Detection of Coal Pilferage using Image Analytics

Vedansh Chandra Dwivedi

A&I Center, Tata Steel

Guided By,

Mr Ayush Mishra

Abstract

Tata Steel Jamshedpur, on an average has an incoming of 6-7 goods train in the industry where each of them contains about 30 wagons approximately loaded with coal. For the past three year there has been an analysis on the weight of the coal that was loaded at the port and the weight of the coal when the train reached the industry and it was inferred that there is a loss of about 1-2% of what was loaded at the ports. On further inspection it was found that pilferage was one of the major reasons for the difference of weights in coal and it was assumed to be nearly equal to 15-19 crore INRs. Now with this project we aim to use the CCTV footages of the trains coming in the industry and predict whether there is any kind of pilferages by using Neural Networks and Image Analytics.

Coal Pilferage Detection using Image Analytics

Image Analytics, as the name suggests, uses images to make predictions. We will be using a Computer Vision Library (OpenCV) to process images in order to create a system that can analyze things based on what it captures as images or videos in the real world. For our understanding, we would be doing a sub-project that incorporates basic understanding of the libraries and techniques that we can use for processing lane lines on the road. This is usually used in self driving cars but this knowledge and experience can be further extended to other projects as well. Once we are well acquainted with all of the techniques we are using for lane detection we would be able to grasp the concepts required for our main project which will be to detect pilferages in goods train by the CCTV footages using Neural Networks.

Detection of Lane Lines

In this project we would be using a sample video of a road with lane lines captured by a moving vehicle and our aim in this project will be to detect the lanes on the streets. For this purpose we would be using an open source computer vision library called 'opencv', numpy and matplotlib.

finding-lanes ▸ lanes.py ▸ ...

```
1  import cv2  # Open CV
2  import numpy as np
3  import matplotlib.pyplot as plt
```

Once we are done with importing the libraries, we will define a function that can process the video clips as individual frames and yield the output on the frames so that we get the output on the video clip itself.

```
cap = cv2.VideoCapture("test2.mp4")
while(cap.isOpened()):
    _, frame = cap.read()
    canny_image = canny(frame)
    cropped_image = region_of_interest(canny_image)
    lines = cv2.HoughLinesP(cropped_image, 2, np.pi/180, 100, np.array([]), minLineLength=40, maxLineGap=5)
    averaged_lines = average_slope_intercept(frame, lines)
    line_image = display_lines(frame, averaged_lines)
    combo_image = cv2.addWeighted(frame, 0.8, line_image, 1, 1)
    cv2.imshow('result', combo_image)
    if cv2.waitKey(1) == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

In the above lines of code, the address of the video is stored in the variable named 'cap'. In our case, the video clip is in our working folder and is named as 'test2.mp4'. First of all, we move the video clip into our working directory for ease of work. A loop is run until the video is being played where each frame of the video is stored in the variable named 'frame'. This frame is passed into a function named 'canny'. The function is as shown below.

```python
def canny(image):
    # This function converts an image to grayscale, reduces noise of gray image and uses canny algorithm to
    find edges

    # convert image to grayscale. cvtColor is used to convert image to different color scales
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    # reducing Noise
    '''
        arg1 - Image to smoothen
        arg2 - kernel grid to be used
        arg3 - deviation
    '''
    blur = cv2.GaussianBlur(gray, (5,5), 0)

    # Simple Edge Detection Canny(image, low_threshold, high_threshold), best low:high :: 1:3
    canny = cv2.Canny(blur, 50, 150)
    return canny
```

This function takes in a parameter as an image. The image is converted into a grayscale image first. This step is done because the image consists of information about each pixel, the information is the intensity of Blue, Green and Red colors (BGR) expressed in the range 0-255 (8-bit representation) for each color. In Grayscale images, the pixel only contains 8 bit information about the intensity of white color. So conversion of image into grayscale makes image smaller and makes processing the image faster. In the very next line, we use GaussianBlur method to smoothen the image and after smoothening of the grayscale image we use the Canny algorithm to detect edges in the image. We use the threshold range of 50-150. The Canny function returns an image with only edges in it and this edged image is returned by this function. Now, as per our loop, the next step involves passing the canny image to a function named region_of_interest() [The function is displayed below]

