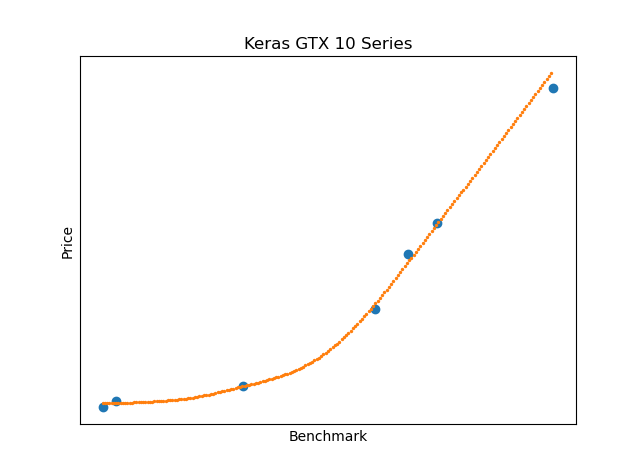
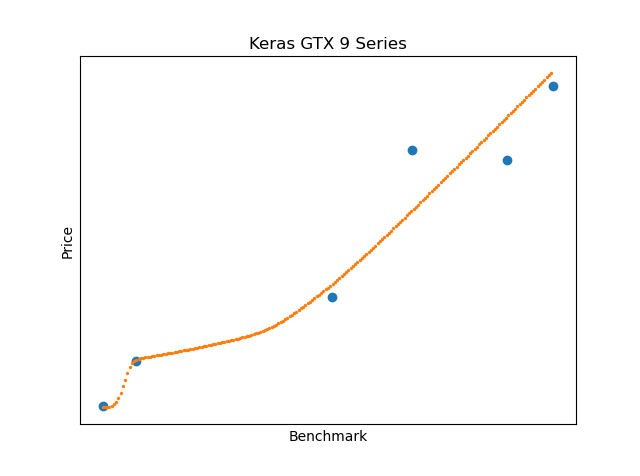
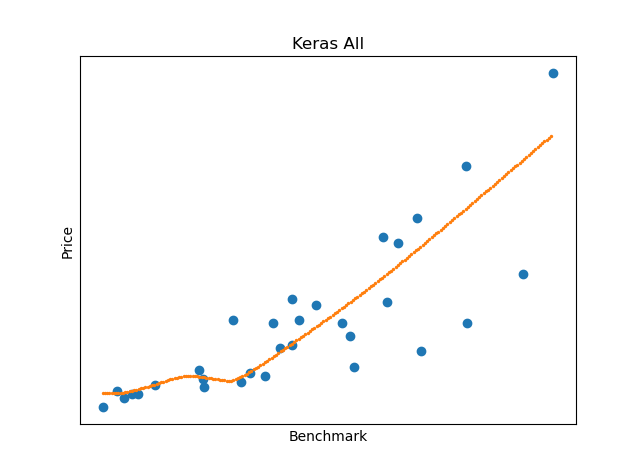
# early stoppping

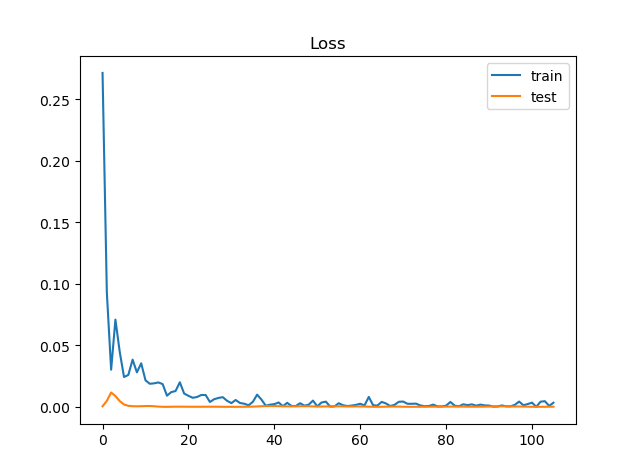
from keras.callbacks import EarlyStopping

