Car Damage Severity Detection

Overview

We have developed a full-fledged deep learning program that can identify car damage. For image categorization, we employed a three-layer network, and we even trained mask rcnn to identify scratch and dent. We've installed our model in AWS EC2 and built a user-friendly angular-based UI that allows users to quickly obtain car damage statistics. Once the picture is uploaded, it will be sent to the model using an API gateway lambda function, which will store it in an S3 bucket then the flask server running on ecs will retrieve from S3 bucket and process the image and will send back the result to our frontend application. By measuring the precise damage length of the car, we will be able to determine which portion of the car is damaged and how serious the damage will be.

Business Use Case

Claims leakage costs a lot of money in the car insurance market. The difference between the optimal and actual settlement of a claim is known as claims leakage. After an accident, we'll determine the extent of the vehicle damage. To cut down on claims leakage throughout the insurance claims processes. There is a visual assessment and validation going on. Because the individual must come and inspect the damage, this takes a long time. We're working on automating this process. Claims will be processed more quickly because of this automation.

Mapping the Problem

We're attempting to automate car damage visual evaluation and validation. We have photographs of cars that have been damaged as input data.

We will break the problem into three steps in order to validate vehicle damage.

1. First, we determine whether or not the provided input image of an automobile is damaged.

2. Next, we determine which side of the car in the photograph has been damaged (front, rear, or side).

3. Last step is to assess the severity of the damage (Minor, Moderate, Severe).

This is a classic classification problem, and we'll use Convolutional Neural Networks because we'll be dealing with photos as input (CNN).

Datasets

There are three different sorts of data:

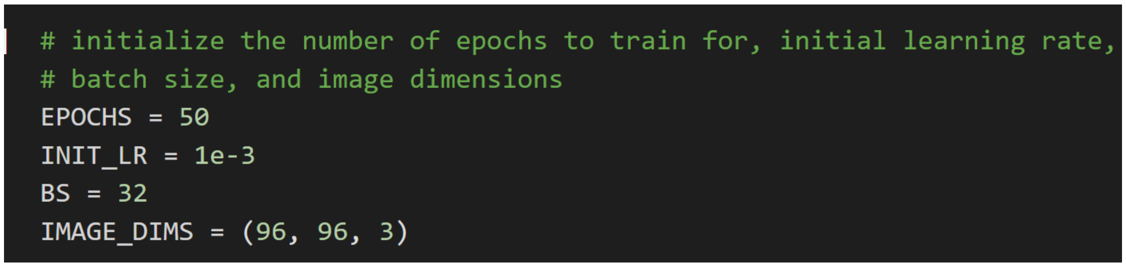
Car damaged, not damaged images in the training and test files.

Damage folders for training and testing on the front, back, and sides.

Damage severity training and testing folders Minor, Moderate, and Severe are the three levels of severity.

