基于 MobileNetV2 的垃圾分类实验

组长：樊施成　组员：于秉宏、万晨阳

1、实验简介

MindSpore 是最佳匹配 Ascend（昇腾）芯片的开源 AI 计算框架，同时也支持 CPU、GPU 平台。访问 MindSpore 官网了解更多：https://www.mindspore.cn/

深度学习计算中，从头开始训练一个实用的模型通常非常耗时，需要大量计算能力。常用的数据如 OpenImage、ImageNet、VOC、COCO 等公开大型数据集，规模达到几十万甚至超过上百万张。网络和开源社区上通常会提供这些数据集上预训练好的模型。大部分细分领域任务在训练网络模型时，如果不使用预训练模型而从头开始训练网络，不仅耗时，且模型容易陷入局部极小值和过拟合。因此大部分任务都会选择预训练模型，在其上做微调（也称为 Fine-Tune）。

本实验以 MobileNetV2+垃圾分类数据集为例，主要介绍如在使用 MindSpore 在 CPU/GPU 平台上进行 Fine-Tune。

垃圾分类信息：

```
{
    '干垃圾': ['贝壳', '打火机', '旧镜子', '扫把', '陶瓷碗', '牙刷', '一次性筷子', '脏污衣服'],
    '可回收物': ['报纸', '玻璃制品', '篮球', '塑料瓶', '硬纸板', '玻璃瓶', '金属制品', '帽子',
'易拉罐', '纸张'],
    '湿垃圾': ['菜叶', '橙皮', '蛋壳', '香蕉皮'],
    '有害垃圾': ['电池', '药片胶囊', '荧光灯', '油漆桶']
}

['贝壳', '打火机', '旧镜子', '扫把', '陶瓷碗', '牙刷', '一次性筷子', '脏污衣服',
'报纸', '玻璃制品', '篮球', '塑料瓶', '硬纸板', '玻璃瓶', '金属制品', '帽子', '易拉罐', '纸张',
'菜叶', '橙皮', '蛋壳', '香蕉皮',
'电池', '药片胶囊', '荧光灯', '油漆桶']

['Seashell', 'Lighter', 'Old Mirror', 'Broom', 'Ceramic Bowl', 'Toothbrush', 'Disposable
Chopsticks', 'Dirty Cloth',
'Newspaper', 'Glassware', 'Basketball', 'Plastic Bottle', 'Cardboard', 'Glass Bottle',
'Metalware', 'Hats', 'Cans', 'Paper',
'Vegetable Leaf', 'Orange Peel', 'Eggshell', 'Banana Peel',
'Battery', 'Tablet capsules', 'Fluorescent lamp', 'Paint bucket']
```

脚本、预训练模型的 Checkpoint 和数据集组织为如下形式：

```
├── main.ipynb # 入口Jupyter Notebook文件
│
├── src_mindspore
│   ├── dataset.py
│   ├── mobilenetv2.py
│   └── mobilenetv2-200_1067_gpu_cpu.ckpt
│
├── results/mobilenetv2.mindir # 待生成的MindSpore0.5.0模型文件
│
├── train_main.py # 将 main.ipynb Notebook 训练模型代码转化为py文件
│
└── datasets/5fbdf571c06d3433df85ac65-momodel/garbage_26x100/ # 数据集
    ├── train/
    ├── val/
    └── label.txt
```

2、实验过程

2.1 超参数简介

本实验中提供的超参数代码部分如下所示:

```
# 训练超参
config = EasyDict({
    "num_classes": 26, # 分类数，即输出层的维度
    "reduction": 'mean', # "mean"或"max"，指定 Head 部分池化采用的方式，可根据需求修改
    "image_height": 224,
    "image_width": 224,
    "batch_size": 24, # 批量大小，根据 CPU 性能调整以避免训练卡顿
    "eval_batch_size": 10, # 评估时的批量大小，可根据实际情况调整
    "epochs": 4, # 训练周期数，可尝试修改以提升模型精度
    "lr_max": 0.01, # 最大学习率，可尝试修改以优化训练效果
    "decay_type": 'constant', # 学习率衰减类型，如"constant"、"exponential"等，可尝试修改以优化训练效果
    "momentum": 0.8, # 动量，可尝试修改以优化训练效果
    "weight_decay": 3.0, # 权重衰减率，可尝试修改以优化训练效果
    "dataset_path": "./datasets/5fbdf571c06d3433df85ac65-momodel/garbage_26x100",
    "features_path": "./results/garbage_26x100_features",
    "class_index": index,
    "save_ckpt_epochs": 1,
    "save_ckpt_path": './results/ckpt_mobilenetv2',
    "pretrained_ckpt": './src_mindspore/mobilenetv2-200_1067_cpu_gpu.ckpt',
    "export_path": './results/mobilenetv2.mindir'
})
```

