Challenge

April 14, 2021

0.1 Technical Challenge to detect artifacts on the back seat of a car.

dataset available on https://sviro.kl.dfki.de/data/

you will need to download and extract these datasets: - https://sviro.kl.dfki.de/download/hyundai-tucson-2/ - https://sviro.kl.dfki.de/download/hyundai-tucson-4/

this notebook runs with tensorflow, keras, opencv, scikit, numpy, and imutils to train and test the images.

First, we create our MiniVGGNet class. This CNN class has 2 groups of CONV => RELU => CONV => RELU => POOL, followed by FC => RELU layer.

The class receives the width and height of filters and depth of image (color or grayscale image), and how many classes are on the output. The class returns the model constructed as a small VGGNet, this is why it is named as MiniVGGNet.

After each Activation layer, it is executed a BatchNormalization operation. It actually slow the learning rate, but enable higher accuracy levels.

This class is an example on the book 'Deep learning for Computer Vision with Python' from Dr. Adrian Rosebrock

[1]:
```
# import the necessary packages
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Dense
from tensorflow.keras import backend as K

class MiniVGGNet:
        @staticmethod
        def build(width, height, depth, classes):
                # initialize the model along with the input shape to be
                # "channels last" and the channels dimension itself
                model = Sequential()
                inputShape = (height, width, depth)
                chanDim = -1
```

```
                # if we are using "channels first", update the input shape
                # and channels dimension
                if K.image_data_format() == "channels_first":
                        inputShape = (depth, height, width)
                        chanDim = 1

                # first CONV => RELU => CONV => RELU => POOL layer set
                model.add(Conv2D(32, (3, 3), padding="same",␣
↪input_shape=inputShape))
                model.add(Activation("relu"))
                model.add(BatchNormalization(axis=chanDim))
                model.add(Conv2D(32, (3, 3), padding="same"))
                model.add(Activation("relu"))
                model.add(BatchNormalization(axis=chanDim))
                model.add(MaxPooling2D(pool_size=(2, 2)))
                model.add(Dropout(0.25))

                # second CONV => RELU => CONV => RELU => POOL layer set
                model.add(Conv2D(64, (3, 3), padding="same"))
                model.add(Activation("relu"))
                model.add(BatchNormalization(axis=chanDim))
                model.add(Conv2D(64, (3, 3), padding="same"))
                model.add(Activation("relu"))
                model.add(BatchNormalization(axis=chanDim))
                model.add(MaxPooling2D(pool_size=(2, 2)))
                model.add(Dropout(0.25))

                # first (and only) set of FC => RELU layers
                model.add(Flatten())
                model.add(Dense(512))
                model.add(Activation("relu"))
                model.add(BatchNormalization())
                model.add(Dropout(0.5))

                # softmax classifier
                model.add(Dense(classes))
                model.add(Activation("softmax"))

                # return the constructed network architecture
                return model
```

To preprocess the images, i.e., resize the images to feed into the CNN, a function is defined to help on the training and testing phase

[2]:
```
# import the necessary packages
import imutils
```

```
import cv2

def preprocess(image, width, height):

        # grab the dimensions of the image, then initialize the padding values
        (h, w) = image.shape[:2]

        # if the width is greater than the height then resize along the width
        # otherwise, the height is greater than the width so resize along the␣
 ↪height
        if w > h:
                image = imutils.resize(image, width=width)
        else:
                image = imutils.resize(image, height=height)

        # determine the padding values for the width and height to obtain the␣
 ↪target dimensions
        padW = int((width - image.shape[1]) / 2.0)
        padH = int((height - image.shape[0]) / 2.0)

        # pad the image then apply one more resizing to handle any rounding␣
 ↪issues
        image = cv2.copyMakeBorder(image, padH, padH, padW, padW, cv2.
 ↪BORDER_REPLICATE)
        image = cv2.resize(image, (width, height))

        # return the pre-processed image
        return image
```

0.2 Here we define the paths of our datasets

0.2.1 training Dataset

Contains seat splited images (right, middle, and left seat), separated into 7 folders, one for each class: - empty seat - baby on a baby chair - kid on a kid chair - adult - generic items layed over the seat - empty baby char - empty kid chair

0.2.2 testing Dataset

Contains the whole image of the back-seat

0.2.3 temp_weights

Each epoch that produces weights with a smaller function loss than the previous recorded weights will be stored on this folder.

