API notebook

This file is the readable version of the code that will be put into the api and possible seperate files as it grows. Everything below was coded for the purpose of allowing us to send an API a base64 image and use a series of ML algorthms to detect and manipulate objects within the image. Currently for this submission I am focusing on a cup object on the table although the model could be trained with other objects.

This file goes along with a h5 file which is the model used that has been trained, as well as a config file. The system config file outline is below in its own box however the best way to understand this is to look at where it is used.

The MRCNN library is used to visualise and run detection on the image.

Imports

In [1]:

```
# FILE NEEDS MODULARISED

import os
import sys
import random
import math
import re
import time
import numpy as np
import tensorflow as tf
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import json
from flask import Flask, request, jsonify
from PIL import Image
from keras.backend import clear_session
import datetime as datetime

#----------------------------
# RCNN IMPORTS
#----------------------------

from mrcnn import utils
from mrcnn import visualize
from mrcnn.visualize import display_images
import mrcnn.model as modellib
from mrcnn.model import log
import cups as cup

app = Flask(__name__)

sys_config=json.load(open("config.json", 'r'))

%matplotlib inline
```

```
Using TensorFlow backend.
```

Config file

Values have been removed from this block for privacy

In [ ]:

```
{
    "api":{
        "base uri":"/api"
    },
    "model directory":"-",
    "weights path" : "-/.h5",
    "device":"/cpu:0",
    "mode":"inference",
    "cup directory":"-"
}
```

Image decoder

In order to support as many image types as possible, the API will only accept bas64 encoded images. base64 is an open file type and can therefore be converter easily from and to many other types. The file type as outlined in the documentation will be png for the final PoC however as the api will be open sourced at the end, base64 was the best way to future proof the solution.

In [2]:

```
import base64
import skimage.io

# This is requred to change base64 into a numpy ndarray

def decode(base64_string):
    if isinstance(base64_string, bytes):
        base64_string = base64_string.decode("utf-8")
    imgdata = base64.b64decode(base64_string)
    img = skimage.io.imread(imgdata, plugin='imageio')
    return img
```

Api Config

These blocks are used for configurations local to the api that are required.

The first is a get_ax() method. This standardises all graphs used and may be removed in future iterations. Inference config allows us to overwrite some values from the model config to suit the system the api is deployed on. create model is used to do just that: create the model we will use. This makes the h5 file into a dataset we can quickly call to

In [3]:

```
def get_ax(rows=1, cols=1, size=16):
    #This fn essentially allows a base size for graphs below
    #Common thing i've seen in notebooks with matplotlib
    _, ax = plt.subplots(rows, cols, figsize=(size*cols, size*rows))
    return ax
```

In [2]:

```
config = cup.CupConfig()

class InferenceConfig(config.__class__):
    # Make sure we only run detection 1 at a time
    # This value may be increased when moved to cloud
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    DETECTION_MIN_CONFIDENCE=0.96
config = InferenceConfig()


def create_model():
    with tf.device(sys_config["device"]):
        clear_session()
        global model
        model = modellib.MaskRCNN(mode=sys_config["mode"], model_dir=sys_config["model dire
                                  config=config)
    try:
        print("Loading weights ", sys_config["weights path"])
        model.load_weights(sys_config["weights path"], by_name=True)
    except:
        print("Weights file unable to be loaded".format(error))
```

Final return

This method is the final return values for the api. This calls to all other methods and compiles their returns into an easily digestible json dump.

In [5]:

```
#rois: [N, (y1, x1, y2, x2)] detection bounding boxes
def final_ret(image, roi):
    img=image[roi[0]:roi[2], roi[1]:roi[3]]
    centre = find_box_center(roi, image)
    plt.imshow(img)
    filename="detect_{:%Y%m%dT%H%M%S}.png".format(datetime.datetime.now())
    plt.savefig(filename)
    return json.dumps({"image":img.tolist(), "centre": centre, "filename":filename})
```

Find Centre

In order to accurately recreate the image in 3D i will need to know the relative and true coordinates of the object in screen space. This method finds the centrepoint of the region of interest and returns a list of all 4 values. We don't need to worry about different x/y sizes of the whole image because it is normalised to 1024x1024 when decoded by the rcnn.