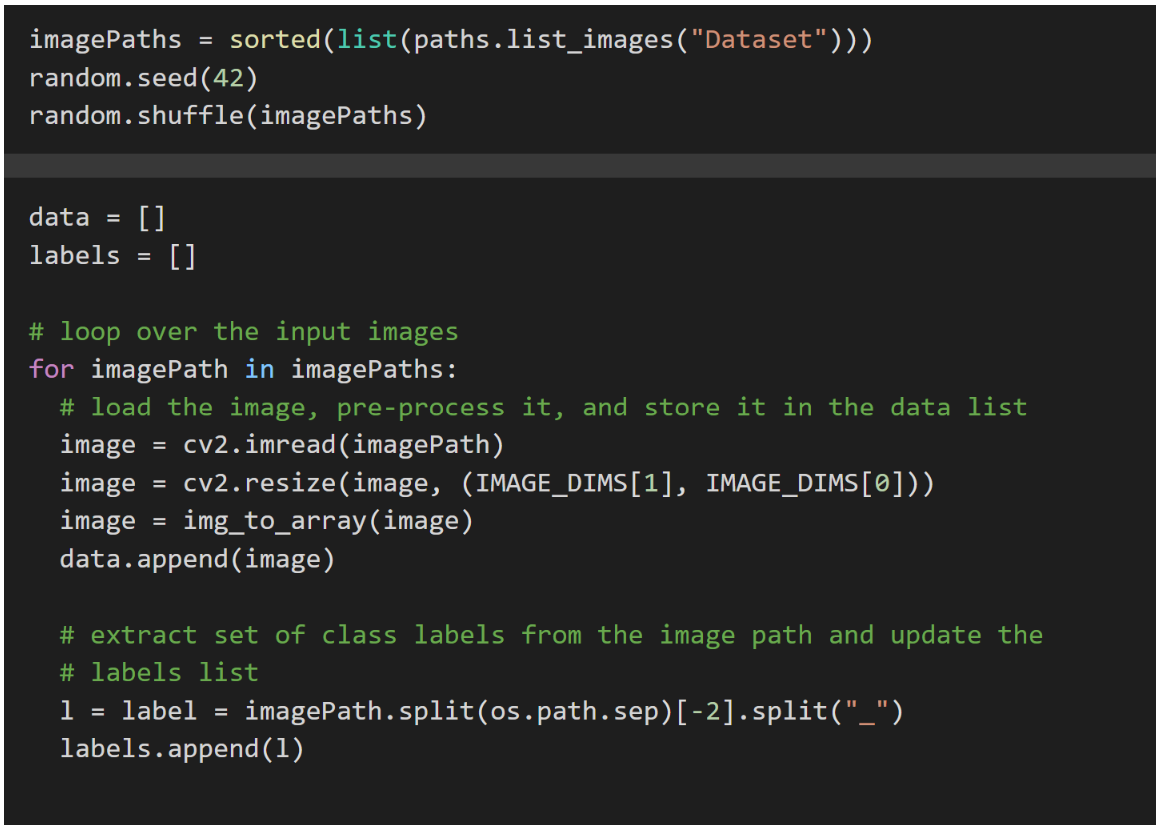
Full Layer Network Model

We began by setting a few parameters, including the number of epochs to train, the initial learning rate, batch size, and image dimensions.



```python
# initialize the number of epochs to train for, initial learning rate,
# batch size, and image dimensions
EPOCHS = 50
INIT_LR = 1e-3
BS = 32
IMAGE_DIMS = (96, 96, 3)
```

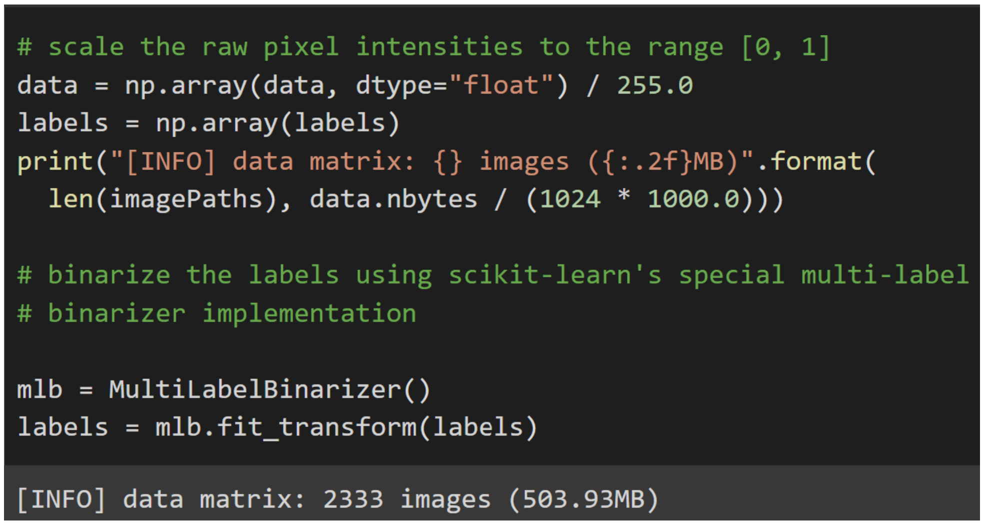
Then Imported the dataset in a variable named as ImagePaths



```python
imagePaths = sorted(list(paths.list_images("Dataset")))
random.seed(42)
random.shuffle(imagePaths)
```

```python
data = []
labels = []

# loop over the input images
for imagePath in imagePaths:
  # load the image, pre-process it, and store it in the data list
  image = cv2.imread(imagePath)
  image = cv2.resize(image, (IMAGE_DIMS[1], IMAGE_DIMS[0]))
  image = img_to_array(image)
  data.append(image)

  # extract set of class labels from the image path and update the
  # labels list
  l = label = imagePath.split(os.path.sep)[-2].split("_")
  labels.append(l)
```

We also created two lists, data and labels, to store the preprocessed pictures and labels, respectively.

