Bottle, Can or Coffee Cup ?!

Materials MADE SMARTER Centre

sustain
future steel manufacturing research hub

How Computer Vision and Machine Learning can be used to Recognise Different Materials to Make Recycling Easier

Discover Materials

03 Investigation of the Code

This Materials Made Smarter Outreach Demonstration of How Computer Vision and Machine Learning can be used to Recognise Different Materials to Make Recycling Easier has been developed by Dr Robert Gibbs with Professor Cinzia Giannetti of Swansea University [↲] for Materials Made Smarter [↲], based upon the NVIDIA DLI "Getting Started with AI on Jetson Nano" course [↲].

This guide looks in detail through the code that makes the project work. An accompanying walkthrough video is available at Discover Materials by scanning the QR code or at

https://discovermaterials.co.uk/resource/bottle-can-or-coffee-cup/

The video forms part of the section

03 Machine Learning and Neural Networks

A playlist of all 4 videos is at
https://www.youtube.com/playlist?list=PLyl3ubsSP6pUkBdTephBtqL7UfIFfGQ_Z

Also available on the Discover Materials website are a glossary of the highlighted technical terms, an electronic version of the printed booklet and further information about the code, the equipment and progressively more detailed project documentation.

Prifysgol Abertawe Swansea University
Y Gyfadran Gwyddoniaeth a Pheirianneg
Faculty of Science and Engineering
Materials and Manufacturing Research Institute

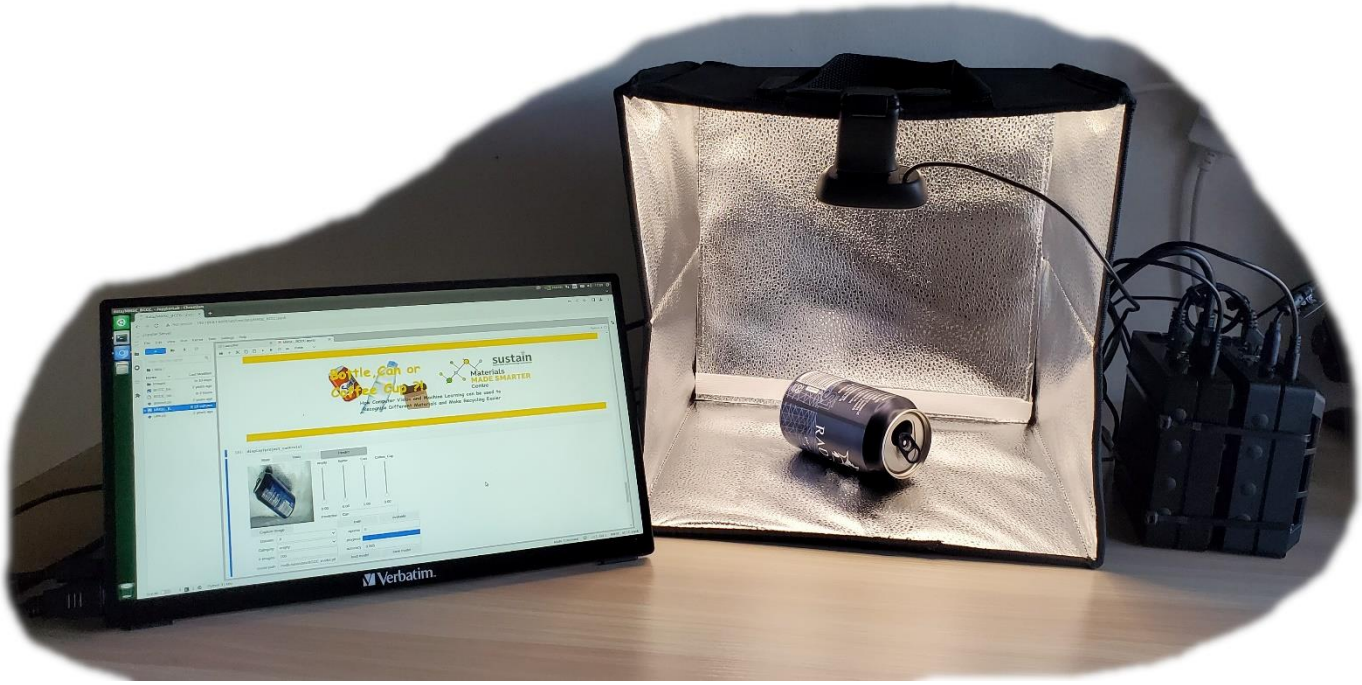
UKRI Engineering and Physical Sciences Research Council