2.2 解题思路

2.2.1 轮次和批次不够大

首先我们尝试调大 batch 和 epoch，可以将最终得分提高至 70 分左右，但是难以继续突破。

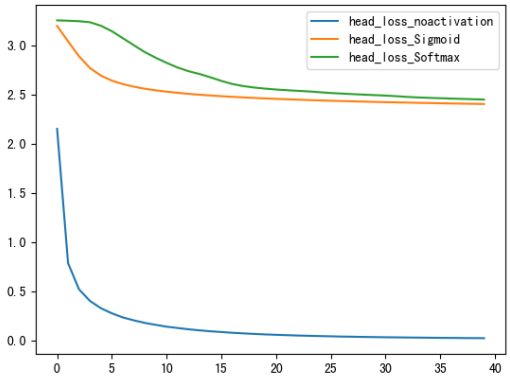
```
    "batch_size": 64, # 鉴于 CPU 容器性能，太大可能会导致训练卡住
    "epochs": 100, # 请尝试修改以提升精度
```

测试点	状态	时长	结果
在 130 张照片上测试模型	✓	15s	得分: 73.08

2.2.2 是否使用激活函数

我们发现原本的代码中没有使用激活函数，因此尝试进行修改。

```
    def __init__(self, input_channel=1280, num_classes=1000,
has_dropout=False, activation="None"):
```



将默认的 None 修改为 Softmax 和 Sigmoid 以后，发现效果没有明显提升，因此不修改。

a source of non-linearity. Additionally, we find that it is important to remove non-linearities in the narrow layers in order to maintain representational power. We demonstrate that this improves performance and provide an intuition that led to this design.

其实我们可以从 MobileNetv2 的论文中看出这个问题，我们的实践也论证了这个观点的准确性。

2.2.3 数据集划分

在原本的代码中（被注释掉的部分）默认都会处理成 train 数据集，数据集划分错误。因此 eval 的时候需要注明。修改后的代码（未注释的部分）如下：

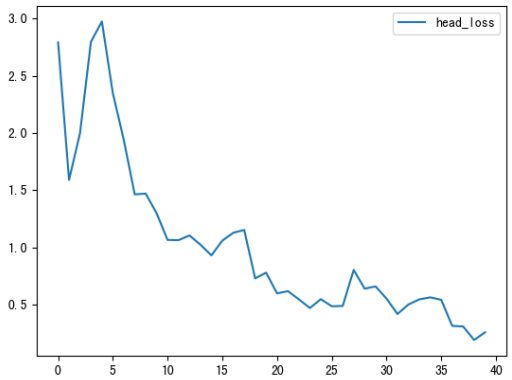
```
#train_dataset = create_dataset(config=config)
#eval_dataset = create_dataset(config=config)
train_dataset = create_dataset(config=config, train = True)
eval_dataset = create_dataset(config=config, train = False)
```

2.2.4 修改优化器

对于以下代码，我们尝试修改优化器，更换成了 Adam：

```
opt = nn.Momentum(head.trainable_params(), lrs, config.momentum,
config.weight_decay)
    # opt = nn.Adam(params=head.trainable_params(),
learning_rate=config.lr_max, weight_decay=config.weight_decay,
use_nesterov=True)
```