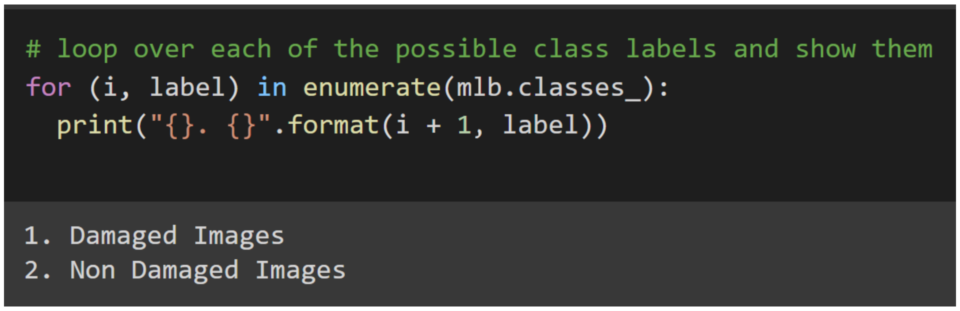


```
# scale the raw pixel intensities to the range [0, 1]
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
print("[INFO] data matrix: {} images ({:.2f}MB)".format(
  len(imagePaths), data.nbytes / (1024 * 1000.0)))

# binarize the labels using scikit-learn's special multi-label
# binarizer implementation

mlb = MultiLabelBinarizer()
labels = mlb.fit_transform(labels)
```

```
[INFO] data matrix: 2333 images (503.93MB)
```

We scale the pixel intensities to the range [0, 1] after converting the data array to a NumPy array. The labels are also converted from a list to a NumPy array. An information message is produced, indicating the size of the data matrix (in megabytes).

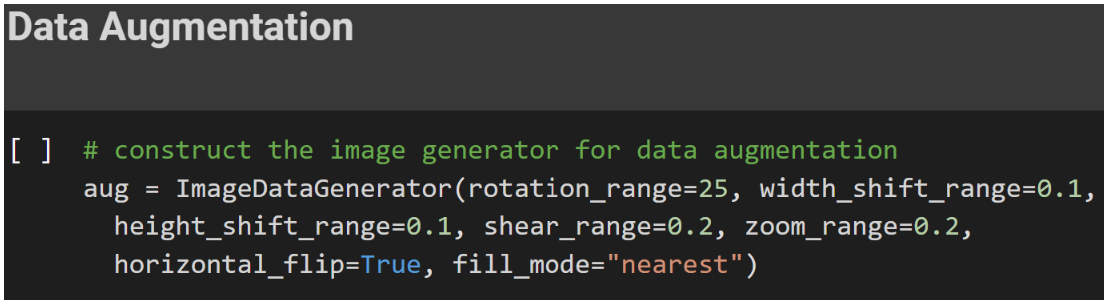
The labels are then binarized using scikit-learn's LabelBinarizer



```
# loop over each of the possible class labels and show them
for (i, label) in enumerate(mlb.classes_):
  print("{}. {}".format(i + 1, label))
```

```
1. Damaged Images
2. Non Damaged Images
```

Lists down the name of labels added



Data Augmentation

```
[ ]  # construct the image generator for data augmentation
     aug = ImageDataGenerator(rotation_range=25, width_shift_range=0.1,
       height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
       horizontal_flip=True, fill_mode="nearest")
```

Since the size of the dataset was very less there was a need to perform data augmentation. The ImageDataGenerator class will be used for data augmentation, which is a way of generating more training data from existing images in our dataset using random transformations (rotations, shearing, etc.). Data augmentation can help you prevent overfitting.