[24]:
```
train_dataset = 'tucson_RGB/tucson/train/RGB'
test_dataset = 'tucson_RGB_wholeImage/tucson/test/RGB_wholeImage'
```

```
# path to weight output during training
temp_weights = 'output'
```

0.3 Training the CNN - MiniVGGNet

The CNN will learn 21 classes: the classes listed above repeated over the 3 available seats on the back seat. Each image on the training dataset will be windowed by a 56 pixels frame.

This next cell outputs the training epochs and at the final, prints the accuracy/loss x epochs of training and validation samples.

From the training dataset, 75% of the images will be used for training, 25% for validation

[12]:
```
# import the necessary packages
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.optimizers import SGD
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import cv2
import os

# initialize the data and labels
data = []
labels = []

# number of total epochs to train
total_epoch = 40

# size of window
filter = 56

# image color depth (RGB => 3)
depth = 3

# loop over the input images
print("[INFO] loading training images...")

notebook_path = os.path.abspath("Challenge.ipynb")
train_path = os.path.join(os.path.dirname(notebook_path), train_dataset)

for imagePath in paths.list_images(train_path):
        # load the image, pre-process it, and store it in the data list
```

```
        image = cv2.imread(imagePath)
        image = preprocess(image, filter, filter)
        image = img_to_array(image)
        data.append(image)

        # extract the class label from the image path and update the labels list
        seat = int(imagePath.split(os.path.sep)[-1].split('_')[-3])
        label = int(imagePath.split(os.path.sep)[-2])
        labels.append('{:0>2d}'.format(label+seat*7))

# scale the raw pixel intensities to the range [0, 1]
print("[INFO] scaling pixel intensities...")
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)

# partition the data into training and testing splits using 75% of
# the data for training and the remaining 25% for testing
print("[INFO] spliting train and test data...")
(trainX, testX, trainY, testY) = train_test_split(data,
        labels, test_size=0.25, random_state=42)

# convert the labels from integers to vectors
print("[INFO] changing labels to vectors...")
lb = LabelBinarizer().fit(trainY)
trainY = lb.transform(trainY)
testY = lb.transform(testY)

# initialize the model
print("[INFO] compiling model...")
model = MiniVGGNet.build(width=filter, height=filter, depth=depth, classes=21)

# configure SGD optimizer
opt = SGD(lr=0.01, decay=0.01 / total_epoch, momentum=0.9, nesterov=True)
model.compile(loss="categorical_crossentropy", optimizer=opt,␣
 ↪metrics=["accuracy"])

# construct the callback to save only the *best* model to disk based on the␣
 ↪validation loss
fname = os.path.sep.join([temp_weights, "weights-{epoch:03d}-{val_loss:.4f}.
 ↪hdf5"])
checkpoint = ModelCheckpoint(fname, monitor="val_loss", mode="min",␣
 ↪save_best_only=True, verbose=1)
callbacks = [checkpoint]

# train the network
print("[INFO] training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
```

```
        batch_size=32, epochs=total_epoch, callbacks=callbacks, verbose=2)

# evaluate the network
print("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=32)
print(classification_report(testY.argmax(axis=1), predictions.argmax(axis=1),␣
 ↪target_names=lb.classes_))

# plot the training + testing loss and accuracy
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, total_epoch), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, total_epoch), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, total_epoch), H.history["accuracy"], label="acc")
plt.plot(np.arange(0, total_epoch), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

```
[INFO] loading training images…
[INFO] scaling pixel intensities…
[INFO] spliting train and test data…
[INFO] changing labels to vectors…
[INFO] compiling model…
[INFO] training network…
Epoch 1/40
141/141 - 57s - loss: 1.2581 - accuracy: 0.6527 - val_loss: 4.2970 -
val_accuracy: 0.2127

