# Project Meal Suggestion

Patrick Bonack, Sebastian Heinemann, Nick Wagner, Felix Lapp, Ya Jiang

March 2021

Github: `https://github.com/5yntek/Meal-Suggestion`

# Contents

# 1 User View

No matter where one lives, "what do I eat?" is a question we all face on a daily basis. A healthy diet is not only important to prevent obesity, diabetes and other chronicle cardiovascular diseases. It is also critical for mental health, as too much consumption of processed food can bring down people's mood.

The project Meal Suggestion combines FoodAI technology with healthy recipe database. This section explains the scope and the purpose of the designed system. Specifically, context, tasks and performance requirements will be discussed.

## 1.1 Context

As technology advances and impacts more aspects of people's life, FoodAI has been developed by programmers and researchers for deep-learning-based image recognition function. Millions of food images from hundreds of classes were trained and made it possible for AI to identify a specific kind of grocery based on an image with precision. This project combines the FoodAI technology with a database for healthy recipes. Simply taking a picture from his or her smart phone or other smart devices with a camera, an end-user can instantly receive the information of available ingredients in the kitchen. All ingredients in the fridge or on the kitchen counter will be identified and recommendations of healthy recipes based on the identified groceries will be sent to the user.

In some cases groceries are contained in special packaging. it is important to find a way to identify those. Food can be contained in its original supermarket packaging, plastic bags, plastic boxes, aluminum foil, glasses or any other container. Groceries in its original packaging can be identified by scanning a barcode. Foods in any of the other containers will likely not be identifiable with certainty which means that our project will be trimmed on detectable food in the beginning. As soon as the base function are working it is intended to look into detection through packaging.

## 1.2 Tasks and Requirements

The first task is raw ingredient identification from pictures. Within the Meal Suggestion APP, an end-user should be able to take pictures of the groceries he or she has in the kitchen and upload them to the APP. The user should also be able to upload previously taken pictures from the photo album. In both cases, given high-resolution images of food ingredients, the task is to identify the food ingredients in the images. The deep-learning Convolutional Neural Network (CNN) in the APP will analyse the images, classify food items in the images into categories of fruit, protein, vegetables and carbohydrates, and identify the exact food items in the images.

The second task is setting up users' choices in the APP. The Meal Suggestion APP should

include a check-process so that the end-users can confirm if the list of identified ingredients is correct or not. If the analysis is not correct, users can correct the error and the error message will be sent to the server to train the Convolutional Neural Network for a more accurate result in the future. The APP will update and download the most up-to-date information of the Convolutional Neural Network once users have internet connection.

In some cases groceries are contained in special packaging. The APP should include a function that gives users the option to scan the bar code on food packages in case the groceries are in supermarket packaging. Food ingredients in other packaging such as plastic bags, plastic boxes, aluminum foil, glasses or any other container should also be able to be included in the ingredient list. For these special packaging food, the APP should allow users to manually add ingredients into the list of identified ingredients.

Another task is pushing recipes to user interface from recipe database on a backend server. After confirming and editing all the available ingredients, the list of ingredients will be uploaded to the server for database of recipes. The database will run on the backend server. The APP sends an inquiry containing all the ingredients the user has at disposal to the backend server. The server sends all filtered recipes back to the app. The database should include only healthy recipes that have the appropriate amount of protein, vitamins, salt and fat intakes. Recipes in the database for which the users have all the ingredients will be filtered out and sent back to the users. The users may set their preferences such as cuisine preference, special diet preference (lactose-intolerant, gluten-free, vegetarian, vegan) as well as other general filters. Given the filters, appropriate recipes should show up in a list.

# 2 Modeller View

This section elaborates the system design choices for the task and performance requirements for the Meal Suggestion APP.

## 2.1 World Model

The Meal Suggestion APP is designed to identify food ingredients in images and gives healthy recipes based on the identified ingredients. The goal of the sensor is to process the environment and construct a model of the world, so that the constructed model is as close to the real world as possible. For this project's AI food recognition process, ingredients to be classified and the environment where the food ingredients sit are of the most importance. Environment should not only include refrigerator but also other cooking or dining areas such as the kitchen counter and dining table. In addition, for ingredients in supermarket package, barcode needs to be added to the world model. It goes without saying that sensors and illumination also need to be taken into consideration.

For detailed world plate and observational plate, please see Section 3.3 "World Plate and Observational Plate".

## 2.2 Architecture Workflow

As shown in Figure 1, the architecture consists of three modules: a database, one backend-service and the mobile App. The mobile App uses the mobile device's camera for ingredient recognition. The channel between Spring Boot Web-Service and the App transfers data of identified ingredients and filtered recipe results. For a more detailed view on data flow and communication, please see the UML diagram in Figure 2.

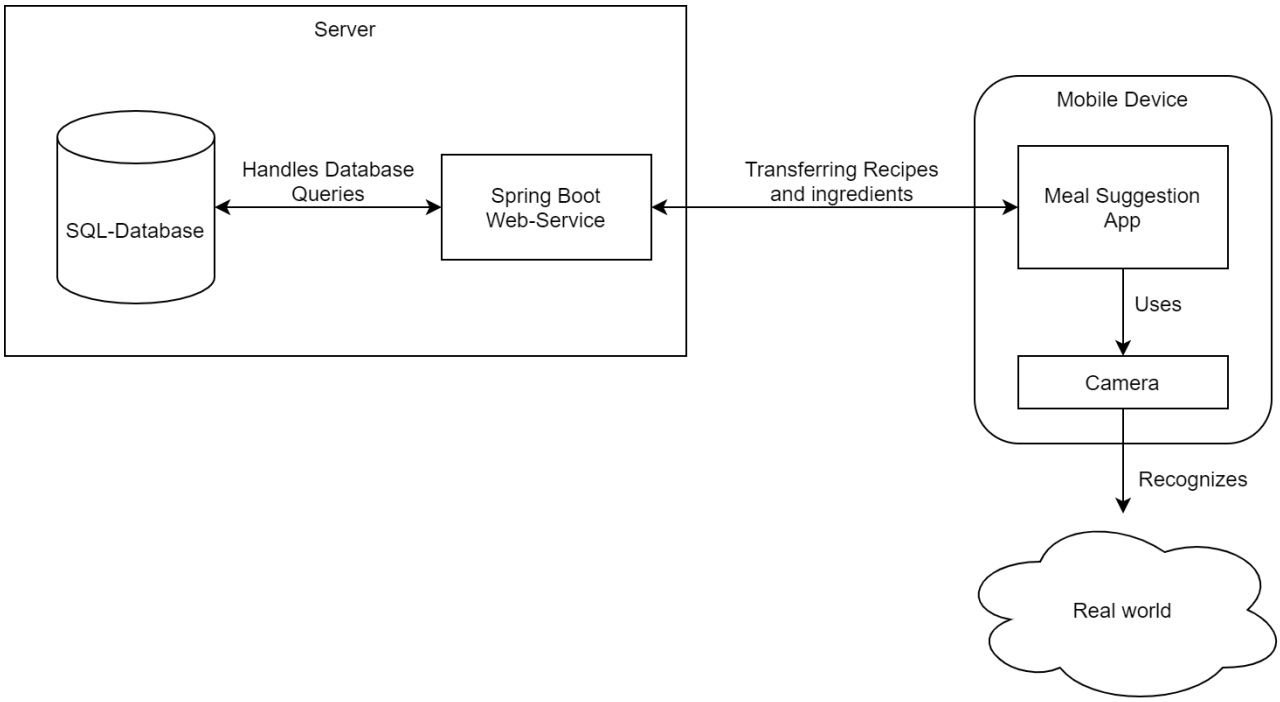



Figure 1: The architecture's system

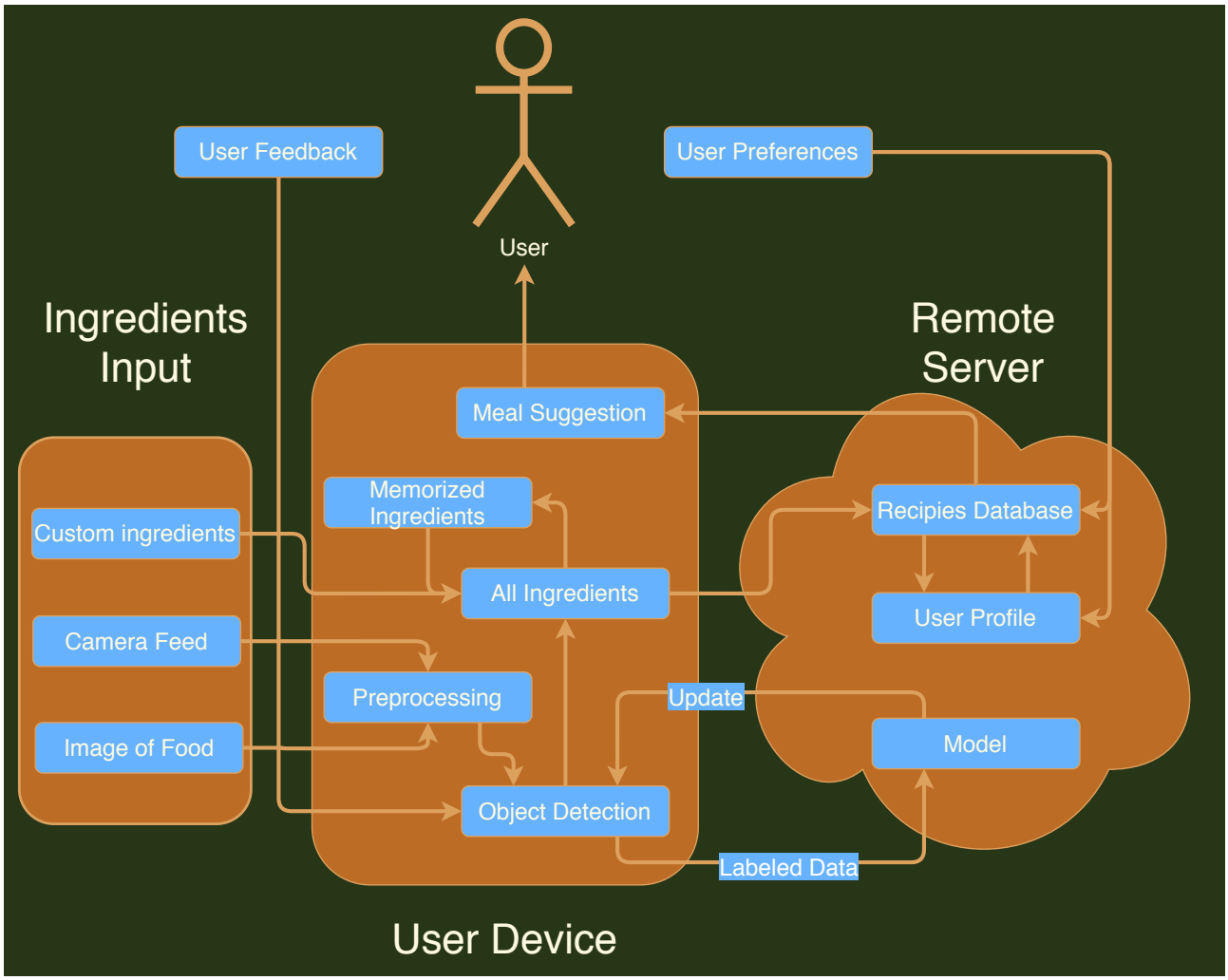



Figure 2: Architecture Workflow

## 2.3 Conceptual Model

Conceptual modelling gives many benefits. Among them the most important ones are to construct a mental picture of the system, to convert tactic knowledge explicitly and to communicate concepts better to others people. In conceptual modelling, processes and objects are the key. Processes can transform an object by creating the object, affecting the object like changing the objects state, or destroying the object. According to the Object-Process Theorem, all domains are able to be modelled by stateful objects and processes and relations among them.

To demonstrate our system design more clearly, we use an Object-Process Methodology (OPM) diagram.

### 2.3.1 OPM

Below is the Object Process Methodology (OPM) for our Meal Suggestion project. Figure 3 is the main diagram and Figure 4 is diagram for sub-process Edit Ingredient List. In the OPM diagrams, objects with solid border line are objects within the system, while objects with dashed border line are objects out of the system environment.