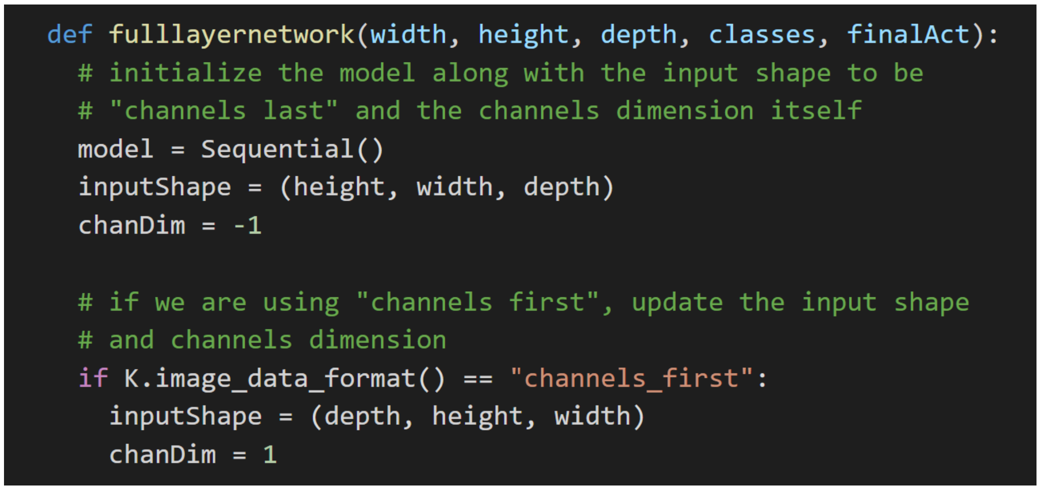
Creating a model

• Each convolutional block has a Conv2D layer and a MaxPooling2D layer, with Batch Normalization layers to normalize the outputs.

• The convolutional blocks will learn feature maps and, as a result, will learn to produce activations for certain regions of the pictures, such as edges.

• The contents of the feature maps are turned into a one-dimensional Tensor that may be utilized in the Dense layers with a Flatten layer.

• The classification is created by combining the dense layers. The first Dense layer's input is also Batch Normalized for the final Dense layer.



```
def fulllayernetwork(width, height, depth, classes, finalAct):
  # 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
```

Five parameters are required by our fulllayernetwork method:

width : The image width dimension.

height : The image height dimension.

depth : The depth of the image — also known as the number of channels.

classes : The number of classes in our dataset

finalAct : The sigmoid function is used as an activation function

Note: We will be working with input photos that are 96 × 96 pixels with a depth of three. Keep this in mind when we discuss the input volume's spatial dimensions as it goes through the network.

Let's begin adding layers to our model now:

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

Our first CONV => RELU => POOL block is seen above.

There are 32 filters in the convolution layer, each having a 3 x 3 kernel. The activation function RELU is used, followed by batch normalization.

Our POOL layer employs a 3 x 3 POOL size to quickly decrease spatial dimensions from 96 × 96 to 32 x 32 (we'll be training our network using 96 x 96 x 3 input pictures, as we'll see in the following section).

We also used dropout in our network design, as you can see from the code block. Dropout operates by disconnecting nodes from one tier to the next at random. Because no one node in the layer is responsible for forecasting a certain class, object, edge, or corner, this process of random disconnects during training batches naturally introduces redundancy into the model.

Before applying another POOL layer, we'll add (CONV => RELU) * 2 layers:

```
# (CONV => RELU) * 2 => POOL
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))
```

We can learn a richer collection of features by stacking numerous CONV and RELU layers together (prior to lowering the volume's spatial dimensions).

Take note of how:

We're expanding the size of our filter from 32 to 64. The smaller the spatial dimensions of our volume get as we progress further into the network, and the more filters we learn.
To avoid shrinking our spatial dimensions too soon, we reduced the maximum pooling size from 3 × 3 to 2 x 2.
At this point, the dropout is done once more.

Add an additional set of (CONV => RELU) * 2 => POOL:

```
# # (CONV => RELU) * 2 => POOL
model.add(Conv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(Conv2D(128, (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))
```

Here, you'll notice that we've raised the filter size to 128. To reduce overfitting once further, 25% of the nodes are dropped out.

Finally, we have a sigmoid classifier and a set of FC => RELU layers:

```
# first (and only) set of FC => RELU layers
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(Dropout(0.5))

# sigmoid classifier
model.add(Dense(classes))
model.add(Activation(finalAct))

# return the constructed network architecture
return model
```

Dense(1024) specifies a fully linked layer with corrected linear unit activation and batch normalization.

Dropout is run once more, this time seeing that we're dropping out 50% of the nodes throughout training. In our fully-connected layers, you'll often utilize a dropout of 40-50 percent, and a considerably lower rate, usually 10-25 percent, in earlier layers (if any dropout is applied at all).

