Garbage Classification on the Atlas 200 DK Using MobileNetV2

Shayan Zargari and Soroush Razavi
Department of Electrical and Computer Engineering,
University of Alberta
(emails: {zargari, s.razavi}@ualberta.ca)

***Abstract*—This paper presents the use of the MobileNetV2 architecture, MindSpore framework, Huawei Ascend910 AI processor, and Atlas 200DK hardware platform to train and deploy a garbage classification application. MobileNetV2 is a convolutional neural network architecture that uses depthwise separable convolutions and a linear bottleneck layer to achieve high accuracy and computational efficiency. MindSpore is an open-source deep-learning framework that supports automatic differentiation and distributed training, while the Huawei Ascend910 AI processor is a specialized AI chip designed for high-performance deep learning. The Atlas 200DK is a versatile hardware platform that supports multiple operating systems and AI development frameworks, making it ideal for AI applications. This project provides guidance on how to develop a garbage classification application using Python and deploy it on the Atlas 200DK.**

I. Introduction

MobileNetV2 is a popular convolutional neural network architecture designed for mobile and embedded vision applications [1]. It was introduced by Google in 2018 as an upgrade to the original MobileNet architecture, to improve its accuracy and computational efficiency. The MobileNetV2 architecture consists of depthwise separable convolutions, which involve a factorization of the standard convolution operation into a depthwise convolution followed by a pointwise convolution. This factorization leads to a significant reduction in the number of parameters and computational cost of the network, without sacrificing accuracy. In addition, MobileNetV2 includes a linear bottleneck layer that further reduces the number of parameters while maintaining the expressive power of the network. The architecture of the MobileNetv2 network is shown in Fig. 1 [2], [3]. MobileNetV2 has demonstrated state-of-the-art performance on a variety of image classification tasks, including the ImageNet dataset while achieving real-time inference on mobile devices. The architecture has also been used in various other computer vision tasks such as object detection, semantic segmentation, and facial recognition.

MindSpore is an open-source deep-learning framework developed by Huawei [4]. It is designed to provide efficient and flexible training and inference capabilities for deep learning models on a variety of hardware platforms, including CPUs, GPUs, and specialized AI chips like the Huawei Ascend910 AI processor. One of the key features of MindSpore is its automatic differentiation capability, which allows users to easily compute gradients and optimize their models during training. MindSpore also supports distributed training across multiple devices, allowing for faster and more efficient model training. The Huawei Ascend910 AI processor is a specialized AI chip designed for high-performance deep learning training and inference. It features a highly parallelized architecture with thousands of processing cores and large amounts of on-chip memory, allowing it to handle large-scale deep-learning models and datasets. The Ascend910 AI processor also includes advanced hardware features such as high-speed memory interfaces, hardware support for low-precision numerical formats, and specialized instruction sets for deep learning operations. In this project, we will use the MindSpore framework to train a MobileNetV2 model on a garbage classification dataset. MindSpore is a deep learning framework developed by Huawei that supports efficient training and inference on a variety of hardware platforms, including the Huawei Ascend910 AI processor. The Ascend910 AI processor is a high-performance AI chip that provides outstanding computing power and memory capacity, making it an excellent choice for training and deploying deep learning models.

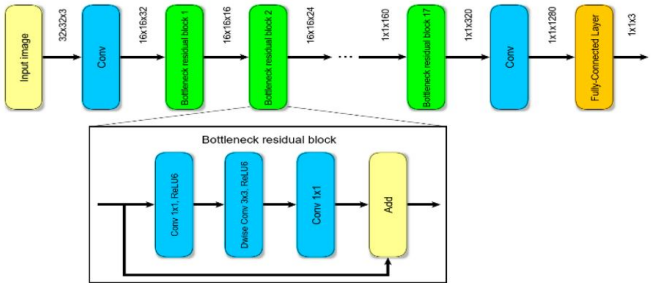

Fig. 1: The architecture of the MobileNetv2 network [3].

The Atlas 200DK is a versatile and powerful hardware platform developed by Huawei to accelerate the development and deployment of AI applications [5]. Equipped with various hardware components and interfaces, it supports multiple operating systems and AI development frameworks, making it ideal for a range of applications including computer vision, speech recognition, natural language processing, and robotics. With its high performance and flexibility, the Atlas 200DK provides developers with a powerful computing platform to build, test, and deploy AI models efficiently. In this project, we will learn how to implement a garbage classification application using the MobileNetV2 model, which has been trained on a garbage classification dataset using the MindSpore framework

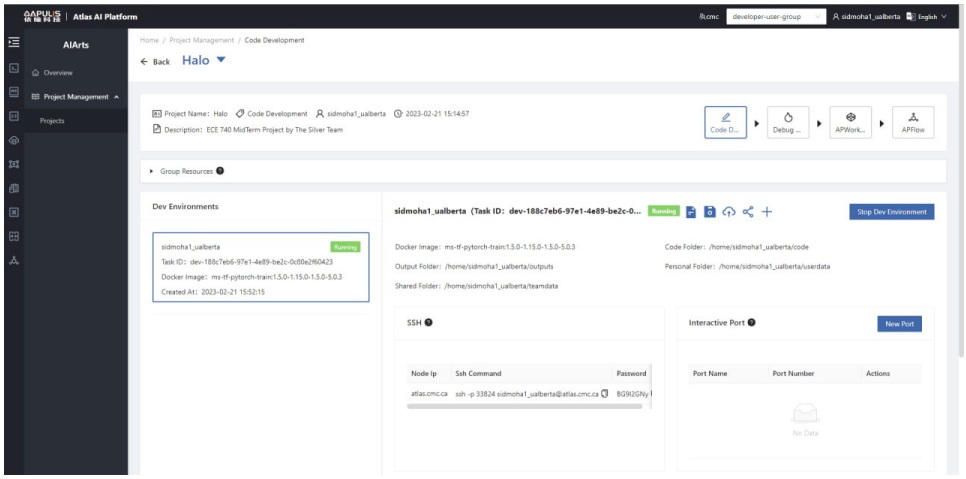

Fig. 2: The project environment on Atlas AI platform.

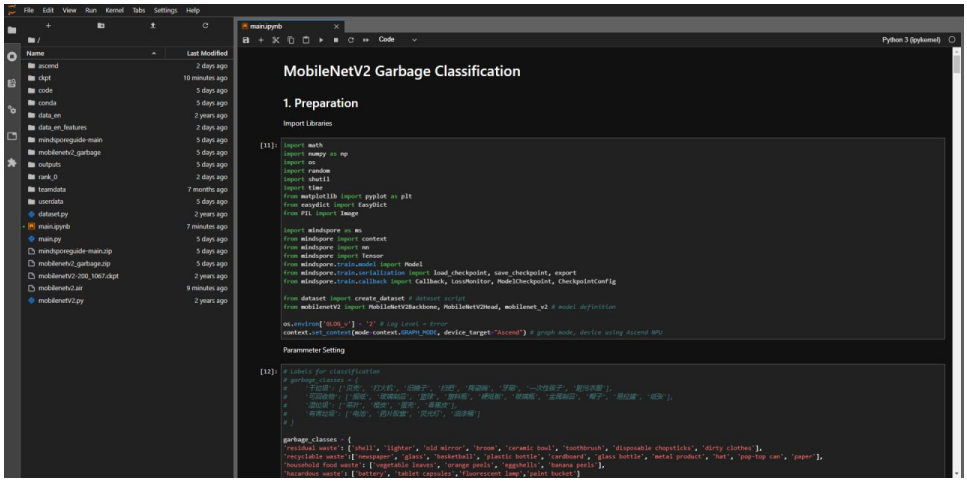

Fig. 3: The decompressed "mobilenetv2_garbage" file and the Jupyter environment.

and Huawei Ascend910 AI processor. This application will use local image data as input and detect garbage objects in images, saving the results to a folder. Through this project, we will gain a better understanding of the application of the Atlas 200DK in the field of AI, as well as learn how to write code for the garbage classification application using Python, deploy applications on the Atlas 200DK, understand the basics of Linux, and master the basic operations of model conversion using the ATC command.