NVIDIA DEEP LEARNING INSTITUTE

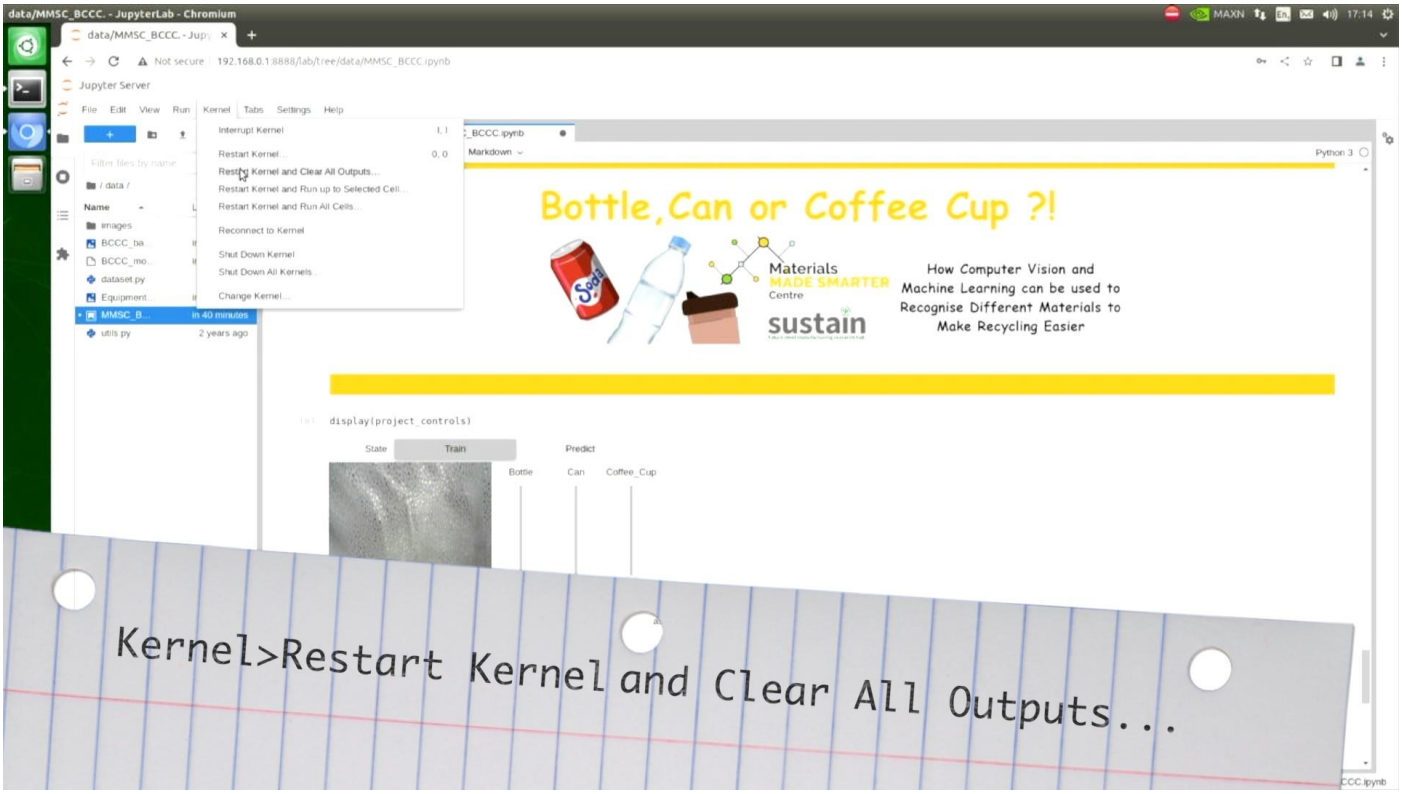
developed by Dr R. Gibbs and Prof. C. Giannetti for Materials Made Smarter,
based upon the NVIDIA DLI
"Getting Started with AI on Jetson Nano" course.
C.G. would like to acknowledge the support of the EPSRC (EP/V061798/1).

visit DISCOVER MATERIALS
https://discovermaterials.co.uk
and learn more about what's happening in the world of materials science!

It is assumed that you have the system up and running according to the guide **01 Getting the System Up and Running** available on the website.



In this guide we're going to look through the code that makes this system work.



First go to **Kernel>Restart Kernel and Clear All Outputs…** to stop the code from running and click on **Restart** when it appears.

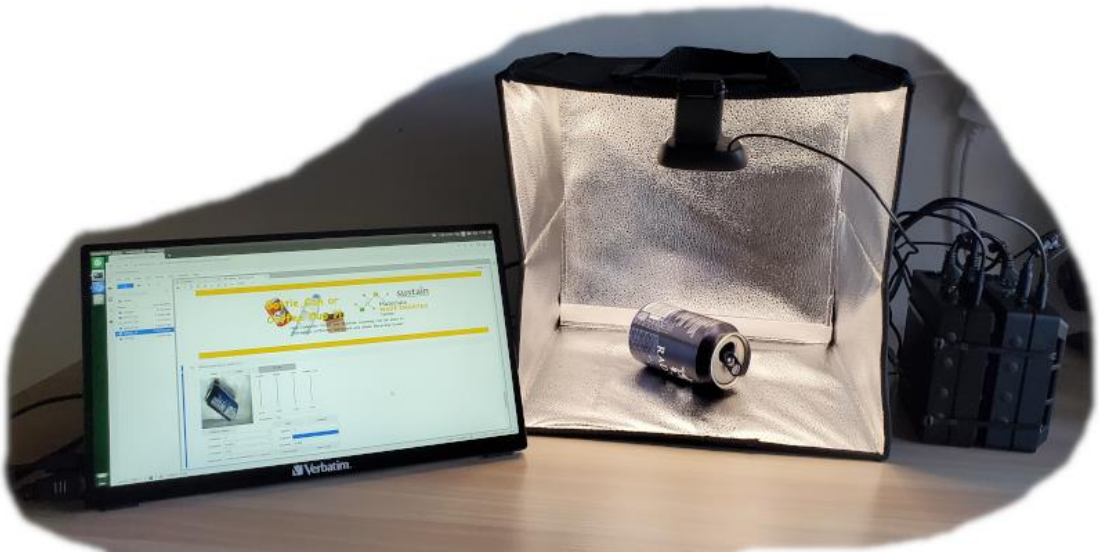
We'll run each of the cells individually as we step through the code. You can execute each cell individually and move onto the next cell by pressing shift-return.

Bottle, Can or Coffee Cup ?!

Machine-Learning Computer-Vision Recognition Outreach Project for the Materials Made Smarter Centre

This project has been developed by the Materials Made Smarter Centre at Swansea University in collaboration with the Sustain Manufacturing Research Hub and Discover Materials to demonstrate how Computer Vision and Machine Learning can be used to recognise different objects to help with the sorting of materials for recycling.

Further information and documentation about this project can be found at
https://discovermaterials.co.uk/resource/bottle-can-or-coffee-cup/



The Project Equipment, a portable monitor and two combined Seeed reComputer J1010 units hosting the NVIDIA Jetson Nano Jetpak project code

The platform this project is built on is the Seeed Studio reComputer J1010 NVIDIA Jetson Nano 2GB Platform with the Arm Cortex A57 CPU and NVIDIA Maxwell GPU and it has been developed by Dr R. Gibbs and Prof. C. Giannetti based upon the NVIDIA DLI "Getting Started with AI on Jetson Nano" course which can be found in the ../classification directory

Professor C. Giannetti would like to acknowledge the support of the EPSRC (EP/V061798/1) in this Materials Made Smarter Project.

Launch Camera

This cell opens access to the Logitech C270 webcam attached to the USB3 port on reComputer 1.
The jetcam library is contained within the nvidia_dli_docker environment.

In [ ]:

```
## Launch Camera ##############################################################################

# Check device number !ls -ltrh
/dev/video* from jetcam.usb_camera
import USBCamera

# Logitech C270 webcam
camera = USBCamera(width=224, height=224, capture_device=0) # confirm the capture_device number

camera.running = True
print("camera started")

#############################################################################################
```

The first cell of the code launches the jetcam.usb_camera library to support the Logitec C270 camera that this code is designed to work with.