Epoch 00001: val_loss improved from inf to 4.29696, saving model to
output/weights-001-4.2970.hdf5
Epoch 2/40
141/141 - 57s - loss: 0.4001 - accuracy: 0.8744 - val_loss: 1.3395 -
val_accuracy: 0.6147

Epoch 00002: val_loss improved from 4.29696 to 1.33948, saving model to
output/weights-002-1.3395.hdf5
Epoch 3/40
141/141 - 56s - loss: 0.2706 - accuracy: 0.9167 - val_loss: 2.1295 -
val_accuracy: 0.7307

Epoch 00003: val_loss did not improve from 1.33948
Epoch 4/40
141/141 - 58s - loss: 0.1890 - accuracy: 0.9387 - val_loss: 7.3117 -
val_accuracy: 0.6993
```

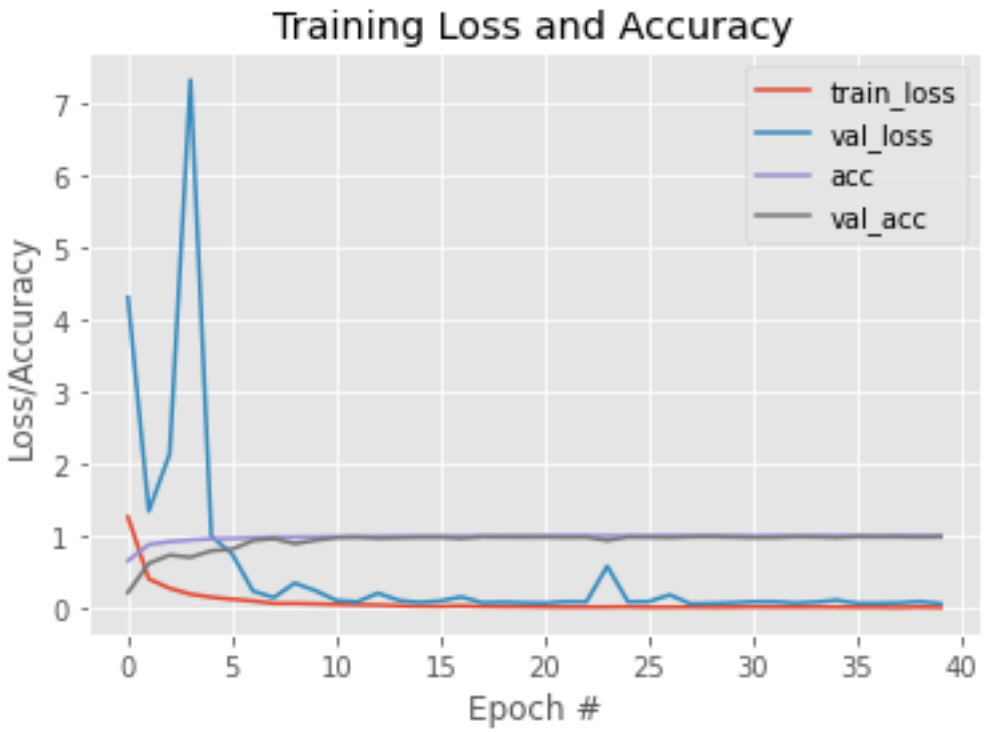
```
Epoch 00004: val_loss did not improve from 1.33948
Epoch 5/40
141/141 - 55s - loss: 0.1468 - accuracy: 0.9558 - val_loss: 0.9912 -
val_accuracy: 0.7893

Epoch 00005: val_loss improved from 1.33948 to 0.99117, saving model to
output/weights-005-0.9912.hdf5
Epoch 6/40
141/141 - 55s - loss: 0.1187 - accuracy: 0.9596 - val_loss: 0.7476 -
val_accuracy: 0.8120