```
def region_of_interest(image):
    # identify the region of interest in the image
    height = image.shape[0]

    # the below coords have been figured out using matplotlib library
    polygons = np.array([
    [(200, height), (1100, height), (550, 250)]
    ])

    # creating a black background
    mask = np.zeros_like(image)
    cv2.fillPoly(mask, polygons, 255)
    masked_image = cv2.bitwise_and(image, mask)
    return masked_image
```

Now, this image takes in as parameter the canny image. The height of the image is stored in a variable named 'height'. We define a polygon with the coordinates that encloses the area of interest to us. The Perspective of Vision causes the lane lines to look as if they are intersecting and thus the region of interest becomes a triangle for our case. Now we create a mask image for which our first step would be to create an array of the same shape as that of our image with all intensities as zeros. Now we have a black image of the same size as that of the image. Next, we fill the area enclosed by polygon vertices on our mask with white color. This yields a black mask with white area of interest on it. Now we perform the AND operation of our canny image with the masked image. This would yield an image that has edges and lie within our area of interest and is returned by our function. Now the next process is to map the points on the cropped image to the Hough Space. This technique detects straight lines in the image. Equation of a line is Y = MX + B. Plot (M,B) in a cartesian. For a straight line, we got a point in Hough Space, what if there was a point in cartesian and we wanted to plot it into Hough Space, for a single line, there lies infinite lines passing through that point. We will get a line in the Hough Space for a single point in the cartesian plane. Hough Space can be used to find a line that passes through a set of given points as for each point in cartesian we have a line in Hough space, so the intersection of lines in the Hough space gives a value (M, B) that can be substituted in y=mx+b to get the

appropriate line. This method is also used to find the line of best fit. Suppose there exists a set of points in cartesian and we need to find the best line that fits into the given points. We will get lines corresponding to each point in the Hough space. We first divide the Hough space into grids or bins. Now, there may exist inter of lines in multiple regions. We select the bin that has the maximum no of intersections in it to get the value (M, B) for the line of best fit. Sometimes, the gradient is equal to infinity for which our equation of line is not the best suited equation to work with, to solve this problem, we use the polar equation of a line $p = x \cos\theta + y \sin\theta$. Using the polar equation, for each line in the cartesian, we get a sinusoidal curve in Hough space. Here the Hough space gives values in (θ, p). The Hough Transformation gives a set of lines that fits our set of points in the cropped image, we need an average of these slope and intercepts to obtain one single line. This is achieved by the following function:

```
def average_slope_intercept(image, lines):
    left_fit = []
    right_fit = []
    for line in lines:
        x1, y1, x2, y2 = line.reshape(4)
        parameters = np.polyfit((x1, x2), (y1, y2), 1)
        slope = parameters[0]
        intercept = parameters[1]
        if slope < 0:
            left_fit.append((slope, intercept))
        else:
            right_fit.append((slope, intercept))
    left_fit_average = np.average(left_fit, axis=0)
    right_fit_average = np.average(right_fit, axis=0)
    left_line = make_coordinates(image, left_fit_average)
    right_line = make_coordinates(image, right_fit_average)
    return np.array([left_line, right_line])
```

Once the average of slope and intercepts are obtained, we display the lines on the image which is the frame in the image. This is done by the display line function