II. Task1 – Train a Garbage Classifier Using the MobileNetV2 Trained Weights

A. Preparation: Project structure designing

1) ***Logging into Atlas AI Platform****:* The Apulis AI Studio manages the Atlas 800 cluster, which is a platform for building, training, and deploying models for developers and data scientists. With its data preprocessing, semi-automated data labeling, distributed training, and automated model-building features, it accelerates AI development and innovation. To create a new project on this platform, follow the steps outlined below. Fig. 2 illustrates the project environment on the Atlas AI platform.

1) Start by navigating to the AIArts option from the home page.
2) Select Projects from the Project Management section to view all current user group tasks or jobs.
3) Choose New Project and select Code Development from the pop-up window to set up a new code-development job.
4) Provide a name and description for the new project to help identify its purpose and goals.
5) Click Submit to create the new project, using a name like "Halo" for example.
6) Once the project is created, begin setting up the code development environment, selecting the appropriate image, dataset, and NPU for the project.

2) ***Downloading the Dataset****:* To get started, we create a directory as our working directory and download the compressed package of the experiment code and dataset.

3) ***Decompress the package****:* To unzip the package, we need to open a terminal and run the commands given below.

```
unzip mobilenetv2_garbage.zip
```

Fig. 3 shows the decompressed "mobilenetv2_garbage" files and the Jupyter environment.

III. Data and Codes explanation

*1) **Dataset**:* The MobileNetV2 code uses the ImageFolder format to manage datasets by default. The structure of the dataset is shown in Fig. 4.

```
└─ImageFolder
    ├─train
    │   class1Folder
    │   class2Folder
    │   ......
    └─eval
        class1Folder
        class2Folder
        ......
```

Fig. 4: The structure of the dataset.

*2) **Imports**:* The Python code necessary for this subsection can be found in Appendix A. It imports the required libraries and modules for training a MobileNetV2 model using the MindSpore framework.

*3) **Configuring parameters**:* To configure the necessary parameters, refer to the Python code provided in Appendix B. This code defines a dictionary named `garbage_classes`, which contains lists of items that belong to various categories of garbage. Within the program, you will find the `class_en` list which includes the English names of all items listed in the `garbage_classes` dictionary. Additionally, the `index_en` dictionary maps the English names of each item to their corresponding indices, which are used as classification labels.

The `config` dictionary contains all the necessary hyperparameters for training a MobileNetV2 model using the MindSpore framework. These hyperparameters include the number of classes, image height and width, batch size, number of epochs, learning rate, momentum, weight decay, and paths to the dataset and checkpoint files.

*4) **Data visualization**:* Next, `create_dataset` code block uses the MindSpore framework to create a dataset object for data visualization. The Python code is provided in Appendix C. It takes two arguments:

1) `dataset_path`: The path to the dataset.
2) `config`: Configuration object containing hyperparameters for the dataset.

After creating the dataset object, the code prints its size using the `"get_dataset_size()"` method. The `"create_dict_iterator"` method is then used to create an iterator for the dataset, with the `"output_numpy"` argument set to `True` so that the iterator returns NumPy arrays. The code retrieves a batch of images and corresponding labels from the dataset using the iterator and stores them in "images" and "labels" variables. Finally, the first four images in the batch are displayed using a for loop and the `"matplotlib"` library. The `"subplot"` function arranges the images into a 2×2 grid, and the `"imshow"` function displays the images. The `"title"` function displays the class label for each image, and the `"xticks"` function removes the x-axis tick labels.

After completing the process, a portion of the processed data is displayed in Figure 5.

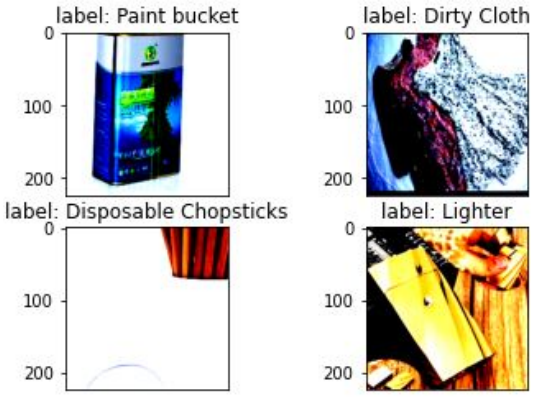

Fig. 5: Visualize some processed data.

*5) **Training strategy: Designing a learning rate**:* Typically, a fixed learning rate, such as 0.01, is utilized for training a model. As the number of training steps increases, the model tends to converge, and the weight parameter's update amplitude gradually decreases to reduce oscillation. However, a dynamically decreasing learning rate can also be applied during model training. There are common policies for reducing the learning rate, including:

1) polynomial decay/square decay
2) cosine decay
3) exponential decay
4) stage decay

Here, the cosine decay approach is employed, with the Python code provided in Appendix D. The code consists of a function that implements cosine decay to generate an array of learning rates. The function accepts several arguments, including:

1) `total_steps`: Total number of steps in training.
2) `lr_init`: Initial learning rate.
3) `lr_end`: Final learning rate.
4) `lr_max`: Maximum learning rate.
5) `warmup_steps`: Total number of steps in warmup epochs.

The function first converts the `lr_init`, `lr_end`, and `lr_max` arguments to floats, then calculates the `decay_steps` as the difference between the `total_steps` and `warmup_steps`. It initializes an empty list to store the learning rate values for all steps in the training and calculates the `inc_per_step` as the rate at which the learning rate should be increased per step during warmup epochs. Then, it iterates over all the steps in the training and calculates the learning rate for each step using the following formula:

1) step $<$ `warmup_steps`: The learning rate is calculated by adding `inc_per_step` multiplied by the step number to `lr_init`.

2) step $\geq$ `warmup_steps`: The learning rate is calculated using cosine decay as follows:

$$\texttt{cosine\_decay} = 0.5 * (1 + math.cos(math.pi * (i - warmup\_steps)/decay\_steps)), \quad (1)$$

where the learning rate is calculated as $(lr\_max - lr\_end) * cosine\_decay + lr\_end$. Finally, the function returns the list of learning rate values for all steps in the training.

*6) **Switch precision function**:* The associate Python code is attached to Appendix E. This function takes a MindSpore neural network model net and a data type `data_type` (e.g. ms.float16, ms.float32) as input. It then converts the model to the specified data type using the `to_float()` method. If the device target is Ascend, it also ensures that the weights of any `nn.Dense` layers in the network are converted to the `ms.float32` data type.

*7) **Training options**:*

i) **Partial training (option 1)**: It involves two steps, namely feature extraction and training the head.

ii) **Full training (option 2)**: It involves training the whole network while using early stopping to prevent over-fitting.

In the following, we will begin by explaining each option and subsequently compare the model losses for the options.

i) **Partial training (option 1)**: With this option, we load the pre-trained MobileNetV2 model that was trained on the ImageNet dataset for fine-tuning. During training, only the last modified FC layer is trained, and the checkpoint is saved.

i-1) **Feature extraction**: The python code is attached to Appendix F. This code defines a function called `extract_features`, which takes three arguments:

a) `net`: Neural network model to extract features from.
b) `dataset_path`: Dataset path.
c) `config`: Configuration object.