Define Machine-Learning Task

The task will be a Classification rather than a Regression Machine-Learning task. Classifying what the camera sees as either a bottle, a can or a coffee cup. There are three Datasets A, B or C. Dataset A contains 50 images each of the example props for the project. Datasets B and C can be used for other examples of training for new images. The images captured for training are stored in, for example, ./images/BCCC_A/Bottle/ etc.

In [ ]:

```
## Define Machine Learning Task ###############################################################

import torchvision.transforms as transforms from
dataset import ImageClassificationDataset

TASK = 'BCCC'

CATEGORIES = ['Bottle', 'Can', 'Coffee_Cup']

DATASETS = ['A', 'B', 'C']

TRANSFORMS = transforms.Compose([
    transforms.ColorJitter(0.2, 0.2, 0.2, 0.2),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

datasets = {}
for name in DATASETS:
    datasets[name] = ImageClassificationDataset('../data/images/'
                     + TASK + '_' + name, CATEGORIES, TRANSFORMS)

print("{} task with {} categories defined".format(TASK, CATEGORIES))

# Set up the data directory location if not there already
DATA_DIR = '/nvdli-nano/data/images/'
!mkdir -p {DATA_DIR}

#############################################################################################
```

We define the machine learning task with categories, bottle, can and coffee_cup, and three possible data sets; A, B and C, so there's lots of flexibility with adding new data to the system.

The transforms convert the image from the camera into the 224 by 224 array that the machine learning model needs as its inputs.

The dataset for each of the categories, and the images directory (which are already present on the system) are created.

Create Widgets used for Data Collection

Creates the widgets that display the dataset and categories being collected, the image from the camera and the capture button which stores an image

In [ ]:

```
## Create Data Collection Widget ###############################################################

import ipywidgets
import traitlets
from IPython.display import displayfrom
from jetcam.utils import bgr8_to_jpeg

# initialize active dataset
dataset = datasets[DATASETS[0]]

# unobserve all callbacks from camera in case we are running this cell for second time
camera.unobserve_all()

# create image from camera
camera_widget = ipywidgets.Image()
traitlets.dlink((camera, 'value'),(camera_widget, 'value'), transform=bgr8_to_jpeg)

# create widgets
dataset_widget = ipywidgets.Dropdown(options=DATASETS, description='Dataset:')
category_widget = ipywidgets.Dropdown(options=dataset.categories, description='Category:')
count_widget = ipywidgets.IntText(description='# Images:')
capture_widget = ipywidgets.Button(description='Capture Image')

# update exisitng count of images at initialization
count_widget.value = dataset.get_count(category_widget.value)

# sets the active dataset
def set_dataset(change):
    global dataset
    dataset = datasets[change['new']]
    count_widget.value = dataset.get_count(category_widget.value)
dataset_widget.observe(set_dataset, names='value')

# update counts when we select a new category def
update_counts(change):
    count_widget.value = dataset.get_count(change['new'])
category_widget.observe(update_counts, names='value')

# save image for category and update counts
def save(c):
    dataset.save_entry(camera.value, category_widget.value)
    count_widget.value = dataset.get_count(category_widget.value)
capture_widget.on_click(save)

data_collection_widget = ipywidgets.VBox([capture_widget,
                                          dataset_widget,
                                          category_widget,
                                          count_widget])

# output
print("camera_widget, data_collection_widget created")

#############################################################################################
```

The next part of the code creates widgets for use in the data collection.

We need to import the widgets library, the traitlets library and the display and image translation libraries.

We use the data set that's selected.

We check that the camera is only being run once so there are no conflicts. And then we create a camera widget that brings the camera input in from the library and transforms it into a jpeg image for display on the screen. This works in real-time and displays the real-time feed from the camera.

We create a drop down menu of all the data sets available (A, B, C) and a drop down menu of all the categories available (Bottle, Can or Coffee_Cup).

A text object that displays the count of the number of images present in each of the folders and a button that tells the system to capture a new image and store it in the relevant folder are defined.

The count widget counts the number of images in each folder is updated whenever the dataset and category menus are changed. The number is also updated whenever there is a change to the count of images in the folder whenever the capture button is pressed,

When the capture button is pressed the jpeg version of the image from the camera is stored it in the relevant folder of the correct category in the correct data set.

All the elements are grouped together for display using ipywidgets vertical and horizontal boxes.

Load the pre-trained RESNET 18 Neural Network Model

There are several large Neural Network models which have been pre-trained on millions of general images to develop general understanding of what the camera is looking at. Through transfer learning, these pretrained models are used as the starting point from which new training is performed to specialise the recognition to a few specific categories. This permits successful models to be built using fewer training images than would be required if trying to learn a situation from scratch. It takes a short time to load in the model that is being used. Three other models are also available for experiementation. The models are stored in the nvidia_dli_docker environment.

In [ ]:

```
############################################################################################

import torch
import torchvision



device = torch.device('cuda') # this makes use of the Graphical Processing Unit built
                              into the Jetson Nano.

# RESNET 18
model = torchvision.models.resnet18(pretrained=True)
model.fc = torch.nn.Linear(512, len(dataset.categories))

# ALEXNET
# model = torchvision.models.alexnet(pretrained=True)
# model.classifier[-1] = torch.nn.Linear(4096, len(dataset.categories))

# SQUEEZENET
# model = torchvision.models.squeezenet1_1(pretrained=True)
# model.classifier[1] = torch.nn.Conv2d(512, len(dataset.categories), kernel_size=1)
# model.num_classes = len(dataset.categories)

# RESNET 34
# model = torchvision.models.resnet34(pretrained=True)
# model.fc = torch.nn.Linear(512, len(dataset.categories))

model = model.to(device)

model_save_button = ipywidgets.Button(description='save model')
model_load_button = ipywidgets.Button(description='load model')
model_path_widget = ipywidgets.Text(description='model path',
                                    value='/nvdli-nano/data/BCCC_model.pth')

def load_model(c):
    model.load_state_dict(torch.load(model_path_widget.value))
model_load_button.on_click(load_model)

def save_model(c):
    torch.save(model.state_dict(), model_path_widget.value)
model_save_button.on_click(save_model)

model_widget = ipywidgets.HBox([model_path_widget,model_load_button,model_save_button])

# output
print("model_widget created")

############################################################################################
```

Here we load the pretrained ResNet 18 model. This is a model that has been trained on millions of different objects to be able to pull out the features for a number of different objects, and we are using this as a transfer of learning so that we don't need to learn from scratch what an object looks like.