```
def display_lines(image, lines):
    line_image = np.zeros_like(image)
    if lines is not None:
        for x1, y1, x2, y2 in lines:
            cv2.line(line_image, (x1, y1), (x2, y2), (255, 0, 0), 10) # arg4 is BGR color (blue) #arg5 is
            line thickness
    return line_image
```

Now as our next step we merge the original frame and the frame with lines and display our frame. This loop goes on till the video is being played.

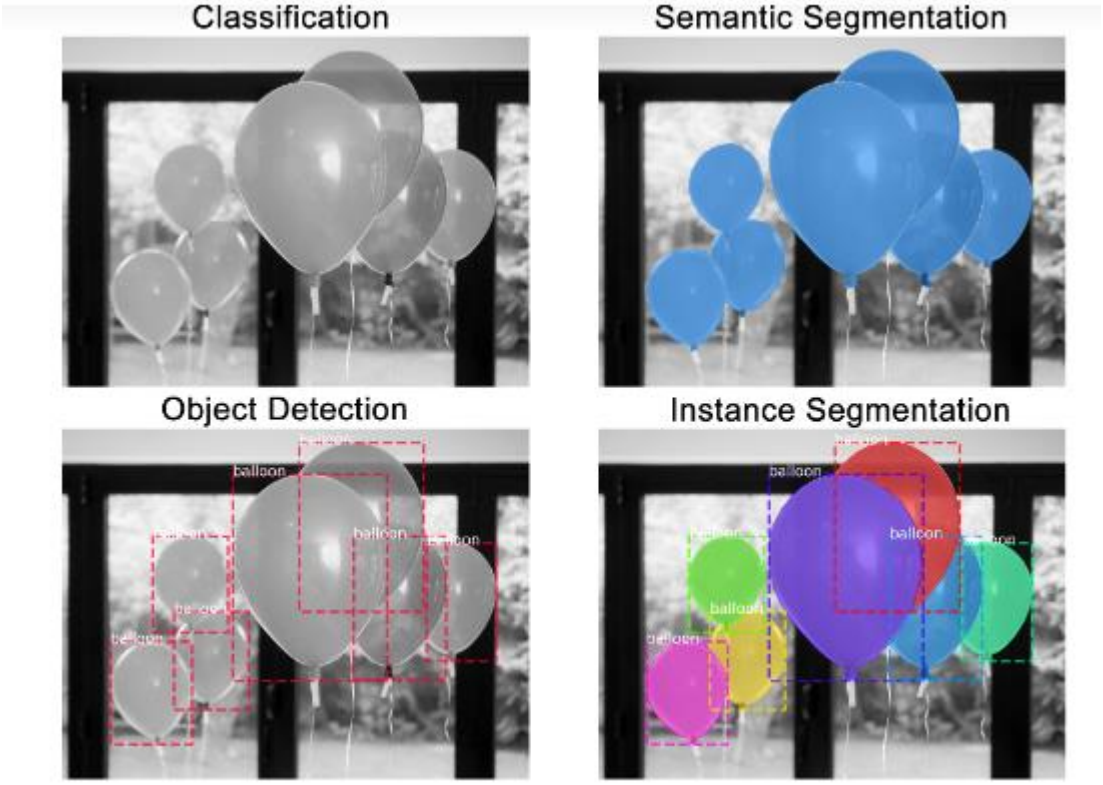
This is how we can successfully detect lane lines using opencv.

Pilferage Detection

The aim of this project is to detect wagons of train and predict whether there has been any kind of pilferages while transmission. This project requires the concepts of masking. We would be using Mask-RCNN. It stands for Regional Convolutional Neural Network that is used for Instance Segmentation.

What is Instance Segmentation?

Instance segmentation is the task of identifying object outlines at the pixel level. Compared to similar computer vision tasks, it's one of the hardest possible vision tasks. Consider the following asks:



- **Classification**: There is a balloon in this image**.**
- **Semantic Segmentation:** These are all the balloon pixels.
- **Object Detection:** There are 7 balloons in this image at these locations. We're starting to account for objects that overlap.
- **Instance Segmentation**: There are 7 balloons at these locations, and these are the pixels that belong to each one.

What is Mask RCNN?

Mask R-CNN (regional convolutional neural network) is a two stage framework. The stages can be described as follows :

- The first stage scans the image and generates proposals (areas likely to contain an object)

- The Second stage classifies the proposals and generates bounding boxes and masks.

Mask R-CNN is an extension of the Faster R-CNN. Faster R-CNN is a popular framework for object detection, and Mask R-CNN extends it with instance segmentation, among other things.

At a high level, Mask R-CNN consists of these modules:

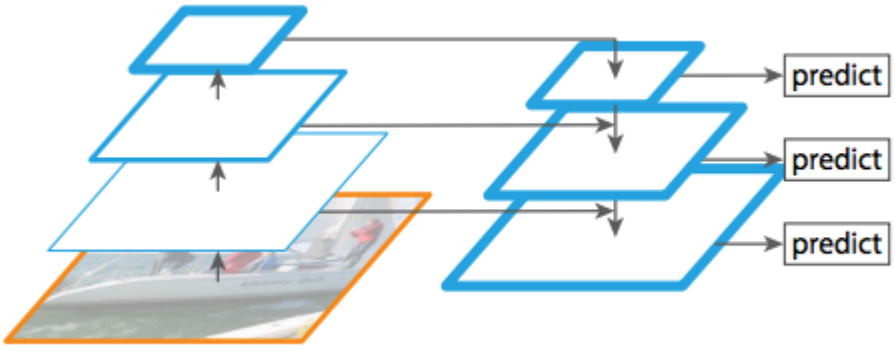
- Backbone
- Region Proposed Network (RPN)
- RoI Classifier & bounding box regressor
- Segmentation Masks

1. Backbone

This is a standard convolutional neural network (typically, ResNet50 or ResNet101) that serves as a feature extractor. The early layers detect low level features (edges and corners), and later layers successively detect higher level features (car, person, sky).

Passing through the backbone network, the image is converted from 1024x1024px x 3 (RGB) to a feature map of shape 32x32x2048. This feature map becomes the input for the following stages.

Feature Pyramid Network