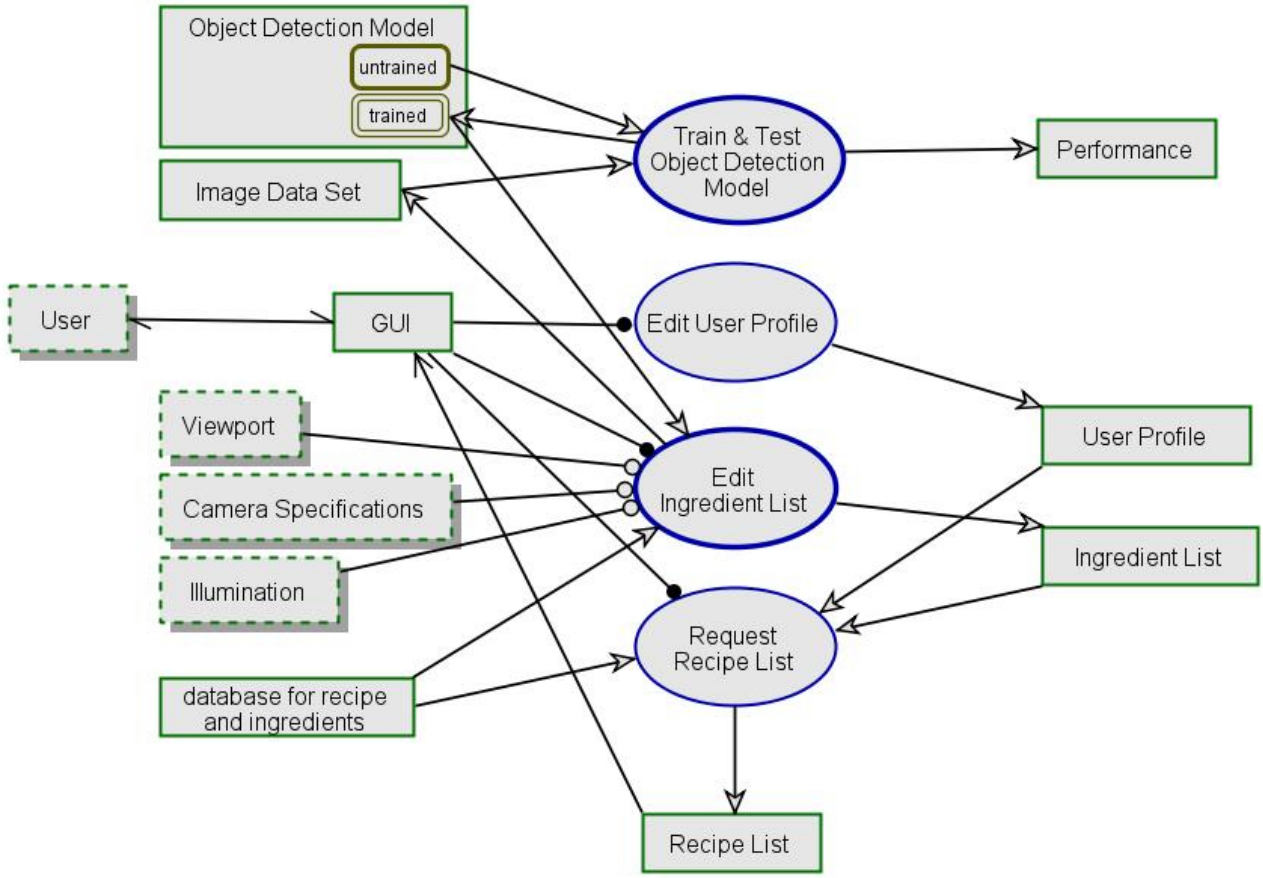



Figure 3: System Top Layer

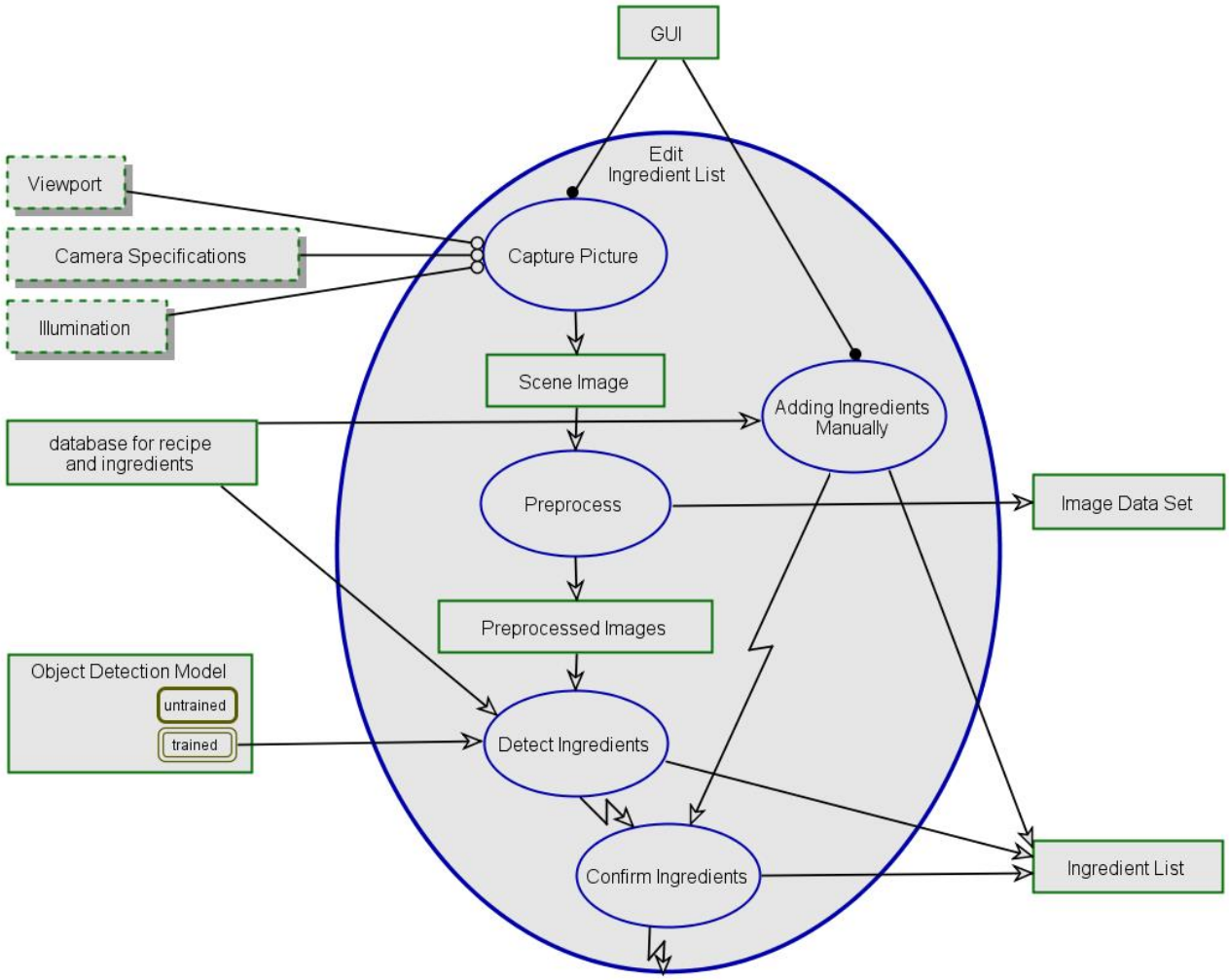



Figure 4: Edit Ingredient List

The first process in Figure 3, the main diagram is Train and Test Object Detection Model. This process requires Object Detection Model and Image Data Set for Performance.

Within the APP through GUI, users can edit their user profile.

After the set-up processes comes the most important process Edit ingredient List. Viewport, Camera Specifications and Illumination are required to this process which later updates the Training Data Set and Ingredient List. More specifically, as labelled in Figure 4, Viewport, Camera Specifications and Illumination are necessary to capture a picture. The captured image is then processed to detect the ingredients in the image. The users also have an option to add the ingredients manually. Both the Detect Object process based on image recognition and Add Manually process based on users' manual typing of the ingredients decide the content in the Ingredient List. In both cases, the ingredients detected or manually added should also be present in the database for recipes and ingredients: as laid out in the ER diagram (Figure 6 under Section 2.6), recipes in the database contain ingredients labeled in the database with ID and name.

The last process in Figure 3 the main diagram is Request Recipe List. This process consumes user profile setting, existing ingredient list and the database for recipe and ingredients. The process will trigger a presentation of filtered recipe list based on the previous processes. This recipe list is presented in GUI.

## 2.4 Sensors

For ingredient identification process, the Meal Suggestion APP takes images of food ingredients taken by the users or uploaded from the digital gadget's photo album. Since users will most likely use their phone or digital tablet for the APP, relevant sensors to the Meal Suggestion APP are cameras and depth sensors.

Cameras for consumer digital products usually use complementary metal-oxide semiconductor (CMOS) sensors because they are usually cheaper and consume less batteries compared to charge-coupled device (CCD) sensors. CCD sensors that are suited more for professional, commercial broadcasting use. An RGB-camera, for example, is a camera with a standard CMOS sensor through which coloured images of objects or person are acquired.

In recent years, more consumer-used digital gadgets have included depth sensors to the gadgets for new technology like Augmented Reality (AR). Apple Inc., for instance, has included LiDAR camera on their iPhone 12 and the new iPad Pro product to support AR capabilities of iOS and help the performance of the gadgets' camera in low light. LiDAR stands for Light Detection and Ranging. In comparison to cameras which measures the color (in need of visible light) for object or person detection, Lidar system consists of a laser and receiver which measure the distance to objects and is suited for 3D object detection.

Although as technology advances there are likely more types of sensors available for consumer digital products, at the moment the Meal Suggestion APP will consider only Lidar and cameras relevant to the project.

## 2.5 Universal Function Approximation

Generally speaking, there are two types of end-to-end system pipelines: one is a traditional shallow pipeline with feature representation and the other is deep recognition pipeline which is a fully data driven approach where a neural net is trained with labelled data.

The first type of pipeline— modular probabilistic modelling with the the system and domain's properties— requires features as input. For instance, if the end-goal is to recognise an apple in an image, features of an apple $\epsilon_1$, $\epsilon_2$,...,$\epsilon_n$ such as the unique geometry and texture of the apple are needed as input for the function F. To minimise error of the output of F, one needs to specify and minimise the errors in features $\epsilon_1$, $\epsilon_2$,...,$\epsilon_n$.

The second type of pipeline—deep recognition pipeline— is fully data-driven. The trained

neural net is presented and trained with labelled data. Figure 5 gives a graphic depiction of the major differences between a traditional shallow pipeline with feature extraction and a deep recognition pipeline. Major advantages of deep recognition pipeline (universal function approximation) are higher accuracy for high dimensional and model free problems and flexibility of the function approximation framework. CNN for object detection is especially good for data that are generated from the same underlying mechanism. However, there are also drawbacks of the fully data-driven approach. For example, the convolutional neural network demand a huge amount of training data and computing power. A lot of variety of implantation choices need to be tuned.

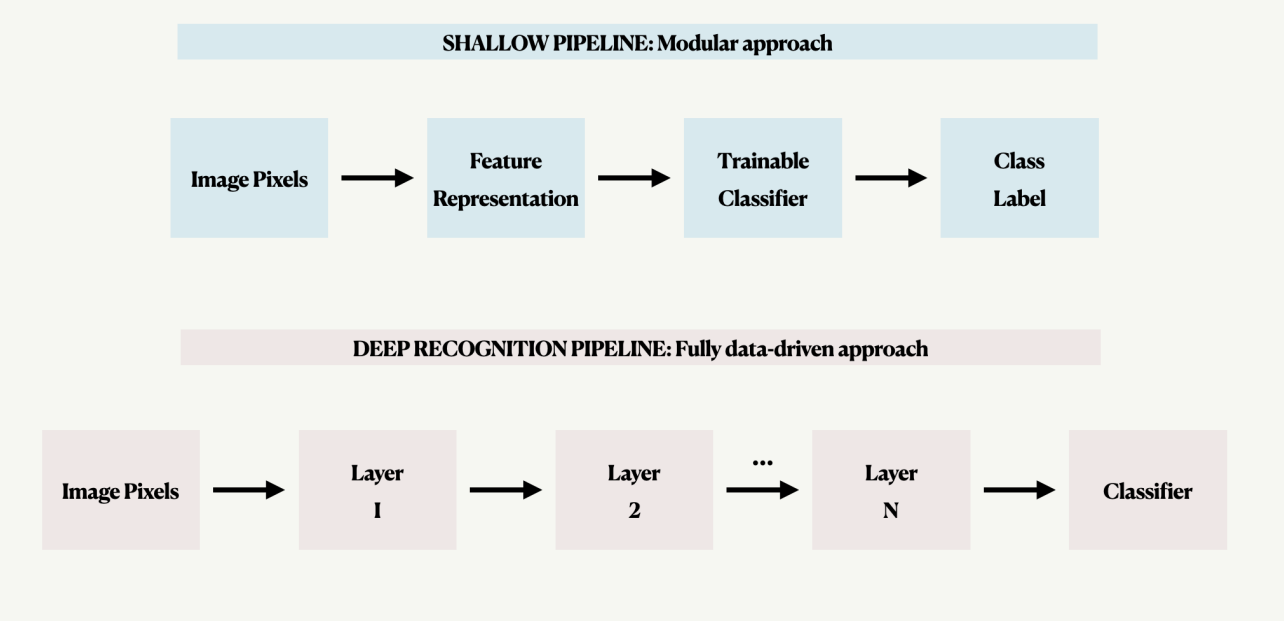



Figure 5: End-to-end system pipelines

For the Meal Suggestion project, we adopt a fully data-driven approach. We produce a lot simulated images of different ingredients in the fridge or kitchen counter environment. All types of ingredients are included and labeled in the simulated images for training the convolutional neural network. We will adapt upon the YOLOv5 model in which meta-parameters are already tuned. For details, please see Section 4.2 "AI Recognition".