There are other models that you can investigate if you remark out, ResNet 18 and replace with the defining statements for Alexnet or Squeezenet or ResNet 34, which is an even bigger 34 layered model

We start with ResNet 18 because it's the smallest of the four models and runs best in the limited memory on the Jetson Nano 2 GB system. But you can experiment yourself if you wish.

We create load, save buttons to pull in a new final layer for the model, if we have one saved. With the mechanism behind those buttons for each of the different models.

This cell takes a while to execute because ResNet is a large model to load into memory, but eventually the cell will finish executing.

Create State and Prediction Widgets

Switch between Training and Prediction states and launch the live real-time state. In the real-time state the system displays a live view of what the camera sees, but waits until the 'recognise' button is pressed before performing a prediction/inference operation with the trained model.

In [ ]:

```
########################################################################################################

import threading
import time
from utils import preprocess
import torch.nn.functional as F

state_widget = ipywidgets.ToggleButtons(options=['Train', 'Predict'], description='State',
                                        value='Train')
prediction_widget = ipywidgets.Text(description='as') # this widget is placed after
                                                      the 'Recognise' widget so that it
                                                      reads Recognise as …
score_widgets = []
for category in dataset.categories:
    score_widget = ipywidgets.FloatSlider(min=0.0, max=1.0, description=category,
                                          orientation='vertical')
    score_widgets.append(score_widget)

recognise_widget = ipywidgets.Button(description='Recognise')

# run prediction on current image
def recognise(c):
    image = camera.value
    preprocessed = preprocess(image)
    output = model(preprocessed)
    output = F.softmax(output, dim=1).detach().cpu().numpy().flatten()
    category_index = output.argmax()
    prediction_widget.value = dataset.categories[category_index]
    for i, score in enumerate(list(output)):
        score_widgets[i].value = score

# during the live state the camera feed thread is run in real-time but and the 'Recognise'
# button is polled  inference is only performed by the recognise function when the button
# is pressed
def live(state_widget, model, camera, prediction_widget, score_widget):
    global dataset
    while state_widget.value == 'Predict':
        recognise_widget.on_click(recognise)

def start_live(change):
    if change['new'] == 'Predict':
        execute_thread = threading.Thread(target=live, args=(state_widget, model, camera,
                                          prediction_widget, score_widget))
        execute_thread.start()
state_widget.observe(start_live, names='value')
predict_widget = ipywidgets.VBox([ipywidgets.HBox(score_widgets),
                                  ipywidgets.HBox([recognise_widget,prediction_widget])])

# outputs
print("state_widget and predict_widget created")

########################################################################################################
```

Now we're going to create the real-time, live-threading, interface that changes between the training state and the prediction state and controls the change between them.

Rather than running in real-time, constantly trying to recognise what the camera sees, we have defined a **Recognise** button, so it only performs a prediction if the **Recognise** button is pressed. This prevents overloading the limited memory of the Jetson Nano 2GB system with uncontrolled data collection.

If the **Recognise** button is pressed, then the image is taken from the camera.

It is processed by the transformations to make it the right size and format for the model to understand. Then the output is the prediction of the model, which calls model and we give it the processed image as an input with this single line

```
output = model(preprocessed)
```

It produces three probabilities of how confident the model is that the image is of a can, a bottle or a coffee cup, providing percentage values of the likelihood.

Those 3 percentage values are then mapped through a softmax function, which effectively takes the most likely of those three values and returns the most probable.

The category index then becomes the most likely of those three categories, the one it has greatest confidence in and the prediction which it then displays the name of the category that matches the category index.

This live state will provide the live feed of the camera constantly and wait for the **Recognise** button to be pressed before making a prediction.

The start live groups together the real time threading of the model and the camera, the prediction widget and the score widgets.

The predict widget then is a grouping of the score widgets, which are the three probabilities, and the recognize and prediction widgets. It organizes the arrangement of this part of the final display.

Training and Evaluation

Define the training widgets, after new data is collected the final layer of the model must be retrained to learn the new images. This cell may take several seconds to execute. The default number of epochs for training the model is 10. The first epoch takes a while to begin as the images are loaded into memmory, but then training proceeds relatively quickly, counting epochs down. Once training is complete the system automatically switches to the 'Predict' state and waits for the 'recognise' button to be pressed.

In [ ]:

```
################################################################################################

BATCH_SIZE = 8

optimizer = torch.optim.Adam(model.parameters())
# optimizer = torch.optim.SGD(model.parameters(), lr=1e-3, momentum=0.9)

epochs_widget = ipywidgets.IntText(description='epochs', value=10)
eval_button = ipywidgets.Button(description='evaluate')
train_button = ipywidgets.Button(description='train')
loss_widget = ipywidgets.FloatText(description='loss')
accuracy_widget = ipywidgets.FloatText(description='accuracy')
progress_widget = ipywidgets.FloatProgress(min=0.0, max=1.0, description='progress')

def train_eval(is_training):
    global BATCH_SIZE, LEARNING_RATE, MOMENTUM, model, dataset, optimizer,
           eval_button, train_button, accuracy_widget, loss_widge

try:
        train_loader = torch.utils.data.DataLoader(
            dataset,
            batch_size=BATCH_SIZE,
            shuffle=True
        )

        state_widget.value = 'Train'
        train_button.disabled = True
        eval_button.disabled = True
        time.sleep(1)

        if is_training:
            model = model.train()
        else:
            model = model.eval()
        while epochs_widget.value > 0:
            i = 0
            sum_loss = 0.0
            error_count = 0.0
            for images, labels in iter(train_loader):
                # send data to device
                images = images.to(device)
                labels = labels.to(device)

                if is_training:
                    # zero gradients of parameters
                    optimizer.zero_grad()

                # execute model to get outputs
                outputs = model(images)

                # compute loss
                loss = F.cross_entropy(outputs, labels)
```

```
                if is_training:
                    # run backpropogation to accumulate gradients
                    loss.backward()

                    # step optimizer to adjust parameters
                    optimizer.step()

                # increment progress
                error_count += len(torch.nonzero(outputs.argmax(1) - labels).flatten())
                count = len(labels.flatten())
                i += count
                sum_loss += float(loss)
                progress_widget.value = i / len(dataset)
                loss_widget.value = sum_loss / i
                accuracy_widget.value = 1.0 - error_count / i

            if is_training:
                epochs_widget.value = epochs_widget.value - 1
            else:
                break
    except e:
        pass
    model = model.eval()

    train_button.disabled = False
    eval_button.disabled = False
    state_widget.value = 'Predict'

train_button.on_click(lambda c: train_eval(is_training=True))
eval_button.on_click(lambda c: train_eval(is_training=False))

training_widget = ipywidgets.VBox([ipywidgets.HBox([train_button,eval_button]),
                                                    epochs_widget,
                                                    progress_widget,
                                                    accuracy_widget])

# loss_widget not included

# output
print("training configured and training_widget created")

#########################################################################################
```

Training and evaluation is the process where the model is trained on all of the data stored in the image's folder or evaluated when a saved model is loaded into the final layer.