In [6]:

```
def find_box_center(roi, image):
    true_y = (roi[0]+roi[2])/2
    true_x = (roi[1]+roi[3])/2
    rel_roi = []
    for i in roi:
        #The shape of the image is normalised to square so we don't need to
        #worry about different x and y shapes
        rel_roi.append((i/image.shape[0])*100)
    rel_y = np.round((rel_roi[0]+rel_roi[2])/2, 2)
    rel_x = np.round((rel_roi[1]+rel_roi[3])/2, 2)

    return ([true_y, true_x, rel_y, rel_x])
```

Detect and crop boxes

So i need to crop out the object to pass to the next model that will give me the rotation and depth. I could have just cropped the detected mask of the object but i found that just removing the refined region of interest would work better as it gives a more consistent view of the full object. This essentially cuts down on mistakes made by the model giving me a more accurate system overall.

The method calls the detection from the RCNN model and uses this to generate the RoIs to pass through the system.

In [7]:

```
def detect_boxes_and_crop(image):
    create_model()
    prediction = model.detect([image])[0]
    imgs=[]
    for roi in prediction['rois']:
        plt.axis('off')
        box_ret = final_ret(image, roi)
        imgs.append(box_ret)
    return imgs
```

The main route

This is the route that will be hit when someone calls to the API. it calls both the decode and the detect methods and sets everything in motion.

This will be changed in the final version of the system.

In [8]:

```
@app.route(sys_config['api']['base uri'], methods=['POST'])
def detect_cup_in_img():
    if not request.json:
        abort(400)
    image = decode(request.json['image'])
    cropped_images=detect_boxes_and_crop(image)
    return jsonify(cropped_images), 200
```

In [9]:

```
if __name__ == '__main__':
    app.run(debug=False)
```

```
 * Serving Flask app "__main__" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off

 * Running on http://127.0.0.1:5000/ (http://127.0.0.1:5000/) (Press CTRL+C
 to quit)

WARNING:tensorflow:From C:\Users\AdamG\Anaconda3\envs\fyp\lib\site-packages
\tensorflow\python\ops\sparse_ops.py:1165: sparse_to_dense (from tensorflow.
python.ops.sparse_ops) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Create a `tf.sparse.SparseTensor` and use `tf.sparse.to_dense` instead.
Loading weights  C:\Users\AdamG\OneDrive\Documents\Projects\Uni\FYP\API\logs
\initial_cups\mask_rcnn_cup_0017.h5

127.0.0.1 - - [27/Nov/2018 18:04:09] "POST /api HTTP/1.1" 200 -
```

Cup Validation

This file is built around validation of the model that has been trained. This file has more printouts than the associated api file and is a better visualisation of what the system does in regards to the RCNN model so far.

In [1]:

```
import os
import sys
import random
import math
import re
import time
import numpy as np
import tensorflow as tf
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import json
from flask import jsonify

sys_config=json.load(open("config.json", 'r'))

# Root directory of the project. We use the working directory of the notebook here as a sta
# this in the notebook.
ROOT_DIR = os.path.abspath("/")

# Import Mask RCNN
sys.path.append(ROOT_DIR)  # To find local version of the library
from mrcnn import utils
from mrcnn import visualize
from mrcnn.visualize import display_images
import mrcnn.model as modellib
from mrcnn.model import log

import cups as cup

%matplotlib inline
```

```
Using TensorFlow backend.
```

Begin a configuration of the cup dataset we are going to use. We use the inference mode of the data for validation as is standard for ML models. We only want to detect on 1 input at a time here because there is no need to have it higher. This block will also display the configuration of the model in full. This is good for tensorflow debug purposes.