A sigmoid classifier completes the model by returning the expected probabilities for each class label

```
# initialize the optimizer
opt = Adam(learning_rate=INIT_LR, decay=INIT_LR / EPOCHS)
```

```
model6.compile(loss="binary_crossentropy", optimizer=opt,
  metrics=["accuracy"])
```

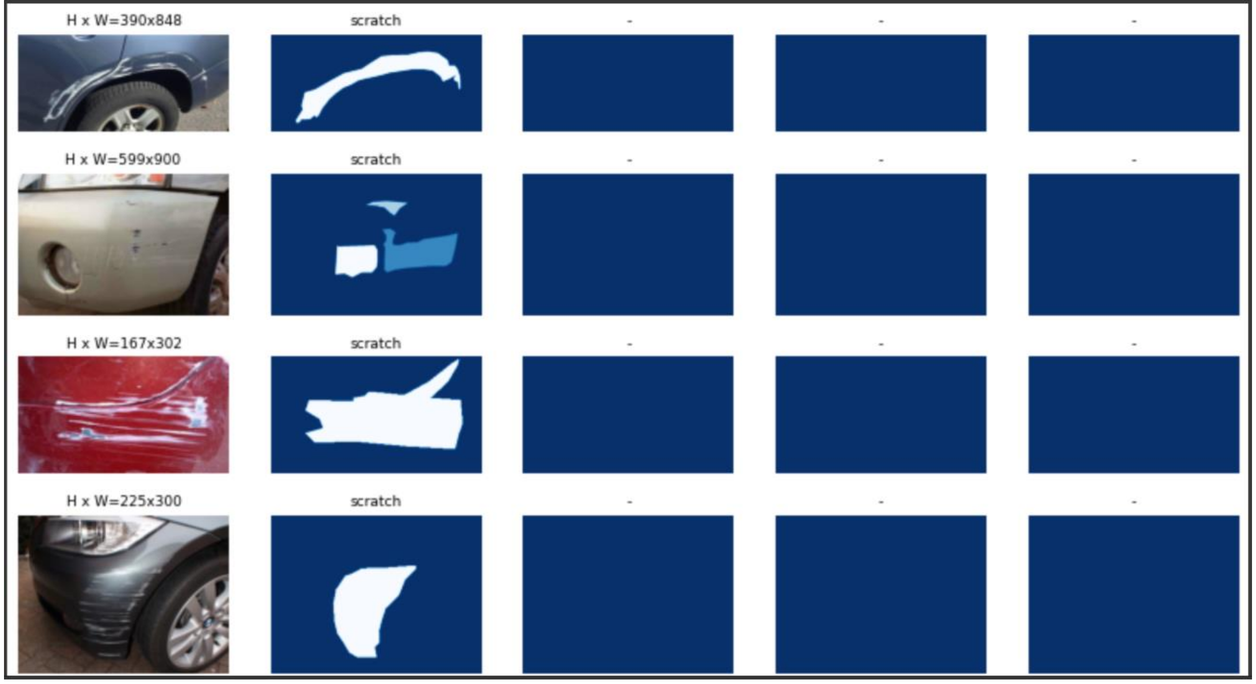
We utilized the Adam optimizer with learning rate decay before compiling our model using binary cross-entropy as the loss function.

We'll use a pre-trained convolutional neural network to extract the features before training the mask r-cnn. We're utilizing resnet101, which stands for residual network of 101 layers, as our convolutional neural network.

With the aid of VGG Annotator, images have been annotated.
The scratch and dent model's weights have been assigned such that we can correctly identify scratch during prediction.

Scratch masks are displayed here.