The purpose of the function is to extract features from the input dataset using the provided neural network model, `net`, and save them to disk. The `dataset_path` is used to load the dataset, and `config` is a configuration object that specifies the hyperparameters and other settings. The function first creates a folder to save the extracted features, using the `os.makedirs()` function. It then loads the dataset using the `create_dataset()` function, passing in the `dataset_path` and `config` objects. It then iterates over the dataset, using the `create_dict_iterator()` function to generate a dictionary of inputs and labels for each batch. For each batch, it generates a filename for the features and label file using the batch index and checks if the files already exist. If the files do not exist, it extracts the features from the input images using the `net` model and saves the features to a file using the `numpy.save()` function, and saves the corresponding labels to a separate file. After defining the `extract_features` function, the code creates an instance of the `MobileNetV2Backbone` neural network model and loads a checkpoint using the `load_checkpoint()` function, passing in the path to the pretrained checkpoint and the `backbone` object. Fig. 6 shows the extracted features for partial training.

```
Complete the batch 1/40
Complete the batch 2/40
Complete the batch 3/40
Complete the batch 4/40
Complete the batch 5/40
Complete the batch 6/40
Complete the batch 7/40
Complete the batch 8/40
Complete the batch 9/40
Complete the batch 10/40
Complete the batch 11/40
Complete the batch 12/40
Complete the batch 13/40
Complete the batch 14/40
Complete the batch 15/40
Complete the batch 16/40
Complete the batch 17/40
Complete the batch 18/40
Complete the batch 19/40
Complete the batch 20/40
Complete the batch 21/40
Complete the batch 22/40
Complete the batch 23/40
Complete the batch 24/40
Complete the batch 25/40
Complete the batch 26/40
Complete the batch 27/40
Complete the batch 28/40
Complete the batch 29/40
Complete the batch 30/40
Complete the batch 31/40
Complete the batch 32/40
Complete the batch 33/40
Complete the batch 34/40
Complete the batch 35/40
Complete the batch 36/40
Complete the batch 37/40
Complete the batch 38/40
Complete the batch 39/40
Complete the batch 40/40
```

Fig. 6: Extract features for partial training.

i-2) **Training the head**: It trains head on top of a pre-trained MobileNetV2 backbone for classification. The Python code is attached to Appendix G. It takes no arguments but instead uses global variables defined in a config object. First, the function loads the training and evaluation datasets using the `create_dataset()` function, passing in the `dataset_path` and `config` objects. It also sets the `step_size` to be the size of the training dataset. Next, the function initializes a MobileNetV2 backbone and loads a pretrained checkpoint using the `load_checkpoint()` function. It then initializes a new MobileNetV2 head using the output channel size of the backbone and the number of classes specified in the `config` object. The two are combined to create a new network using the `mobilenet_v2()` function. A softmax cross entropy loss function, a cosine decay learning rate schedule and a stochastic gradient descent optimizer is defined. It then defines a `WithLossCell` object that combines the head and loss function and trains the network using the `TrainOneStepCell` object. The function also sets the network in training mode. The training loop then begins, where for each epoch, the function shuffles the index list of the dataset and loops through each batch, loading the pre-extracted features and labels from the `features_path` and computing the loss using `train_step()`. The function appends the epoch loss to the `history` list, prints the epoch time and loss, and saves the network checkpoint at specified intervals. Finally, the function evaluates the

```
epoch: 1, time cost: 17.82689666748047, avg loss: 1.1597609519958496
epoch: 2, time cost: 1.409881353378296, avg loss: 0.3269261121749878
epoch: 3, time cost: 1.0584471225738525, avg loss: 0.18703912198543549
epoch: 4, time cost: 0.8494408130645752, avg loss: 0.10365275293588638
epoch: 5, time cost: 1.0643246173858643, avg loss: 0.08110307157039642
epoch: 6, time cost: 1.232433557510376, avg loss: 0.06297467648983002
epoch: 7, time cost: 1.1565189361572266, avg loss: 0.055956847965717316
epoch: 8, time cost: 1.056850910186767676, avg loss: 0.0522956252098083 5
epoch: 9, time cost: 1.256248950958252, avg loss: 0.050349265336990356
epoch: 10, time cost: 1.0997943878173828, avg loss: 0.049569401890039444
```

Fig. 7: Time cost and average loss for partial training - option 1.

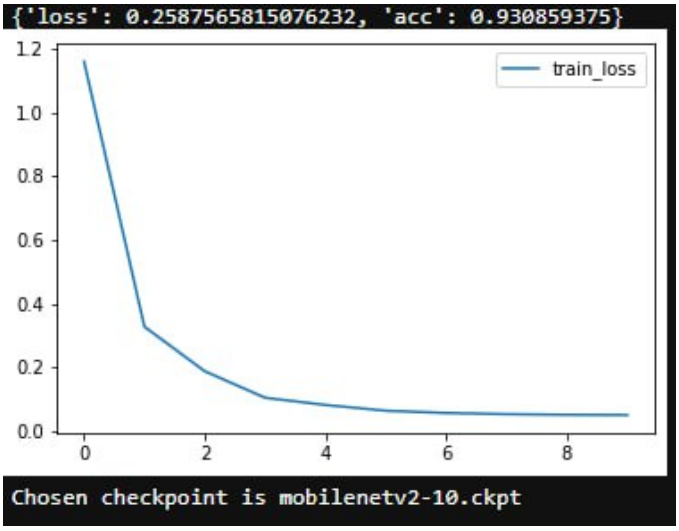

Fig. 8: Train loss and model evaluation including accuracy and loss over the test set - option 1.

trained network on the evaluation dataset using the `Model` object and prints the evaluation accuracy. Next, the `train_head()` function is called, which trains a new head on top of a MobileNetV2 backbone for image classification and returns a list of epoch losses. A block of code is then executed to plot the epoch losses over time using `matplotlib`. Finally, the code sets the name of the selected checkpoint file `CKPT` to be "mobilenetv2" concatenated with the number of epochs specified in the `config` object and the file extension "`.ckpt`".

Fig. 7 shows the time cost and average loss for partial training. Finally, Fig. 8 plots the training loss and also illustrates the model evaluation including accuracy and loss over the test set.

ii) **Full training (option 2)**: In this option, the frozen layers will still be involved in the forward pass, but will not learn as the learning rate is set to 0, and the weight will not be updated. It should be noted that training a complex network often requires hundreds of epochs to achieve the desired accuracy, and it can be difficult to predict when that accuracy will be achieved. Therefore, it is common practice to save checkpoints periodically during training and evaluate their accuracy to choose the best one at the end of training.

ii-1) **Early stopping and validation with EvalCallback**: Generalization capability is crucial for a deep learning network. However, during training, the network may tend to overfit. We can detect overfitting when the accuracy of the model increases on the training dataset but decreases on the validation dataset. In such cases, we can implement "early stopping" by stopping the training process if the validation accuracy does not increase for a certain number of epochs, usually 5 epochs. The Python code for this is provided in Appendix H. The `EvalCallback` class is used to monitor the performance of the model on the validation dataset during training and to perform early stopping to prevent overfitting. This class evaluates the accuracy and loss of the model on the validation dataset during training and stops the training if the validation accuracy does not improve for a specified number of epochs. The constructor of the EvalCallback class takes several arguments:

a) `model`: The model to be evaluated.
b) `eval_dataset`: The validation dataset to be used for evaluation.
c) `history`: A dictionary that stores the training history, including epoch number, training loss, validation loss, and validation accuracy.
d) `eval_epochs`: The frequency at which to evaluate training.

The class has several methods that are called by the MindSpore framework during the training process:

a) `epoch_begin()`: It is called at the beginning of each epoch and initializes a list to store the loss values for each batch.
b) `step_end()`: It is called at the end of each batch and appends the loss value of the batch to the list of losses.
c) `epoch_end()`: It is called at the end of each epoch and calculates the average training loss for the epoch. It then evaluates the model on the validation dataset and stores the epoch number, training loss, validation loss, and validation accuracy in the history dictionary. If the validation accuracy does not improve for a certain number of epochs, as specified by the `count_max` variable, the training is stopped early by calling the `request_stop()` function.

ii-2) **Training on original dataset**: The training function is defined in Appendix I that create an instance of a MobileNetV2 model on a specified dataset. The function first creates training and evaluation datasets using the `create_dataset` function with the given configuration `config`. The dataset size is obtained using the `get_dataset_size` method of the training dataset. It then creates a MobileNetV2 network by defining a backbone and a head and combining them using the `mobilenet_v2` function. The parameters of the backbone are frozen, and a pre-trained checkpoint is loaded. The loss function is defined as `nn.SoftmaxCrossEntropyWithLogits` with `sparse` set to `True` and `reduction` set to `mean`. More specifically, `nn.SoftmaxCrossEntropyWithLogits` is a popular loss function used for multi-class classification tasks. It combines the softmax activation function and the cross-entropy loss function into a single operation. The `sparse` argument is used when the labels are not one-hot encoded. In this case, the label is a single integer

that corresponds to the correct class, rather than a vector of binary values. The `True` value for `sparse` tells the function to use sparse categorical cross-entropy instead of regular categorical cross-entropy. The `reduction` argument specifies how the loss should be reduced over the batch. The `mean` option calculates the mean loss across all samples in the batch. This is often preferred over `sum` because it makes the loss value independent of the batch size, which allows for easier comparison between different runs and datasets.