In [2]:

```
config = cup.CupConfig()

class InferenceConfig(config.__class__):
    # Make sure we only run detection 1 at a time
    # This value may be increased when moved to cloud
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    DETECTION_MIN_CONFIDENCE=0.96

config = InferenceConfig()
config.display()
```

```
Configurations:
BACKBONE                       resnet101
BACKBONE_STRIDES               [4, 8, 16, 32, 64]
BATCH_SIZE                     1
BBOX_STD_DEV                   [0.1 0.1 0.2 0.2]
COMPUTE_BACKBONE_SHAPE         None
DETECTION_MAX_INSTANCES        100
DETECTION_MIN_CONFIDENCE       0.96
DETECTION_NMS_THRESHOLD        0.3
FPN_CLASSIF_FC_LAYERS_SIZE     1024
GPU_COUNT                      1
GRADIENT_CLIP_NORM             5.0
IMAGES_PER_GPU                 1
IMAGE_CHANNEL_COUNT            3
IMAGE_MAX_DIM                  1024
IMAGE_META_SIZE                14
IMAGE_MIN_DIM                  800
IMAGE_MIN_SCALE                0
IMAGE_RESIZE_MODE              square
IMAGE_SHAPE                    [1024 1024    3]
LEARNING_MOMENTUM              0.9
LEARNING_RATE                  0.001
LOSS_WEIGHTS                   {'rpn_class_loss': 1.0, 'rpn_bbox_loss': 1.0,
'mrcnn_class_loss': 1.0, 'mrcnn_bbox_loss': 1.0, 'mrcnn_mask_loss': 1.0}
MASK_POOL_SIZE                 14
MASK_SHAPE                     [28, 28]
MAX_GT_INSTANCES               100
MEAN_PIXEL                     [123.7 116.8 103.9]
MINI_MASK_SHAPE                (56, 56)
NAME                           cup
NUM_CLASSES                    2
POOL_SIZE                      7
POST_NMS_ROIS_INFERENCE        1000
POST_NMS_ROIS_TRAINING         2000
PRE_NMS_LIMIT                  6000
ROI_POSITIVE_RATIO             0.33
RPN_ANCHOR_RATIOS              [0.5, 1, 2]
RPN_ANCHOR_SCALES              (32, 64, 128, 256, 512)
RPN_ANCHOR_STRIDE              1
RPN_BBOX_STD_DEV               [0.1 0.1 0.2 0.2]
RPN_NMS_THRESHOLD              0.7
RPN_TRAIN_ANCHORS_PER_IMAGE    256
STEPS_PER_EPOCH                100
TOP_DOWN_PYRAMID_SIZE          256
TRAIN_BN                       False
```

```
TRAIN_ROIS_PER_IMAGE           200
USE_MINI_MASK                  True
USE_RPN_ROIS                   True
VALIDATION_STEPS               50
WEIGHT_DECAY                   0.0001
```

This fn essentially allows a base size for graphs below. This is a common thing i've seen in notebooks with matplotlib and see no reason not to use it here.

In [3]:

```
def get_ax(rows=1, cols=1, size=16):
    _, ax = plt.subplots(rows, cols, figsize=(size*cols, size*rows))
    return ax
```

Within our dataset of annotated cup images, I have split into train and validation images. This section below uses the 3 validation images which have annotations already attached for AP values later. The printout states 2 layers in the image (BG and Cup). This is how the RCNN splits up images, into background and chosen objects.