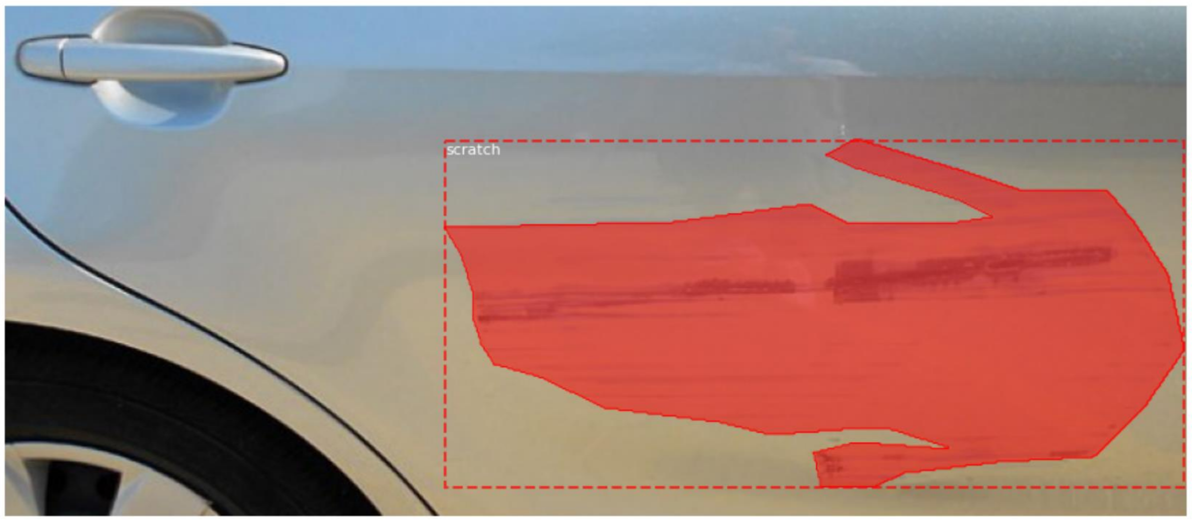
```
# Load and display random samples
image_ids = np.random.choice(dataset.image_ids, 5)
for image_id in image_ids:
    image = dataset.load_image(image_id)
    mask, class_ids = dataset.load_mask(image_id)
    visualize.display_top_masks(image, mask, class_ids, dataset.class_names)
```



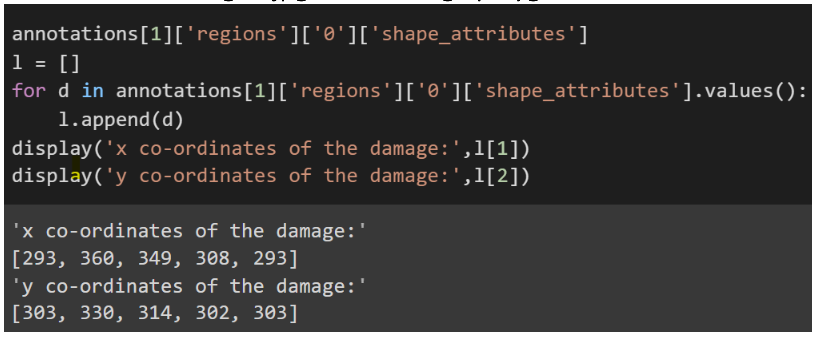
With the use of annotated masks, we'll construct bounding boxes for a car image, allowing us to produce areas if any damage is detected.

```python
image_id = random.choice(dataset.image_ids)
image = dataset.load_image(image_id)
mask, class_ids = dataset.load_mask(image_id)
# Compute Bounding box
bbox = utils.extract_bboxes(mask)

# Display image and additional stats
print("image_id ", image_id, dataset.image_reference(image_id))
log("image", image)
log("mask", mask)
log("class_ids", class_ids)
log("bbox", bbox)
# Display image and instances
visualize.display_instances(image, bbox, mask, class_ids, dataset.class_names)
```

```
image_id  39 /content/custom/train/image42.jpg
image                    shape: (280, 650, 3)        min:    0.00000  max:  255.00000  uint8
mask                     shape: (280, 650, 1)        min:    0.00000  max:    1.00000  bool
class_ids                shape: (1,)                 min:    1.00000  max:    1.00000  int32
bbox                     shape: (1, 4)               min:   73.00000  max:  649.00000  int32
```