但是发现效果并不如原本的 Momentum 优化器，因此不修改。

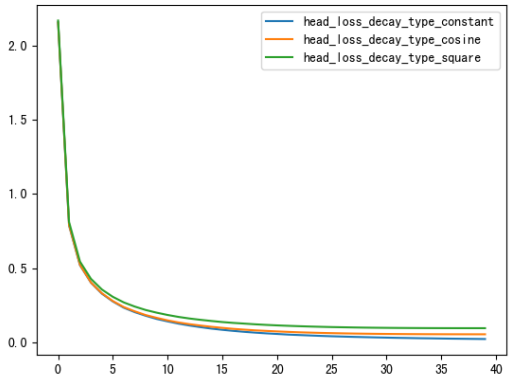


图表 1 使用 Adam 优化器的 loss 曲线

2.2.5 调整学习率变化策略

学习率优化的策略是:

```
lrs = build_lr(config.epochs * step_size,lr_init=config.lr_init,
lr_max=config.lr_max, warmup_steps=config.warmup_steps,
decay_type=config.decay_type)
```



原本的 lr_init 和 warmup_steps 都是 0。尝试修改 lr_init 为 0.01，warmup_steps 为 5，

也即经历 5 步之后从 0.01 到 0.1，再根据 cosine 曲线下降学习率，但是效果没有显著提升，因此不做修改。并尝试使用各种不同的下降策略，比如 square 和 constant，最后对比如图所示。最终选定为 lr_init 和 warmup_steps 都是 0，下降曲线为 cosine

2.2.6 双训练策略（backbone 微调）

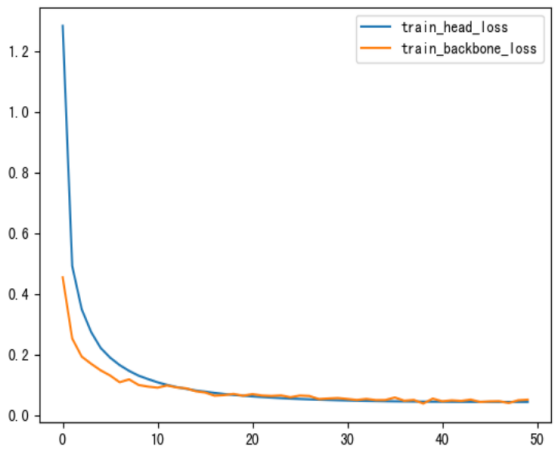
我们调整了训练策略，在训练 head 之后固定 head，微调 backbone。

代码如下：

```
def train_backbone():
    train_dataset = create_dataset(config=config2, training=True,
repeat=3)
    eval_dataset = create_dataset(config=config2, training=False)
    step_size = train_dataset.get_dataset_size()

    backbone = MobileNetV2Backbone()
    head = MobileNetV2Head(input_channel=backbone.out_channels,
num_classes=config.num_classes, reduction=config.reduction,
has_dropout=config.has_dropout, activation=config.activation)
    network = mobilenet_v2(backbone, head)

    # 加载预先训练好的头部权重
    load_checkpoint(os.path.join(config.save_ckpt_path, f"mobilenetv2-
{config.epochs}.ckpt"), net=network)

    # 解冻背部参数以进行微调
    for param in backbone.get_parameters():
        param.requires_grad = True

    for param in head.get_parameters():
       param.requires_grad = False

    loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True,
reduction='mean')
    lrs = build_lr(config2.epochs * step_size,
lr_init=config2.lr_init, lr_max=config.lr_max,
warmup_steps=config2.warmup_steps, decay_type=config2.decay_type)
    opt = nn.Momentum(network.trainable_params(), lrs,
config2.momentum, config2.weight_decay)

    net_with_loss = nn.WithLossCell(network, loss)
    train_step = nn.TrainOneStepCell(net_with_loss, opt)
    train_step.set_train()

    # 重新训练
```

```
    history = list()
    for epoch in range(config2.epochs):
        epoch_start = time.time()
        losses = []
        data_iter =
train_dataset.create_dict_iterator(output_numpy=True)
        for data in data_iter:
            image = Tensor(data["image"])
            label = Tensor(data["label"])
            losses.append(train_step(image, label).asnumpy())
        epoch_seconds = (time.time() - epoch_start)
        epoch_loss = np.mean(np.array(losses))

        history.append(epoch_loss)
        print("epoch: {}, time cost: {}, avg loss: {}".format(epoch +
1, epoch_seconds, epoch_loss))
        if (epoch + 1) % config.save_ckpt_epochs == 0:
            save_checkpoint(network,
os.path.join(config.save_ckpt_path,
f"mobilenetv2_backbone_{epoch+1}.ckpt"))
    print('validating the model...')
    eval_model = Model(network, loss, metrics={'acc', 'loss'})
    acc = eval_model.eval(eval_dataset, dataset_sink_mode=False)
    print(acc)
    return history
```

