NuNet Data Gathering and Training Steps

1. Equipment Used

Digital camera or smartphone with a depth-sensing feature (e.g., iPhone or Android).

Digital weighing scale for precise food measurements.

2. Procedure

Step 1: Collecting RGB Images

Orientation: Each dish was captured from four angles—top, front, left, and right—to provide more perspectives.

(From testing model's performance based on 1 angle image versus 4 angle images, performance doesn't seem to improve much when providing 4 angle images, so capturing a top-down image would be easier and provide equivalent model mass prediction performance)

Specifications: Images were taken in portrait mode with consistent lighting.

Step 2: Weighing Food

Each dish was weighed using a digital scale to record:

- **Initial weight:** Measured before consuming.
- **Partial weights:** Measured during consumption to capture variations of mass to further improve model's predictions for various masses of food.
- Containers' weights were deducted to ensure only the food mass was recorded.

These weights were recorded alongside the dish name in a structured json file:

{

"food_name": "Ayam Penyet_1",

"file_path": " Ayam Penyet /Ayam Penyet_1.jpg",

"weight": 600,

"depth_map_file": "depth_maps/ Ayam Penyet_1_Depth.jpg"

},

Keeping the data in this format allows us to easily convert this json into either MiniCPMv2's food recognition model's training format or NuNet's mass estimation training format. With Food name for identification of the food, file_path to ensure that specific food's image path is recorded, weight/mass for mass estimation model and its depth map file locations for mass estimation model as well.

You can reference masterfile.json to see the format that was used in ITP2 and can ignore source and ingredients as we found those were not needed after already recording those down.

Step 3: Generating/Obtaining Depth Maps

For generating/obtaining Depth maps, 2 methods can be used, either manually or automatically via Apple's depth estimation model, due to the cumbersome process of manually obtaining depth maps however, Apple's depth estimation model is advised to be used instead, and from model's performance testing, accuracy for mass estimation is almost negligible when comparing to manually collected depth map images.

(Around 1-2% mass estimation variation when comparing manually collected versus depth maps generated by Apple's model)

Manual Depth Extraction:

For iOS: Images with embedded depth metadata were transferred via wired connections or cloud services (e.g., Google Drive).

(Note, taking images without using portrait mode specifically will result in the images not having depth captured, also if the files were transferred via telegram or WhatsApp to obtain the photos in computers for their depth, the depth maps in those images will be lost, requiring manually connecting phones to the computer and extracting the photos manually to preserve the depth maps).

Depth maps were extracted using Photopea (https://www.photopea.com/), isolating the depth layer and saving it as JPG files.

For Android: The Hasaranga (https://www.hasaranga.com/dmap/) Depth Map Generator was used for dual output (RGB and depth).

Automated Depth Map Generation:

Apple's Depth Pro model was employed to generate additional depth maps for RGB images lacking depth data.

You can go to https://huggingface.co/apple/DepthPro or https://github.com/apple/ml-depth-pro and follow their instructions to setup Apple's depth estimation model.

(The model itself is already download and provided in the project directory but if any errors were encountered, the documentation from Apple's model from either their Huggingface or Github can be referenced)

The script DLDepth.ipynb was used to obtain the depth images based on all images found in the directory stated in the script, ignoring all the files that already have depth images created and mass producing all the depth images required for fine-tuning.

Step 4: Storing and Organizing Data

Images and depth maps were stored in separate directories:

RGB Images: /Dataset/Images/

Depth Maps: /Dataset/Depth Maps/

Metadata (e.g., dish name, angles, weights) was stored in structured files (train_data.txt, test_data.txt).

The scripts **convert_to_CorrectFormat.py**, **convert_to_CorrectFormat_1PerFoodType.py** and **convert_to_CorrectFormat_4PerFoodType.py** can be used to convert masterfile.json to NuNet's test_data.txt and train_data.txt, both of these files being used for training and test scripts later on.

Step 5: Training/Fine-tuning

1. Prerequisites

Before initiating the training or fine-tuning process, ensure the following:

All RGB images and depth maps are collected and stored in their respective directories (/Dataset/Images/ for RGB images and /Dataset/Depth Maps/ for depth maps).

Metadata files (train_data.txt, test_data.txt) are properly formatted and include essential fields such as dish name, weight, and depth map paths.

Required Python Libraries: Install the libraries listed in requirements.txt using the following command: **pip install -r requirements.txt**

2. Training Preparation

Data Preprocessing:

The dataset is split into 80% training and 20% testing subsets. For NuNet, the split can vary based on available weighted images (e.g., 920 for training and 180 for testing).

Augmentation is applied to the training set using data_augment.py to improve the model's ability to generalize. Techniques include rotation, flipping, and color adjustments.

Model Configuration:

The fine-tuning process uses full parameter fine-tuning for NuNet and MiniCPM-V2 models.