While the backbone described above works great, it can be improved upon. The Feature Pyramid Network (FPN) was introduced by the same authors of Mask R-CNN as an extension that can better represent objects at multiple scales.

FPN improves the standard feature extraction pyramid by adding a second pyramid that takes the high level features from the first pyramid and passes them down to lower layers. By doing so, it allows features at every level to have access to both, lower and higher-level features.

Our implementation of Mask RCNN uses a ResNet101 + FPN backbone

2. Region Proposed Network

The RPN is a light weight neural network that scans the image in a sliding-window fashion and finds areas that contain objects.
The regions that the RPN scans over are called *anchors*. Which are boxes distributed over the image area, as show on the left. This is a simplified view, though. In practice, there are about 200K anchors of different sizes and aspect ratios, and they overlap to cover as much of the image as possible.

How fast can the RPN scan that many anchors? Pretty fast, actually. The sliding window is handled by the convolutional nature of the RPN, which allows it to scan all regions in parallel (on a GPU). Further, the RPN doesn't scan over the image directly (even though we draw the anchors on the image for illustration). Instead, the RPN

scans over the backbone feature map. This allows the RPN to reuse the extracted features efficiently and avoid duplicate calculations. With these optimizations, the RPN runs in about 10 ms according to the Faster RCNN paper that introduced it. In Mask RCNN we typically use larger images and more anchors, so it might take a bit longer.

The RPN generates two outputs for each anchor:

- **Anchor Class:** One of the two classes: foreground or background. The FG class implies that there is likely an object in that box.
- **Bounding Box Refinement:** A foreground anchor (also called positive anchor) might not be centered perfectly over the object. So the RPN estimates a delta (% chainge in x,y, width, height) to refine the anchor box to fit the object better.

Using the RPN predictions, we pick the top anchors that are likely to contain objects and refine their location and size. If several anchors overlap too much, we keep the one with the highest foreground score and discard the rest (referred to as Non-max Suppression). After that we have the final *proposals* (regions of interest) that we pass to the next stage.

3. ROI Classifier & Bounding Box Regressor

This stage runs on the regions of interest (ROIs) proposed by the RPN. And just like the RPN, it generates two outputs for each ROI:

1. **Class:** The class of the object in the ROI. Unlike the RPN, which has two classes (FG/BG), this network is deeper and has the capacity to classify regions to specific classes (person, car, chair, ...etc.). It can also generate a *background* class, which causes the ROI to be discarded.

2. **Bounding Box Refinement:** Very similar to how it's done in the RPN, and its purpose is to further refine the location and size of the bounding box to encapsulate the object.

ROI Pooling

Using the RPN predictions, we pick the top anchors that are likely to contain objects and refine their location and size. If several anchors overlap too much, we keep the one with the highest foreground score and discard the rest (referred to as Non-max Suppression). After that we have the final *proposals* (regions of interest) that we pass to the next stage.

ROI pooling refers to cropping a part of a feature map and resizing it to a fixed size. It's similar in principle to cropping part of an image and then resizing it (but there are differences in implementation details).

The authors of Mask R-CNN suggest a method they named ROIAlign, in which they sample the feature map at different points and apply a bilinear interpolation. In our implementation, we used TensorFlow's crop_and_resize function for simplicity and because it's close enough for most purposes.

4. Segmentation Masks

If we stop at the end of the last section then you have a Faster R-CNN framework for object detection. The mask network is the addition that Mask R-CNN paper introduced. The mask branch is a convolutional network that takes the positive regions selected by the ROI classifier and generates masks for them. The generated masks are low resolution. But they are *soft* masks, represented by float numbers, so they hold more details than binary masks. The small mask size helps keep the mask branch light. During training, we scale down the ground-truth masks to compute the loss, and during

inferencing we scale up the predicted masks to the size of the ROI bounding box and that gives us the final masks, one per object.

Implementation

For this project, we will take a sample video of a goods train carrying coal (Video used in this project: https://www.youtube.com/watch?v=_FfW5Fo6NnI&t=6s). Using the following piece of code we divide the videos into frames and store them into a folder for later use. The function is as follows:

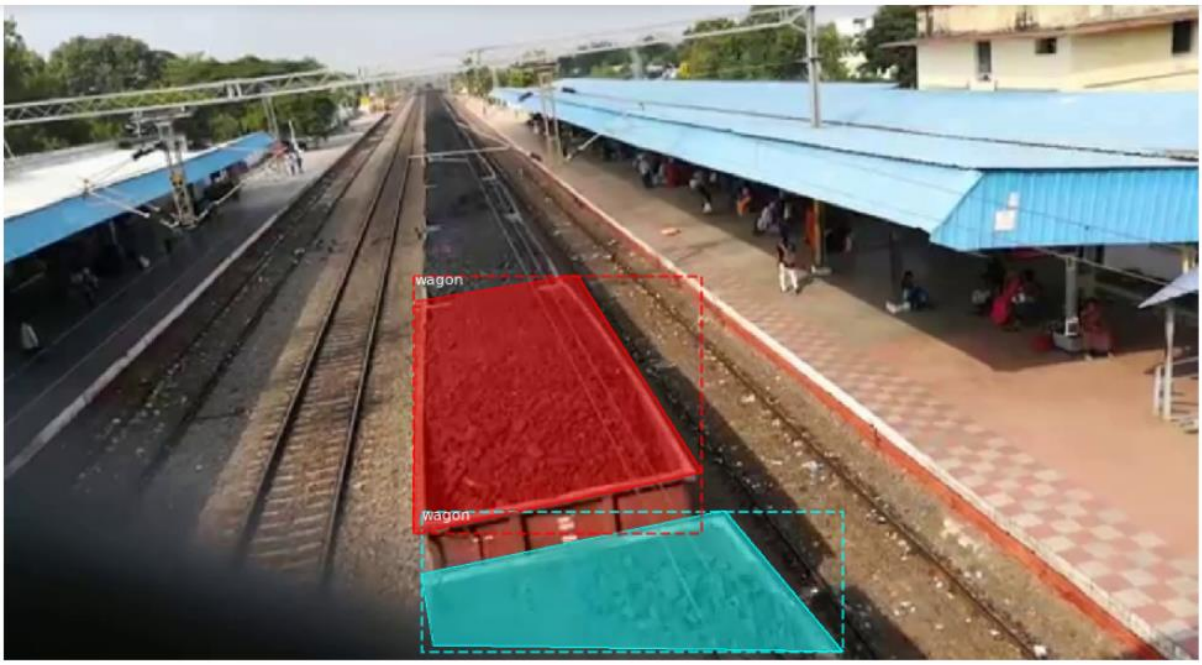
```
import cv2
def vid2frame(path):
    cap = cv2.VideoCapture(path)
    count = 0
    success = 1
    while success:
      success, image = cap.read()
      cv2.imwrite("test/frame%d.jpg" % count, image)
      count += 1
    return count