## 2.6 Database and Backend

This section contains information about what happens on the webserver.
The entire list of known recipes is stored in a single database laying on a web server. Since each recipe consists of a specific amount of ingredients, there is a second table for every known ingredient. A detailed ER-diagram for the database's structure can be seen in figure 6.
It is important, that the set of ingredients which can be recognized by the AI is a subset of all ingredients in the database.
There will also be a backend service running on the same server. It is supposed to manage

database access and make it accessible using HTTP-endpoints.

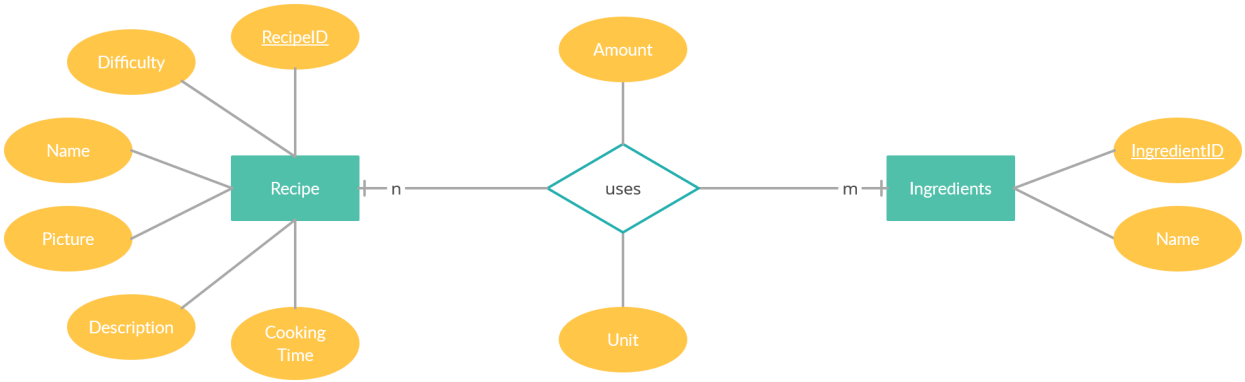
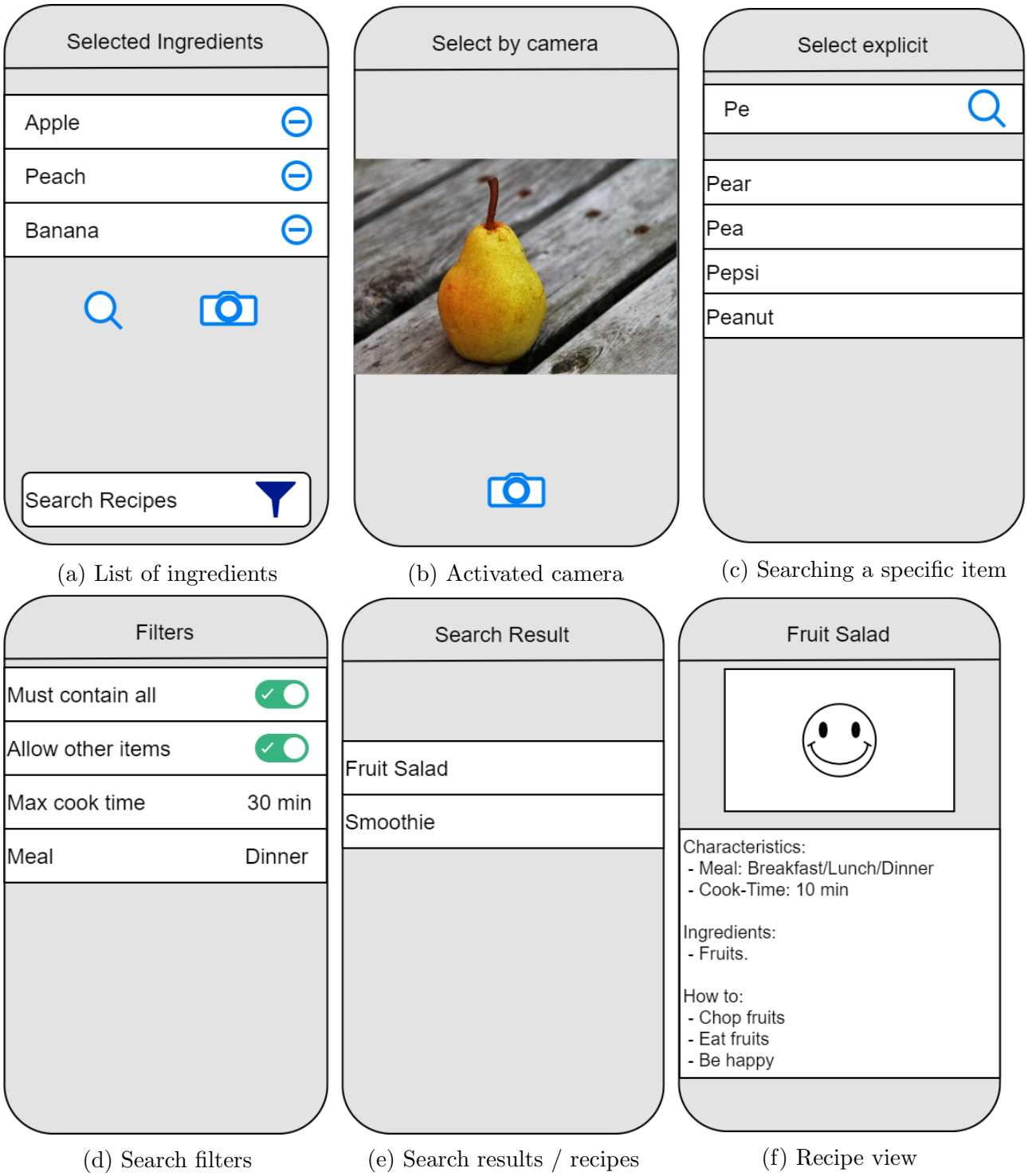



Figure 6: ER-model

## 2.7 Mobile Application

Within the APP, users should be able to check if the list of identified ingredients is correct. This will be achieved by a list of current ingredients providing the possibility to remove and add ingredients (Figure 7a). For the groceries in special packaging, the APP allows users to scan the bar code on food packages or manually add ingredients into the list of identified ingredients. This will be achieved by implementing a search bar for displaying all ingredients which are known for the database. A prototype for this search-bar-screen is displayed in Figure 7c. Users may set their meal preferences such as cuisine preference, special diet preference (lactose-intolerant, gluten-free, vegetarian, vegan) as well as other general filters (see Figure 7d) in the recipe searching step. Given the filters, appropriate recipes should show up in a list as seen in Figure 7e. After selecting one, the app displays a handy overview including a how-to for the selected recipe (Figure 7f).The database will run on the backend server and will get requests from the app on the users' phone. The app sends an inquiry containing all the ingredients the user has at their disposal to the backend. The backend has to provide all recipes back to the app which contain the given ingredients. The database structure is described in the ER-model in Figure 6.




(a) List of ingredients (b) Activated camera (c) Searching a specific item

(d) Search filters (e) Search results / recipes (f) Recipe view

Figure 7: Prototype for the app's screens

# 3 Simulation

## 3.1 Camera Noise

As discussed in Section 2.4 Sensors, camera is relevant to the Meal Suggestion APP.

In reality, when taking a picture, photons strike the material during exposure time and electrons may be emitted. This will form a charge which will be amplified and digitised, resulting noise to the picture. Simulated hologram, on the other hand, does not include camera noise. Hence, for a better simulation, noise factor needs to be considered and added to the simulated hologram so that the end-result images are more realistic as if they are taken by a real camera. Noise factor is especially important with CMOS sensors, since they in general generate more noise.

The EMVA 1288 Standard lays out models which characterise the CMOS and CCD cameras well. [4] The following Figure 8 is a graphical explanation of a mathematical model of a single pixel from the EMVA 1288 Standard document.

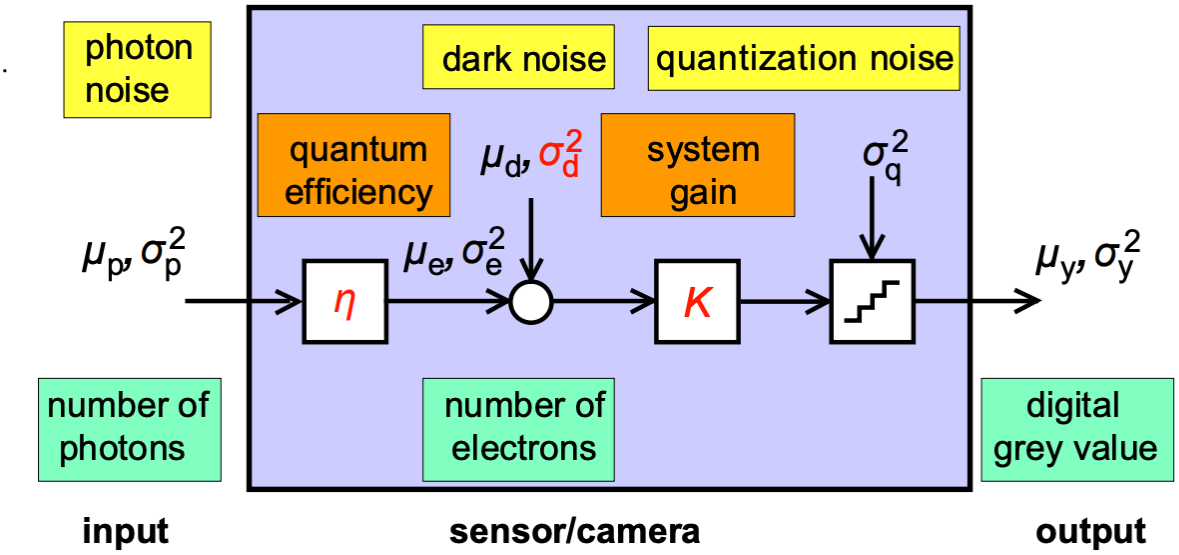



Figure 8: Noise: mathematical model of a single pixel

In this model, the camera is a single pixel that takes number of photons as input and returns digital grey value as output. During the transformation, dark noise and quantisation noise are added. Based on probability theories and some assumptions, the mean and the variance of the output as well as the signal-to-noise ratio (SNR) can be derived:

$$\mu_y = K\mu_d + K\eta\mu_p$$

$$\sigma_y^2 = K^2\sigma_d^2 + K(\mu_y - K\mu_d)$$

$$SNR = \frac{\eta\mu_p}{\sqrt{\sigma_d^2 + \sigma_q^2/K^2 + \eta\mu_p}}$$

where p = the number of photons hitting the camera, d = the dark noise, y = output
$\mu$ = mean, $\sigma^2$ = variance, and K = sensitivity of the camera.

For formula derivation and more details, please see paper [5].

In his blog Modeling Noise for Image Simulations [5] , Kyle M. Douglass demonstrates a model of simulating read noise and dark current based on factors such as photon shot noise and the number of photoelectrons. He also shows the detailed code of adding realistic camera noise to the simulated holograms. Since his code handles mostly pixel-to-pixel variations in the output of the grey values, the type of noise in his model is temporal noise.

Adding noise to simulated holograms is an important part of simulation. For this project, we are adopting the noise adding method kindly shared by Kyle M. Douglass. [5]

## 3.2 World Plate and Observation Plate

To get a better understanding of what is important for our system we defined a world model consisting of the world plate and the observation plate, containing all parameters that play a role in recognizing ingredients from images. The implementation of the simulation will be later based on some these parameters. Furthermore, defining all parameters that influence the recognition of an object helps to keep track of the relevant data.

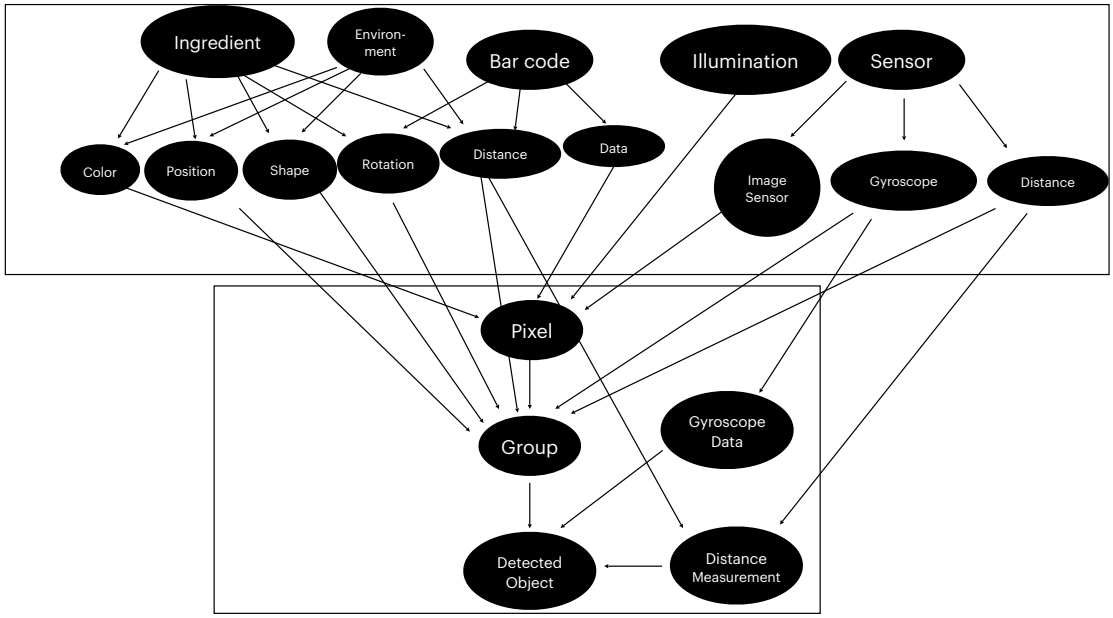



Figure 9: world plate  observational plate

From 9 we get the following parameters for the world plate:

- $\Theta color = \Theta red, \Theta green, \Theta blue$
- $\Theta position = \Theta x, \Theta y, \Theta z$
- $\Theta shape = \Theta polygon$
- $\Theta rotation = \Theta u, \Theta v, \Theta w$
- $\Theta data = \Theta spacing\_of\_lines, \Theta width\_of\_lines$
- $\Theta illumination = \Theta solar\_rad, \Theta fridge\_lighting, \Theta kitchen\_lighting, \Theta camera\_flash$

And the parameters for the observational plate:

- $\Theta ij(pixeli, j)$
- $\Theta g(groupofpixels)$

# 4 Implementation

## 4.1 Simulation

In this chapter, we will take a look on the simulation. It will be shown, how a simulation-based image generator can be created which produces domain-like images. Afterwards the use of annotations which are required to evaluate and proof the correctness of the model's outcomes are pointed out. The annotations could also be used to train the model.

