项目：Android系统中的系统异常及检测机制研究

**异常检测程序测试报告**

二○二三年八月

**目 录**

[1 性能异常测试报告 1](#_Toc135916670)

[1.1 整体报告 1](#_Toc135916671)

[1.1.1 概述 1](#_Toc135916672)

[1.1.2 测试目标 1](#_Toc135916673)

[1.2 测试环境 1](#_Toc135916674)

[1.3 测试方案 2](#_Toc135916675)

[1.4 测试结果 2](#_Toc135916676)

[2 功耗异常测试报告 7](#_Toc135916677)

[2.1 测试环境 7](#_Toc135916678)

[2.2 测试方案 7](#_Toc135916679)

[2.2.1 短时异常检测 7](#_Toc135916680)

[2.2.2 长时异常检测 8](#_Toc135916681)

[2.2.3 异常定位 8](#_Toc135916682)

[2.3 测试结果 8](#_Toc135916683)

[2.3.1 短时异常检测 8](#_Toc135916684)

[2.3.2 长时异常检测 10](#_Toc135916685)

[2.3.3 异常定位 10](#_Toc135916686)

[3 模型开销分析 11](#_Toc135916687)

[3.1 客户端——服务器部署测试方案 11](#_Toc135916688)

[3.2 本地部署 12](#_Toc135916689)

[3.3 部署方式总结 14](#_Toc135916690)

[4 附录 15](#_Toc135916691)

本文中的实验均以一加 9 Pro 手机作为硬件平台，其配置如表1所示。为了防止非原生系统可能对实验的干扰，本文禁用了设备上的 Bootloader 安全机制，并通过 fastboot 刷入了基于 Android 12 的 LineageOS 19.2 类原生操作系统，其 Linux 内核版本为 5.4.191，同时刷入了 Magisk 工具以获取设备上的 Root 权限。

表1 实验平台

|  |  |
| --- | --- |
| 相关平台 | 名称 |
| 手机型号 | OnePlus 9 Pro LE2120 |
| Android版本 | Android 12 |
| 操作系统版本 | Lineage OS 19-20221101-NIGHTLY-lemonadep |
| 内存大小 | 12GB(可用内存：10.99GB) |
| CPU版本 | 高通骁龙™ 888 移动平台 |
| CPU核心数目 | 8 |
| GPU版本 | Qualcomm, Adreno (TM) 660 |

# **性能异常测试报告**

本章会从性能异常检测机制测试环境、测试结果两方面对性能异常检测进行说明。

## **整体报告**

### 概述

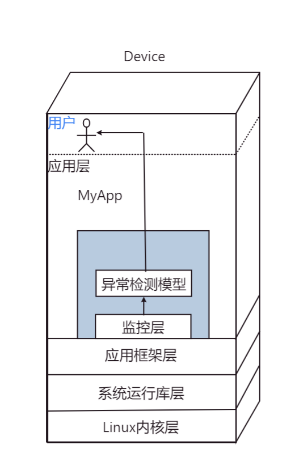
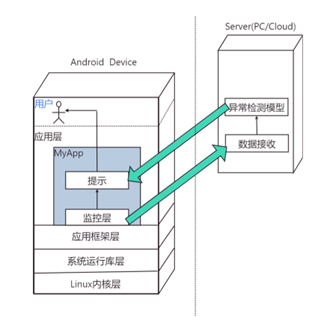
本文主要通过使用CPU异常、GPU异常以及内存异常三个决策树模型随系统是否处于性能异常进行检测，检测的机理是利用手机资源不足时整机和前台应用的资源使用情况都会和正常情况下的不同，采集与资源使用相关的监控数据，用于训练机器学习模型：模型的输入为各项监控数据，模型的输出为异常分类结果。模型训练完成后，可对实时数据进行推理，分类各种情况的异常。

### 测试目标

本文的测试目标在于验证基于决策树的性能异常检测算法的可行性，获得在真实环境中引入的开销，为后期优化提供方向。

## **测试环境**

实验测试分为基于本地检测的架构以及基于服务器云端检测的架构，如图1.1(a)(b)所示。根据系统结构图可知，本系统主要三个主要的功能层：一是监控层，主要负责记录系统状态，包括监控 APP 的 CPU、内存、温度等指标，二是异常检测层，这是整个系统最核心的部分，主要负责建立异常检测的机器学习模型，采用基于服务器云端检测的框架时还需要在服务器上接收来自客户 APP 发送的信息并经过模型推理后传回给 APP，相关的客户端实现工作作为一个在 Android 平台上开发的 APP 实现，开发语言为 Java。监控服务作为一个前台Service运行在手机上，决策树模型运行在本地时在前台Service里，运行在服务器上时是python程序，以Socket方式通信。

(a)本地架构 (b)服务器架构

图1.1 不同实验结构

## **测试方案**

对于三种异常检测决策树模型（CPU 异常、GPU 异常、内存异常），实验对三种常见类型的 APP 进行了实验，一是淘宝，该类应用主要对应的是社交购物类应用程序；二是原神，该类应用主要对应的是大型游戏类应用程序；三是抖音，该类应用主要对应的是视频类应用程序。通过对这三个典型应用程序构建性能异常的决策树模型，可以对不同前台下的模型效果有一个普适性的认知。每种异常检测模型只能检测对应类型的异常，对于混合异常的检测需要通过三个模型的结果进行判断。

## **测试结果**

对于 CPU 异常的检测，表 1.1、表 1.2以及表 1.3显示了针对三种不同应用程

序的 CPU 异常检测决策树模型在测试集上的训练结果。

表1.1 针对前台为淘宝应用的 CPU 异常检测模型效果

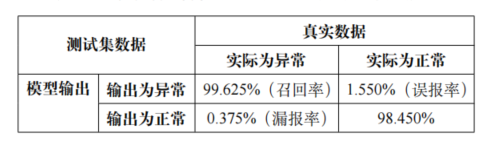


表1.2 针对前台为原神应用的 CPU 异常检测模型效果

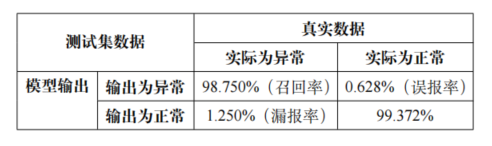
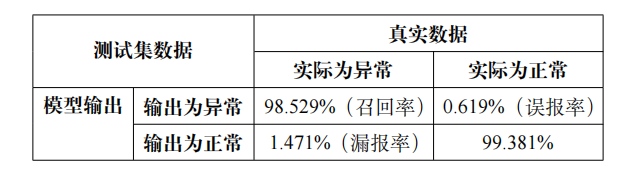