In [4]:

```
# Load validation dataset
dataset = cup.CupDataset()
dataset.load_cup(sys_config["cup directory"], "val")

# Must call before using the dataset as per mrcnn
dataset.prepare()

print("Images: {}\nClasses: {}".format(len(dataset.image_ids), dataset.class_names))
```

```
Images: 3
Classes: ['BG', 'cup']
```

In [5]:

```
# Create model in inference mode
with tf.device(sys_config["device"]):
    model = modellib.MaskRCNN(mode=sys_config["mode"], model_dir=sys_config["model director
                              config=config)

try:
    print("Loading weights ", sys_config["weights path"])
    model.load_weights(sys_config["weights path"], by_name=True)
except:
    print("Weights file unable to be loaded".format(error))
```

```
WARNING:tensorflow:From C:\Users\AdamG\Anaconda3\envs\fyp\lib\site-packages
\tensorflow\python\ops\sparse_ops.py:1165: sparse_to_dense (from tensorflow.
python.ops.sparse_ops) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Create a `tf.sparse.SparseTensor` and use `tf.sparse.to_dense` instead.
Loading weights  C:\Users\AdamG\OneDrive\Documents\Projects\Uni\FYP\API\logs
\initial_cups\mask_rcnn_cup_0017.h5
```

Here we start to actually use the model we have trained. Below is a print of a rough prediction of where the detected cup is in the image. For the process of this initial prototype I used a small training set of 10 images and 17 epochs (100 steps/e) in the interest of time. A more extensive training set will be used in the final product.

This block chooses a random image from the validation set to used. This image is saved and used throughout the remainder of the notebook.

In [6]:

```
image, image_meta, gt_class_id, gt_bbox, gt_mask =\
    modellib.load_image_gt(dataset, config, random.choice(dataset.image_ids), use_mini_mask

# Run object detection
results = model.detect([image], verbose=1)

# Display results
ax = get_ax()
r = results[0]
visualize.display_instances(image, r['rois'], r['masks'], r['class_ids'],
                            dataset.class_names, r['scores'], ax=ax,
                            title="Predictions")
```

```
Processing 1 images
image                    shape: (1024, 1024, 3)       min:    0.00000  max:
255.00000  uint8
molded_images            shape: (1, 1024, 1024, 3)    min: -123.70000  max:
151.10000  float64
image_metas              shape: (1, 14)               min:    0.00000  max:
1024.00000  int32
anchors                  shape: (1, 261888, 4)        min:   -0.35390  max:
   1.29134  float32
```

Predictions

In [7]:

```
print (r['rois'])
print ("-------------------")
print (r['rois'][0])
```

```
[[384 680 608 926]
 [429 435 735 725]
 [329  37 633 415]]
-------------------
[384 680 608 926]
```

In [8]:

```
#rois: [N, (y1, x1, y2, x2)] detection bounding boxes
#Will return x, y, relative x, relative y, and cropped image
def final_ret(image, roi):
    img=image[roi[0]:roi[2], roi[1]:roi[3]]
    centre = find_box_center(roi, image)
    plt.imshow(img)
    plt.savefig("logs/Detection_test_{}".format(roi))
    return {"image":img.tolist(), "centre": centre}
```

In [9]:

```
def find_box_center(roi, image):
    true_y = (roi[0]+roi[2])/2
    true_x = (roi[1]+roi[3])/2
    rel_roi = []
    for i in roi:
        #The shape of the image is normalised to square so we don't need to
        #worry about different x and y shapes
        rel_roi.append((i/image.shape[0])*100)
    rel_y = np.round((rel_roi[0]+rel_roi[2])/2, 2)
    rel_x = np.round((rel_roi[1]+rel_roi[3])/2, 2)

    return ([true_y, true_x, rel_y, rel_x])
```

In [10]:

```
def detect_boxes_and_crop(image):
    imgs=[]
    for i in range(len(r['rois'])):
        plt.axis('off')
        box_ret = final_ret(image, r['rois'][i])
        filename=("detect_tst_{}.png".format(i))
        imgs.append(box_ret)
    return imgs
```

In [11]:

```
image_list=detect_boxes_and_crop(image)
with open('logs/test_data.json', 'w') as outfile:
    json.dump(image_list, outfile)
```