**Tools** First of all, the choice of the tools that were used are summarized by the performance tasks:

- Realtime is not needed: images can be rendered in non-Realtime applications.
- physically based rendering: The images should be as close to the real world as possible.
- Automatically generation: Enables the computation of hundreds of images needed for the database.
- Providing Annotations: In order to evaluate the performance of the model, annotated images have to be provided.
- Low budget: Open source is preferred

The decision was made in favour of Blender (version 2.92) with its included Python version 3.7. One additional criterion was, that it is well known by the developer.

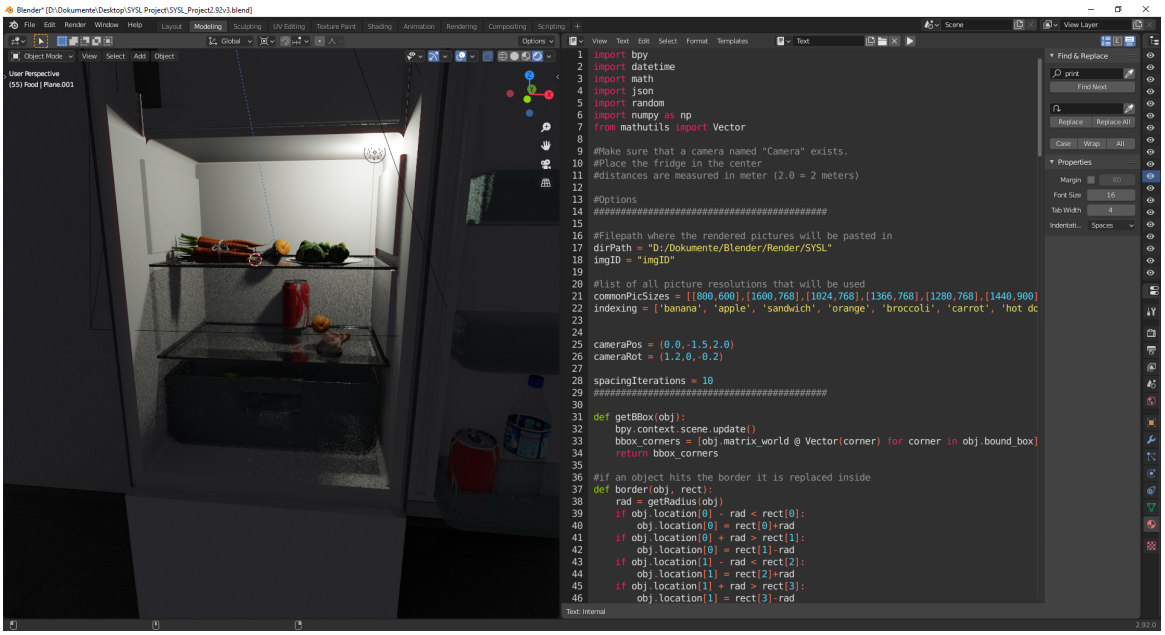



Figure 10: blender project

**Design the world plate** The simulation needs, like the task and performance chapter, an analysis of the world plate, but in a more technical way. The task is to render food in its environment (kitchen). Therefore a set of parameters has to be found, that are influential to the task of capturing food images in real life. These so called "Wold Parameters" can be plugged into the blender project to render with underlying physically correctness. To cover as many scenarios as possible none of them are fixed values, meaning that they will be unique for every single image. The simulation chooses these parameters by random value generators.

**World parameter** Camera: A simulated camera device:

- Position: (x,y,z)
- Rotation: (x,y,z)
- Lens: The focal length of the Camera
- Resolution: the pixel size of the rendered image (width, height)
- Bloom: Simulates diffraction of the lens
- (noise)

Illumination:

- Color: (r,g,b)
- Intensity: float
- Spectacular: softness of the reflective light

Food:

- Size
- Position
- Rotation
- Visibility: Which of the food is shown in the scene

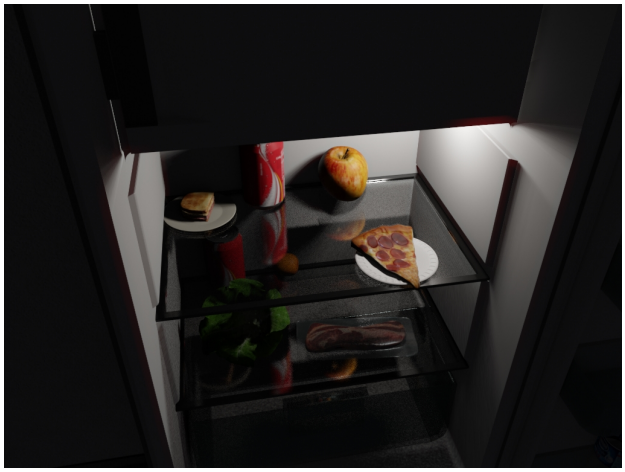
**How does the simulator work?** First of all, the Blender project shows a scene that contains an empty fridge, a camera object and all food elements that can be placed in the fridge. All of the world parameters are set initially. The core of the simulation is a python script that does all the magic. All the functionalities of Blender can be accessed and executed by python functions. A few parameters at the top of the script can be changed by the user to set the target destination, number of images that will be rendered and the labelling. By running the script, the generation of a new scene is progressed, the world parameter are chosen randomly, the rendering of the images is done and the output data is saved fully automatically. The process traverses following steps:

- Choose and set the world parameter randomly
- Choose a set of the food elements and place it in the fridge
- Render and save the images
- For every ingredient of the image create a mask
- Loop that for all images

**Output** The output is a folder, containing a predefined amount of images. Setting the world parameters randomly has the effect that every image looks different.

(a) good conditions

(b) overexposure



(c) bad lightning conditions

(d) bad camera positioning

Figure 11: different looking examples of the created images

After that the annotations of each image will be created. The importance of the annotation will be shown later on in the evaluation and the training section. The procedure is explained using picture 12 as an example, but is done for every image in the output directory.

Figure 12: image 271

In order to create annotations the blender project creates masks for the image. A mask of a food object is nothing less than an image with an alpha channel, where the alpha channel covers every pixel that is not associated with the food object. The number of masks is determined by the number of food elements that are shown on the image. In this example four masks are created.

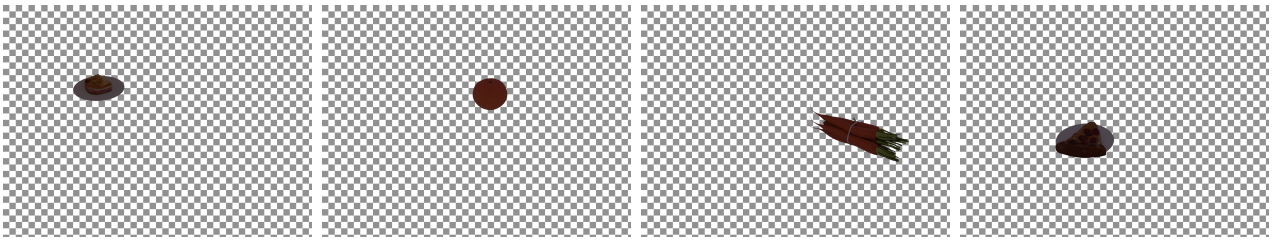


Figure 13: masks of image 271

A script determines the bounding boxes of every mask by looping over pixel coordinates and observing the alpha channel. The bounding box is denoted as the minimum and maximum x and y coordinates of the pixels with an alpha channel unequal 0.

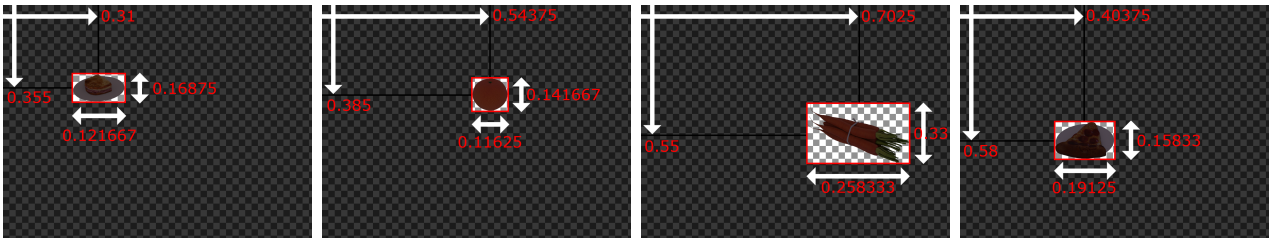



Figure 14: bounding boxes of the masks of image 271

The annotation needs the ids of all objects to which the image and the masks are related to.

That's the reason why the blender project saves the masks named with those indices. The assignment of the id is given by the models object array.

```
'banana', 'apple', 'sandwich', 'orange', 'broccoli','carrot', 'hot dog', 'pizza',
'donut', 'cake'
```

Index 0 relies on 'banana', index 2 on 'apple' and so on. Therefore a mask named 0.png will show a banana. The very last step of the simulation is to create the annotation files. For every image one file is created, where the ids and the bounding boxes are pasted in. Every row belongs to one food object and contains the id, the x and y position, the width and height of the bounding box.

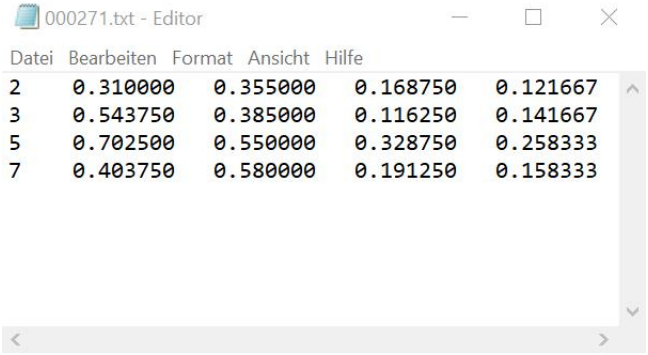



Figure 15: label file 271

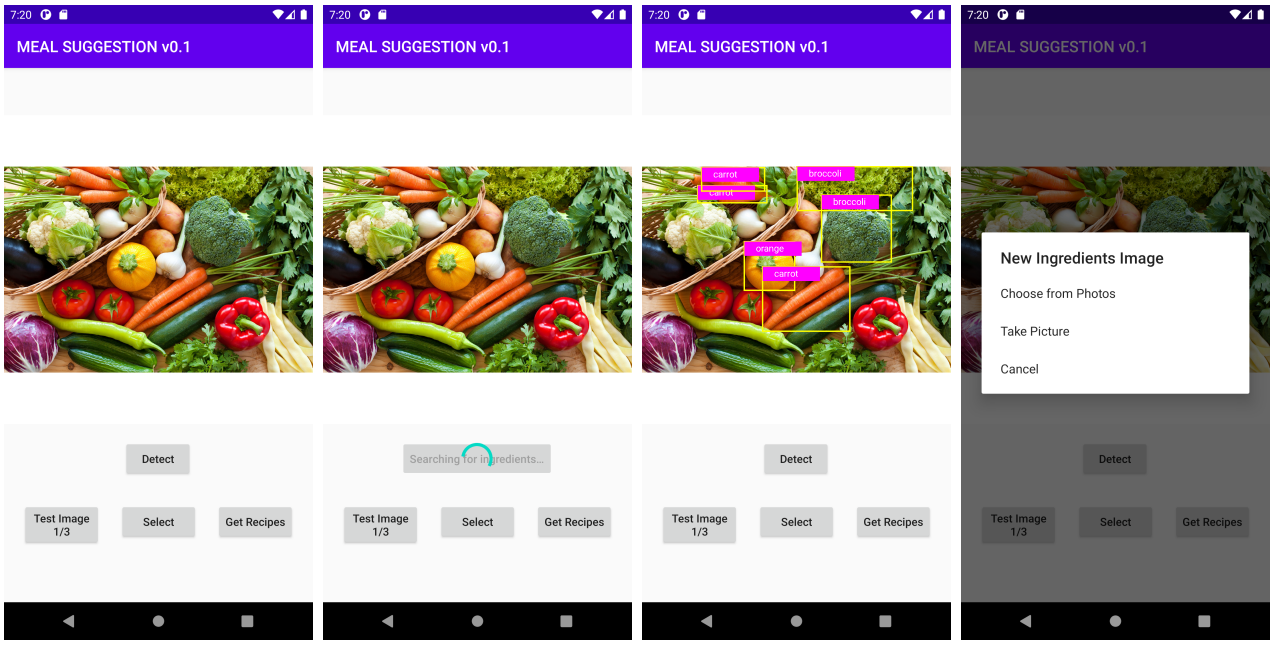
## 4.2 AI Recognition

The scope of this portion of our projects includes modifying and retrain a pre-trained checkpoint of the model `https://github.com/ultralytics/yolov5` for our needs, extracting food-relevant data from Coco dataset [2] and combine it in a working android-app with the purpose of detecting food-ingredients on pictures and highlighting them.