Then, A loss scale manager is defined using a fixed loss scale value of `1024`. The optimizer method is momentum with the specified learning rate and weight decay, and a cosine learning rate decay schedule. In particular, the momentum optimizer is a stochastic gradient descent (SGD) optimizer that uses a moving average of the gradient to update the weights of the model.

Finally, a `Model` object is created using the network, loss function, optimizer, and loss scale manager. A dictionary `history` is initialized to store the training history. A model checkpoint callback is also defined and added to the list of callbacks.

Fig. 9 shows train loss, evaluation loss, and evaluation accuracy for training on the original dataset. Fig. 10 shows the train loss and validation loss versus epochs, while Fig. 11 illustrates the validation accuracy versus epochs and the chosen checkpoint, `mobilenetv2-10_40.ckpt`.

8) ***Training option selection for model deployment on Atlas 200 DK****:* Our experiment with the defined parameter revealed that partial training (option 1) achieves an accuracy of 93.08% after 10 epochs, whereas full training (option 2) can attain an accuracy of 92.69% after 15 epochs. Thus, we have decided to use option 1 to deploy the model on Atlas 200 DK due to its superior accuracy.

9) ***Perform model inference****:* To perform inference, the best checkpoint is loaded using the `load_checkpoint` API. It's important to note that when loading the model using this method, the data should be transferred to the original network for inference purposes, rather than the training network with an optimizer and a loss function.

```
epoch: 1, train_loss: 1.472878, eval_loss: 0.603598, eval_acc: 0.820312
epoch: 2, train_loss: 0.653814, eval_loss: 0.496359, eval_acc: 0.855859
epoch: 3, train_loss: 0.510012, eval_loss: 0.475901, eval_acc: 0.861328
epoch: 4, train_loss: 0.466544, eval_loss: 0.402654, eval_acc: 0.880078
epoch: 5, train_loss: 0.415677, eval_loss: 0.334792, eval_acc: 0.910156
epoch: 6, train_loss: 0.389445, eval_loss: 0.307962, eval_acc: 0.918359
epoch: 7, train_loss: 0.371341, eval_loss: 0.301674, eval_acc: 0.924219
epoch: 8, train_loss: 0.330095, eval_loss: 0.294413, eval_acc: 0.919141
epoch: 9, train_loss: 0.362163, eval_loss: 0.288576, eval_acc: 0.923828
epoch: 10, train_loss: 0.376955, eval_loss: 0.277859, eval_acc: 0.926953
epoch: 11, train_loss: 0.326532, eval_loss: 0.288898, eval_acc: 0.923047
epoch: 12, train_loss: 0.340352, eval_loss: 0.309259, eval_acc: 0.915625
epoch: 13, train_loss: 0.354801, eval_loss: 0.288785, eval_acc: 0.919531
epoch: 14, train_loss: 0.365854, eval_loss: 0.290794, eval_acc: 0.919922
epoch: 15, train_loss: 0.338070, eval_loss: 0.308589, eval_acc: 0.919922
```

Fig. 9: Train loss, evaluation loss, and evaluation accuracy for training on the original dataset - option 2.

i) **Preprocessing**: For this aim, `image_process` function is used to preprocess the image by subtracting the mean and dividing by the standard deviation. It also transposes the dimensions of the image array to be in the correct order for the neural network. The Python code for this function is attached to Appendix J. Another function, `infer_one`, is defined that takes a network A function called `infer_one` is created that takes a neural network object and an image path as inputs. This function loads the image from the specified path, resizes it to the dimensions defined in the configuration object, preprocesses the image using the `image_process` function, feeds it through the network, and outputs the predicted class of the image. The results of the inference using this function are depicted in Figure 12.

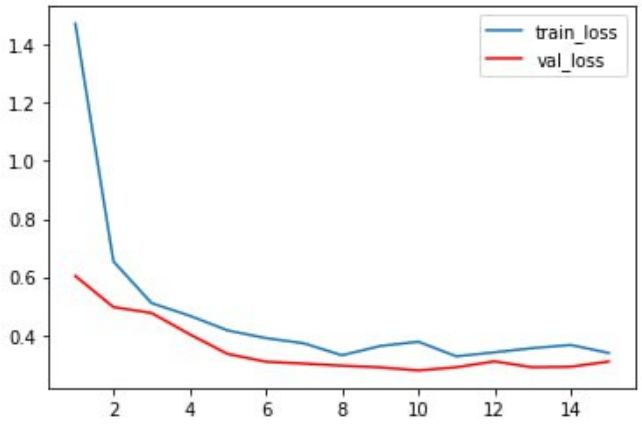

Fig. 10: Train loss and validation loss versus epochs - option 2.

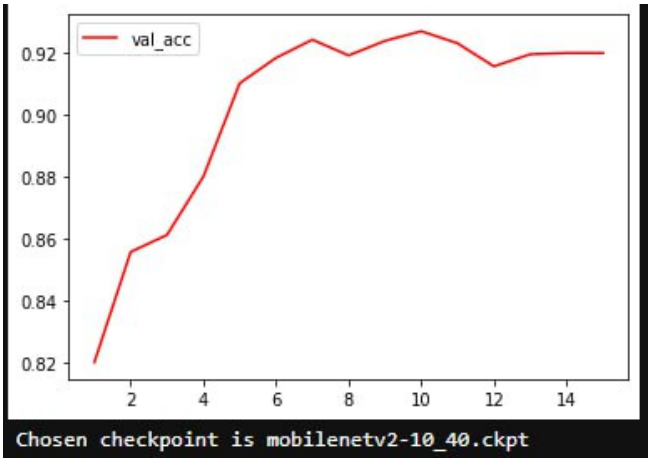

Fig. 11: Validation accuracy versus epochs - option 2.

```
data_en/test/Cardboard/00091.jpg Cardboard
data_en/test/Cardboard/00092.jpg Cardboard
data_en/test/Cardboard/00093.jpg Cardboard
data_en/test/Cardboard/00094.jpg Cardboard
data_en/test/Cardboard/00095.jpg Cardboard
data_en/test/Cardboard/00096.jpg Cardboard
data_en/test/Cardboard/00097.jpg Cardboard
data_en/test/Cardboard/00098.jpg Cardboard
data_en/test/Cardboard/00099.jpg Cardboard
```

Fig. 12: Model inference results.

ii) **Exporting AIR model file:** We export the AIR model file for model conversion and inference on Atlas 200 DK. The Python code is attached to Appendix K. This

code creates an instance of the MobileNetV2 network for image classification. It defines the backbone of the network with a specified number of output channels and a head that includes the number of classes for classification. The `mobilenet_v2` function is used to combine the backbone and head into a complete network. A saved checkpoint is then loaded into the network using the `load_checkpoint` function. Next, a random input of size $1 \times 3 \times 224 \times 224$ is generated, and the network is exported using the export function of the MindSpore framework. The exported network is saved in the AIR format with the file name `"mobilenetv2.air"` as shown in Fig. 13.