表1.3 针对前台为抖音应用的 CPU 异常检测模型效果



根据上述的实验结果，对于三个不同前台应用建立的 CPU 异常决策树模型的异常误报率（正常情况误报成异常）、异常漏报率（异常情况漏报成正常）均很低，且都具有较高的召回率（异常数据中正确识别的比例）。模型效果良好。除了决策树模型，在实验的过程中，还选取了朴素贝叶斯分类模型以及 SVM分类模型这两种模型进行对比，如图 1.2所示，通过对比 F1 − score 发现，对于三种不同的前台应用，决策树模型都具有最好的效果，SVM 模型效果最差，SVM 模型与朴素贝叶斯两个模型在精度上效果较差，会产生许多误报的情况，模型效果不尽如人意。

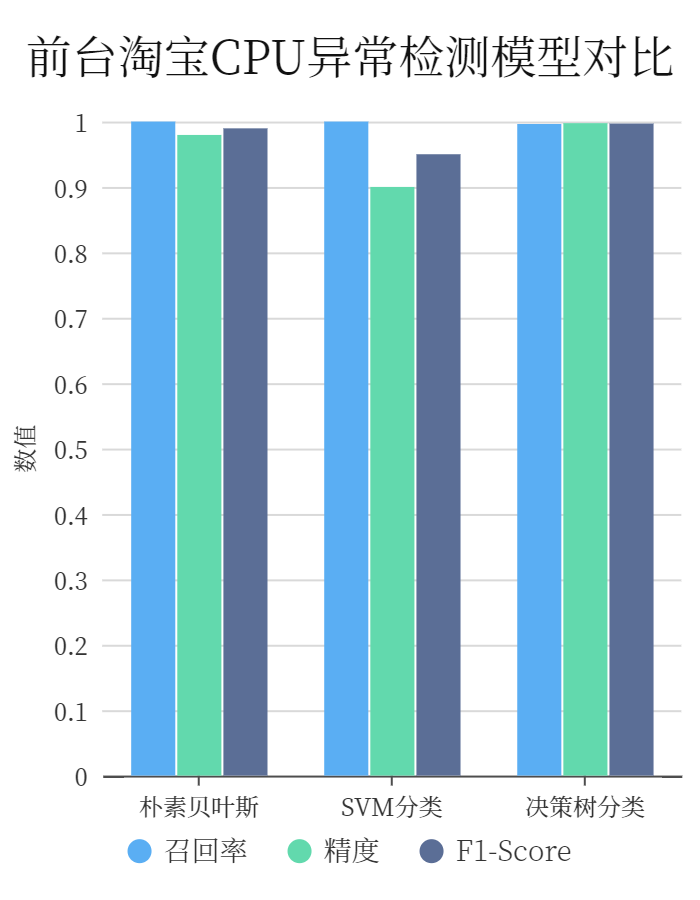
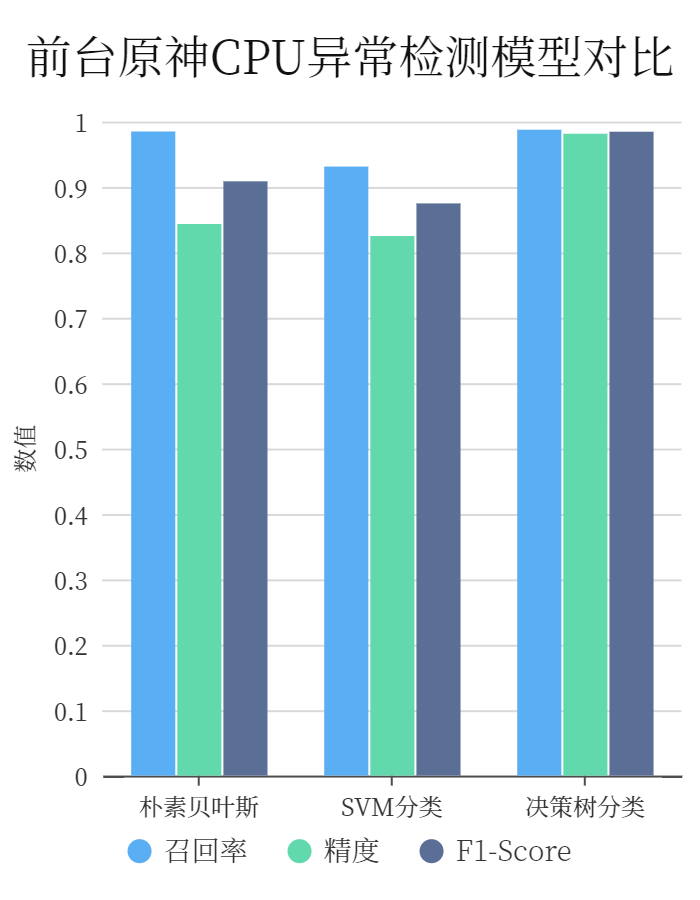
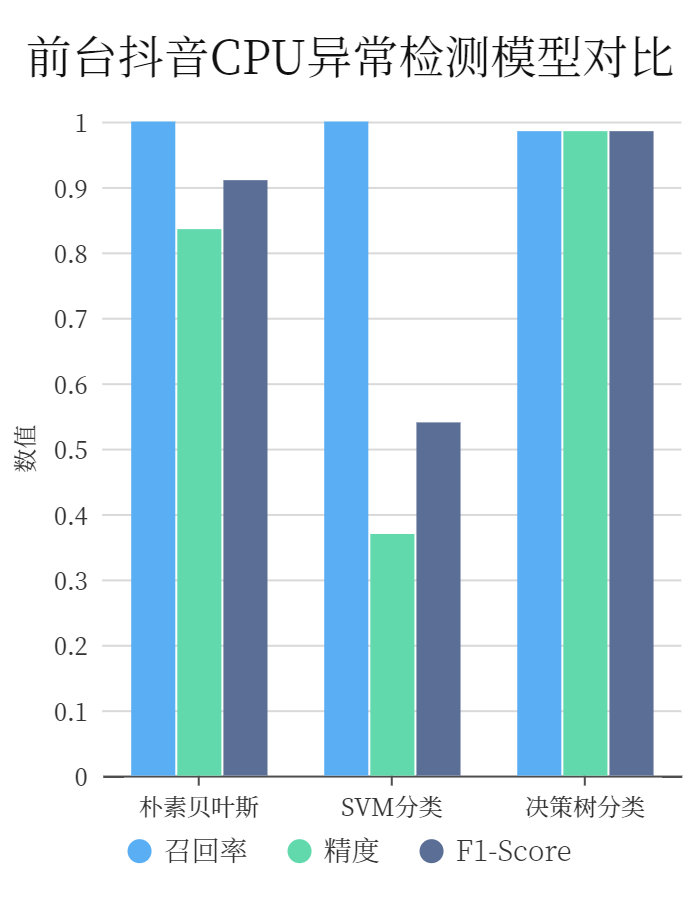
 