### 4.2.1 The App: Object Detection

The object detection part of our app is a prototype for demonstration and should be regarded as such. There are many features missing we would like to include in the (theoretical) final version of this app. For example we would like to give the user the possibility to notify us, if an ingredient is not correctly detected and in the best case provide us with high quality labeled data.

The app currently includes the following features, as seen in figure 16.




(a) start screen (b) searching objects (c) detected objects (d) select picture

Figure 16: Different states of Object Detection app

**Button Test Image x/3:** Clicking on this button will provide you with 1 out of 3 example pictures of ingredients in a fridge.

**Button Select** Button Select lets you choose pictures stored in your smartphone or take a picture with your camera module.

**Button Detect** Detect will search for ingredients in the chosen picture and highlight them with a bounding box together with the name of the detected class.

**Button Get Recipes** Get recipes button is a dummy, as the part of the app that communicates with the database back-end is currently separated in its own application.

This app is an Android Studio project built with gradle. There are no other gradle dependencies. The app based on an example provided by pytorch [1].

### 4.2.2 YOLOv5s

The actual object detection is done by a model called YOLOv5s. It is the smallest version of YOLOv5 [7], an open-source implementation of the latest version of YOLO. In the project we use the PyTorch scripted YOLOv5s model to detect objects of the food category. Classes with

the supertype food is a subset of the 80 COCO classes the model is originally trained on. Due to lack of more training data the current model only uses data provided by COCO. Our plan was to generate labeled data by simulation and use it as additional training and testing data. While simulation at time of writing this is progressing well, it is not at a point we can use the to train the model.

YOLOv5 is trained on 80 classes of the coco Dataset. We are only interested in the following 10:

```
'banana', 'apple', 'sandwich', 'orange', 'broccoli','carrot', 'hot dog', 'pizza',
'donut', 'cake'
```

In a later state of this project we would like to add more appropriate classes if we have the necessary data.

### 4.2.3 The models

Our initial goal was to test three different approaches using YOLOv5. 1) Using an existing pre-trained model as is. 2) Retrain the model with only COCO-Data containing food. 3) Train the model with simulated data.

1) and 2) were successful, 3) is work in progress.

### 4.2.4 The pre-trained model

The concrete base model we used is `yolov5s`. It is the smallest, but also the fastest YOLO model. Not only is the time consumed training the model by far the smallest, also the inference speed for object detection is the highest. Every YOLO model is based on the COCO database and has therefore 80 detectable classes. As previously stated we are actually only using the 10 classes containing food.

We solved this problem simply by only using the inference data of these 10 classes while ignoring the remainder.

The result is quite good. To fairly compare this model for our purpose with the newly trained model, we only use validation data containing food and only use the respective labels of the food classes. The actual comparison between the two approaches are located in a later section.

### 4.2.5 The trained model

Our YOLOv5 model is trained on 16.255 food related images from the COCO-Database. The initial weights are those of YOLOv5s.

The training-pipeline of the model included the following steps:

- Download Coco-Data (images and labels) and install COCO-API for python
- Extract only relevant Data containing food
- Create labels that are usable for YOLO which is using the Darknet format
- Organize Directories of Data
- Create a `.yaml` file describing the classes and Data-Location
- Train the YOLOv5 model on COCO-Data, hyper-parameter tuning included.
- Visualize results

All needed information to install the Python COCO-API and download the COCO data you find here `https://github.com/cocodataset/cocoapi`. Particularly useful was the Jupyter Notebook `/cocoapi/PythonAPI/pycocoDemo.ipynb`.

Steps 2-4 were done via our jupyter notebook `pycoco_ingredients.ipynb`

In step 5 we created two `.yaml` files `food.yaml` and `food_coco.yaml`. We have to create two different `.yaml` files as the output layer of the pre-trained model and our model are not of the same size. In figure 17 you can see the final folder structure described in those `.yaml` files. The first model contains only the 10 food related classes and its data is stored in the folder `/food`. The latter contains all 80 COCO classes and the data is stored in `/food_coco`. The image data and their labels are fundamentally the same. The only difference are the class ids of food objects. The folder `/food_coco/images` is actually a hard-link to `/food/images`. No need to waste disk-space.

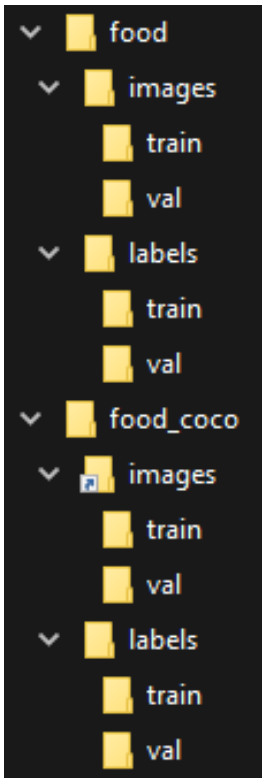



Figure 17: folder structure of training and validation data

After these preparation steps we are now able to train the model in step 6. You can recreate the process by using the jupyter notebook `/yolov5/train_food.ipynb`. The actual code used to start the process is:

```
python train.py --img 640 --batch 16 --epochs 5 --data
food.yaml --weights yolov5s.pt
```

We train the model with initial weights defined by `yolov5s.pt` with the data defined by `food.yaml` for 5 epochs. `img` declares the size of one (resized) image used in the training. `batch` defines the size of one batch.

5 epochs is clearly not enough for a model used in the final product. Due to our training-machine being not that powerful and

the training process is quite time consuming, we decided that for a prototype the results were good enough.

### 4.2.6 Difference YOLO & COCO labels

Darknet [3] and therefore YOLO needs for every image a `.txt` file with the same basename. These files define one object per row. The format is `class xcenter ycenter width height`. The box-values are normalized, meaning every value is between 0-1. Class numbers are indexed and start by zero.

The label data of COCO is stored in a single JSON file for all images. Each annotation of every object has its own entry. The annotation is linked to an image by the value `imageid`. The bounding-box information for an object stored in the value `bbox`. This bounding-box format is `top_left_x_position top_left_y_position width height`.

In figure 18 you can see an example provided by YOLOv5 describing darknet bounding boxes.

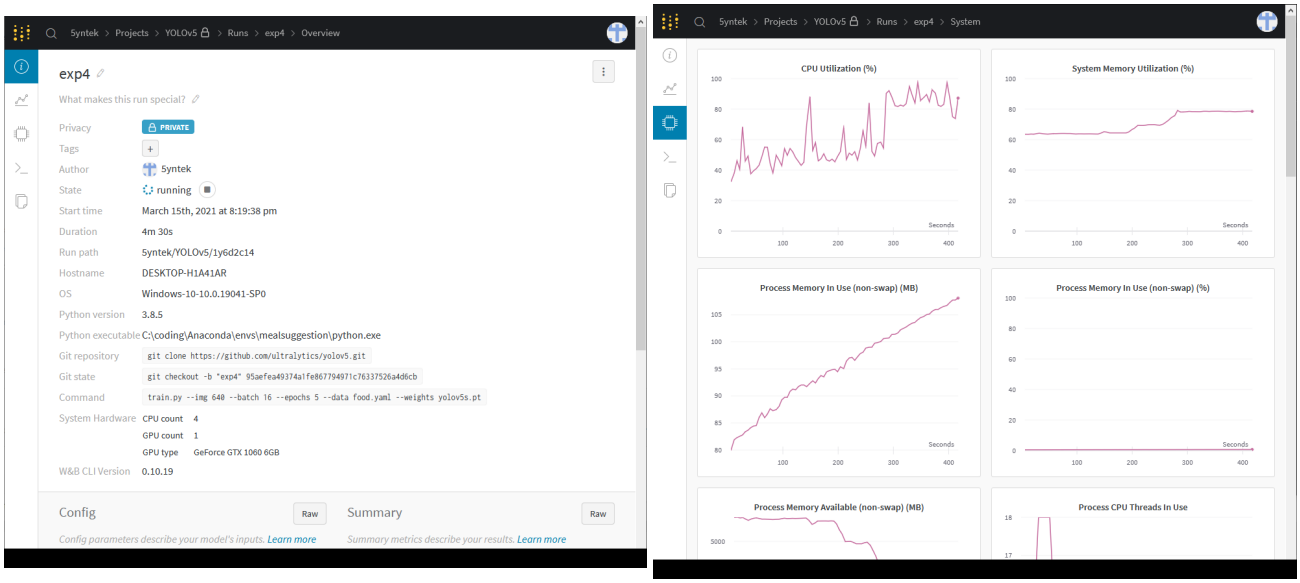

Figure 18: Bounding boxes as defined in YOLOv5

### 4.2.7 Visualization of training

YOLOv5 provides an integrated visualization and logging toolkit for the training-process with Weights & Biases [6]. You need to install it with `pip install wandb`. After creating an account the information about your training is automatically stored online easily shareable between group members. In addition YOLOv5 provides local logging and creates some useful images about the quality of the resulting model.

Wandb does not only provides information about the training results but also the training

process itself e.g information about the system (figure 19a and used resources (figure 19b). Here you can also see the most relevant information about our training machine.




(a) wandb: run overview

(b) wandb: system resources used

Figure 19: wandb: system information

While training you are continuously provided with information about the training progress. Particularly demonstrative were information about the used batches, see figure 20.

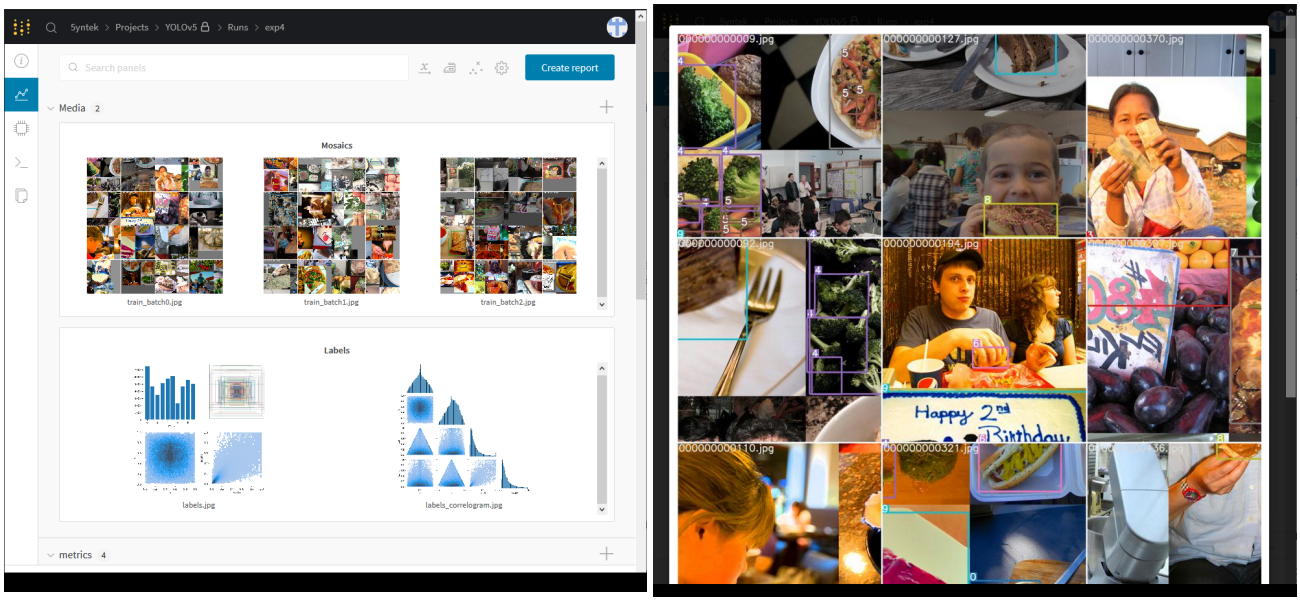



Figure 20: wandb: batch information

Multiple diagrams are created to analyse the quality of the model. The confusion matrix (figure 21) and the F1 curve (figure 22) seems to be especially meaningful. Further diagrams include,