Epoch 00006: val_loss improved from 0.99117 to 0.74762, saving model to
output/weights-006-0.7476.hdf5
Epoch 7/40
141/141 - 55s - loss: 0.0945 - accuracy: 0.9689 - val_loss: 0.2325 -
val_accuracy: 0.9307

Epoch 00007: val_loss improved from 0.74762 to 0.23251, saving model to
output/weights-007-0.2325.hdf5
Epoch 8/40
141/141 - 56s - loss: 0.0601 - accuracy: 0.9789 - val_loss: 0.1417 -
val_accuracy: 0.9580

Epoch 00008: val_loss improved from 0.23251 to 0.14167, saving model to
output/weights-008-0.1417.hdf5
Epoch 9/40
141/141 - 56s - loss: 0.0607 - accuracy: 0.9793 - val_loss: 0.3423 -
val_accuracy: 0.8887

Epoch 00009: val_loss did not improve from 0.14167
Epoch 10/40
141/141 - 56s - loss: 0.0540 - accuracy: 0.9807 - val_loss: 0.2377 -
val_accuracy: 0.9347

Epoch 00010: val_loss did not improve from 0.14167
Epoch 11/40
141/141 - 56s - loss: 0.0454 - accuracy: 0.9840 - val_loss: 0.1027 -
val_accuracy: 0.9720

Epoch 00011: val_loss improved from 0.14167 to 0.10272, saving model to
output/weights-011-0.1027.hdf5
Epoch 12/40
141/141 - 56s - loss: 0.0408 - accuracy: 0.9873 - val_loss: 0.0833 -
val_accuracy: 0.9780

Epoch 00012: val_loss improved from 0.10272 to 0.08332, saving model to
output/weights-012-0.0833.hdf5
```

```
Epoch 13/40
141/141 - 56s - loss: 0.0381 - accuracy: 0.9882 - val_loss: 0.2026 -
val_accuracy: 0.9633

Epoch 00013: val_loss did not improve from 0.08332
Epoch 14/40
141/141 - 55s - loss: 0.0284 - accuracy: 0.9916 - val_loss: 0.1021 -
val_accuracy: 0.9687

Epoch 00014: val_loss did not improve from 0.08332
Epoch 15/40
141/141 - 56s - loss: 0.0219 - accuracy: 0.9922 - val_loss: 0.0754 -
val_accuracy: 0.9753

Epoch 00015: val_loss improved from 0.08332 to 0.07545, saving model to
output/weights-015-0.0754.hdf5
Epoch 16/40
141/141 - 56s - loss: 0.0214 - accuracy: 0.9927 - val_loss: 0.0962 -
val_accuracy: 0.9753

Epoch 00016: val_loss did not improve from 0.07545
Epoch 17/40
141/141 - 56s - loss: 0.0263 - accuracy: 0.9913 - val_loss: 0.1514 -
val_accuracy: 0.9613

Epoch 00017: val_loss did not improve from 0.07545
Epoch 18/40
141/141 - 56s - loss: 0.0195 - accuracy: 0.9936 - val_loss: 0.0705 -
val_accuracy: 0.9820

Epoch 00018: val_loss improved from 0.07545 to 0.07054, saving model to
output/weights-018-0.0705.hdf5
Epoch 19/40
141/141 - 56s - loss: 0.0159 - accuracy: 0.9953 - val_loss: 0.0775 -
val_accuracy: 0.9793

Epoch 00019: val_loss did not improve from 0.07054
Epoch 20/40
141/141 - 56s - loss: 0.0159 - accuracy: 0.9949 - val_loss: 0.0691 -
val_accuracy: 0.9773

Epoch 00020: val_loss improved from 0.07054 to 0.06906, saving model to
output/weights-020-0.0691.hdf5
Epoch 21/40
141/141 - 56s - loss: 0.0118 - accuracy: 0.9960 - val_loss: 0.0651 -
val_accuracy: 0.9800

Epoch 00021: val_loss improved from 0.06906 to 0.06509, saving model to
```

```
output/weights-021-0.0651.hdf5
Epoch 22/40
141/141 - 55s - loss: 0.0100 - accuracy: 0.9976 - val_loss: 0.0846 -
val_accuracy: 0.9747