The number of epochs, is defaulted to 10, but can be changed. The evaluation button is used when loading a model in. The train button is used when training a model from scratch. The loss widget accuracy widget and progress widget are all values that can be interrogated to understand how well the training performed.

In this particular demonstration we only display the accuracy and the progress widget, the loss widget isn't displayed, but you can choose to display it if you want to.

The training function contains the mechanics of passing through each image in the training dataset folders and determining the loss function = the measure of the error in predicting that image compared to the actual classification of image. The back propagation process then in improves the weights and biases of the final layer of the resulting model to predict for the three categories defined.

The widget are again then grouped to make the arrangement on the final display.

Define Project Controls

Groups and arranges all the created widgets into the display for the project

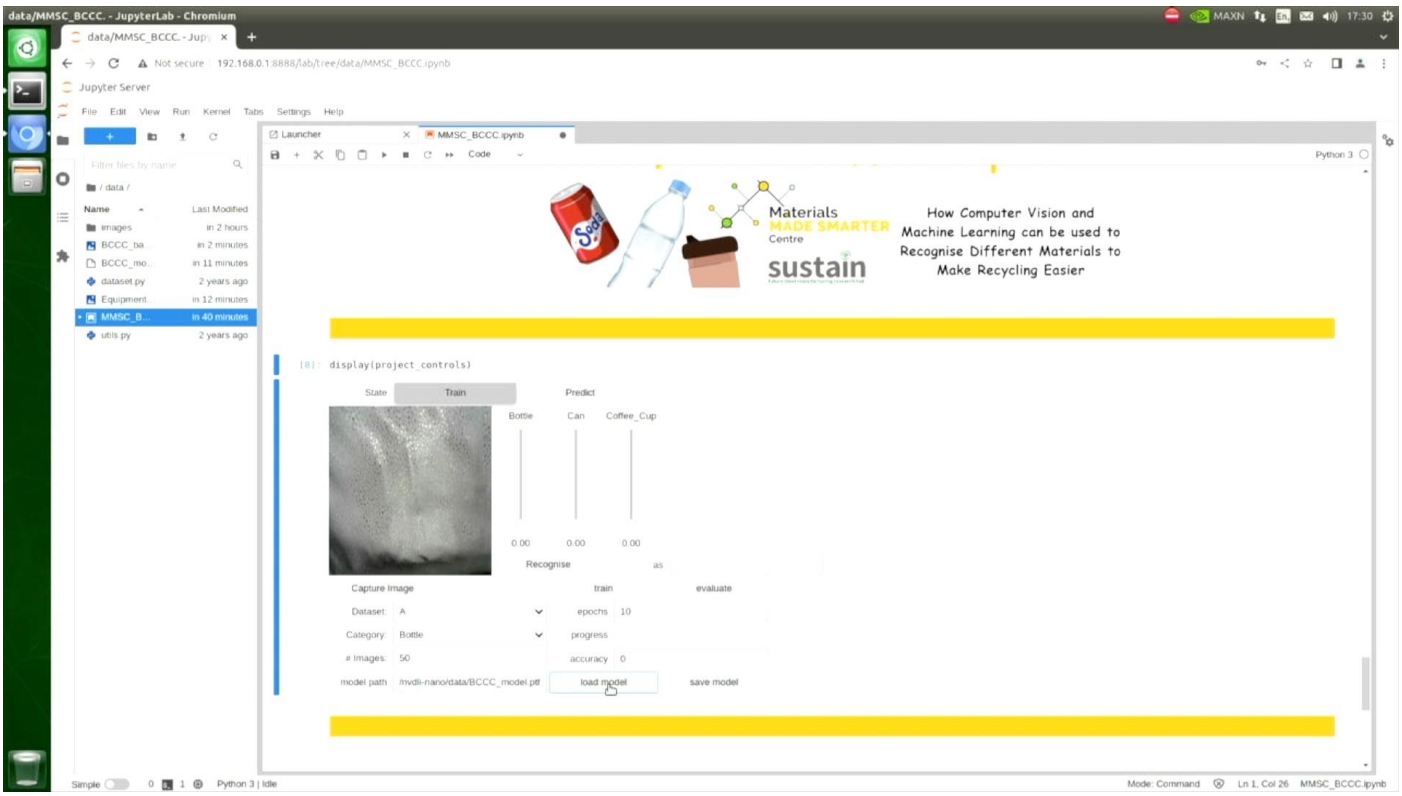
In [ ]:
```
## define project controls ###########################################################################
constructed from:
# camera_widget, data_collection_widget
# modelsave_widget
# state_widget, predict_widget and score_widgets
# training_widget
project_controls = ipywidgets.VBox([state_widget,
                                    ipywidgets.HBox([camera_widget,predict_widget]),
                                    ipywidgets.HBox([data_collection_widget,training_widget]),
                                    model_widget])
```

Finally, we group all of our individual widget sub-groups widgets together into one collection called the project controls.