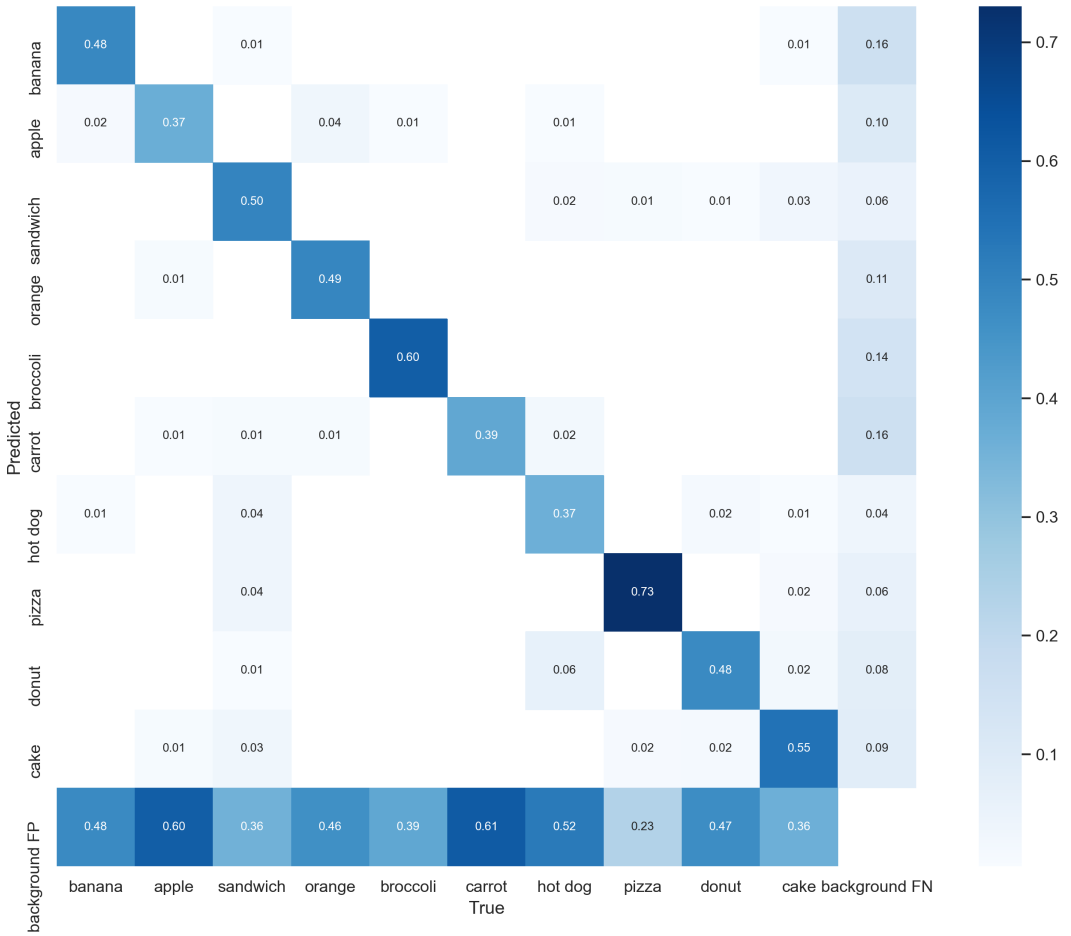



Figure 21: confusion matrix

P-, R- and PR-curve, information about labels and their correlation, multiple metrics for every training epoch and images about the difference of the predicted and actual objects of each test-batch. More information about the evaluation results are located in section 5.1 Model Result Evaluation.

## 4.3 Database and Backend

Database and Backend Service are both laying on a server which is accessible via internet. The database uses MariaDB and the backend was done using Java and Spring Boot. This service provides some HTTP-endpoints under port 7010 to ensure the app's functionality. These are listed in table 1.

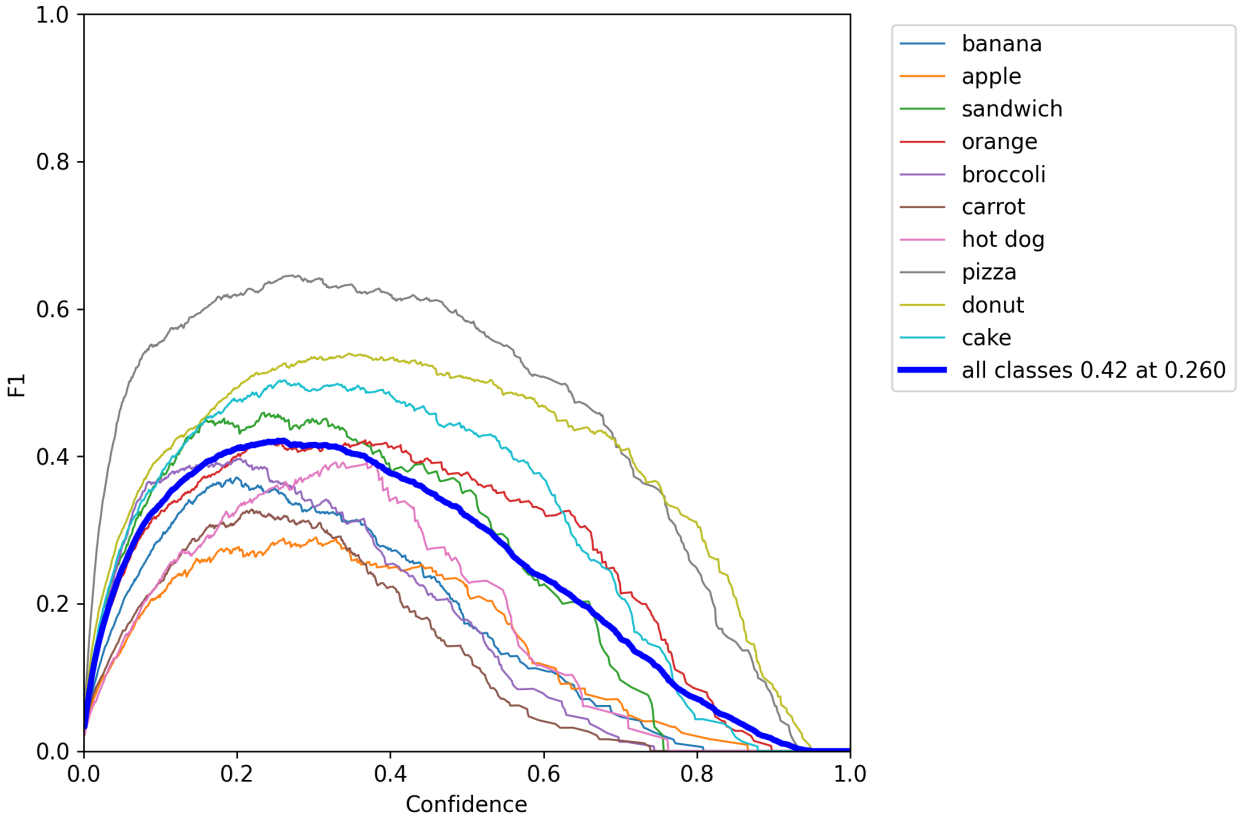



Figure 22: F1-Curve

| **Endpoint** | **Method** | **Parameters** | **Description** |
|---|---|---|---|
| `/recipes` | GET | None | Responds with a list of recipes formatted using JSON |
| `/recipes` | POST | Request Body: List of IDs formatted using JSON | Responds with a list of recipes just like the GET-Request on the same URL but only with recipes containing all ingredients represented by the given IDs |
| `/ingredients` | GET | None | Responds with a list of ingredients formatted using JSON |

Table 1: HTTP-Endpoints of the backend service

## 4.4 Mobile Application

The mobile app (later called "app") has been done using the react-native framework and expo client. Several packages and dependencies have been used additionally. They can be found in file `MealSuggestionApp/package.json` (see GitHub-Repository).
The app consists of 5 Screens:

1. **Loading Screen**
   Checks connection to the backend service and fetches all known ingredients. This is important so the amount of requests can be low later when searching ingredients (figure 23a)

2. **Ingredients Screen**
   Shows a list of ingredients which have been added. You can add ingredients using the search screen. In figure 23b you can see how it looks like when two ingredients have been added.

3. **Search Screen**
   Shows a search bar to find ingredients based on their name. By click on a suggested ingredient, the selected one will be added to the ingredients screen (figure 23c).

4. **Recipe List Screen**
   Shows a list of recipes which contain the ingredients being selected in the ingredients screen. In figure 23d you can see one result. This one contains both ingredients selected in 23b.

5. **Recipe Screen**
   After clicking on a recipe on the previously described list, this screen will open to show a detailed description of the selected recipe (figure 23e)

There was also a sixth screen planned which should make it possible to take a picture of any food/ingredient to automatically add the recognized ingredients to the ingredient list.
Unfortunately this was not possible using react native and pytorch in time of this project. This is why there is a second app being only responsible for taking pictures and recognizing the ingredients. More information about that can be found in section 4.2.
In further development, these apps can be combined into one so that the recognized ingredients will automatically be added to the existing ingredient list.

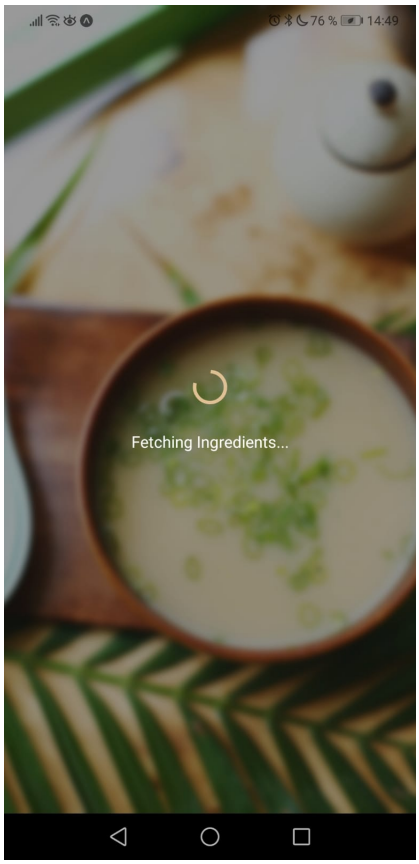
The screens shown in figure 23 can be repoduced by the following steps:

1. Start the app and wait until it loaded the ingredients (figures 23a and 23a)

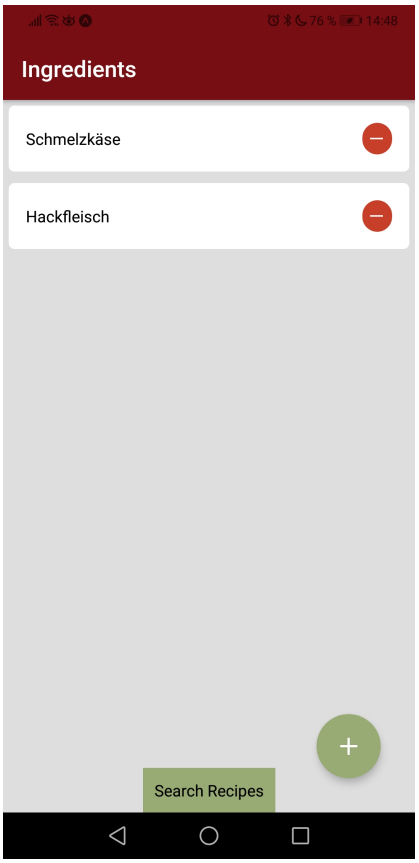
2. Click on the Plus-Icon (bottom right) in the ingredients screen and select "Search ingredient". Screen 23c will open.