图1.2 不同CPU异常检测模型效果对比

同理对于内存异常的检测，表 1.4、表 1.5以及表 1.6显示了针对三种不同应用程序的内存异常检测决策树模型在测试集上的训练结果。

表1.4 针对前台为淘宝应用的内存异常检测模型效果

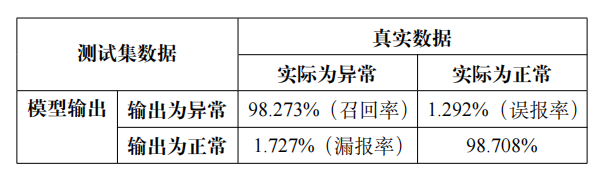


表1.5 针对前台为原神应用的内存异常检测模型效果

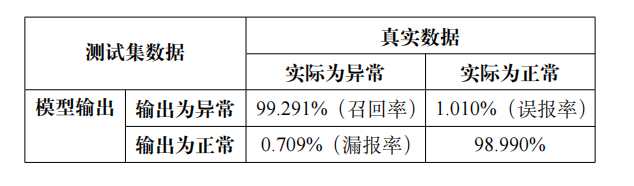
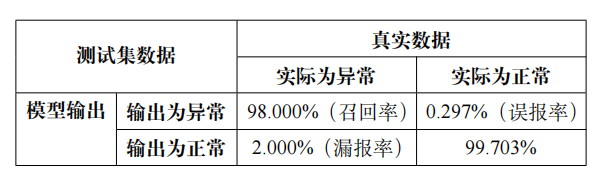


表1.6 针对前台为抖音应用的内存异常检测模型效果



同理对于GPU异常的检测，表 1.7、表 1.8以及表 1.9显示了针对三种不同应用程序的 GPU异常检测决策树模型在测试集上的训练结果。

表1.7 针对前台为淘宝应用的 GPU 异常检测模型效果

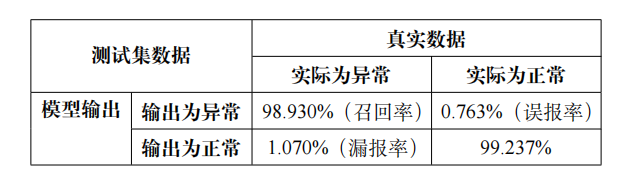


表1.8 针对前台为原神应用的 GPU 异常检测模型效果

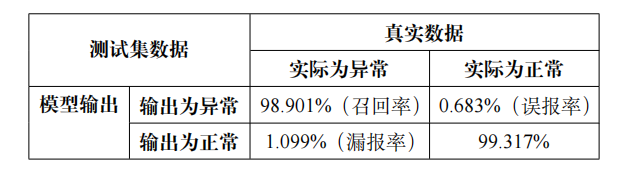
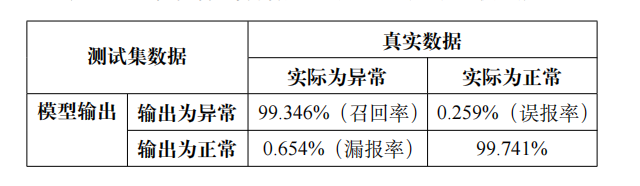
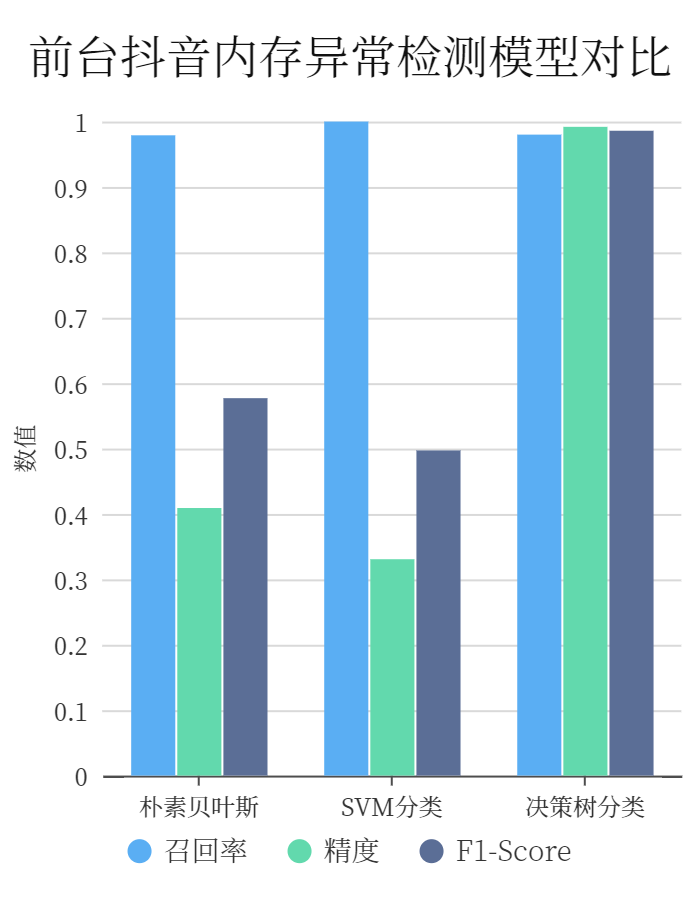
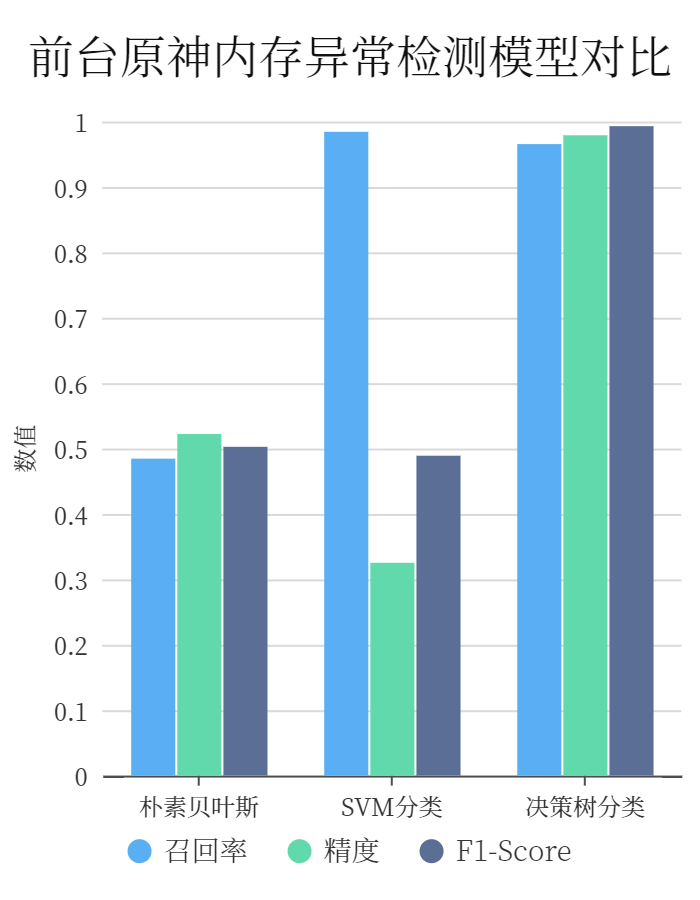
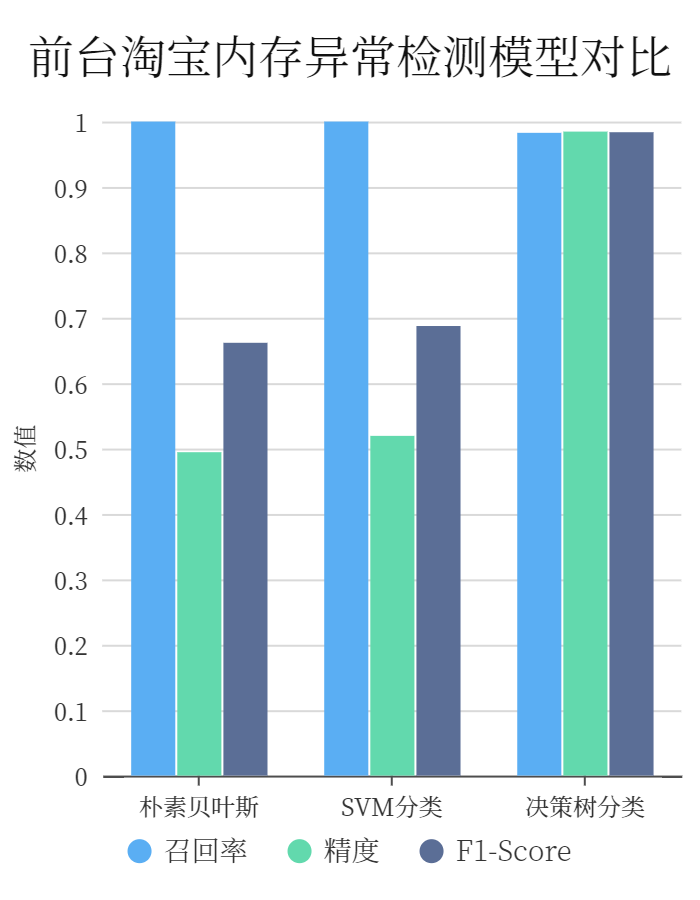