Launch the Project Controls

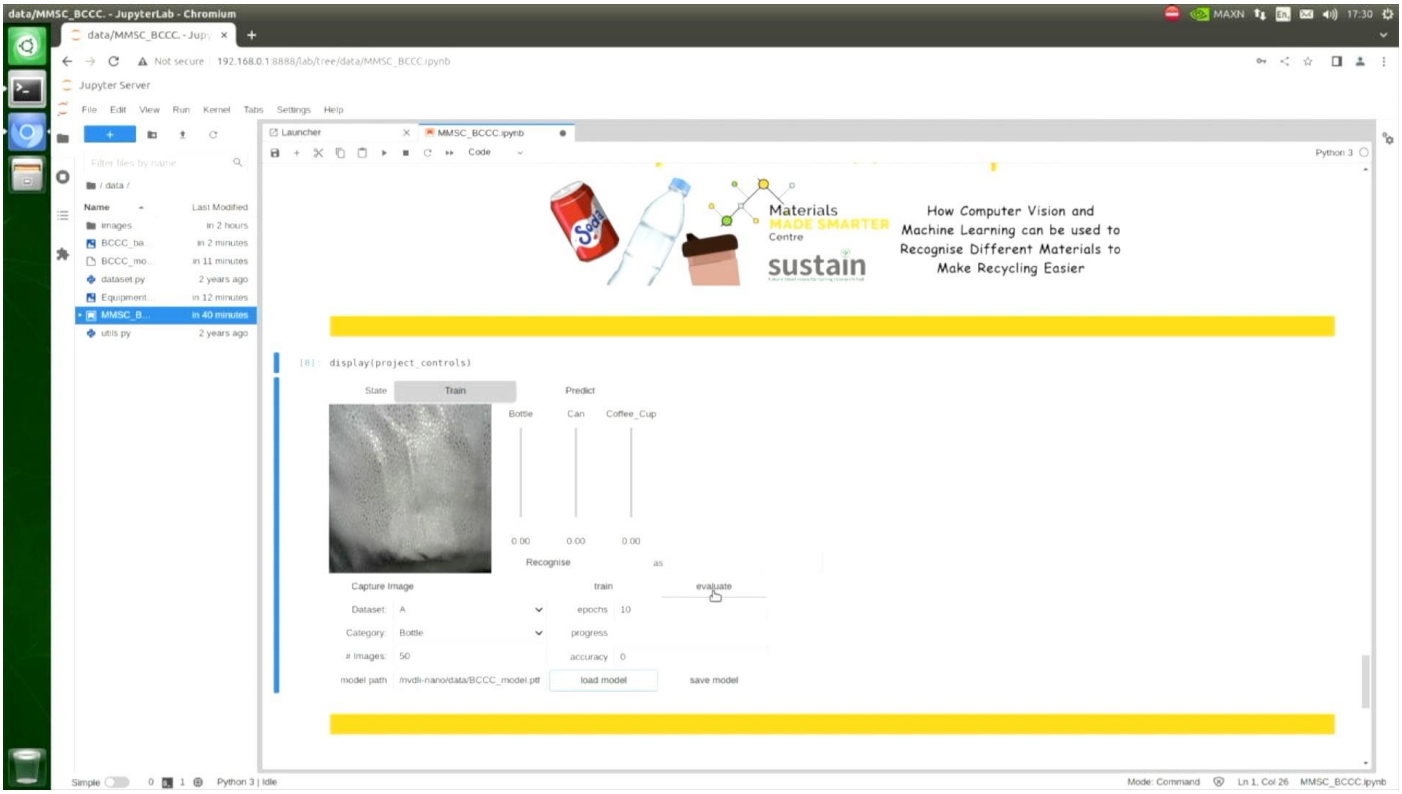
In [ ]:
```
display(project_controls)
```

Finally, we can display all the project controls and we can see all of those elements that were discussed arranged within the project controls interface with the live camera feed, the data sets, bottles, number of images in each half the categories.

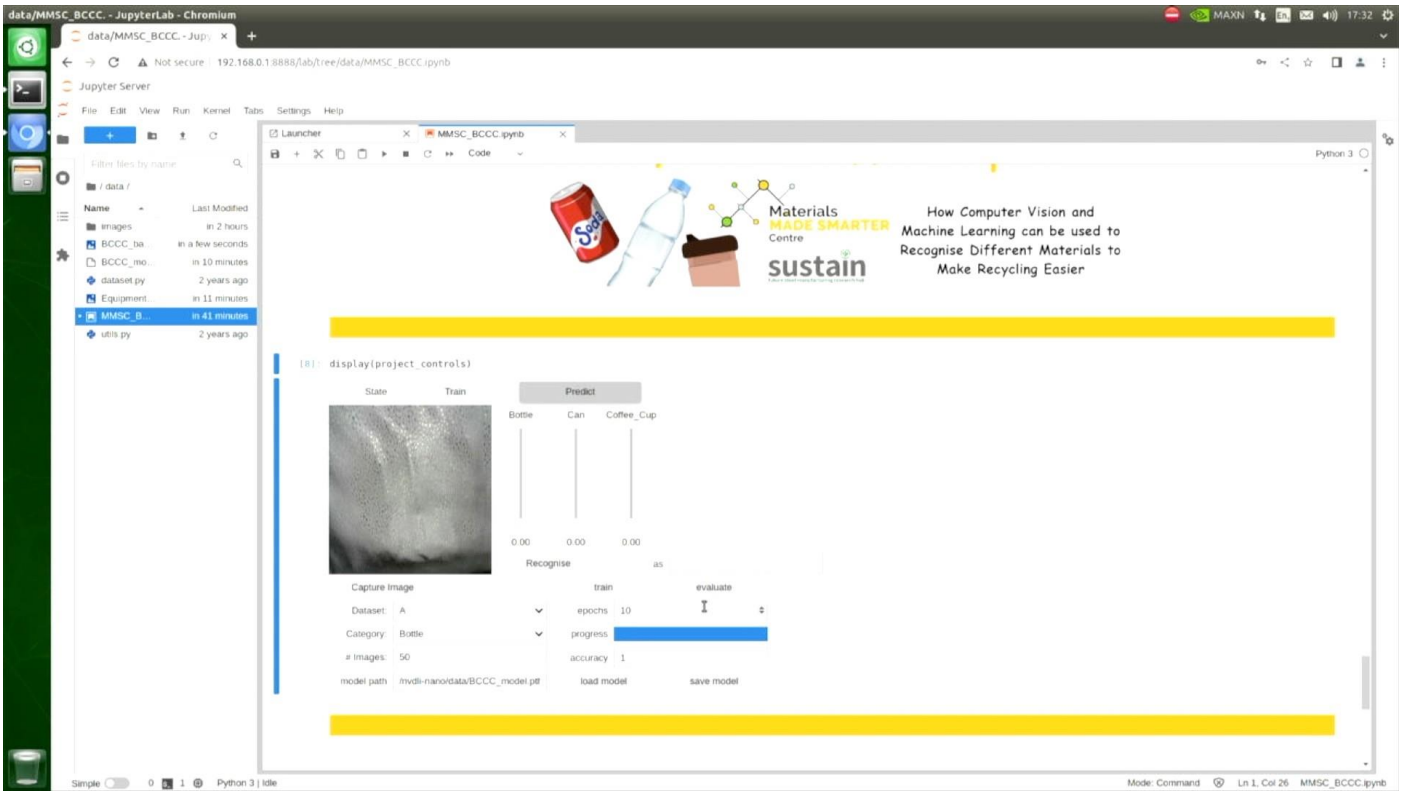


Pressing the train button which executes the training procedure as described in the guide **02 How to Use the Project Controls to Identify Objects** available on the website.