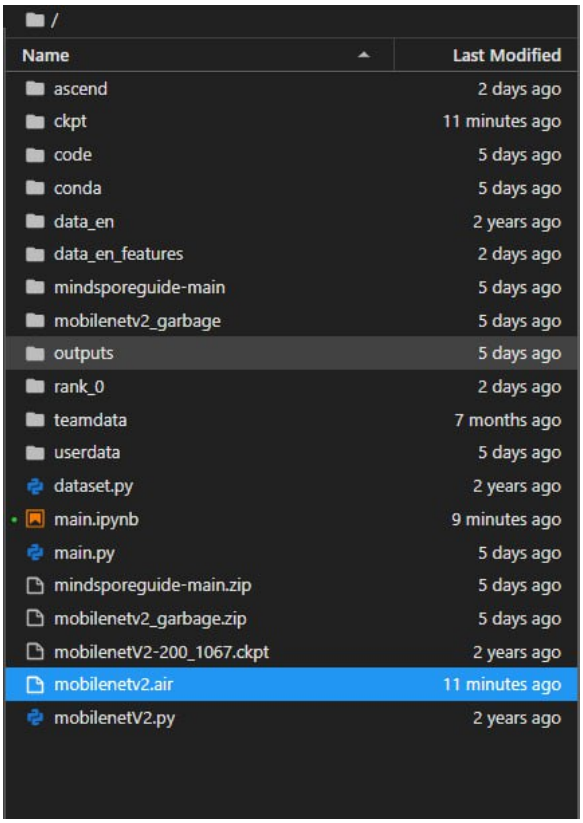

Fig. 13: Exporting AIR model file result.

IV. Introduction to Deep Learning based Computer Vision and Edge Computing

Here, we will demonstrate how to build an image classification app on the Atlas200 DK using the GoogLeNet network to classify objects in images and output the top five classes with the highest confidence scores. We provide detailed step-by-step instructions and explanations. The building blocks of the application pipeline are shown in Fig. 14 below.

The code for this project is available as a GitHub repository. We first log in to the board, then download the repository to the board and finally run the experiments for the image and video branches of the project step-by-step.

i) Download the GitHub repository to the board: We log in to the board using HwHiAiUser and run the following command:

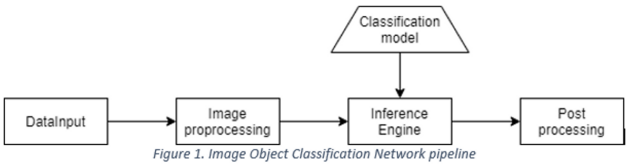

Fig. 14: Image object classification network pipeline.

```
mkdir /home/HwHiAiUser/HIAI_PROJECTS
cd /home/HwHiAiUser/HIAI_PROJECTS
git clone https://github.com/Atlas200dk/
    sample_image_classification_c73_python.
    git
cd sample_image_classification_c73_python
```

ii) Prepare GoogleNet model: We download the `'googlenet.prototxt'` and `'googlenet.caffemodel'` and upload them to the board. Then, we create a model directory using the following commands:

```
mkdir model && cd model
```

Finally, we upload `'googlenet.prototxt'` and `'googlenet.caffemodel'` to the 'model' folder.

iii) Model Conversion: We run the following command in the same directory as the downloaded model files to convert the model from Caffe to Offline Model format:

```
atc --framework=0 --model="googlenet.prototxt
    " --weight="googlenet.caffemodel" --
    input_shape="data:1,3,224,224" --
    input_fp16_nodes="data" --input_format=
    NCHW --output="googlenet" --output_type=
    FP32 --soc_version=Ascend310
```

More specifically, the ascend tensor compiler (ATC) serves as both a model and operator compiler, allowing for offline model conversion, custom operator development, and IR graph construction. Using the ATC, an open-source network model or a single-operator description file (in JSON format) defined by Ascend IR can be converted into an offline model that is suitable for Ascend AI Processors.

iv) Run the application to see the results:

```
cd ..
python3 main.py
```

After running the **Image Classification**, we can see the result prediction on the test image. Fig. 15, 16, and 17 show the images and console output in the VScode file explorer.

V. Task 2 – Deploying Garbage Classification on the Atlas 200 DK Using MobileNetV2

In this section, we will explore the process of deploying garbage classification code on the Atlas 200 DK. This inference experiment aims to use the Atlas 200 DK to classify garbage objects in images by using local image data as input and saving the results to a designated folder. To achieve this, we utilize the AscendCL library to allocate runtime resources and load the model for inference.

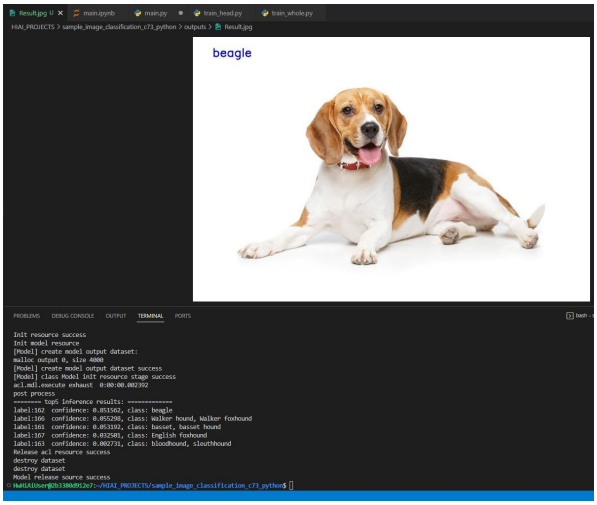

Fig. 15: Image object classification network pipeline for ‘dog1.jpg’.

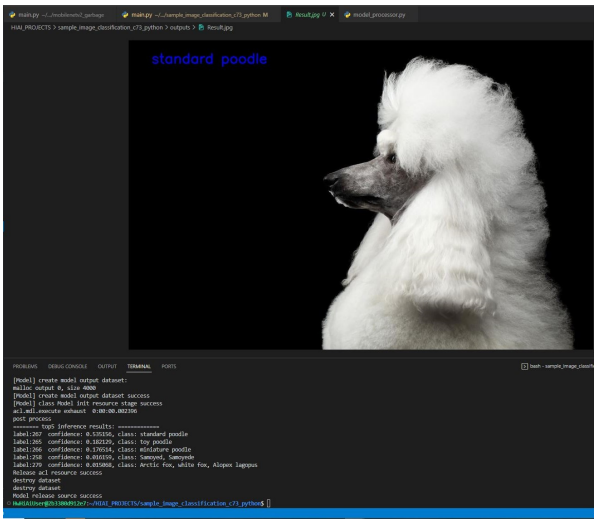

Fig. 16: Image object classification network pipeline ‘dog2.jpg’.

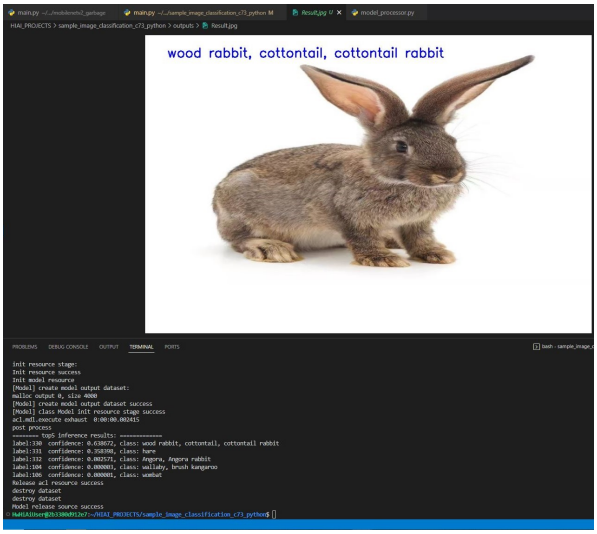

Fig. 17: Image object classification network pipeline ‘rabit.jpg’.

*1) **Converting the MobileNetV2 AIR Model for Inference**:* We upload the `mobilenetv2.air` model file to Atlas 200 DK. Open a terminal on Atlas 200DK, and in the same folder of the model file, run the following conversion command:

```
atc --model=./mobilenetv2.air --framework=1 --output
    =garbage --soc_version=Ascend310 --input_shape="
    data:1,3,224,224" --input_format=NCHW
```

Here is the breakdown of the different options used in this command:

1) `--model=./mobilenetv2.air`: Specifies the path to the input pre-trained model in the AIR format.
2) `--framework=1`: Specifies the input model framework type as MindSpore, which is represented by the number 1.
3) `--output=garbage`: Specifies the name of the output directory for the generated offline model files.
4) `--soc_version=Ascend310`: Specifies the type of Ascend AI processor that the model will be deployed on. In this case, it is the Ascend310.
5) `--input_shape="data:1,3,224,224"`: Specifies the input shape of the model in NCHW format. Here, the input shape is 1 image with 3 channels and a size of 224×224 pixels.
6) `--input_format=NCHW`: Specifies the input data format for the model. NCHW stands for ”number of samples, number of channels, height, and width.”