表1.9 针对前台为抖音应用的 GPU 异常检测模型效果



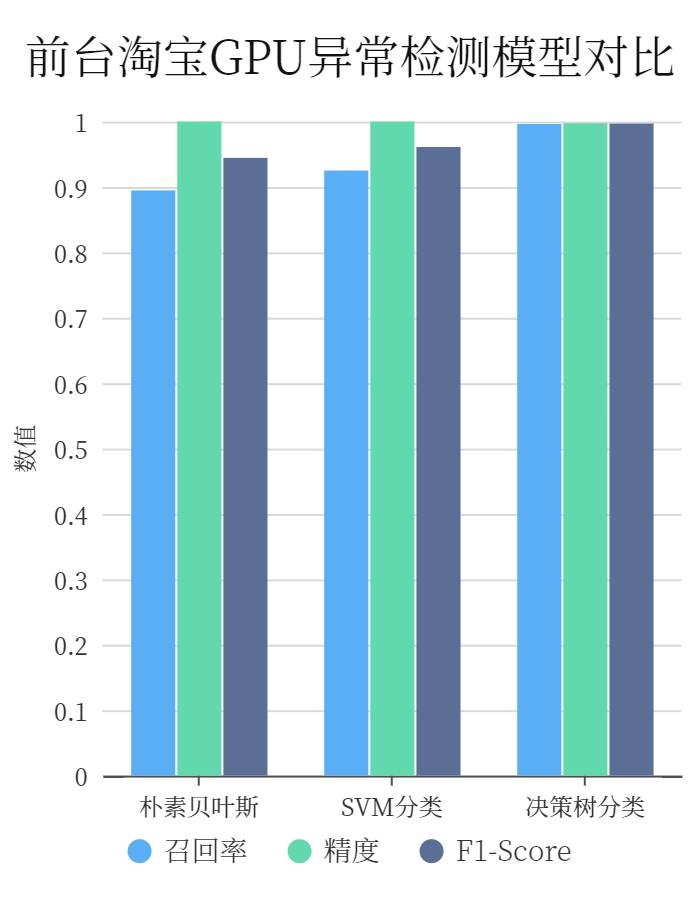
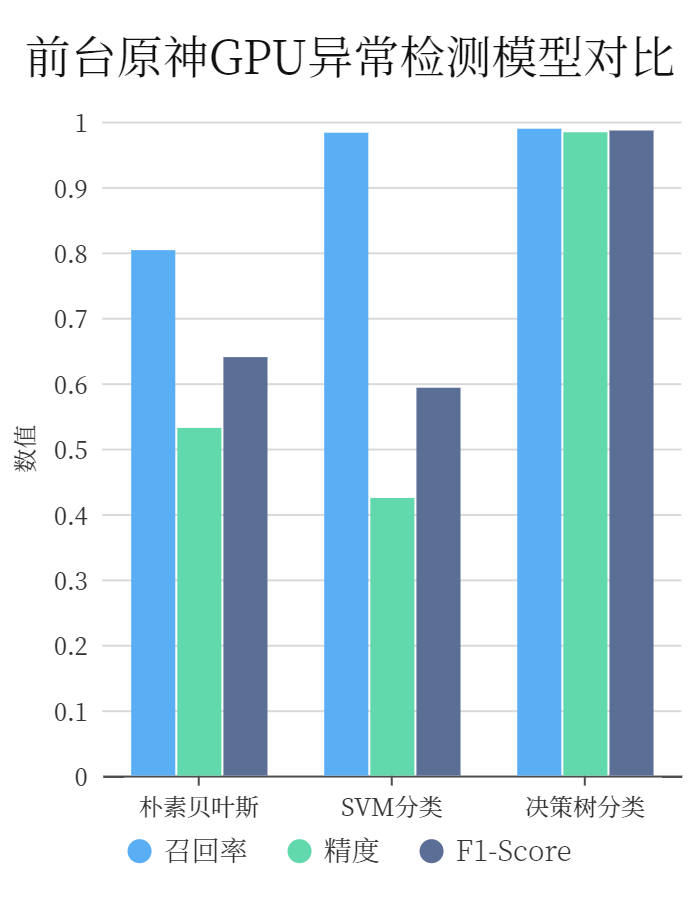
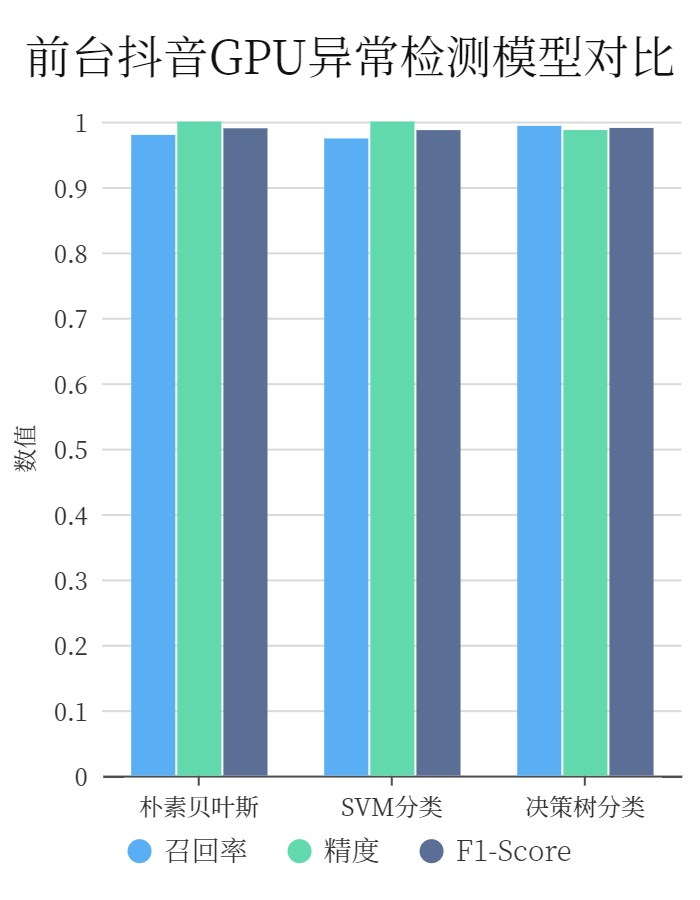
根据上述的实验结果，对于三个不同前台应用建立的 GPU 异常决策树模型的异常误报率（正常情况误报成异常）、异常漏报率（异常情况漏报成正常）均很低，且都具有较高的召回率（异常数据中正确识别的比例），模型效果良好。