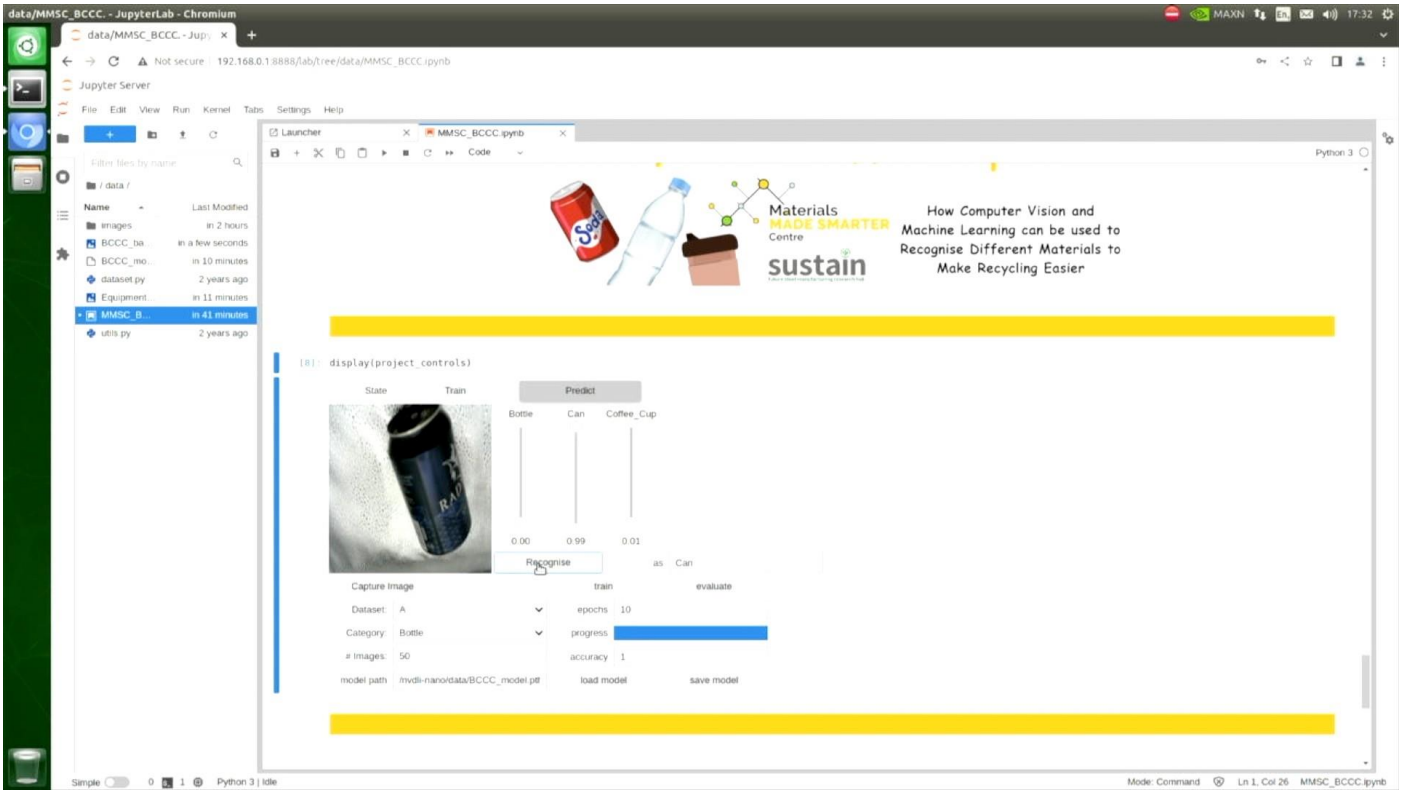
On this occasion we already have the model saved so we can load that in by pressing **load**.