在采取双训练策略后，模型训练的效果有了一定提升，模型最终得分突破 80 分。



测试点	状态	时长	结果
在 130 张照片上测试模型	✓	15s	得分: 80.0

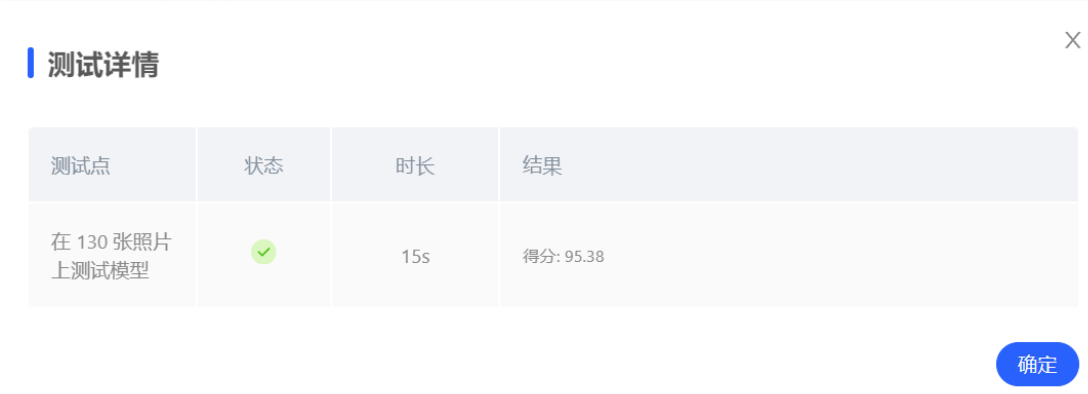
```
2023-12-26 10:20:13.646200 epoch: 32, time cost: 25.177817344665527, avg loss: 0.10888128727674484
2023-12-26 10:20:40.788000 epoch: 33, time cost: 26.952067613601685, avg loss: 0.10826720297336578
2023-12-26 10:21:11.548800 epoch: 34, time cost: 30.51374912261963, avg loss: 0.10171453654766083
2023-12-26 10:21:48.516100 epoch: 35, time cost: 36.682568073272705, avg loss: 0.11207113415002823
2023-12-26 10:22:24.125700 epoch: 36, time cost: 35.486202001571655, avg loss: 0.10442236810922623
2023-12-26 10:22:55.433800 epoch: 37, time cost: 31.09327220916748, avg loss: 0.09706676751375198
2023-12-26 10:23:30.227700 epoch: 38, time cost: 34.49223971366882, avg loss: 0.09837847948074341
2023-12-26 10:24:10.156300 epoch: 39, time cost: 39.71071982383728, avg loss: 0.09596900641918182
2023-12-26 10:24:43.877100 epoch: 40, time cost: 33.44046664237976, avg loss: 0.09045547991991043
2023-12-26 10:25:11.246100 epoch: 41, time cost: 27.21184754371643, avg loss: 0.09351005405187607
2023-12-26 10:25:38.420300 epoch: 42, time cost: 26.983229637145996, avg loss: 0.09068454802036285
2023-12-26 10:26:06.157600 epoch: 43, time cost: 27.523974657058716, avg loss: 0.09352456033229828
2023-12-26 10:26:33.702000 epoch: 44, time cost: 27.363704681396484, avg loss: 0.09015733748674393
2023-12-26 10:27:03.103200 epoch: 45, time cost: 29.26657271385193, avg loss: 0.08482315391302109
2023-12-26 10:27:30.067300 epoch: 46, time cost: 26.83229160308838, avg loss: 0.09471592307090759
2023-12-26 10:27:59.625400 epoch: 47, time cost: 29.29099941253662, avg loss: 0.09084121137857437
2023-12-26 10:28:27.279200 epoch: 48, time cost: 27.428106546401978, avg loss: 0.0841466635465622
2023-12-26 10:28:54.232100 epoch: 49, time cost: 26.69651198387146, avg loss: 0.09023458510637283
2023-12-26 10:29:23.387700 epoch: 50, time cost: 28.84040927886963, avg loss: 0.08270444720983505
2023-12-26 10:29:23.632800 validating the model...
2023-12-26 10:29:31.006600 {'loss': 0.18489053298960414, 'acc': 0.9538461538461539}
2023-12-26 10:29:31.339100 Chosen checkpoint is mobilenetv2-100.ckpt
2023-12-26 10:29:31.339200 training is ok!!!
2023-12-26 10:29:32.082400 SYSTEM: Finishing...
```