The output terminal screenshots for the successful conversion of the ATC model and generation of the OM model in the specified directory are presented in Fig. 18 and Fig. 19, respectively.

Fig. 18: Output terminal result for generating OM model.

*2) **Preprocessing**:* we use OpenCV to read the provided ‘jpg’ images and perform normalization by doing the following steps:

```
 mean=[0.485*255, 0.456*255, 0.406*255]
std=[0.229*255, 0.224*255, 0.225*255]
image = (np.array(image) - mean) / std
```

and Resize (to $[224, 224]$), the same as in the training process. The `preprocess` function, located in `model_processor.py`, is illustrated in Fig. 20. It takes an original image as input and resizes it to a required size specified by `self._model_width` and `self._model_height`. It then normalizes the pixel values by subtracting the mean values (`[0.485*255, 0.456*255, 0.406*255]`) and dividing by the standard deviation (`[0.229*255, 0.224*255, 0.225*255]`).

*3) **Deploying the Garbage Classification Model**:* Next, we deploy the garbage classification code on the Atlas 200 DK by generating the OM model as shown in Fig. 18.

Before deploying, we need to make some modifications to the code.

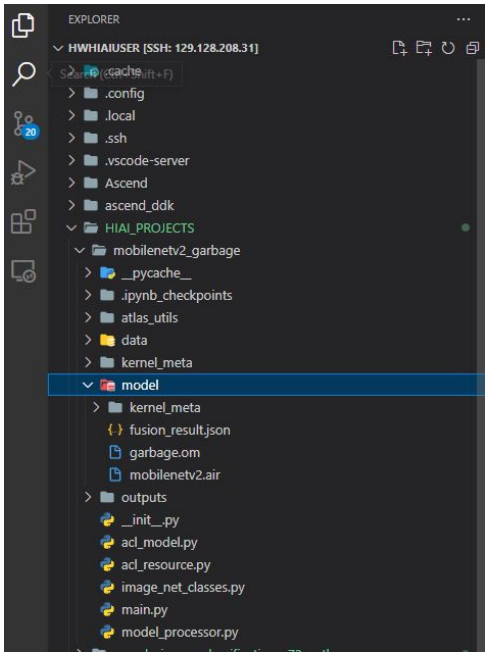

Fig. 19: The generated OM model.

```
def preprocess(self,img_original):
    '''
    preprocessing: resize image to model required size, and normalize value
    '''
    scaled_img_data = cv2.resize(img_original, (self._model_width, self._model_height))
    normalized_img = (scaled_img_data - np.array([0.485*255, 0.456*255, 0.406*255])) / np.array([0.229*255, 0.224*255, 0.225*255])
    # Caffe model after conversion, need input to be NCHW, the orignal image is NHWC, need to be transposed (use .copy() to change memory format)
    preprocessed_img = np.asarray(normalized_img, dtype=np.float32).transpose([2,0,1]).copy()

    return preprocessed_img
```

Fig. 20: `preprocess` function for image normalization.

1) First, we normalize the images as shown in Fig. 20 by modifying the `preprocess` function, located in `model_processor.py`.
2) Second, we revise the `image_net_classes.py` file with the related classes as shown in Fig. 21
3) Then, we download sample images that include a bottle, dirty clothes, and a newspaper.
4) To perform inference, we revise data paths in the `main.py` file as follows:
   `MODEL_PATH = "./model/garbage.om"`
   `DATA_PATH = "./data/newspaper.jpg"`

Following these steps, we can see the results of our model's inference on the sample images. Fig. 22, 23, and 24 show the images and console output in the VScode file explorer.

Fig. 21: Defining the image classes.

VI. Conclusion

This project provides a comprehensive overview of the use of the MobileNetV2 architecture, MindSpore framework, Huawei Ascend910 AI processor, and Atlas 200DK hardware platform for the development and deployment of a garbage classification task. Through this project, we have learned how to train a MobileNetV2 model on a garbage classification dataset using the MindSpore framework and Huawei Ascend910 AI processor, and how to deploy the trained model on the versatile and powerful Atlas 200DK hardware platform. This project has provided us with a deeper understanding of the capabilities of these cutting-edge technologies and their potential applications in the field of AI. By mastering the basic operations of model conversion using the ATC command and writing code for the garbage classification application using Python, we have gained valuable practical experience that will be useful in developing AI applications in the future.

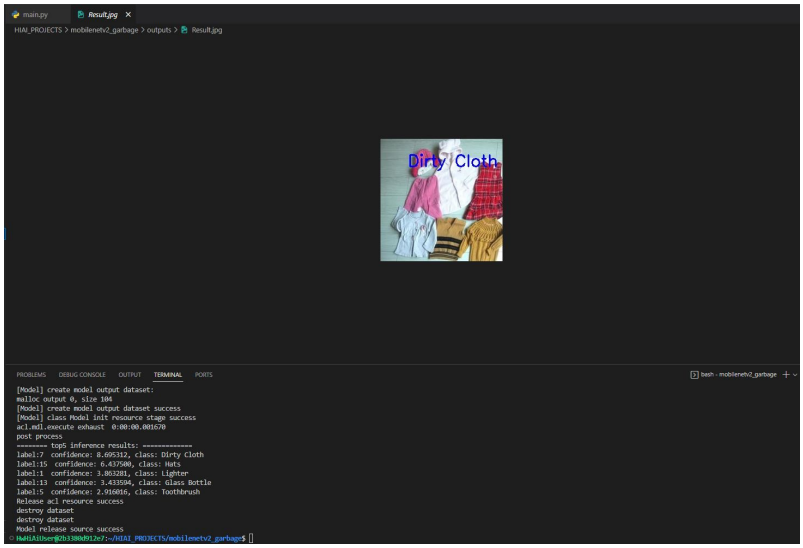

Fig. 22: Image object classification network pipeline for 'dirtycloth.jpg'.

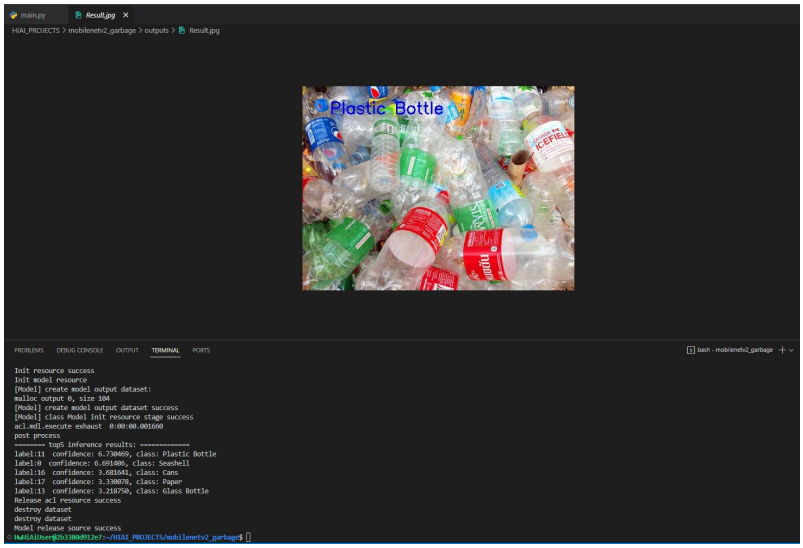

Fig. 23: Image object classification network pipeline 'bottle.jpg'.