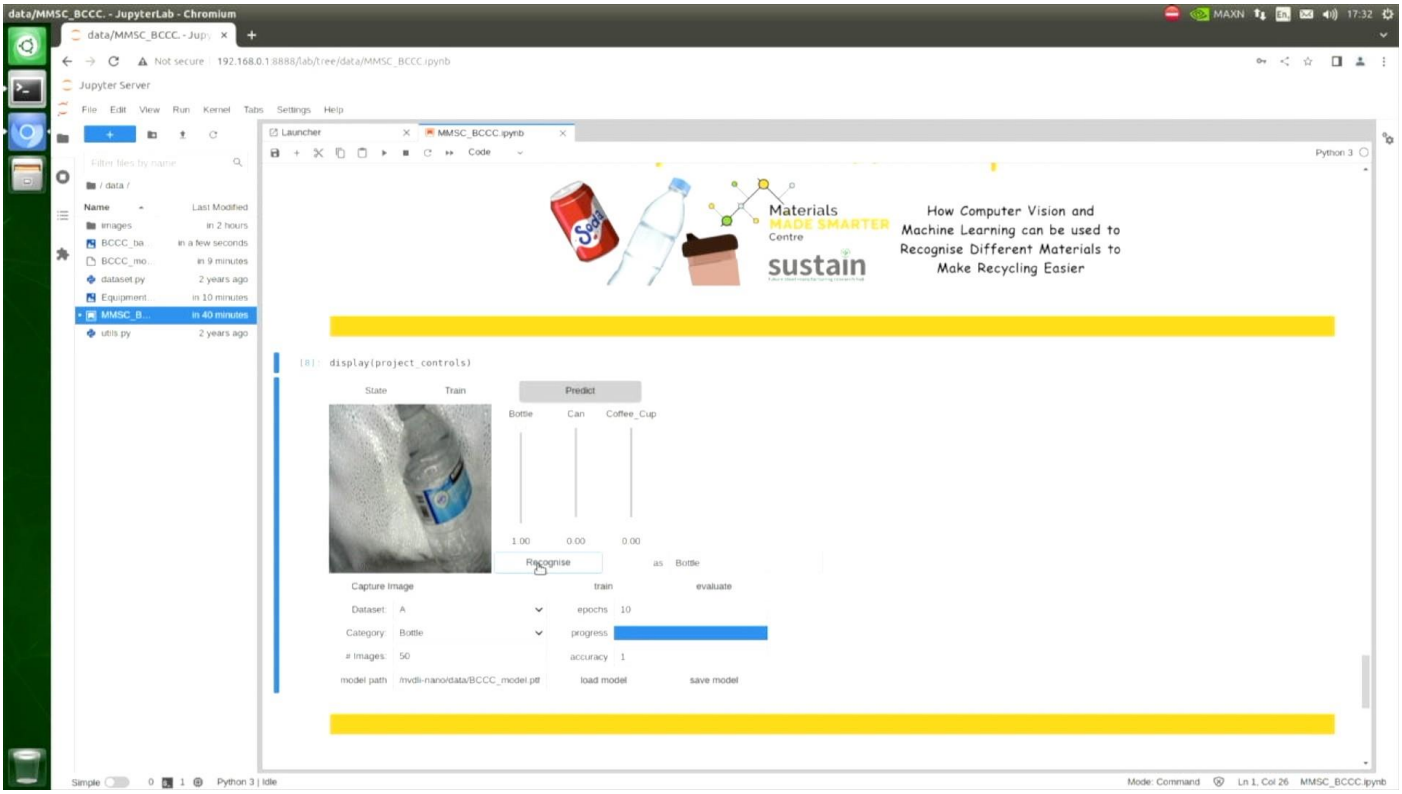
Before the model can be used to predict anything, we have to load it in as the final layer of the ResNet model, so we press the **evaluate** button. This process takes approximately a minute but is faster than training from scratch.



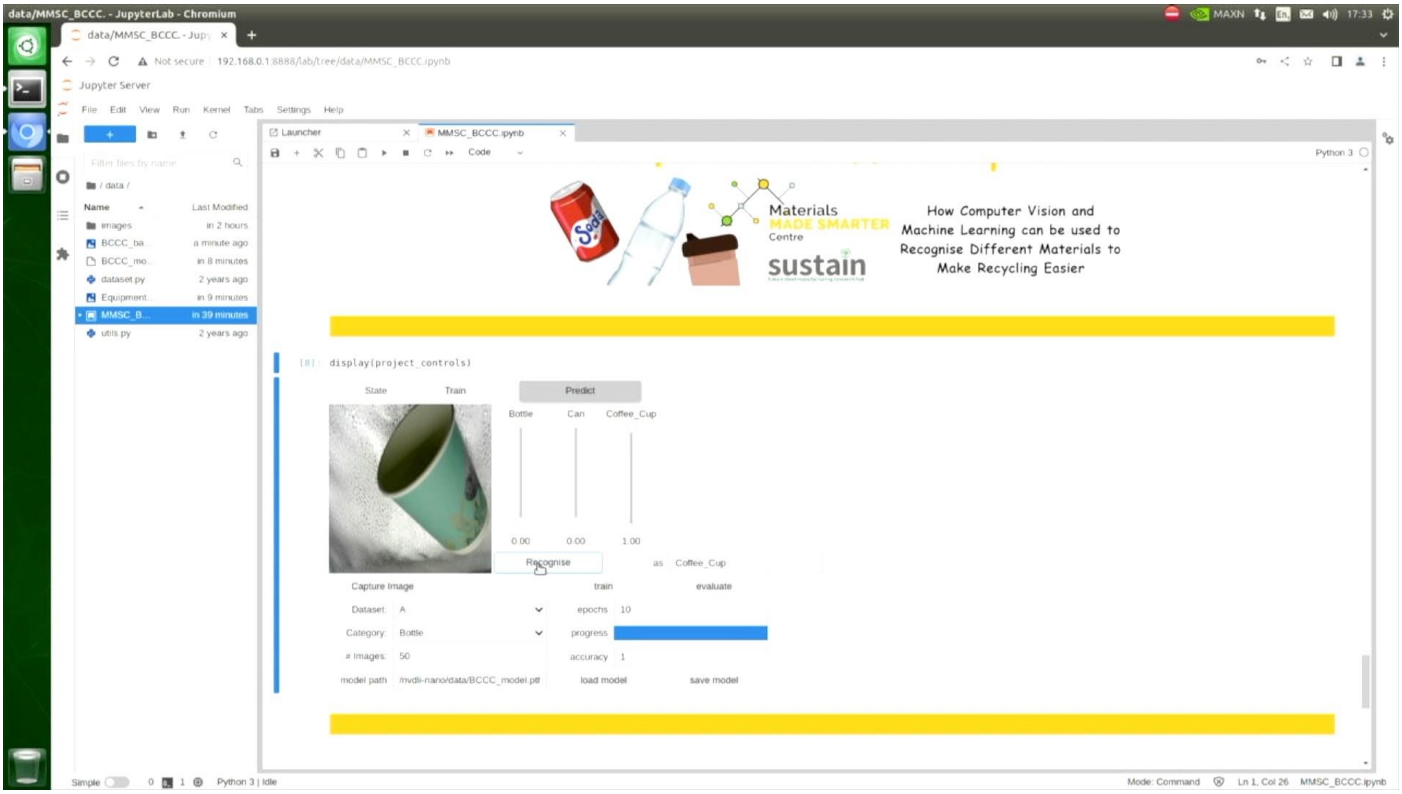
Once the model has loaded into the final layer the system switches to the predict mode.



We can put in an object to be recognized, and we press the **Recognise** button, it will create a prediction of that object, the can.



the Bottle



and the Coffee Cup

Loading a saved model is a lot faster than training from scratch, as in guide **02 How to Use the Project Controls to Identify Objects** available on the website.

In the guide **04 Improving the Performance for New Objects** (available on the website) we'll investigate how you can improve the performance on recognising objects it has never seen before.

In the meantime have fun with the system on the three objects it's been trained to recognise.

See you there.