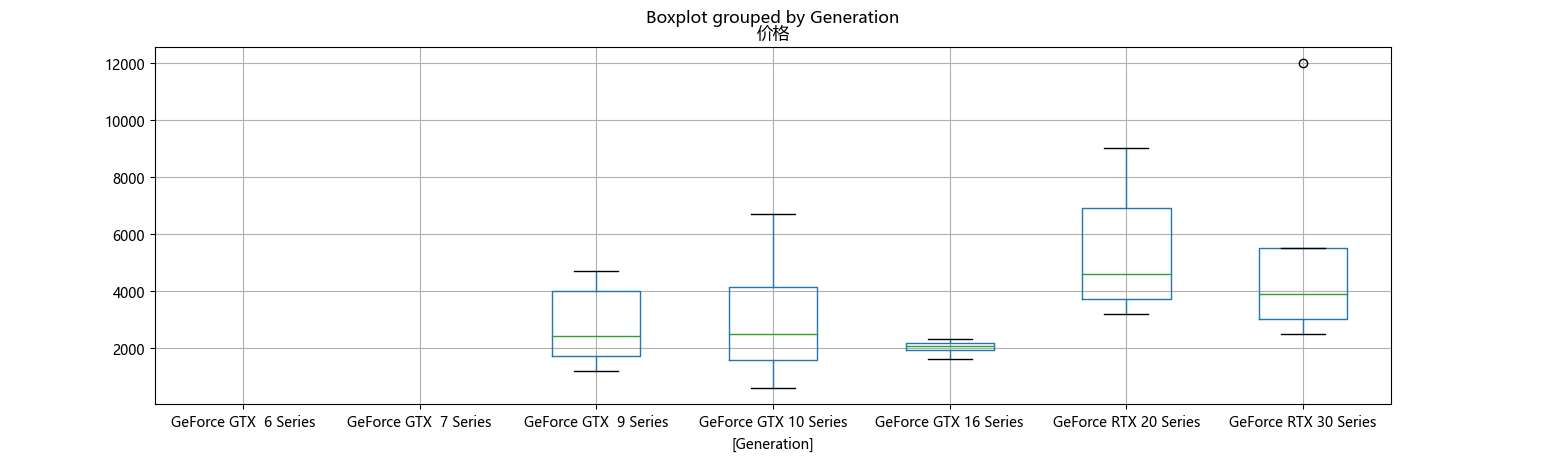
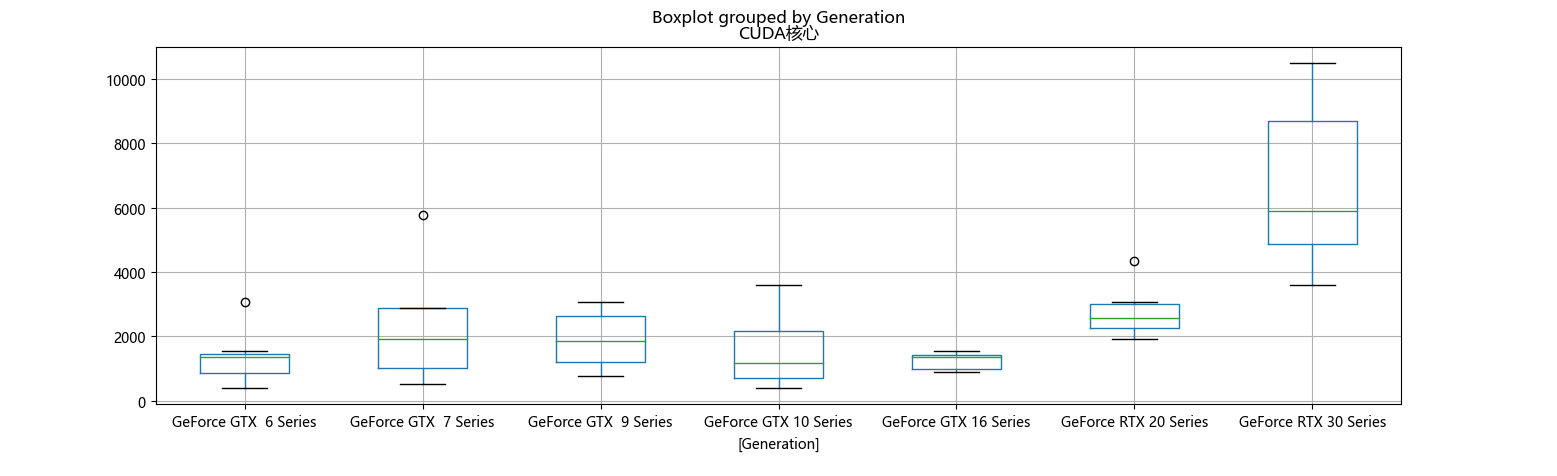
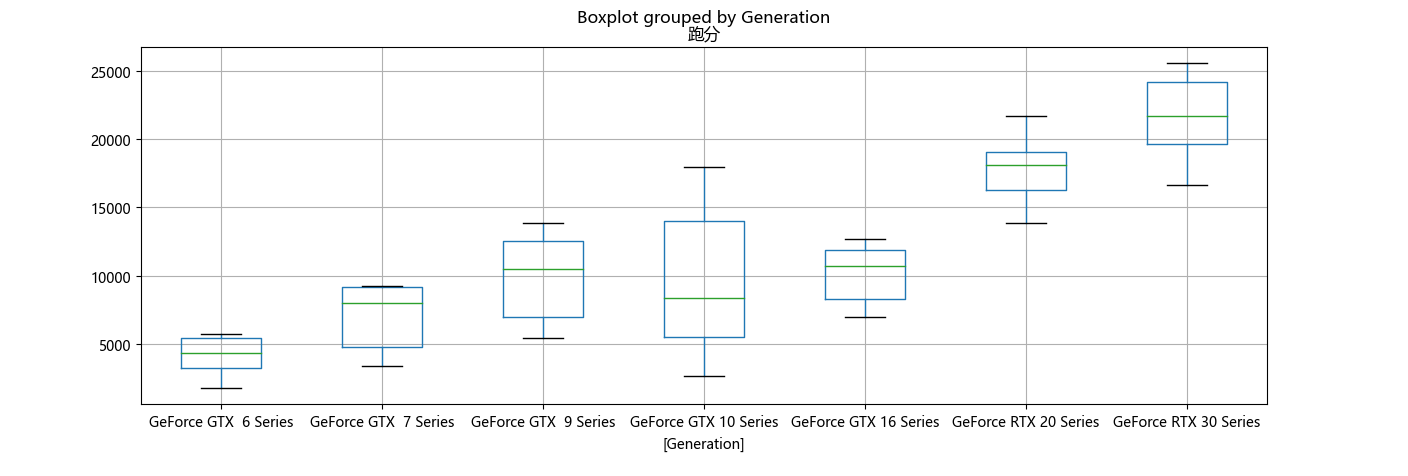