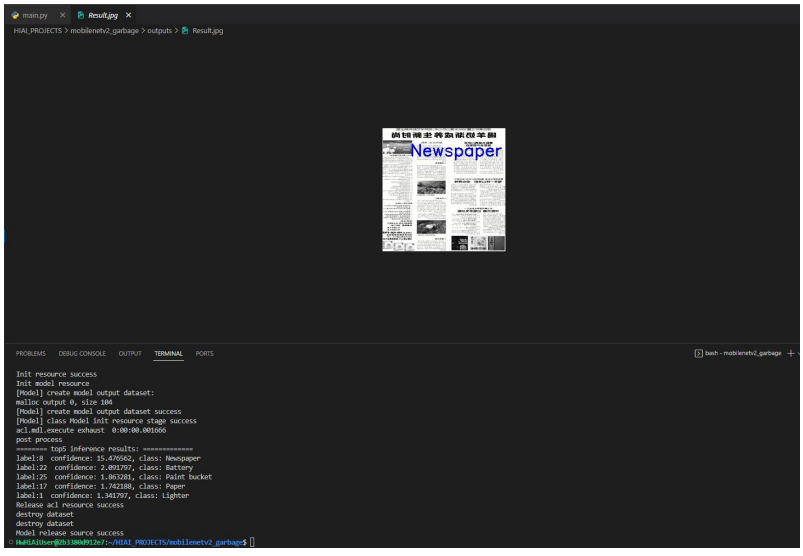

Fig. 24: Image object classification network pipeline 'newspaper.jpg'.

APPENDIX

A. Appendix A

```
import math
import numpy as np
import os
import random
import shutil
import time
from matplotlib import pyplot as plt
from easydict import EasyDict
from PIL import Image
import mindspore as ms
from mindspore import context
from mindspore import nn
from mindspore import Tensor
from mindspore.train.model import Model
from mindspore.train.serialization import
    load_checkpoint, save_checkpoint, export
from mindspore.train.callback import Callback,
    LossMonitor, ModelCheckpoint, CheckpointConfig
from dataset import create_dataset # dataset script
from mobilenetV2 import MobileNetV2Backbone,
    MobileNetV2Head, mobilenet_v2 # model definition

os.environ['GLOG_v'] = '2' # Log Level = Error
context.set_context(mode=context.GRAPH_MODE,
    device_target="Ascend") # graph mode, device
    using Ascend NPU
```

B. Appendix B

```
# Labels for classification

class_en = ['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']

index_en = {'Seashell': 0, 'Lighter': 1, 'Old Mirror
    ': 2, 'Broom': 3, 'Ceramic Bowl': 4, 'Toothbrush
    ': 5, 'Disposable Chopsticks': 6, 'Dirty Cloth':
     7,
        'Newspaper': 8, 'Glassware': 9, 'Basketball
            ': 10, 'Plastic Bottle': 11, 'Cardboard
            ': 12, 'Glass Bottle': 13, 'Metalware':
             14, 'Hats': 15, 'Cans': 16, 'Paper':
            17,
        'Vegetable Leaf': 18, 'Orange Peel': 19, '
            Eggshell': 20, 'Banana Peel': 21,
        'Battery': 22, 'Tablet capsules': 23, '
            Fluorescent lamp': 24, 'Paint bucket':
            25}

# Hyper-Parameters
config = EasyDict({
   "num_classes": 26,
   "image_height": 224,
   "image_width": 224,
   #"data_split": [0.9, 0.1],
   "backbone_out_channels":1280,
   "batch_size": 64,
   "eval_batch_size": 8,
   "epochs": 10,
   "lr_max": 0.05,
   "momentum": 0.9,
   "weight_decay": 1e-4,
   "save_ckpt_epochs": 1,
   "save_ckpt_path": "./ckpt",
   "dataset_path": "./data_en",
   "class_index": index_en,
   "pretrained_ckpt": "./mobilenetV2-200_1067.ckpt"
       # mobilenetV2-200_1067.ckpt
       mobilenetv2_ascend.ckpt
})
```

C. Appendix C

```
ds = create_dataset(dataset_path=config.dataset_path
    , config=config, training=False)
print(ds.get_dataset_size())
data = ds.create_dict_iterator(output_numpy=True).
    _get_next()
images = data['image']
labels = data['label']

for i in range(1, 5):
   plt.subplot(2, 2, i)
   plt.imshow(np.transpose(images[i], (1,2,0)))
   plt.title('label: %s' % class_en[labels[i]])
   plt.xticks([])
plt.show()
```

D. Appendix D

```
def cosine_decay(total_steps, lr_init=0.0, lr_end
    =0.0, lr_max=0.1, warmup_steps=0):
   """
   Applies cosine decay to generate learning rate
       array.

   Args:
     total_steps(int): all steps in training.
     lr_init(float): init learning rate.
     lr_end(float): end learning rate
     lr_max(float): max learning rate.
     warmup_steps(int): all steps in warmup epochs.

   Returns:
     list, learning rate array.
   """
   lr_init, lr_end, lr_max = float(lr_init), float(
       lr_end), float(lr_max)
   decay_steps = total_steps - warmup_steps
   lr_all_steps = []
   inc_per_step = (lr_max - lr_init) / warmup_steps
       if warmup_steps else 0
   for i in range(total_steps):
      if i < warmup_steps:
         lr = lr_init + inc_per_step * (i + 1)
      else:
         cosine_decay = 0.5 * (1 + math.cos(math.pi
             * (i - warmup_steps) / decay_steps))
         lr = (lr_max - lr_end) * cosine_decay +
             lr_end
      lr_all_steps.append(lr)

   return lr_all_steps
```

E. Appendix E

```
def switch_precision(net, data_type):
   if context.get_context('device_target') == "
       Ascend":
      net.to_float(data_type)
      for _, cell in net.cells_and_names():
         if isinstance(cell, nn.Dense):
            cell.to_float(ms.float32)
```

F. Appendix F

```
def extract_features(net, dataset_path, config):
    features_folder = dataset_path + '_features'
    if not os.path.exists(features_folder):
        os.makedirs(features_folder)
    dataset = create_dataset(dataset_path=
        dataset_path, config=config)
    step_size = dataset.get_dataset_size()
    if step_size == 0:
        raise ValueError("The step_size of dataset is
            zero. Check if the images count of train
            dataset is more \
           than batch_size in config.py")

    for i, data in enumerate(dataset.
        create_dict_iterator(output_numpy=True)):
        features_path = os.path.join(features_folder,
            f"feature_{i}.npy")
        label_path = os.path.join(features_folder, f"
            label_{i}.npy")
        if not os.path.exists(features_path) or not os
            .path.exists(label_path):
            image = data["image"]
            label = data["label"]
            features = net(Tensor(image))
            np.save(features_path, features.asnumpy())
            np.save(label_path, label)
        print(f"Complete the batch {i+1}/{step_size}")
    return

backbone = MobileNetV2Backbone() #last_channel=
    config.backbone_out_channels
load_checkpoint(config.pretrained_ckpt, backbone)
extract_features(backbone, config.dataset_path,
    config)
```

G. Appendix G

```
def train_head():
    train_dataset = create_dataset(dataset_path=
        config.dataset_path, config=config)
    eval_dataset = create_dataset(dataset_path=config
        .dataset_path, config=config)
    step_size = train_dataset.get_dataset_size()

    backbone = MobileNetV2Backbone()
    # Freeze parameters of backbone. You can comment
        these two lines.
    for param in backbone.get_parameters():
      param.requires_grad = False
    load_checkpoint(config.pretrained_ckpt, backbone)

    head = MobileNetV2Head(input_channel=backbone.
        out_channels, num_classes=config.num_classes)
    network = mobilenet_v2(backbone, head)
    #switch_precision(network, ms.float16)

    # define loss, optimizer, and model
    loss = nn.SoftmaxCrossEntropyWithLogits(sparse=
        True, reduction='mean')
    lrs = cosine_decay(config.epochs * step_size,
        lr_max=config.lr_max, warmup_steps=0)
    opt = nn.Momentum(head.trainable_params(), lrs,
        config.momentum, config.weight_decay)
    net = nn.WithLossCell(head, loss)
    train_step = nn.TrainOneStepCell(net, opt)
    train_step.set_train()

    # train
    history = list()
    features_path = config.dataset_path + '_features'
    idx_list = list(range(step_size))
    for epoch in range(config.epochs):
        random.shuffle(idx_list)
        epoch_start = time.time()
        losses = []
        for j in idx_list:
            feature = Tensor(np.load(os.path.join(
                features_path, f"feature_{j}.npy")))
            label = Tensor(np.load(os.path.join(
                features_path, f"label_{j}.npy")))
            losses.append(train_step(feature, 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-{
                epoch+1}.ckpt"))

    # evaluate
    eval_model = Model(network, loss, metrics={'acc',
        'loss'})
    acc = eval_model.eval(eval_dataset)
    print(acc)

    return history
```

```
if os.path.exists(config.save_ckpt_path):
    shutil.rmtree(config.save_ckpt_path)
os.makedirs(config.save_ckpt_path)

history = train_head()

plt.plot(history, label='train_loss')
plt.legend()
plt.show()

CKPT = f'mobilenetv2-{config.epochs}.ckpt'
print("Chosen checkpoint is", CKPT)
```