2.2.7 改正图像预测函数

我们注意到原本给定的 predict 函数错了：cv2.imread()读的颜色空间是 BGR，而训练用的是 RGB，因此将代码改成：

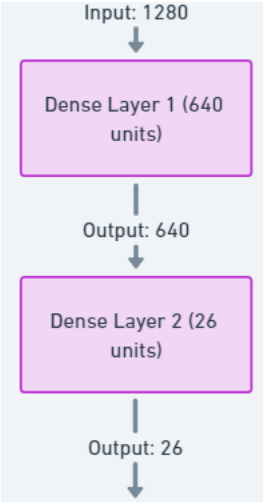
```
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
```

将预测函数改正以后，模型性能显著提升，得分突破 95 分：



2.2.8 模型结构修改和调整

考虑修改 head 层数：原本的 head 包含一个全局平均池化层和一个丢弃层，但只有一个全连接层。这会导致维数下降不够平滑。**因此我们修改 head 结构，增加一个全连接层，将输入通道数 input_channel 转换到其一半的大小。**



考虑到模型结构变得更加复杂，因此我们需要继续增加训练轮数 epoch 的值，为了防止可能的过拟合出现，我们在代码中尝试加入 dropout 丢弃层，丢弃率为 0.2。

1、不修改 head 层数，不添加 dropout 丢弃层：

测试点	状态	时长	结果
在 130 张照片上测试模型	✓	15s	得分: 95.38

2、不修改 head 层数，添加 dropout 丢弃层：

测试点	状态	时长	结果
在 130 张照片上测试模型	✓	11s	得分: 94.62

3、head 中增加一个全连接层，并添加 dropout 丢弃层：

代码如下：

```
if has_dropout:
    head = [
        GlobalAvgPooling(),
        nn.Dense(input_channel, int(0.5*input_channel),
has_bias=False),
        nn.Dropout(0.2),
        nn.Dense(int(0.5*input_channel), num_classes, has_bias=False),
    ]
```

测试点	状态	时长	结果
在 130 张照片上测试模型	✓	15s	得分: 96.15

4、head 中增加一个全连接层，但不添加 dropout 丢弃层：

测试点	状态	时长	结果
在 130 张照片上测试模型	✓	11s	得分: 97.69

因此我们最终模型结构采用在 head 中添加一个全连接层，不添加 dropout 丢弃层的网络结构，**最终模型测试的得分为 97.69 分。**

模型的结构如下：

```
class MobileNetV2Head(nn.Cell):

    def __init__(self, input_channel=1280, hw=7, num_classes=1000,
reduction='mean',has_dropout=True, activation="None"):
        super(MobileNetV2Head, self).__init__()
        # head = ([GlobalAvgPooling(), nn.Dense(input_channel,
num_classes, has_bias=True)] if not has_dropout else
        #         [GlobalAvgPooling(), nn.Dropout(0.2),
nn.Dense(input_channel, num_classes, has_bias=True)])
        if has_dropout:
            head = [
                GlobalAvgPooling(),
                nn.Dense(input_channel, int(0.5*input_channel),
has_bias=False),
                #nn.Dropout(0.2),
                nn.Dense(int(0.5*input_channel), num_classes,
has_bias=False),
            ]
        else:
            head = [
                GlobalAvgPooling(),
                nn.Dense(input_channel, int(0.5*input_channel),
has_bias=False),
                nn.Dense(int(0.5*input_channel), num_classes,
has_bias=False),
            ]
        self.head = nn.SequentialCell(head)
```

```
        self.need_activation = True

        if activation == "Sigmoid":
            self.activation = nn.Sigmoid()
        elif activation == "Softmax":
            self.activation = nn.Softmax()
        else:
            self.need_activation = False
```