Epoch 00022: val_loss did not improve from 0.06509
Epoch 23/40
141/141 - 56s - loss: 0.0087 - accuracy: 0.9984 - val_loss: 0.0832 -
val_accuracy: 0.9767

Epoch 00023: val_loss did not improve from 0.06509
Epoch 24/40
141/141 - 56s - loss: 0.0108 - accuracy: 0.9967 - val_loss: 0.5761 -
val_accuracy: 0.9333

Epoch 00024: val_loss did not improve from 0.06509
Epoch 25/40
141/141 - 55s - loss: 0.0122 - accuracy: 0.9969 - val_loss: 0.0826 -
val_accuracy: 0.9787

Epoch 00025: val_loss did not improve from 0.06509
Epoch 26/40
141/141 - 56s - loss: 0.0075 - accuracy: 0.9978 - val_loss: 0.0868 -
val_accuracy: 0.9767

Epoch 00026: val_loss did not improve from 0.06509
Epoch 27/40
141/141 - 57s - loss: 0.0076 - accuracy: 0.9978 - val_loss: 0.1797 -
val_accuracy: 0.9727

Epoch 00027: val_loss did not improve from 0.06509
Epoch 28/40
141/141 - 56s - loss: 0.0099 - accuracy: 0.9964 - val_loss: 0.0554 -
val_accuracy: 0.9827

Epoch 00028: val_loss improved from 0.06509 to 0.05540, saving model to
output/weights-028-0.0554.hdf5
Epoch 29/40
141/141 - 56s - loss: 0.0065 - accuracy: 0.9980 - val_loss: 0.0604 -
val_accuracy: 0.9847

Epoch 00029: val_loss did not improve from 0.05540
Epoch 30/40
141/141 - 56s - loss: 0.0084 - accuracy: 0.9967 - val_loss: 0.0692 -
val_accuracy: 0.9787

Epoch 00030: val_loss did not improve from 0.05540
Epoch 31/40
```

```
141/141 - 57s - loss: 0.0128 - accuracy: 0.9949 - val_loss: 0.0847 -
val_accuracy: 0.9720

Epoch 00031: val_loss did not improve from 0.05540
Epoch 32/40
141/141 - 56s - loss: 0.0097 - accuracy: 0.9969 - val_loss: 0.0851 -
val_accuracy: 0.9733

Epoch 00032: val_loss did not improve from 0.05540
Epoch 33/40
141/141 - 56s - loss: 0.0088 - accuracy: 0.9969 - val_loss: 0.0656 -
val_accuracy: 0.9820

Epoch 00033: val_loss did not improve from 0.05540
Epoch 34/40
141/141 - 56s - loss: 0.0111 - accuracy: 0.9969 - val_loss: 0.0792 -
val_accuracy: 0.9807

Epoch 00034: val_loss did not improve from 0.05540
Epoch 35/40
141/141 - 56s - loss: 0.0091 - accuracy: 0.9967 - val_loss: 0.1059 -
val_accuracy: 0.9720

Epoch 00035: val_loss did not improve from 0.05540
Epoch 36/40
141/141 - 56s - loss: 0.0067 - accuracy: 0.9984 - val_loss: 0.0621 -
val_accuracy: 0.9827

Epoch 00036: val_loss did not improve from 0.05540
Epoch 37/40
141/141 - 55s - loss: 0.0079 - accuracy: 0.9978 - val_loss: 0.0626 -
val_accuracy: 0.9813

Epoch 00037: val_loss did not improve from 0.05540
Epoch 38/40
141/141 - 56s - loss: 0.0034 - accuracy: 0.9991 - val_loss: 0.0672 -
val_accuracy: 0.9820

Epoch 00038: val_loss did not improve from 0.05540
Epoch 39/40
141/141 - 56s - loss: 0.0101 - accuracy: 0.9967 - val_loss: 0.0861 -
val_accuracy: 0.9787