H. Appendix H

```
class EvalCallback(Callback):
    def __init__(self, model, eval_dataset, history,
        eval_epochs=1):
        self.model = model
        self.eval_dataset = eval_dataset
        self.eval_epochs = eval_epochs
        self.history = history
        self.acc_max = 0
        # early stopping, accuracy stops increasing in
            5 epochs
        self.count_max = 5
        self.count = 0

    def epoch_begin(self, run_context):
        self.losses = []

    def step_end(self, run_context):
        cb_param = run_context.original_args()
        loss = cb_param.net_outputs
        self.losses.append(loss.asnumpy())

    def epoch_end(self, run_context):
        cb_param = run_context.original_args()
        cur_epoch = cb_param.cur_epoch_num
        train_loss = np.mean(self.losses)

        if cur_epoch % self.eval_epochs == 0:
            metric = self.model.eval(self.eval_dataset,
                dataset_sink_mode=False)
            self.history["epoch"].append(cur_epoch)
```

```
        self.history["eval_acc"].append(metric["acc
            "])
        self.history["eval_loss"].append(metric["
            loss"])
        self.history["train_loss"].append(
            train_loss)
        if self.acc_max < metric["acc"]:
           self.count = 0
           self.acc_max = metric["acc"]
        else:
           self.count += 1
           if self.count == self.count_max:
              run_context.request_stop()
        print("epoch: %d, train_loss: %f, eval_loss
            : %f, eval_acc: %f" %(cur_epoch,
            train_loss, metric["loss"], metric["acc
            "]))
```

I. Appendix I

```
from mindspore.train.loss_scale_manager import
    FixedLossScaleManager
LOSS_SCALE = 1024
def train():
   train_dataset = create_dataset(dataset_path=
       config.dataset_path, config=config)
   eval_dataset = create_dataset(dataset_path=config
       .dataset_path, config=config)
   step_size = train_dataset.get_dataset_size()

   backbone = MobileNetV2Backbone() #last_channel=
       config.backbone_out_channels
   # Freeze parameters of backbone. You can comment
       these two lines.
   for param in backbone.get_parameters():
     param.requires_grad = False
   # load parameters from pretrained model
   load_checkpoint(config.pretrained_ckpt, backbone)

   # head = MobileNetV2Head(num_classes=config.
       num_classes, last_channel=config.
       backbone_out_channels)
   head = MobileNetV2Head(input_channel=backbone.
       out_channels, num_classes=config.num_classes)
   network = mobilenet_v2(backbone, head)

   # define loss, optimizer, and model
   loss = nn.SoftmaxCrossEntropyWithLogits(sparse=
       True, reduction='mean')
   loss_scale = FixedLossScaleManager(LOSS_SCALE,
       drop_overflow_update=False)
   lrs = cosine_decay(config.epochs * step_size,
       lr_max=config.lr_max)
   opt = nn.Momentum(network.trainable_params(), lrs
       , config.momentum, config.weight_decay,
       loss_scale=LOSS_SCALE)
   model = Model(network, loss, opt,
       loss_scale_manager=loss_scale, metrics={'acc'
       , 'loss'})

   history = {'epoch': [], 'train_loss': [], '
       eval_loss': [], 'eval_acc': []}
   eval_cb = EvalCallback(model, eval_dataset,
       history)
   cb = [eval_cb]
   ckpt_cfg = CheckpointConfig(save_checkpoint_steps
       =config.save_ckpt_epochs * step_size,
       keep_checkpoint_max=config.epochs)
   ckpt_cb = ModelCheckpoint(prefix="mobilenetv2",
       directory=config.save_ckpt_path, config=
       ckpt_cfg)
   cb.append(ckpt_cb)
   model.train(15, train_dataset, callbacks=cb,
       dataset_sink_mode=False)
   return history
```

```
if os.path.exists(config.save_ckpt_path):
   shutil.rmtree(config.save_ckpt_path)

history = train()

plt.plot(history['epoch'], history['train_loss'],
    label='train_loss')
plt.plot(history['epoch'], history['eval_loss'], 'r'
    , label='val_loss')
plt.legend()
plt.show()

plt.plot(history['epoch'], history['eval_acc'], 'r',
     label = 'val_acc')
plt.legend()
plt.show()

CKPT = 'mobilenetv2-%d_40.ckpt' % (np.argmax(history
    ['eval_acc']) + 1) # Choose the best Checkpoint,
     modify the value '40' according to dataset and
    batchsize.
print("Chosen checkpoint is", CKPT)
```

J. Appendix J

```
def image_process(image):
   """Precess one image per time.

   Args:
      image: shape (H, W, C)
   """
   mean=[0.485*255, 0.456*255, 0.406*255]
   std=[0.229*255, 0.224*255, 0.225*255]
   image = (np.array(image) - mean) / std
   image = image.transpose((2,0,1))
   img_tensor = Tensor(np.array([image], np.float32)
       )
   return img_tensor

def infer_one(network, image_path):
   image = Image.open(image_path).resize((config.
       image_height, config.image_width))
   logits = network(image_process(image))
   pred = np.argmax(logits.asnumpy(), axis=1)[0]
   print(image_path, class_en[pred])

def infer():
   backbone = MobileNetV2Backbone(last_channel=
       config.backbone_out_channels)
   head = MobileNetV2Head(input_channel=backbone.
       out_channels, num_classes=config.num_classes)
   network = mobilenet_v2(backbone, head)
   print(os.path.join(config.save_ckpt_path, CKPT))
   load_checkpoint(os.path.join(config.
       save_ckpt_path, CKPT), network)
   for i in range(91, 100):
      infer_one(network, f'data_en/test/Cardboard
          /000{i}.jpg')

infer()
```

K. Appendix K

```
backbone = MobileNetV2Backbone(last_channel=config.
    backbone_out_channels)
head = MobileNetV2Head(input_channel=backbone.
    out_channels, num_classes=config.num_classes)
network = mobilenet_v2(backbone, head)
load_checkpoint(os.path.join(config.save_ckpt_path,
    CKPT), network)
```

```
input = np.random.uniform(0.0, 1.0, size=[1, 3, 224,
    224]).astype(np.float32)
export(network, Tensor(input), file_name='
    mobilenetv2.air', file_format='AIR') # MindSpore
    1.0
```

REFERENCES

[1] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," pp. 4510–4520, 2018.

[2] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pp. 4510–4520, Computer Vision Foundation / IEEE Computer Society, 2018.

[3] U. Seidaliyeva, D. Akhmetov, L. Ilipbayeva, and E. T. Matson, "Real-time and accurate drone detection in a video with a static background," *Sensors*, vol. 20, no. 14, p. 3856, 2020.

[4] Huawei, "Mindspore." https://e.huawei.com/in/products/cloud-computing-dc/atlas/mindspore, March 2020.

[5] Huawei, "Atlas 200 dk ai developer kit." https://e.huawei.com/in/products/cloud-computing-dc/atlas/atlas-200, March 2020.