附加题

构建和执行 mindspore Lite 端侧图像分类任务，实现一个 Android 图像分类的 APP。

开发环境依赖

Mindspore Lite 框架介绍与编译

MindSpore Lite 是 MindSpore 全场景 AI 框架的端侧引擎，目前 MindSpore Lite 作为华为 HMS Core 机器学习服务的推理引擎底座。MindSpore Lite 1.0.0 已经开源，开源之后，其接口易用性、算子性能与完备度、第三方模型的广泛支持等方面，得到了众多手机应用开发者的广泛认可。

由于发行版存在内部的依赖版本冲突，我们选择下载源码进行编译安装。编译 x86_64 架构 Release 版本，同时设定线程数。

```
git clone https://gitee.com/mindspore/mindspore.git -b r1.0
bash build.sh -I x86_64 -j32
```

编译完成后，进入 `mindspore/output/`目录，可查看编译后生成的文件。文件分为三部分：

- `mindspore-lite-{version}-converter-{os}.tar.gz`：包含模型转换工具 `converter`。
- `mindspore-lite-{version}-runtime-{os}-{device}.tar.gz`：包含模型推理框架 `runtime`、基准测试工具 `benchmark` 和性能分析工具 `timeprofiler`。
- `mindspore-lite-{version}-minddata-{os}-{device}.tar.gz`：包含图像处理库 `imageprocess`。

执行解压缩命令，获取编译后的输出件。

```
tar -xvf mindspore-lite-{version}-converter-{os}.tar.gz
tar -xvf mindspore-lite-{version}-runtime-{os}-{device}.tar.gz
tar -xvf mindspore-lite-{version}-minddata-{os}-{device}.tar.gz
```

至此我们已经获得了需要的 Minspore Lite 框架内容。

APP 开发环境依赖

- Android Studio >= 3.2 (推荐4.0以上版本)
- NDK 21.3
- CMake 3.10.2
- Android SDK >= 26
- JDK >= 1.8