Epoch 00039: val_loss did not improve from 0.05540
Epoch 40/40
141/141 - 56s - loss: 0.0033 - accuracy: 0.9993 - val_loss: 0.0580 -
val_accuracy: 0.9827
```

```
Epoch 00040: val_loss did not improve from 0.05540
[INFO] evaluating network…
              precision    recall  f1-score   support

          00       0.95      0.96      0.96        85
          01       1.00      1.00      1.00        71
          02       1.00      0.98      0.99        49
          03       1.00      1.00      1.00        96
          04       0.94      0.92      0.93        49
          05       1.00      1.00      1.00        68
          06       0.99      1.00      0.99        82
          07       0.99      0.99      0.99       263
          08       1.00      0.95      0.98        22
          09       1.00      1.00      1.00        39
          10       1.00      0.93      0.96        42
          11       0.93      0.96      0.95        54
          12       0.96      1.00      0.98        26
          13       1.00      1.00      1.00        41
          14       0.98      0.99      0.98       231
          15       1.00      1.00      1.00        27
          16       1.00      0.98      0.99        49
          17       0.96      0.99      0.97        78
          18       0.95      0.91      0.93        58
          19       1.00      1.00      1.00        24
          20       1.00      1.00      1.00        46

    accuracy                           0.98      1500
   macro avg       0.98      0.98      0.98      1500
weighted avg       0.98      0.98      0.98      1500
```



0.4 Testing the results

As stated on the previous cell, the best weight was produced on epoch 28, saved at 'output/weights-028-0.0554.hdf5' These weights produced a validation accuracy of 0.9827 and loss of 0.0554. This result could be better if fine tuned by learning rate, batch size, class weight, etc.

0.5 The whole back seat image is taken for tests.

On the next cell, we take a whole back seat image and split it into 3 images, left seat (l), middle seat (m), and right seat (r). Each of these images will be feeded into the network for prediction, resulting into 21 possible classes, 7 artifacts classes repeated over 3 seats.

The label classes are the following: humanize = { 1: 'l-empty', 2: 'l-baby', 3: 'l-kid', 4: 'l-adult', 5: 'l-item', 6: 'l-baby chair', 7: 'l-chair', 8: 'm-empty', 9: 'm-baby', 10:'m-kid', 11:'m-adult', 12:'m-item', 13:'m-baby chair', 14:'m-chair', 15: 'r-empty', 16: 'r-baby', 17:'r-kid', 18:'r-adult', 19:'r-item', 20:'r-baby chair', 21:'r-chair', }

[29]:
```
# import the necessary packages
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.models import load_model
from matplotlib import pyplot as plt
import numpy as np
import imutils
import cv2
```

```
filter = 56
depth = 3

# load the pre-trained network
print("[INFO] loading pre-trained network...")
model = load_model('output/weights-028-0.0554.hdf5')

# randomy sample a few of the input images
imagePaths = list(paths.list_images(test_dataset))
imagePaths = np.random.choice(imagePaths, size=(5,),replace=False)

# Are we handling a whole image?
whole_image = True

# imagePaths = ['tucson_RGB_wholeImage/tucson/train/RGB_wholeImage/
 ↪tucson_train_imageID_0_GT_6_6_0.png']

# loop over the image paths
for imagePath in imagePaths:

        # read image from dataset
        image = cv2.imread(imagePath)

        # Are we handling a whole image? -> let's split it into 3
        if whole_image:
                (h, w) = image.shape[:2]
                # generate 3 slices of 550hx250w
                img0 = image[70:(70+550), 130:(130+250)].copy()
                img1 = image[70:(70+550), 363:(363+250)].copy()
                img2 = image[70:(70+550), 582:(582+250)].copy()
                #cv2.imshow("left", img0)
                #cv2.imshow("middle", img1)
                #cv2.imshow("right", img2)
                #cv2.waitKey(0)
                test = [ img0, img1, img2 ]
        else:
                test = [ image ]

        output = image
        predictions = []