vid2frame('test.mp4')
```

Next we need a weight's file for which I used a file which has been trained on the COCO dataset. Although the COCO dataset does not contain the wagon class (we are performing wagon detection). It contains a lot of other images (~120K), so the trained weights have already learned a lot of the features common in natural images, which really helps. And, second, the accuracy depends upon the size of the dataset we provide. From the frames that we got from breaking down the video, we will select some random images for training purpose, the number of images will directly affect the accuracy of our model. After selecting the images, we will annotate them, which means that we would mark the regions of interest in our images and then save those regions in a json format. For annotating images in this project we used the VIA (VGG Image Annotator). Once the annotation is complete we download the information as json file. Now we have to store the images and the json file in 'datasets/wagon/train' folder where wagon is name of our working directory. Now we need to define a WagonDataset class in wagon.py that can make use of the images for training. After the datasets have been loaded in the wagon.py file, we can verify our dataset.

```
# Load random image and mask.
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)
```



Once we verified that our datasets have been successfully loaded, we now need to create a configuration file for our project. The configurations for this project are similar to the base configuration used to train the COCO dataset, so I just needed to override 3 values. As I did with the `Dataset` class, I inherit from the base `Config` class and add my overrides:

```
class WagonConfig(Config):
    """Configuration for training on the toy  dataset.
    Derives from the base Config class and overrides some values.
    """
    # Give the configuration a recognizable name
    NAME = "wagon"

    # We use a GPU with 12GB memory, which can fit two images.
    # Adjust down if you use a smaller GPU.
    IMAGES_PER_GPU = 2

    # Number of classes (including background)
    NUM_CLASSES = 1 + 1   # Background + train

    # Number of training steps per epoch
    STEPS_PER_EPOCH = 100

    # Skip detections with < 70% confidence
    DETECTION_MIN_CONFIDENCE = 0.7
```

Our next step is to train our Mask R-CNN model. Since our implementation uses ResNet101 and FPN so we need a powerful GPU. To start the training, we need to open

a prompt from the ‘wagon’ directory. The code will automatically download weights from the repository. To start training, we give the following command in our prompt:

```
python3 balloon.py train --
dataset=C:\Users\Vedansh\Mask_RCNN\datasets\wagon\ --
weights=coco
```

Once the model is trained, it is ready to detect wagons and mask them in the videos.

Now, since we are able to mask the wagons, we can extend this knowledge to detect and mask coal and detect irregularities and patterns and detect for pilferages. Also, to attain more accuracy we can train our model with more wide varieties of data such as with blurred footages, low lighting condition footages, footage in various weather conditions, footages from various angles etc. The accuracy of the model depends upon the amount and variety of the datasets that we provide.

Conclusion

This project requires a firm base of understanding of neural networks and concepts to be used for Image Analytics (from setting up the environment to implementing the concepts). This project gave an impetus to my potentiality and will help me in numerous aspects in the imminent future.