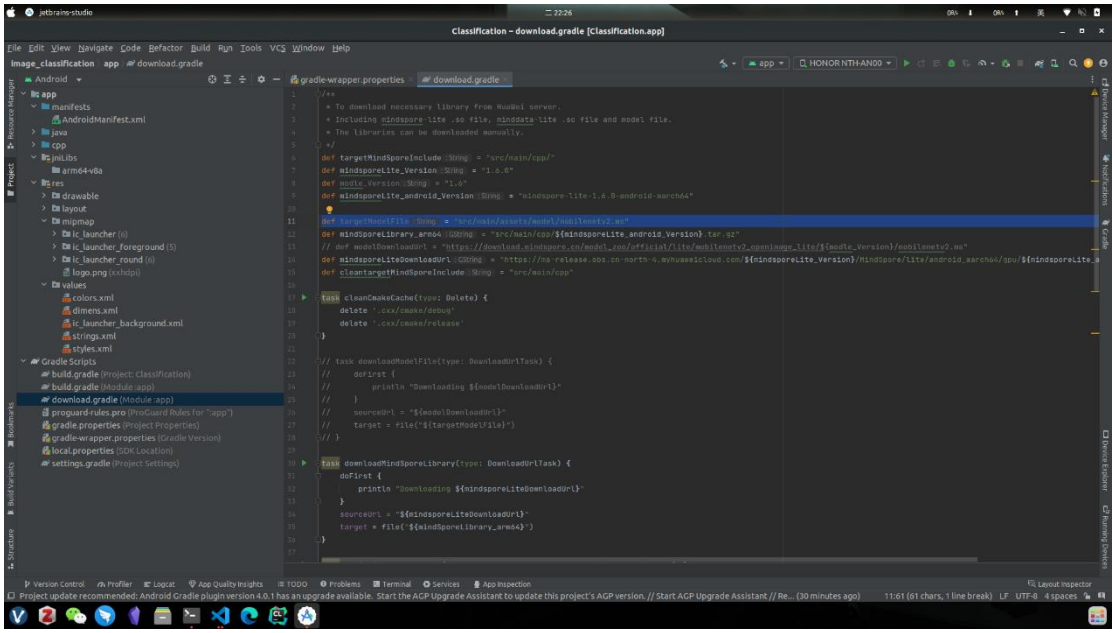
工程文件介绍

文件组织

APP 开发过程中的工程文件组织结构如下：

```
app
├── src/main
|   ├── assets # 资源文件
|   |   └── mobilenetv2.ms # 存放模型文件
|   |
|   ├── cpp # 模型加载和预测主要逻辑封装类
|   |   ├── ..
|   |   ├── mindspore-lite-1.0.0-minddata-arm64-cpu # MindSpore Lite版本
|   |   ├── MindSporeNetnative.cpp # MindSpore调用相关的JNI方法
|   |   └── MindSporeNetnative.h # 头文件
|   |   └── MsNetWork.cpp # MindSpore接口封装
|   |
|   ├── java # java层应用代码
|   |   └── com.mindspore.himindsporedemo
|   |       ├── gallery.classify # 图像处理及MindSpore JNI调用相关实现
|   |       |   └── ...
|   |       └── widget # 开启摄像头及绘制相关实现
|   |           └── ...
|   |
|   ├── res # 存放Android相关的资源文件
|   └── AndroidManifest.xml # Android配置文件
|
├── CMakeList.txt # cmake编译入口文件
|
├── build.gradle # 其他Android配置文件
├── download.gradle # 工程依赖文件下载
└── ...
```

官方示例代码中的模型文件通过 `app/download.gradle` 脚本在 APP 构建时自动下载，并放置在 `app/src/main/assets` 工程目录下，我们在此处进行更换，将其改为我们自己的模型。



之后我们使用 Android Studio 连接手机进行调试。Mindspore 的 demo 提供了 UI 等 APP 内容，我们只需要测试我们自己的模型是否可用。

端侧推理代码编写

在 JNI 层调用 MindSpore Lite C++ API 实现端测推理。推理代码流程如下：

1. 加载 MindSpore Lite 模型文件，构建上下文、会话以及用于推理的计算图。

- 加载模型文件：创建并配置用于模型推理的上下文

```
1. // Buffer is the model data passed in by the Java layer
2. jlong bufferLen = env->GetDirectBufferCapacity(buffer);
3. char *modelBuffer = CreateLocalModelBuffer(env, buffer);
```

- 创建会话

```
1. void **labelEnv = new void *;
2. MSNetWork *labelNet = new MSNetWork;
3. *labelEnv = labelNet;
4.
5. // Create context.
6. lite::Context *context = new lite::Context;
7. context->thread_num_ = numThread;  //Specify the number of threads to run inference
8.
9. // Create the mindspore session.
10.labelNet->CreateSessionMS(modelBuffer, bufferLen, context);
11.delete(context);
```

- 加载模型文件并构建用于推理的计算图

```
1. void MSNetWork::CreateSessionMS(char* modelBuffer, size_t bufferL
   en, std::string name, mindspore::lite::Context* ctx)
2. {
3.     CreateSession(modelBuffer, bufferLen, ctx);
4.     session = mindspore::session::LiteSession::CreateSession(ctx)
   ;
5.     auto model = mindspore::lite::Model::Import(modelBuffer, buff
   erLen);
6.     int ret = session->CompileGraph(model);
7. }
```