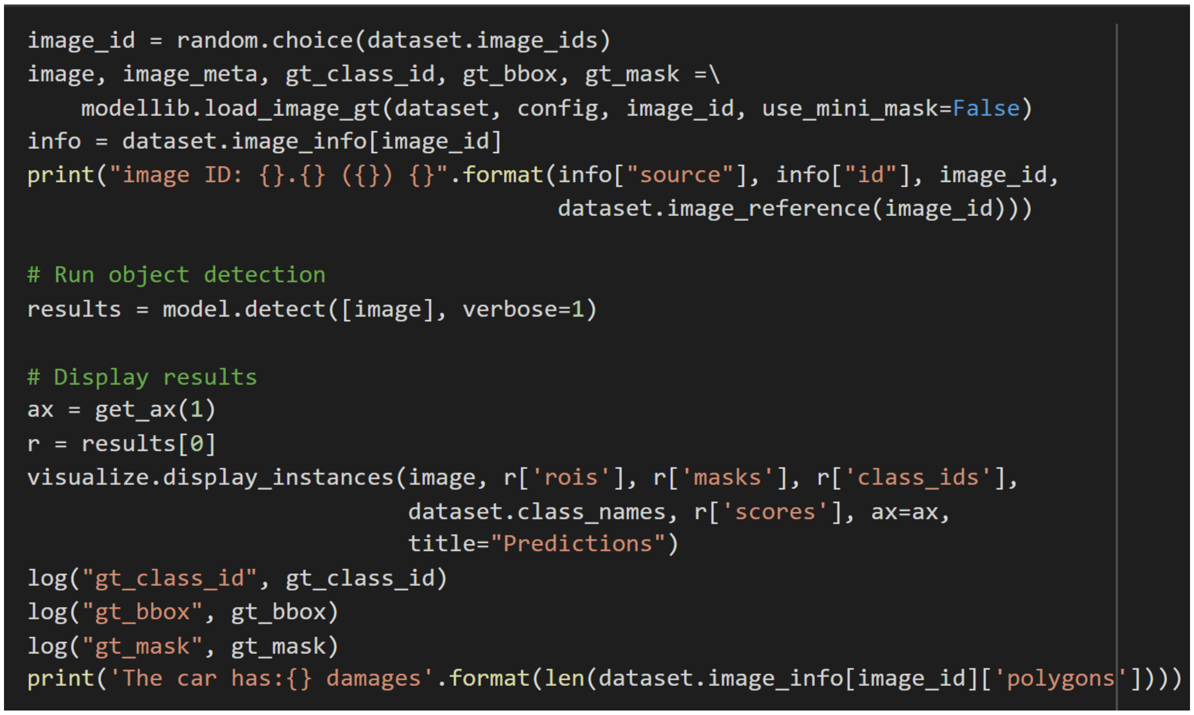
We can see some of the elements that make up picture annotations. It primarily contains the x and y coordinates of all labeled damages ('polygon') as well as the class name ('scratch') for each car image.
If we want to determine the area of a marked/detected automotive damage, we need to know the x and y coordinates of the polygon.
This is for the 'image2.jpg' 2nd damage polygon.