后面两种模型同样对比了SVM以及朴素贝叶斯两个baseline,发现决策树模型效果最好。



(a) (b) (c)

图1.3 不同内存异常检测模型效果对比

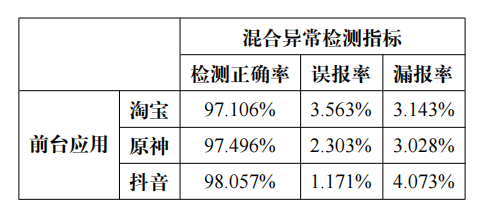
  

(a) (b) (c)

图1.4 不同GPU异常检测模型效果对比

当认为三个检测模型时彼此独立的时候，根据概率乘法定理，得出进行混合异常检测时的模型效果如表1.10。

表1.10 混合异常效果



对于整机模型，当采用整机模型进行检测时，准确率如下：CPU：100%，Mem：100%，GPU： 99.95%

此处的准确率可能偏高，因为实验中模拟的异常应用行为比较激进，导致整机资源使用情况波动明显。在实际情况下，整机模型的准确率可能会有一定下降，但前台应用的行为改变较小。

# **功耗异常测试报告**

为了验证功耗异常检测算法和功耗异常定位算法的有效性和实际效果，本章会对异常检测算法中应用的FCN方法和CNN方法进行评估并与其他基线算法作对比，同时启动多个异常应用，评估异常定位算法有无漏报和误报。

## **测试环境**

测试过程中的负载基本和数据采集流程一致，在实验中，本文使用多类应用作为前后台应用模拟了设备上的不同负载。本文随机运行了多个常见的后台应用（如微信、QQ、网易云音乐）并使这些应用在后台运行，然后随机运行 1 个前台应用，并保持该前台应用运行 30 分钟。在此过程中，异常应用可能会在随机的时间中触发。

## **测试方案**

### 短时异常检测

为了检验模型的效果，本文在实验设备上采集了超过 5000 条（约 41 小时）的数据作为数据集，其中训练集和测试集的比例约为 。本文使用了三层 MLP 算法作为基线算法，并使用模型在测试集上的准确率（accuracy）、精确率（precision）、召回率（recall）作为两者评价的指标，其计算公式如式2.1所示：

其中 TP、TN、FP、FN 分别表示真阳性、真阴性、假阳性和假阴性的样本个数。

此外为衡量模型的准确性，需构建一个函数衡量模型输出的好坏，这一般是真实结果和模型输出结果之间的差异，称为损失函数(Loss Function)，本文使用交叉熵损失函数作为模型的损失函数，其计算公式如式2.2所示：

其中 是样本数量， 是预先设置的类别的数量， 取值为 0 或 1，当第 个样本的真实类别为第 类，则 取 1，当第 个样本的真实类别不是第 类，则 取 0。 是模型输出的概率，即模型预测第 个样本为第 类的概率，在模型中由 softmax 函数给出。

### 长时异常检测

对于长时异常，本文采集了超过 1300 条（约 217 小时）的数据作为数据集，训练集和测试集的比例约为 。为了检验不同模型的效果，实验首先使用了 sklearn 库中的 GridSearchCV 函数对所有模型进行参数搜索，对模型的参数进行微调，并使用最高准确率的模型作为对比模型。

实验选择了模型训练和推理所用的时间和模型在测试集上的 5 类异常的判别准确率作为评价指标。

### 异常定位

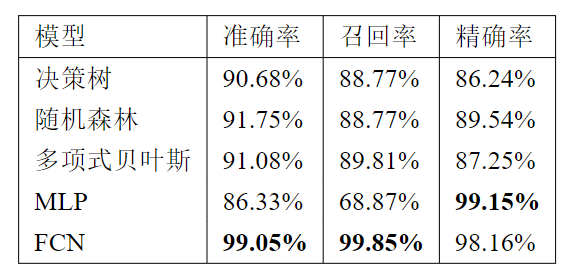
当异常定位算法定位到异常的应用后，会将应用的信息保存到文件中。为了检验异常定位算法的效果，本文运行了一个或两个异常应用。运行约 30 分钟后，读取记录异常应用信息的文件并人工判断文件记录的异常应用是否都是先前运行的异常应用，是否存在漏判和错判的情况。

## **测试结果**

### 短时异常检测

通过对比决策树算法、随机森林算法、贝叶斯算法、MLP 和 FCN 模型的准确率，数据显示 FCN 模型的准确率更高，两种深度学习训练过程中的准确率变化如图2.1所示，在模型迭代次数 1000 次时训练得到的模型的各个指标如表2.1所示，为了更方便观察，最优的结果被加粗表示，可见 FCN 的各项指标都较好，MLP 模型精确率较高，但是召回率和准确率均较低，表明 MLP 模型对于异常的识别不够灵敏。