2. 将输入图片转换为传入 MindSpore 模型的 Tensor 格式。

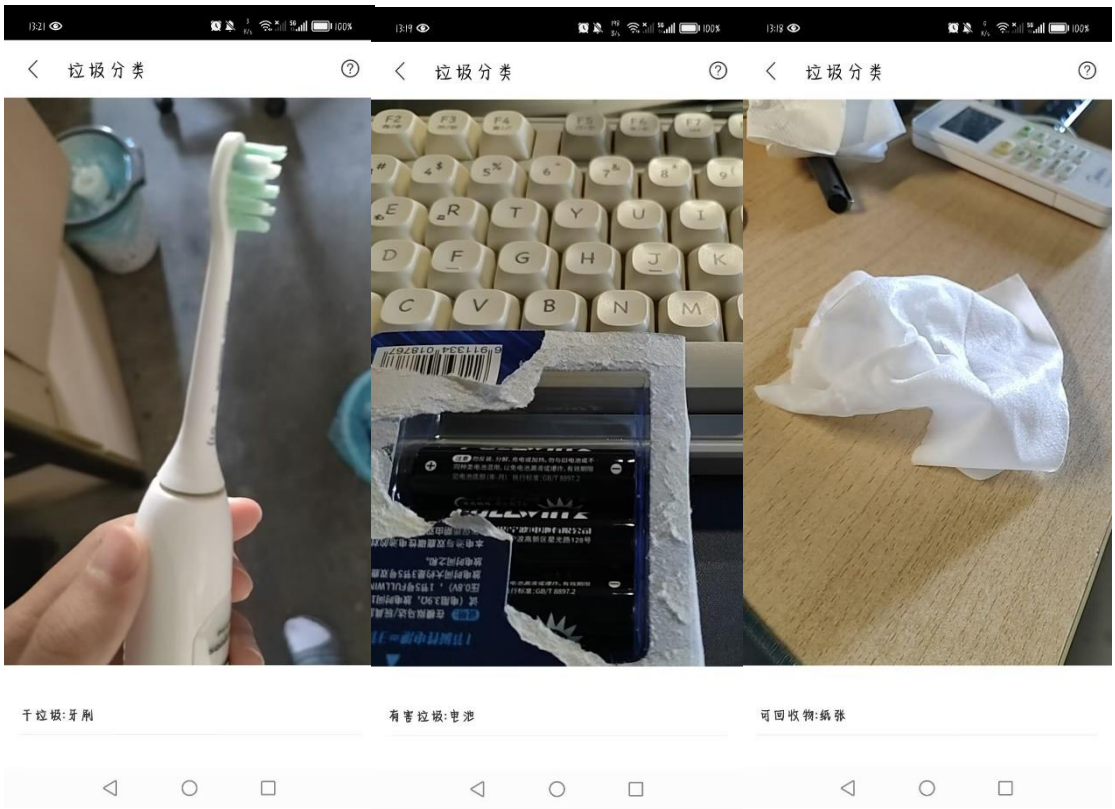
```
1. if (!BitmapToLiteMat(env, srcBitmap, &lite_mat_bgr)) {
2.  MS_PRINT("BitmapToLiteMat error");
3.  return NULL;
4. }
5. if (!PreProcessImageData(lite_mat_bgr, &lite_norm_mat_cut)) {
6.  MS_PRINT("PreProcessImageData error");
7.  return NULL;
8. }
9.
10.ImgDims inputDims;
11.inputDims.channel = lite_norm_mat_cut.channel_;
12.inputDims.width = lite_norm_mat_cut.width_;
13.inputDims.height = lite_norm_mat_cut.height_;
14.
15.// Get the mindsore inference environment which created in LoadMo
   del().
16.void **labelEnv = reinterpret_cast<void **>(netEnv);
17.if (labelEnv == nullptr) {
18. MS_PRINT("MindSpore error, labelEnv is a nullptr.");
19. return NULL;
20.}
21.MSNetWork *labelNet = static_cast<MSNetWork *>(*labelEnv);
22.
23.auto mSession = labelNet->session();
24.if (mSession == nullptr) {
25. MS_PRINT("MindSpore error, Session is a nullptr.");
26. return NULL;
27.}
28.MS_PRINT("MindSpore get session.");
29.
```

```
auto msInputs = mSession->GetInputs();
if (msInputs.size() == 0) {
 MS_PRINT("MindSpore error, msInputs.size() equals 0.");
 return NULL;
}
auto inTensor = msInputs.front();

float *dataHWC = reinterpret_cast<float *>(lite_norm_mat_cut.data_ptr_);
// Copy dataHWC to the model input tensor.
memcpy(inTensor->MutableData(), dataHWC,
     inputDims.channel * inputDims.width * inputDims.height * sizeof(float));
```

3. 对输入 Tensor 按照模型进行推理，获取输出 Tensor，并进行后处理。

```
// After the model and image tensor data is loaded, run inference
auto status = mSession->RunGraph();
```

测试结果



我们的测试结果中可以看到我们的模型正确的输出了分类结果。平均的推断用时大约在150ms 左右，与 MobileNetV2 论文中的数据接近。

Network	Top 1	Params	MAdds	CPU
MobileNetV1	70.6	4.2M	575M	113ms
ShuffleNet (1.5)	71.5	**3.4M**	292M	-
ShuffleNet (x2)	73.7	5.4M	524M	-
NasNet-A	74.0	5.3M	564M	183ms
MobileNetV2	**72.0**	**3.4M**	**300M**	**75ms**
MobileNetV2 (1.4)	**74.7**	6.9M	585M	**143ms**