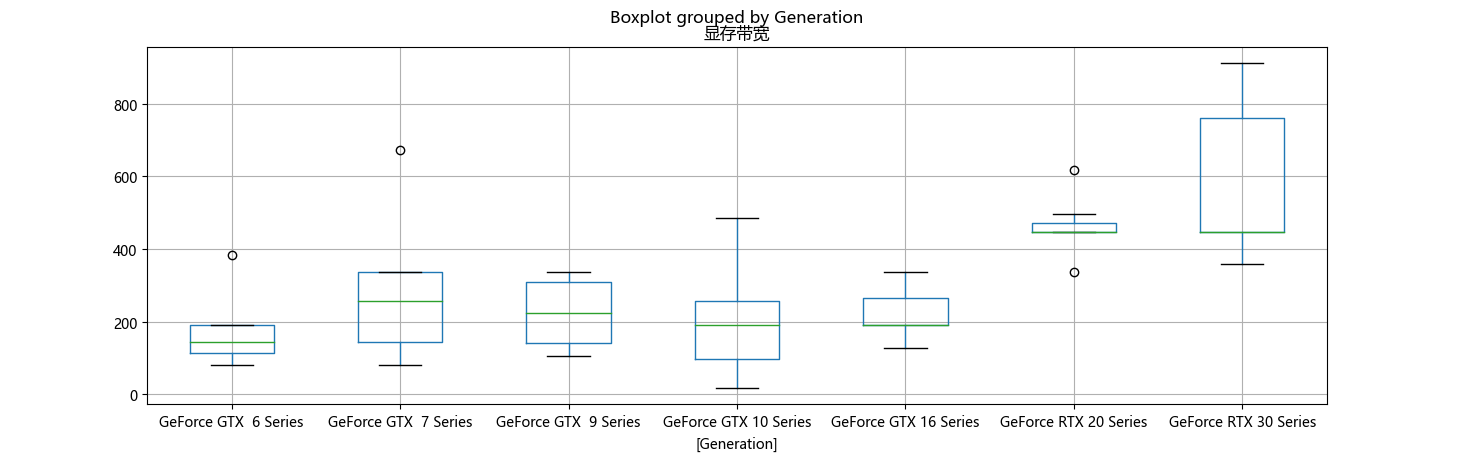
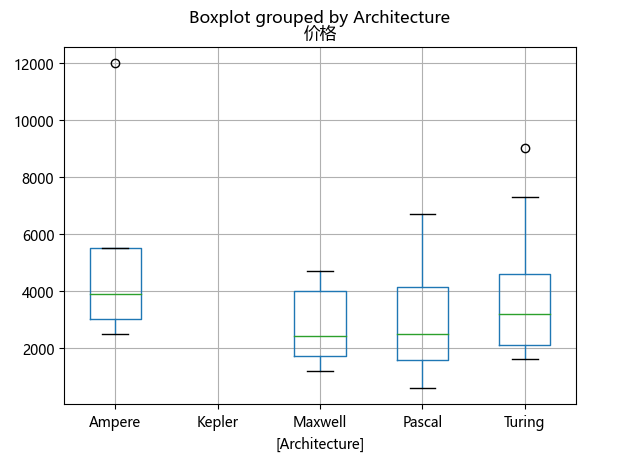
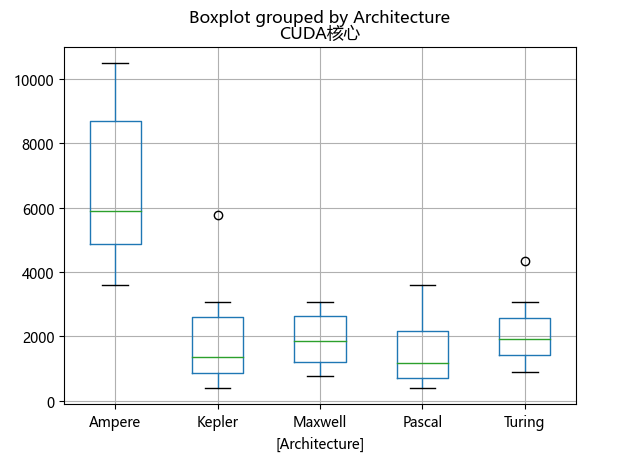
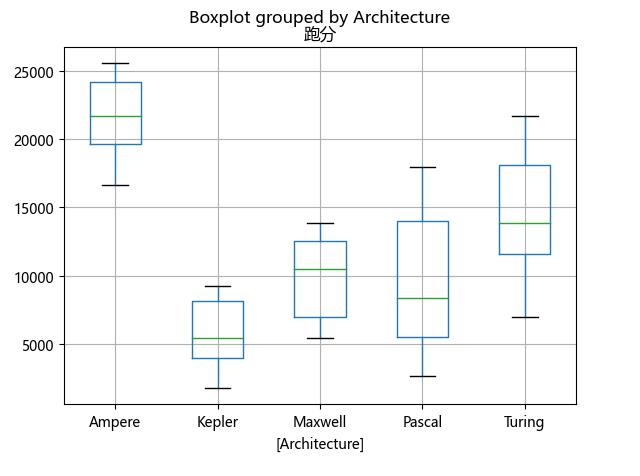
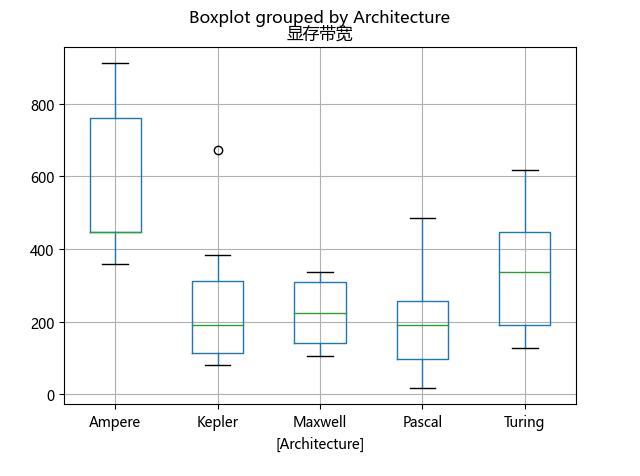
early\_stopping = EarlyStopping(monitor='val\_loss', patience=90, verbose=2)