表2.1混合异常效果



同时，本实验还对 FCN 模型的损失函数进行了分析，如图2.2所示，FCN 模型的训练集的损失函数在训练过程中会收敛到 0 附近，说明该模型的训练效果较好。

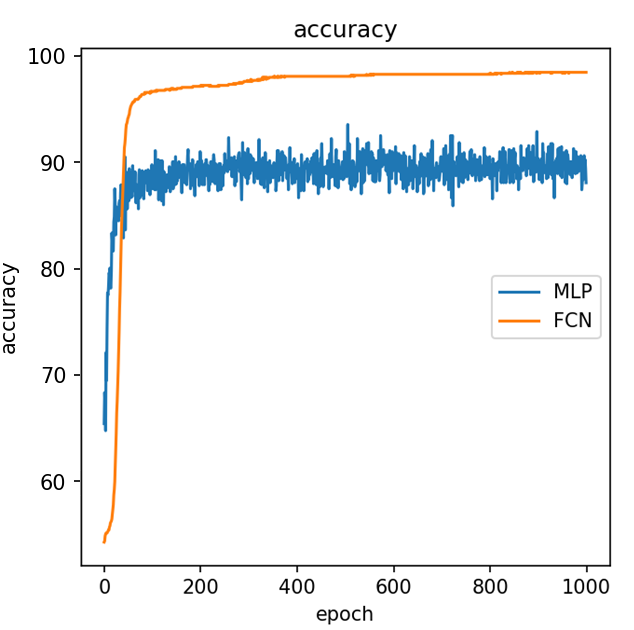


图2.1 训练过程中模型准确率变化

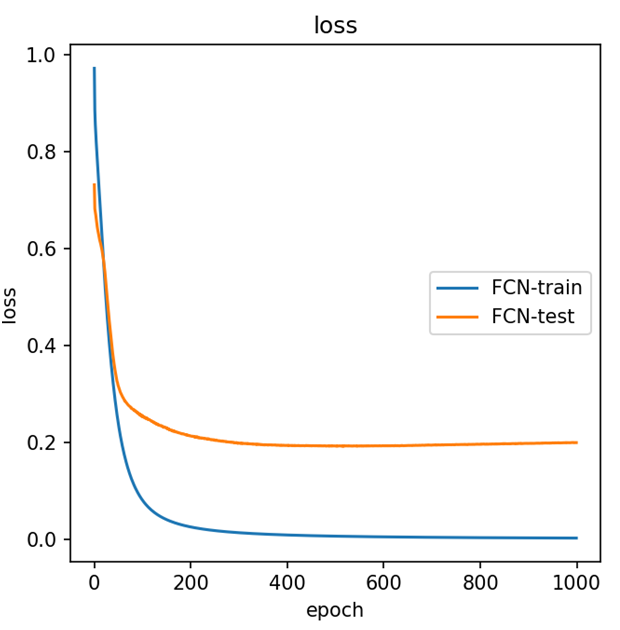
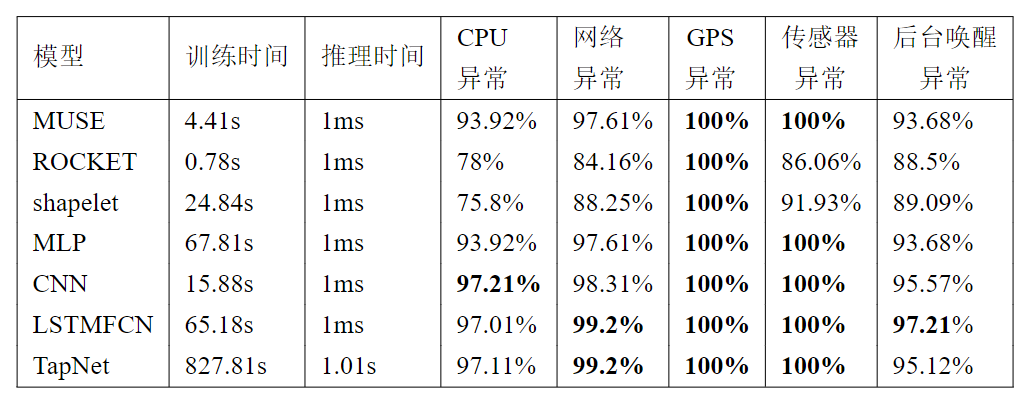


图2.2 训练过程中 FCN 模型损失函数变化

### 长时异常检测

表2.2为7个时间序列分类模型在测试平台上的训练时间、推理时间和模型对于本文的长时异常数据集中5 类异常的准确率表。为了更方便观察，最优的结果被加粗表示。

表2.2 长时异常模型效率与准确率



得益于在卷积神经网络处理时序数据时，能够捕获数据的时间相关性，以提取更准确的特征，基于卷积神经网络的模型 CNN 和 LSTM-FCN 都在较短的训练时间和推理时间上获得了较好的效果。传统机器学习方法与 MLP 对于时间序列数据的时序变化特征提取效果不佳，对于部分异常情景不能有效识别，因此准确率较低。而同样基于卷积神经网络的 Tapnet 算法的结构复杂度较高，推理时间过长，模型部署到手机上可能会对手机的性能和续航产生较大影响。

### 异常定位

在实验场景下，经过多次不同前台应用测试，对于大部分异常类型，单异常应用和双异常应用识别准确率均为 100%。但对于短时网络异常，定位算法会有小概率会错误地将后台的音乐应用识别为异常应用，误报的原因是由于后台音乐应用在下载音乐时也会频繁地访问网络，导致识别错误。用户也可以将识别错误的异常应用加入白名单。

# **模型开销分析**

## 客户端——服务器部署**测试方案**

当使用客户端——服务器部署方式时，异常检测模型位于服务器端，通过建立 Android 设备与目标服务器之间的 Socket 套接字 TCP 连接，将采集到的数据即模型的输入处理之后发送到服务器，由服务器推理之后将模型预测结果返回给 Android 设备。

对于客户端——服务器部署方式的开销主要包括评估客户端 Android 设备的CPU 使用、内存使用、能量消耗以及网络流量四个方面，实验评估结果如下：

通过对监控状态下的 CPU 开销进行统计，发现 CPU 利用率呈现周期性变化，每次监控结束进行数据保存和发送给服务器时，CPU 利用率会有短暂上升，其他监控时间段的 CPU 利用率约等于 0。Android Profiler 呈现 CPU 波动情况如图1.5所示，横轴表示时间，纵轴表示 CPU 利用率，App 指我们的应用，Others 是其他应用（淘宝等），虚线表示线程数，其周期对应一次采样时间间隔。图3.1的开销主要是进行监控数据保存的时间点 t，峰值的 CPU 占用率大概会占用 1%—4% 的 CPU 资源。当进行中间的监控时间时，对于 CPU 资源的利用率几乎可以忽略不计。



图3.1 客户端——服务器部署异常检测的 CPU 资源周期性变化

通过使用 adb shell dumpsys cpuinfo 对 10 分钟内的 CPU 平均利用率进行统计，本文得出了采样频率为 10s 时，10 分钟内平均 CPU 利用率大概为 0.4375%，CPU占用率极低，不会造成性能问题。