Specify the training hyperparameters in the configuration file or script:

- ***Learning Rate: Default set to 1e-4.***
- ***Epochs: Standard configurations include 3, 75, and 150. (You can change max_epochs in train.py to increase or decrease based on your requirements)***
- ***Batch Size: Default set to 16.***
- ***Optimizer: Adam optimizer is used with a weight decay of 1e-5.***

```
    if args.strategy == "None":
        print(f'args.strategy:{args.strategy}')
        trainer = pl.Trainer(
            fast_dev_run=args.fast_dev_run,
            logger=logger,
            devices=1,
            accelerator='cuda',
            min_epochs=1,
            max_epochs=75,
            precision=args.precision,
            enable_checkpointing=False,
            callbacks=[TrainCallback()],
        )

    trainer.fit(model, dm)


if __name__ == "__main__":
    parser, args_dict = get_parser()
    main(parser, args_dict)
```

3. Training the Model

You can run main.py which will run both train.py and test.py (test.py being ran after train.py has finished training the model).

```
    # Load the Wandb run ID
    wandb_run_id = 'Example'

    # Initialise the WandbLogger with the saved run ID
    logger = WandbLogger(id=wandb_run_id, resume="allow",
                         project=project_name, group=group_name)
    logger.experiment.tags = logger.experiment.tags + run_tags
    logger.experiment.notes = str(args.description)

    # Path to your saved checkpoint
    checkpoint_path = "experiment/Example.ckpt"
```

You'll have to create a wandb account and use their api key provided to you after creating your account then input it into wandb_ran_id, chaning "Example" to your API key as well as changing checkpoint_path's Example.ckpt to your api key as the model's checkpoint file name is based on your api key.

```
# NuNet Mass Estimation Model
parser, args_dict = get_parser()
args = parser.parse_args([])
checkpoint_path = r"experiment/Example.ckpt"
nUNet_model = RGB_D_Constructor.load_from_checkpoint(checkpoint_path, args=args)
nUNet_model.eval()
```

In Run_TEST_WithDepthEsti.py, the same will have to be done with changing "Example.ckpt" to your api key, E.g (31gs56sgd31grfg113.ckpt).

4. Evaluation

Model Performance:

The fine-tuned models are evaluated on the test dataset. Metrics such as Mean Absolute Percentage Error (MAPE) and classification accuracy are calculated.

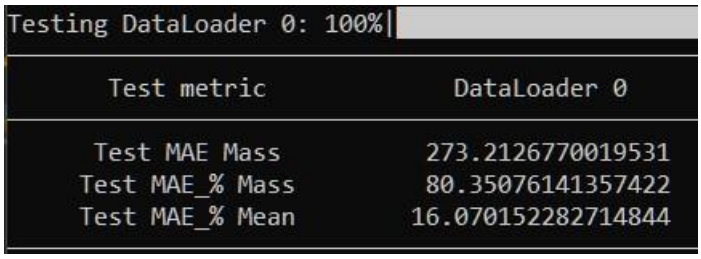
For NuNet, testing includes scenarios like:

Training with local datasets versus a mix of local and Nutrition5K datasets.

Testing on unseen food categories to assess generalizability, to see if the model is able to predict what a food's mass is without being trained on that specific food's images and mass (E.g training on chicken rice but testing on noodles to see if it can still somewhat predict what the noodle's mass is without being trained on noodle images and mass).

In Training Results folder, these were the recorded model's performance based on the different testing criteria, E.g **N5K-Pretrained- (UNSEEN) - 1 Image and Angle Per Food Type (3 Epoch).jpg, N5K-Pretrained-1 Image and Angle Per Food Type (75 Epoch).jpg,** and many more that was tested during ITP2.

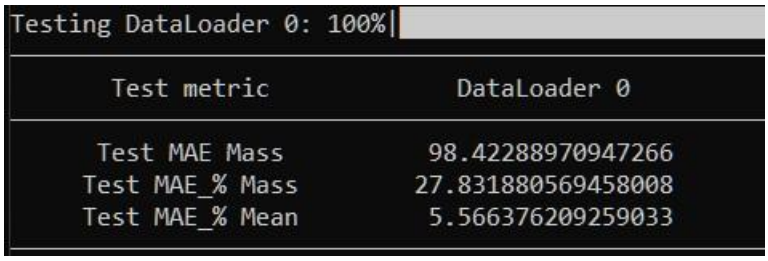
*(Important note, when utilizing the evaluation scripts, **Ignore Test MAE_% Mean**, as that's from the first iteration of the code, **Use Test MAE_% Mass for the right MAPE Percentage**, basically ignore the final line in the Test metric results, using the second last line as the MAPE)*



Testing DataLoader 0: 100%

Test metric	DataLoader 0
Test MAE Mass	273.2126770019531
Test MAE_% Mass	80.35076141357422
Test MAE_% Mean	16.070152282714844

N5K-Pretrained- (UNSEEN) - 1 Image and Angle Per Food Type (3 Epoch).jpg



Testing DataLoader 0: 100%

Test metric	DataLoader 0
Test MAE Mass	98.42288970947266
Test MAE_% Mass	27.831880569458008
Test MAE_% Mean	5.566376209259033

N5K-Pretrained-1 Image and Angle Per Food Type (75 Epoch).jpg