3. Insert "Sch" and select "Hackfleisch" and "Schmelzkäse". You will be navigated back to screen 23b,

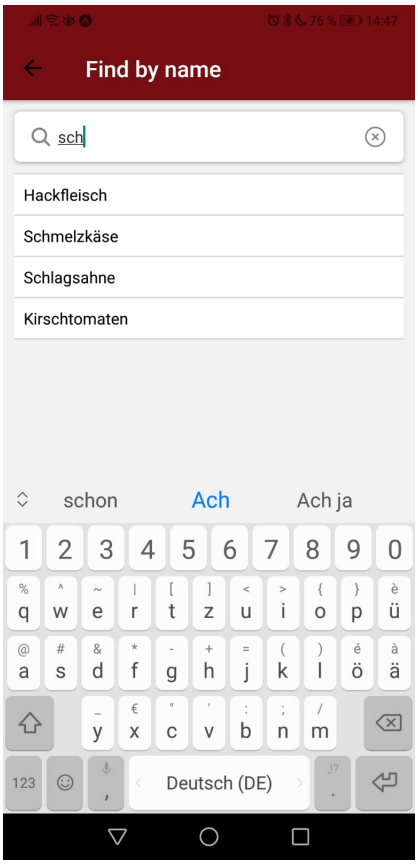
4. Click on "Search Recipes" (bottom) to reach screen 23d
5. Select "Käse-Lauch-Suppe". You will reach screen 23e



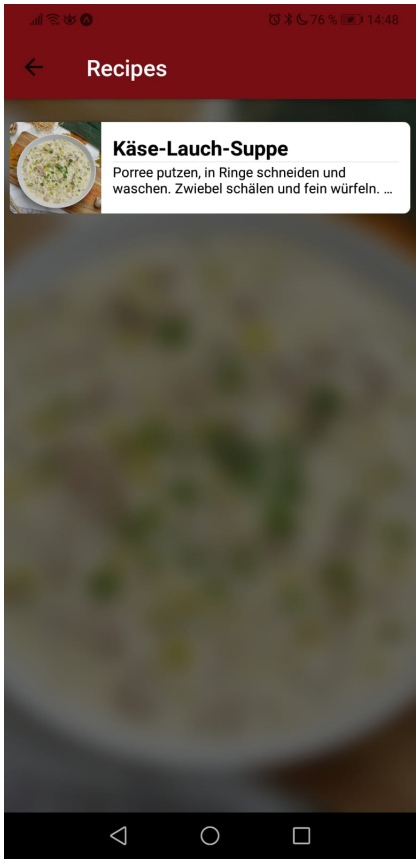

(a) Loading Screen



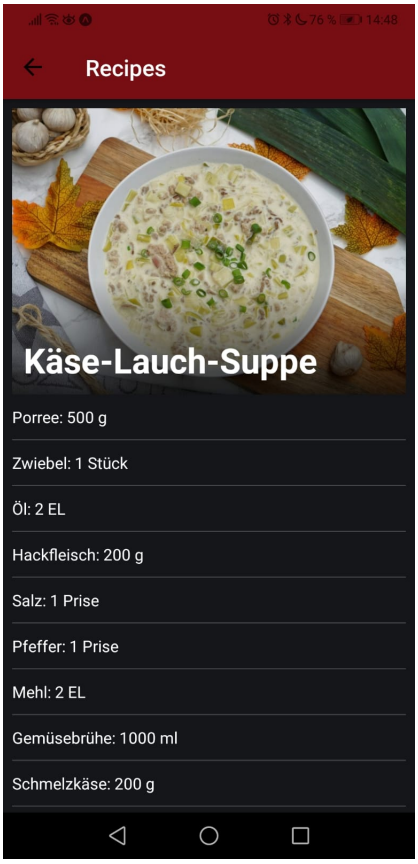

(b) Ingredients Screen. Two ingredients have been added




(c) Search Screen




(d) List of recipes matching the searched ingredients




(e) Recipe Screen



Figure 23: Implemented screens of the app

# 5 Evaluation and Validation

## 5.1 Model Result Evaluation

We have two models to compare, the pre-trained YOLOv5s and our retrained model. Further we have two sources of validation data. First we can use the validation data provided by COCO, second we will use our simulated data.

To compare those approaches we will mainly use the provided analysis tools of YOLOv5, namely `/yolov5/test.py`.

For each pair of data and model we need a specific `.yaml` file stored in `/yolov5/data/`. Therefore we have a total of 4 `.yaml` files: `food.yaml`, `food_coco.yaml`, `simulation.yaml` and `simulation_coco.yaml`.

To rerun our tests you can use our provided jupyter notebook `/yolov5/train_food.ipynb we used for the training`. The actual command to start the testing process looks like this: `!python test.py --weights <name>.pt --data <name>.yaml --img 640 --iou 0.65 weights` is the model that we want to test. To test the trained model you have to choose the correct folder `/yolov5/runs/train/exp/weights/best.pt`. The pre-trained model can be found here: `/yolov5/weights/yolov5s.pt`. Each training process creates its own separate output folder (exp, exp1 etc).

As mentioned in a previous section we had to think about how to fairly compare the two models with different output layers in our use-case. We decided to only use validation data containing food and only use the respective labels of the food classes. That means for the pre-trained model, every metric concerning non-food classes has to be ignored.

### 5.1.1 COCO Data

Our Validation set contains 708 images. Each of those pictures has at least one representative of the food category.

In figure 24 you can see the first batch of validation data with class labels. Figure 25 shows the comparison of predicted classes by the pre-trained model on the left and the retrained model on the right.

Is figure 26 there is a comparison of the F1-curve in which you can see the first batch of validation data with class labels. Figure 27 compares the confusion matrix of models. Naturally the matrix on the left of the pre-trained model shows all 80 classes even if the data actually only contains information of 10 classes. Therefore most entries of the matrix are empty. The matrix on right of the trained model is far more densely packed.

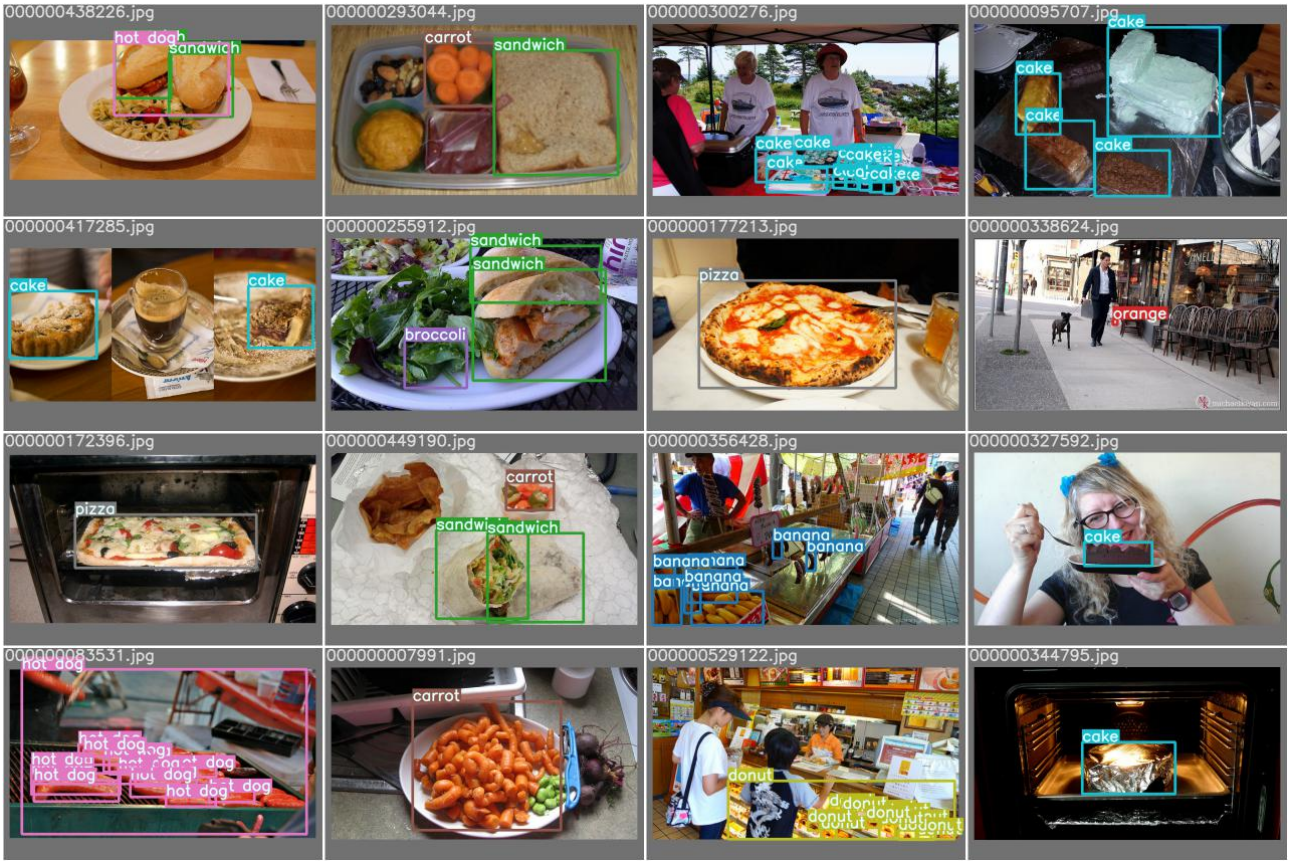
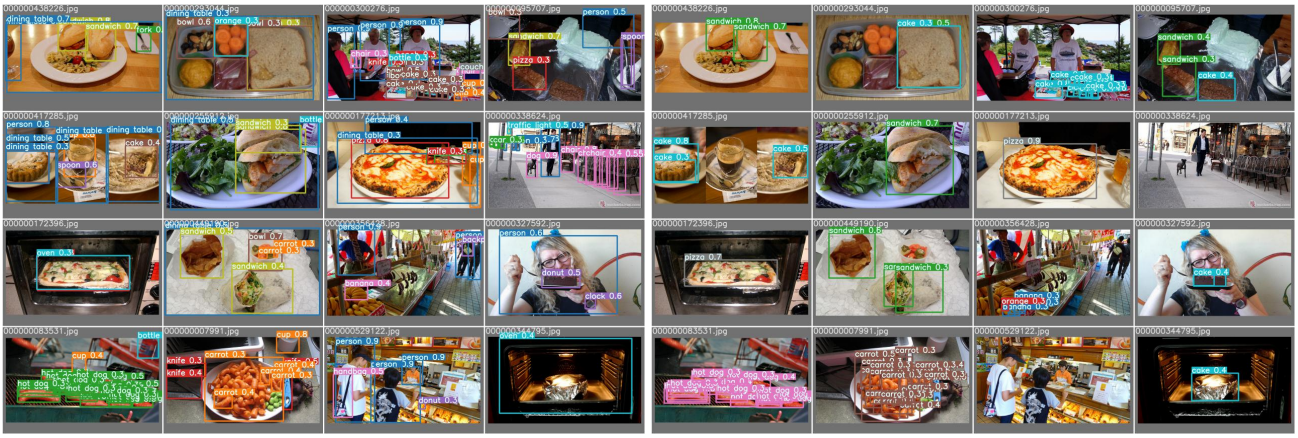
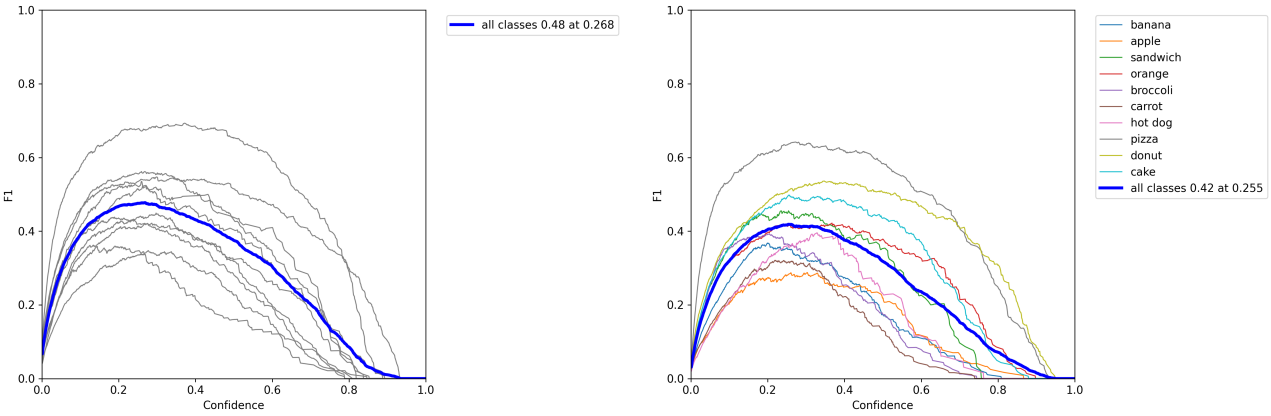



Figure 24: COCO: labeled data



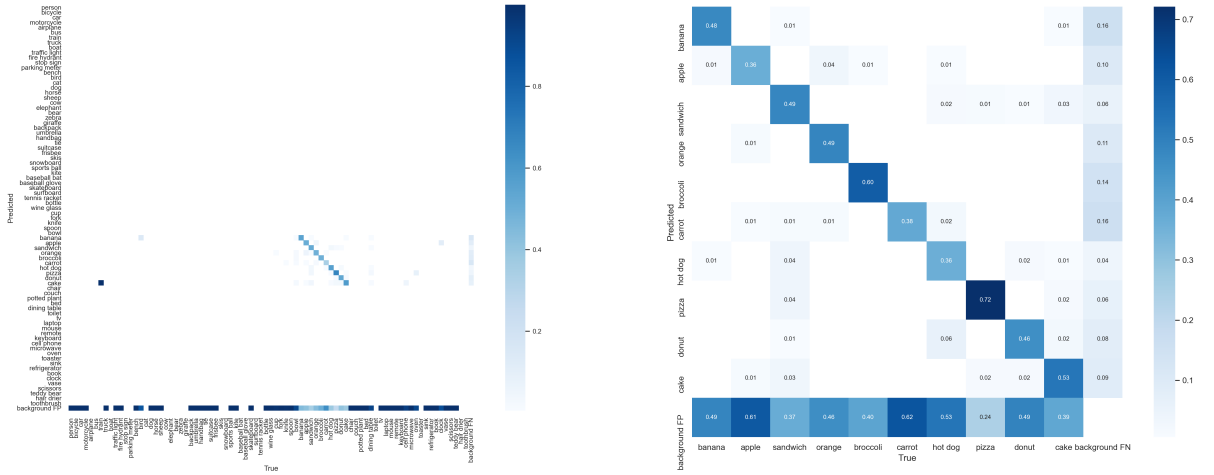
(a) predicted by pre-trained model (b) predicted by trained model

Figure 25: COCO: predicted data




(a) pre-trained model (b) trained model

Figure 26: COCO: F1 Curve




(a) pre-trained model (b) trained model

Figure 27: COCO: confusion matrix

### 5.1.2 Simulation Data

With the simulator described in section 4.1 we created a data set with 500 images and labels. In comparison with the COCO data-set each of these pictures only contains the interior of a refrigerator.