# 训练

history = model.fit(train\_X, train\_y, epochs=1000, batch\_size=20, validation\_data=(test\_X, test\_y), verbose=2,shuffle=False, callbacks=[early\_stopping])







import matplotlib.pyplot as plt

import numpy as np

from sklearn.cluster import KMeans

import pandas as pd

from pandas import DataFrame

data = pd.read\_excel('C:/Users/Fraxinus/Desktop/BigData/GPU.xlsx', 'Sheet3', index\_col=0)

values = data.iloc[:, 14:16].astype(float)

values.dropna(inplace=True)

X = DataFrame(values)

estimator = KMeans(n\_clusters=6)#构造聚类器

estimator.fit(X)#聚类

label\_pred = estimator.labels\_ #获取聚类标签

#绘制k-means结果

x0 = X[label\_pred == 0]

x1 = X[label\_pred == 1]

x2 = X[label\_pred == 2]

x3 = X[label\_pred == 3]

x4 = X[label\_pred == 4]

x5 = X[label\_pred == 5]

plt.scatter(x0.iloc[:, 0], x0.iloc[:, 1], c = "red", marker='o', label='label0')

plt.scatter(x1.iloc[:, 0], x1.iloc[:, 1], c = "green", marker='o', label='label1')

plt.scatter(x2.iloc[:, 0], x2.iloc[:, 1], c = "blue", marker='o', label='label2')

plt.scatter(x3.iloc[:, 0], x3.iloc[:, 1], c = "orange", marker='o', label='label3')

plt.scatter(x4.iloc[:, 0], x4.iloc[:, 1], c = "yellow", marker='o', label='label4')

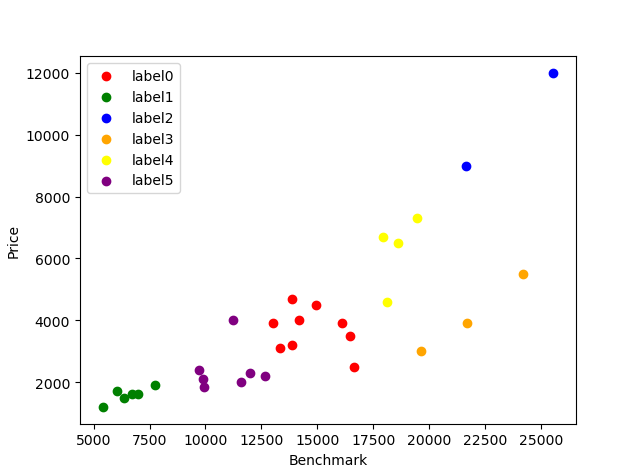
plt.scatter(x5.iloc[:, 0], x5.iloc[:, 1], c = "purple", marker='o', label='label5')

plt.xlabel('Benchmark')

plt.ylabel('Price')

plt.legend(loc=2)

plt.show()



import numpy as np

import pandas as pd

from pandas import Series, DataFrame

import matplotlib.pyplot as plt

import scipy.signal

data = pd.read\_csv('C:/Users/Fraxinus/Desktop/BigData/dat.csv', index\_col=0)

data = DataFrame(data)

data.dropna(axis=1, inplace=True)

data.plot()

#plt.show()

x = data.iloc[:, 0]

print(x)

h = np.array([0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])

a = scipy.signal.convolve(x, h)

a = DataFrame(a[9:-10])

print(a)

a.plot()

plt.show()