图3.2显示了程序运行时的占用内存情况，由于应用程序以 APP 运行在 Android 手机终端，占用内存比实现在系统层有较大的提升，由于模型的异常检测在服务器端实现，本地主要是监控模块以及监控数据发送给服务器的网络通信模块，整个 APP 的大小在 95MB 左右，通过 adb shell dumpsys meminfo 统计占用内存的情况发现 APP 的 Total PSS 也在 95MB 左右。

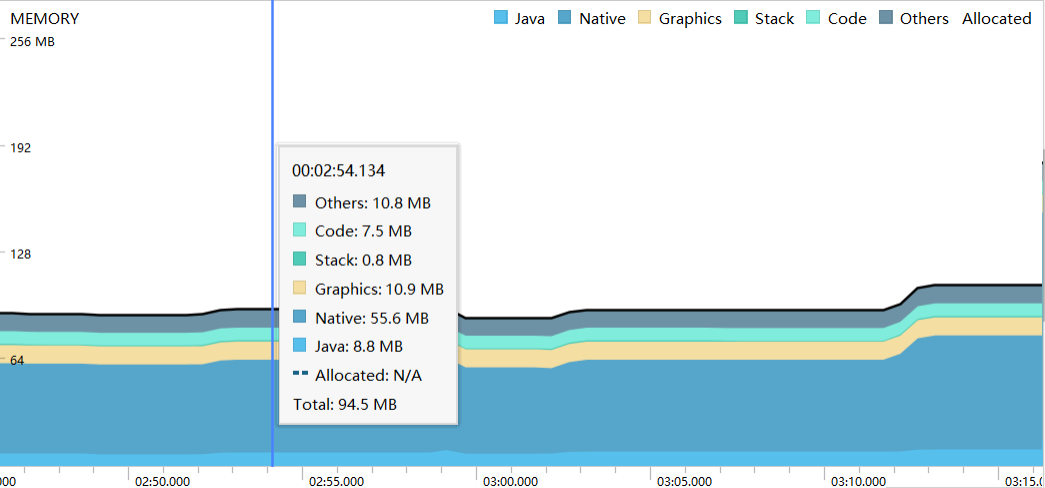


图3.2客户端——服务器部署异常检测的内存开销

通过对监控状态下的能耗开销进行统计，发现 Android 系统能耗呈现周期性变化，每次监控结束进行数据保存发送时，能耗会有短暂上升，其他监控时间段的能耗较低。Android Profiler 呈现能量消耗情况如图 3.3所示，横轴表示时间，纵轴表示能耗，App 指我们的应用，Others 是其他应用（淘宝等），虚线表示线程数，其周期对应一次采样时间间隔。图1.7的能耗开销主要包括监控数据保存与发送的时间点，应用程序对相关资源消耗仍然均为轻量（Light）。

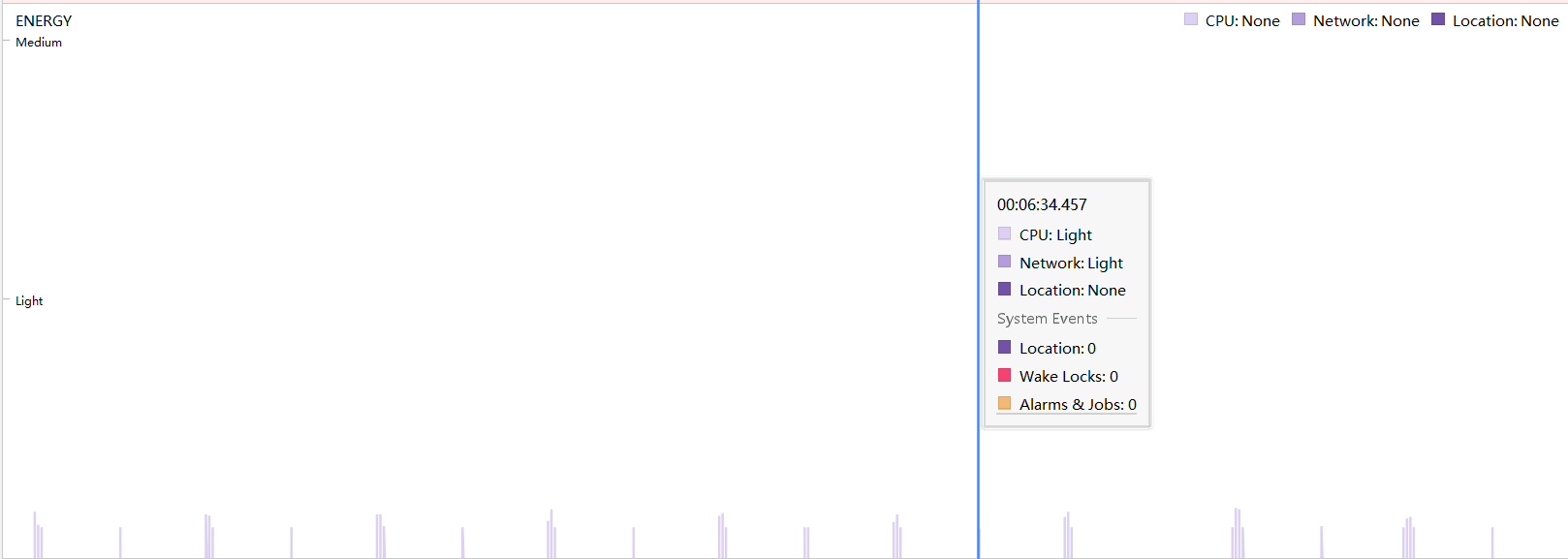


图3.3 客户端——服务器部署异常检测的能耗开销

通过使用 adb shell dumpsys batterystats 对于 10 分钟内能耗消耗进行统计，本文得出了采样频率为 10s 时，运行在后台 10 分钟内的能耗资源平均开销大概为0.428mAh，在实验设备上测得息屏待机 10 分钟消耗的电量约为 2.6257mAh，相当于息屏待机 16.3%，能耗均处于很低的水平。

## **本地部署**

本地部署的主要方式是将相关模型在Android系统端进行实现，基于由外部PC使用Sklearn框架训练出的模型经过sklearn2pmml导出pmml模型和pmml-android序列化，将其部署到Android系统端。

本地部署模型的开销主要包括CPU使用、内存使用、能量消耗三个方面，本地部署无需网络传输，对于网络资源无消耗。实验评估结果如下：

通过对于监控状态下的CPU开销进行了统计，发现CPU利用率呈现周期性变化，每次监控结束进行模型推理时以及数据保存时，CPU利用率会有短暂上升，其他监控时间段的CPU利用率约等于0。Android Profiler呈现进行异常检测的CPU波动如图1.8所示，横轴表示时间，纵轴表示CPU利用率，App指我们的应用，Others是其他应用（淘宝等），虚线表示线程数，其周期对应一次采样时间间隔。图3.4的开销主要包括两个时间节点，一是进行本地异常检测的时间点，二是进行监控数据保存的时间点，当进行本地异常检测时，峰值的CPU占用率大概会占用1-6%的CPU资源。当处于中间的监控时间时，对于CPU资源的利用率几乎可以忽略不计。