We provide the same diagrams for simulation data as for the COCO set. In figure 28 you can see again a batch of validation data with class labels. The next figure 29 shows the comparison of predicted classes between the two models.

Figure 30 shows the F1-Curve and figure 31 provides the confusion matrix for both models.

While the F1-score for both models using simulation-data is higher on average there is much more variance. We are confident to get far better results after training our models on specialized fridge data.

### 5.1.3 Manual comparison

In this section we did a small analysis of our model results with simulation data. Our goal is to provide a modified confusion matrix. Four aspects are need discussion. Classical true negatives do not provide information. After all every part of the image that is not falsely identified as ingredient is a true negative, meaning there are always arbitrarily many true positives. False positives we define as objects being predicted as a wrong class. False negatives are missing predictions of labeled objects.

Secondly we deem ingredients as correctly predicted if a cluster of objects is identified as one object and if the data provides a cluster of objects but the models identify multiple instances of the object. This is mainly important for carrots in our data.

In the analysis of the pre-trained model we ignore every predicted class not being an ingredient.

Lastly if an object is identified as two different classes, we only consider that with highest score.

You can find the validation batches used for this analysis 32 at `/model_validation/`.

Some thoughts about the models: The pre-trained model has problems to differentiate cake from a sandwich and oranges from peaches, resulting in a good portion of false positives. The model has problems detecting donuts missing almost every single one. Our simulator sometimes creates objects behind other objects making them hard to detect even for humans.

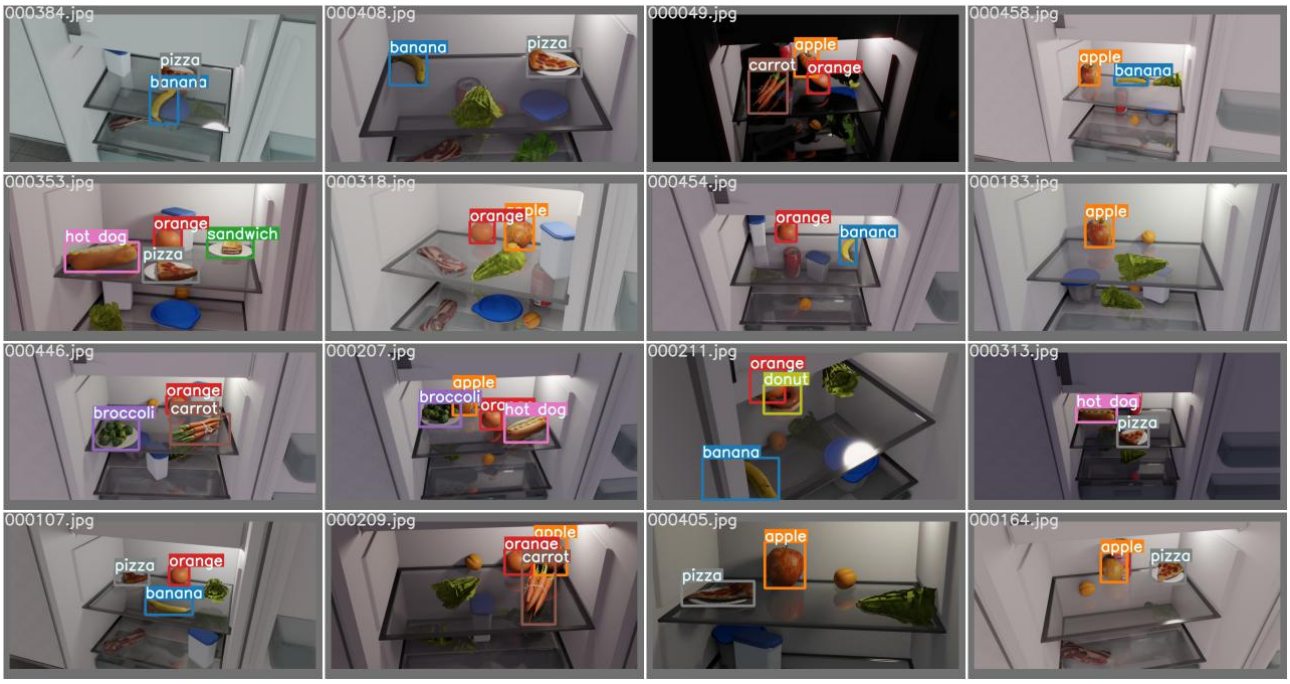
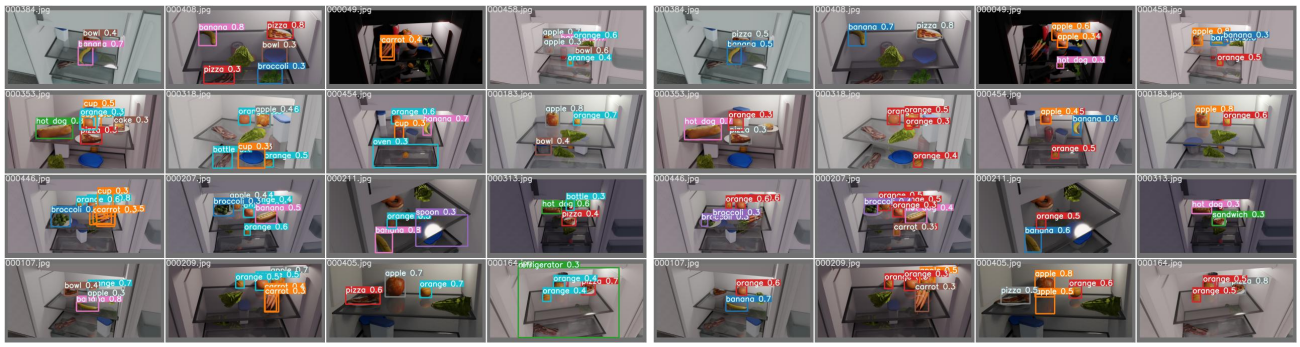
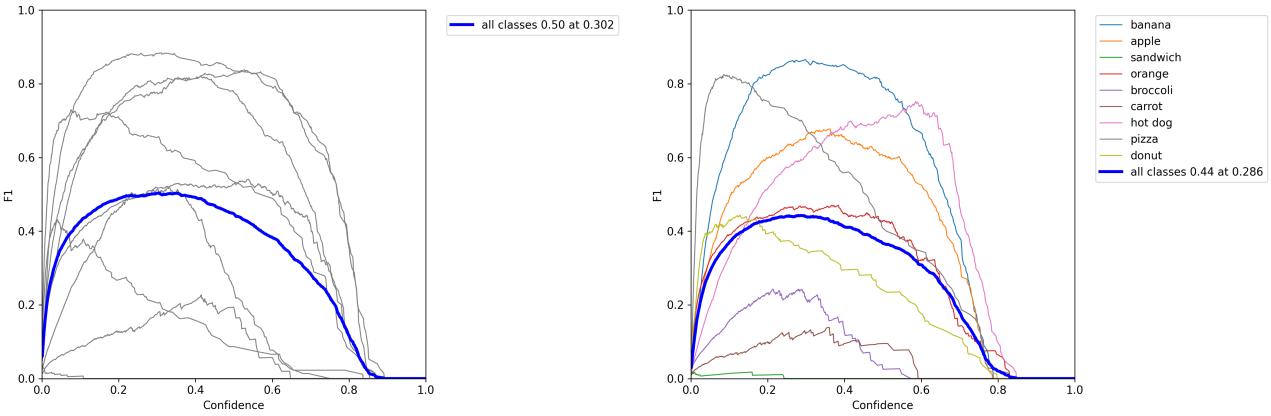



Figure 28: Simulation: labeled data



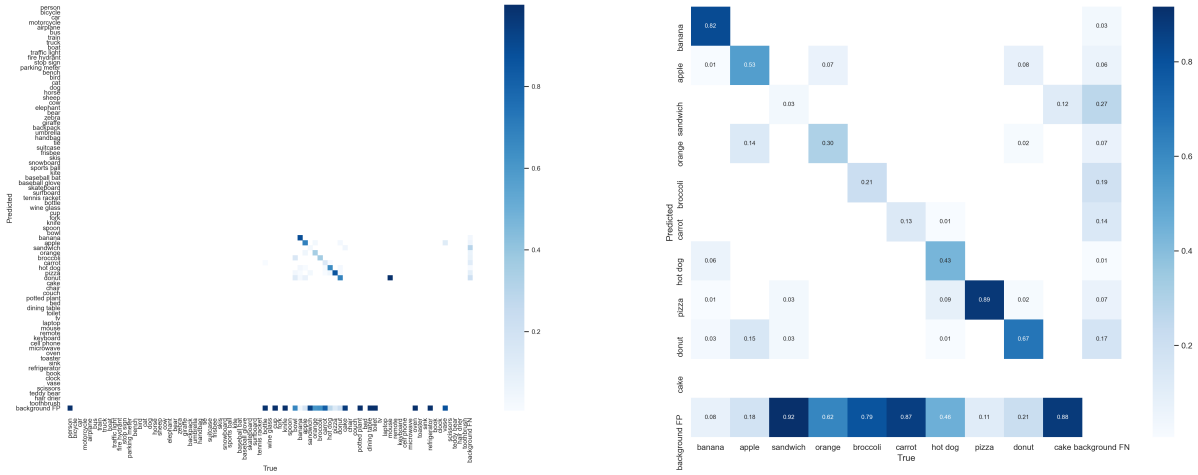
(a) predicted by pre-trained model (b) predicted by trained model

Figure 29: Simulation: predicted data




(a) pre-trained model (b) trained model

Figure 30: Simulation: F1 Curve




(a) pre-trained model (b) trained model

Figure 31: Simulation: confusion matrix

| | True | False | *ground truth* |
|---|---|---|---|
| True | 92 | 30 | |
| False | 37 | - | |
| *prediction* | | | |

(a) pre-trained model

| | True | False | *ground truth* |
|---|---|---|---|
| True | 78 | 20 | |
| False | 50 | - | |
| *prediction* | | | |

(b) trained model

Figure 32: Manual comparison on simulation data with total of 128 labeled objects

#### 5.1.4 Final comments

Our prediction should only help the user to faster identify ingredients. False or missing prediction are therefore not a huge problem. Most important for us are the true positive predictions. Thus, as you can see, both models have decent results but are by no means good enough for production. We think the data confirms that the pre-trained model has currently slightly better results. Therefore we are using it in the app. The plan is to replace it by a model trained on a huge amount of simulated data with large variety with a lot more epochs (at least 500 until the score does not seem in raise anymore).

### 5.2 Prototype

In this section we would like to compare the prototype we developed with the planned functionalities. Since the implementation is described in detail in section 4 we will focus mainly on the parts of the timeframe of the project.
In subsection 1.2 we explained that ingredients may be stored in different packages such as containers or simply the original supermarket packaging. Our prototype is able to detect the types of food given in the table in subsubsection 4.2.2. Since the machine learning model was originally designed to detect the raw food it is not possible for us to recognize these foods through packaging at this stage of the project. The option to scan barcodes is not implemented as well, as is the function to take videos of the fridge to analyse them for ingredients. Therefore the only way to get ingredients, aside from manually inserting them, is to take a picture of the fridge. In subsection 2.4 we talked about possibly using LiDAR to support image recognition via the camera function. Since our goal was to build a working prototype first this is also not implemented in the prototype because it is not a basic functionality, necessary for the prototype to work.
One more functionality we advertised in subsection 1.2 is the ability for the user to select a special cuisine to filter recipes. Since our prototype does not contain a big amount of recipes this function would not have been very effective nor needed.
The last thing we did not implement is the feedback function we talked about in Figure 2. This functionality is supposed to improve the quality of the system with the help of the user. Since the prototype is not delivered to any users and we did not have the infrastructure to implement and test this idea it could not be part of the prototype.
All the points above would be part of the product if it would go into production because we think they would improve the quality of the product a lot. For this reason these functionalities

are included and analyzed in the planning document. Summarizing we can present a prototype which consists of two apps with which we can take pictures created by our simulation and deliver recipes according to the detected ingredients to the user.

# References


[1] android-demo-app. https://github.com/pytorch/android-demo-app/tree/master/ObjectDetection. Date: 2021-06-04.

[2] COCO - common objects in context. https://cocodataset.org/#home. Date: 2021-06-04.

[3] Darknet: Open source neural networks in c. https://pjreddie.com/darknet/. Date: 2021-06-04.

[4] Emva standard. https://www.emva.org/wp-content/uploads/EMVA1288-3.0.pdf. Date: 2021-09-04.

[5] read noise. http://kmdouglass.github.io/posts/modeling-noise-for-image-simulations/. Date: 2021-09-04.

[6] wandb. https://wandb.ai. Date: 2021-06-04.

[7] Yolov5. https://github.com/ultralytics/yolov5. Date: 2021-06-04.