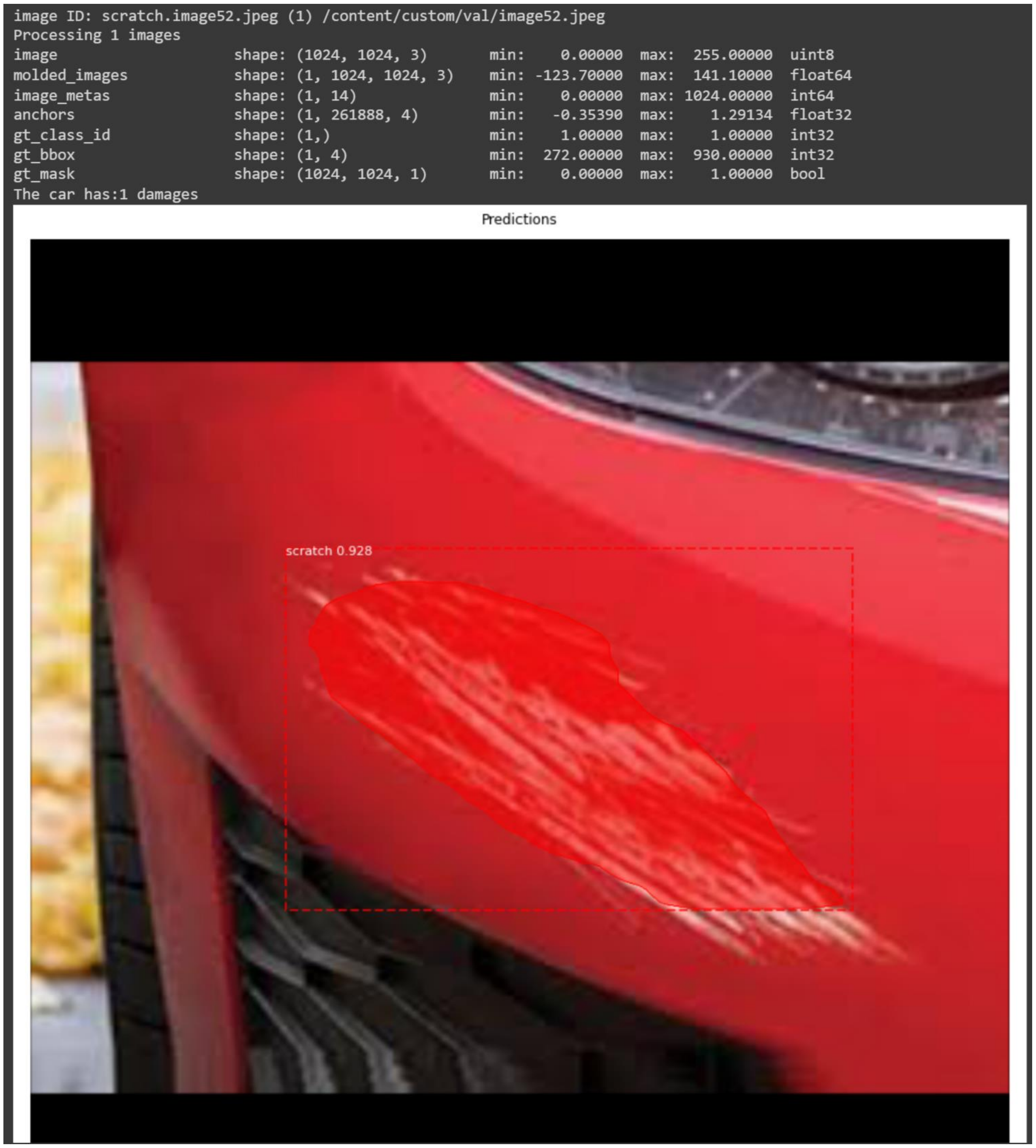
```
annotations[1]['regions']['0']['shape_attributes']
l = []
for d in annotations[1]['regions']['0']['shape_attributes'].values():
    l.append(d)
display('x co-ordinates of the damage:',l[1])
display('y co-ordinates of the damage:',l[2])
```

```
'x co-ordinates of the damage:'
[293, 360, 349, 308, 293]
'y co-ordinates of the damage:'
[303, 330, 314, 302, 303]
```

Prediction on a random validation image



```
image_id = random.choice(dataset.image_ids)
image, image_meta, gt_class_id, gt_bbox, gt_mask =\
    modellib.load_image_gt(dataset, config, image_id, use_mini_mask=False)
info = dataset.image_info[image_id]
print("image ID: {}.{} ({}) {}".format(info["source"], info["id"], image_id,
                                       dataset.image_reference(image_id)))

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

# Display results
ax = get_ax(1)
r = results[0]
visualize.display_instances(image, r['rois'], r['masks'], r['class_ids'],
                            dataset.class_names, r['scores'], ax=ax,
                            title="Predictions")
log("gt_class_id", gt_class_id)
log("gt_bbox", gt_bbox)
log("gt_mask", gt_mask)
print('The car has:{} damages'.format(len(dataset.image_info[image_id]['polygons'])))
```



```
image ID: scratch.image52.jpeg (1) /content/custom/val/image52.jpeg
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:  141.10000  float64
image_metas              shape: (1, 14)               min:    0.00000  max: 1024.00000  int64
anchors                  shape: (1, 261888, 4)        min:   -0.35390  max:    1.29134  float32
gt_class_id              shape: (1,)                  min:    1.00000  max:    1.00000  int32
gt_bbox                  shape: (1, 4)                min:  272.00000  max:  930.00000  int32
gt_mask                  shape: (1024, 1024, 1)       min:    0.00000  max:    1.00000  bool
The car has:1 damages
```

Conclusion

We have seen in this article how convolutional neural network can be used to do image classification and how mask rcnn can be used in order to tackle a challenge of instance segmentation in machine learning or computer vision To put it another way, it can distinguish between various things in a picture or video. It takes a picture and returns the object's bounding boxes, classes, and masks. The accuracy of the model will keep on increasing as we provide more dataset.

References

https://www.kaggle.com/lplenka/coco-car-damage-detection-dataset

https://www.pyimagesearch.com/2018/04/16/keras-and-convolutional-neural-networks-cnns/

https://medium.com/analytics-vidhya/car-damage-classification-using-deep-learning-d29fa1e9a520