        humanize = { 1: 'l-empty', 2: 'l-baby', 3:'l-kid', 4:'l-adult', 5:
 ↪'l-item', 6:'l-baby chair', 7:'l-chair',
                                8: 'm-empty', 9: 'm-baby', 10:'m-kid', 11:
 ↪'m-adult', 12:'m-item', 13:'m-baby chair', 14:'m-chair',
```

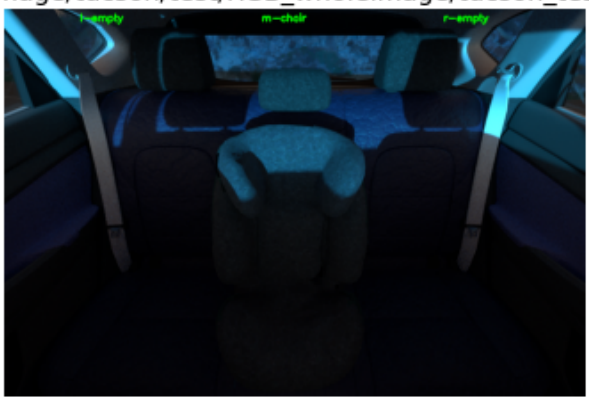
```
                                15: 'r-empty', 16: 'r-baby', 17:'r-kid', 18:
↪'r-adult', 19:'r-item', 20:'r-baby chair', 21:'r-chair',}

        for idx, img in enumerate(test):
                # pre-process the ROI and classify it
                roi = preprocess(img, filter, filter)
                roi = np.expand_dims(img_to_array(roi), axis=0) / 255.0
                pred = model.predict(roi).argmax(axis=1)[0] + 1
                predictions.append(str(pred))

                cv2.putText(output, str(humanize.get(pred)), (125+idx*300, 20),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.55, (0, 255, 0), 2)

        # show the output image
        print("[INFO] prediction: {}".format(" ".join(predictions)))
        plt.imshow(output)
        plt.title(imagePath)
        plt.axis('off')
        plt.show()
```

```
[INFO] loading pre-trained network…
[INFO] prediction: 4 8 18
```

tucson_RGB_wholeImage/tucson/test/RGB_wholeImage/tucson_test_imageID_267.png

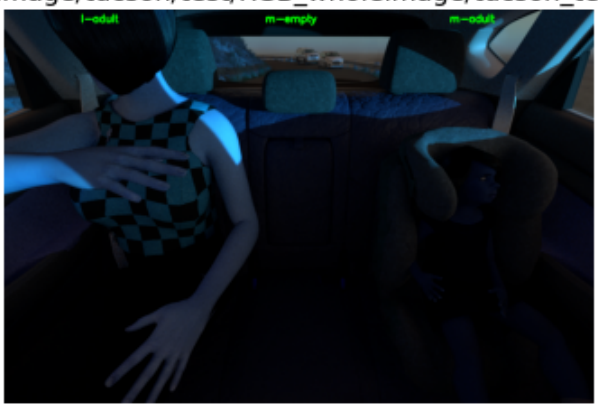
```
[INFO] prediction: 1 14 15
```

tucson_RGB_wholeImage/tucson/test/RGB_wholeImage/tucson_test_imageID_354.png



[INFO] prediction: 4 8 11

tucson_RGB_wholeImage/tucson/test/RGB_wholeImage/tucson_test_imageID_94.png



[INFO] prediction: 4 8 18

tucson_RGB_wholeImage/tucson/test/RGB_wholeImage/tucson_test_imageID_471.png

```
[INFO] prediction: 5 8 15
```

tucson_RGB_wholeImage/tucson/test/RGB_wholeImage/tucson_test_imageID_421.png

0.6 Conclusion

It is possible to see that the CNN recognizes always correctly the seat position but makes some mistakes on the respective artifact class. Still, success is greater than the error. As the accuracy of the network reached 98%, it tells us that the CNN needs some more improvement. In this case, weighting the classes will probably fix this issue, once images for empty seat are greater in number than other classes.