图3.4 本地部署异常检测的CPU资源周期性变化

通过使用adb shell dumpsys cpuinfo对于10分钟内CPU平均利用率统计，本文得出了采样频率为10s时，都进行异常检测的平均CPU利用率大概为0.412%，不会影响前台的性能。

图3.5显示了程序运行时的占用内存情况，由于应用程序以APP应用在Android手机终端，占用内存比实现在系统层有较大的提升，整个APP的大小在120MB左右，通过adb shell dumpsys meminfo统计占用内存的情况发现APP的Total PSS也在120MB左右，但是用于实现异常检测的决策树模型本身占用内存大小并不大。

|  |
| --- |
|  |
|  |  |

图3.5本地部署异常检测的内存开销

通过对于监控状态下的能耗开销进行了统计，发现Android系统能耗呈现周期性变化，每次监控结束进行模型推理时以及保存时，能耗会有短暂上升，其他监控时间段的能耗较低，总体变化趋势与CPU趋势相同。Android Profiler呈现接连两次采样都能量消耗情况如图3.6所示，横轴表示时间，纵轴表示能耗，App指我们的应用，Others是其他应用（淘宝等），虚线表示线程数，其周期对应一次采样时间间隔。图1.10的能耗开销主要包括两个时间节点：一是进行本地异常检测的时间点，二是进行监控数据保存的时间点，当进行本地异常检测时，能耗会有短暂上升，但是应用程序对相关资源消耗仍然均为轻量（Light）。

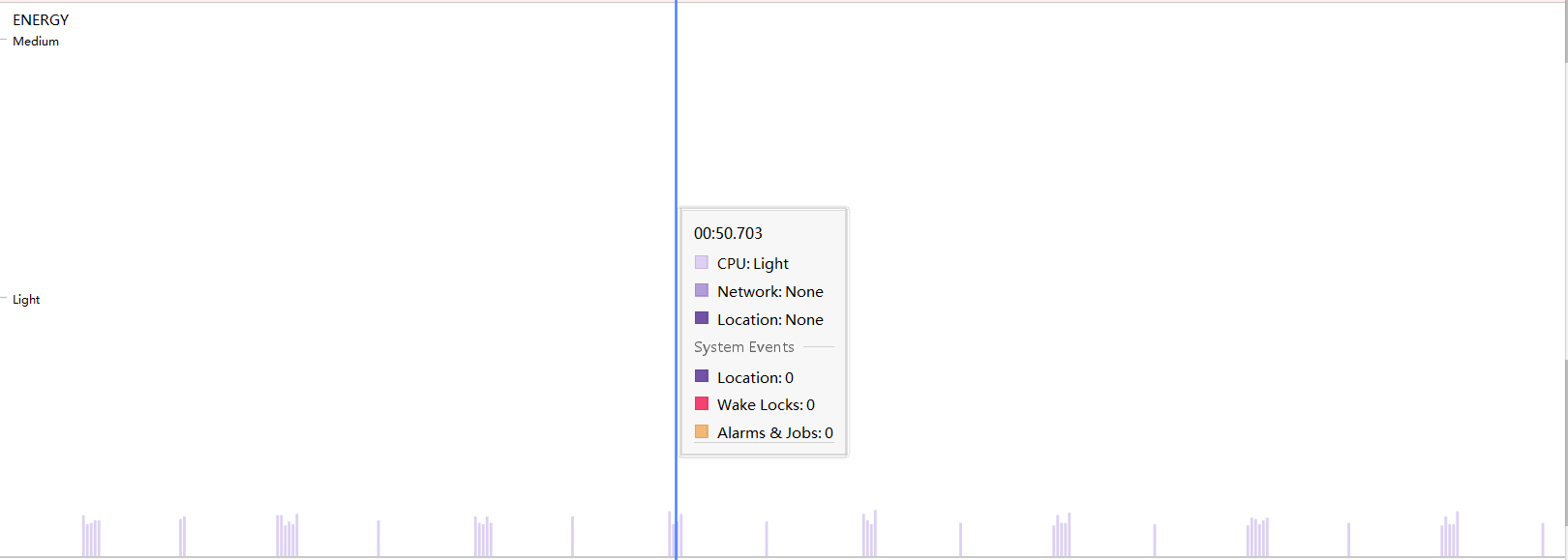


图3.6本地部署异常检测的能耗开销

通过使用adb shell dumpsys batterystats对于10分钟内能耗消耗进行统计，本文得出了采样频率为10s时，平均10分钟能耗大概为0.429mAh，在实验设备上测得息屏待机10分钟消耗的电量约为2.6257mAh，相当于息屏待机16.3%，能耗均处于很低的水平。

本地部署方式下，进行一次异常检测的模型推理时间开销如图3.7所示，三个决策树性能异常检测模型平均每个模型从载入到运行完成一次推理时间在12ms左右，时间开销可以接受。

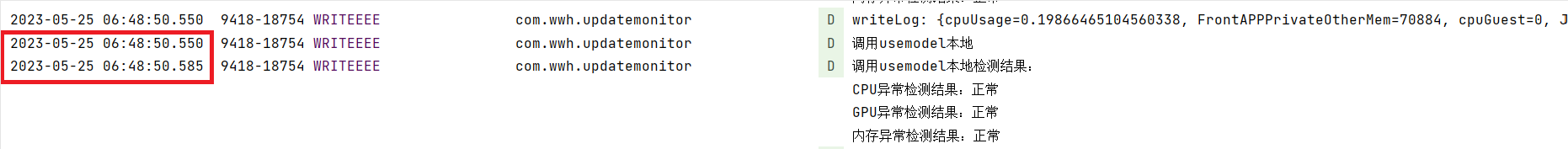


图3.7本部部署模型推理时间开销

## 部署方式总结

综上，对比两种部署方式，使用服务器通信的方法部署模型在内存占用、CPU使用以及能量消耗上均比本地部署消耗更低，具有更好的效果，对于对本机数据有隐私保护需求的用户，可以采用本地部署的方式，两种模型对于资源的消耗均在可以接受的范围内，一般不会因为程序本身给用户带来新的问题。

# **附录**

本阶段提交的文件清单如下所示：

.

├─code

│ ├─performance

│ │ ├─performance\_Decision # 性能异常检测机器学习模型

│ │ ├─pmml-android-master # 决策树模型序列化

│ │ ├─server # 远程服务器代码

│ │ └─updatemonitor # 性能异常检测APP代码

│ └─power

│ ├─power\_model # 功耗异常检测深度学习模型代码

│ ├─powermonitorapp # 功耗异常检测应用

│ └─powerdataacquisitionapp # 功耗异常数据采集应用

├─模型训练手册

├─代码使用手册

└─异常检测程序测试报告 # 本文件