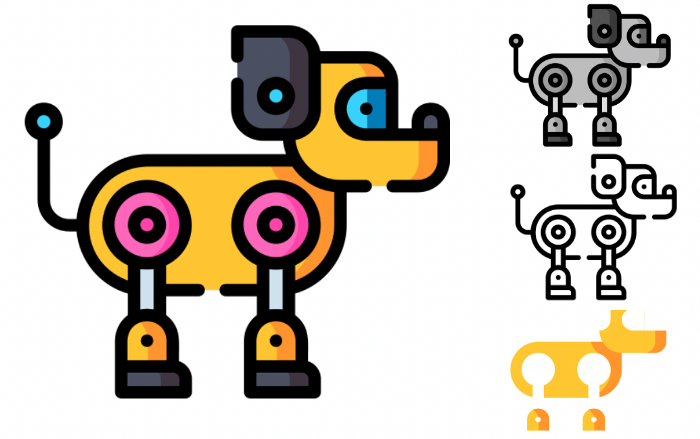
**Feature Engineering with Image Data**

**Cropping, grayscale, RGB channels, intensity thresholds, edge detection and colour filters**



(source: [flaticon](https://www.flaticon.com/free-icon/robotic-dog_9004669?related_id=9004669&origin=pack))

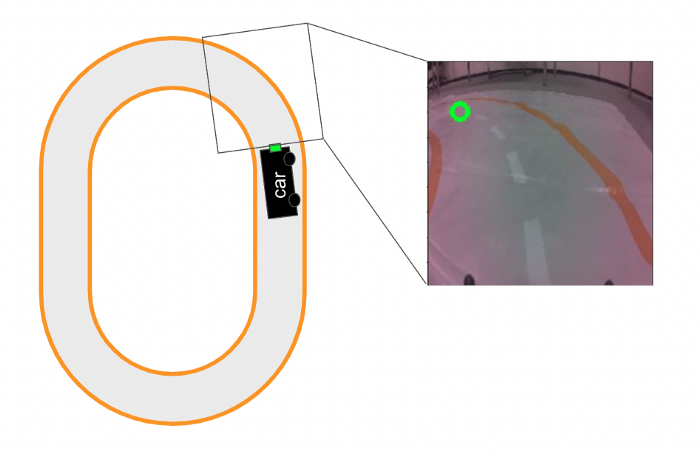
With feature engineering, we immediately think about tabular data. Yet, we can also get features for image data. The goal is to extract the most important aspects of the image. Doing so will make it easier to find a mapping between our data and the target variable.

This means you can use less data and train smaller models. A smaller model reduces the time needed to make predictions. This is particularly useful when deploying on edge devices. An additional benefit is that you can be more certain about what your model is using to make those predictions.

We will walk through some methods of image feature engineering using Python:

* Cropping
* Grayscalling
* Selecting RGB channels
* Intensity thresholds
* Edge detection
* Colour filters (i.e. extracting pixels in a given colour range)

To keep things interesting, we will be doing this for an automated car. As seen below, we want to train a model using images of a track. The model will then be used to make predictions that direct the car. To end, we will discuss the limitations of feature engineering from image data.



Automate car with camera sensor (source: author)

**Feature engineering vs augmentation**

Before we dive into that, it is worth discussing image augmentation. This method has similar goals to feature engineering. Yet, it achieves them in different ways.

**What is data augmentation?**

Data augmentation is when we systematically or randomly alter data using code. For images, this includes methods like flipping, adjusting colour and adding random noise. These methods allow us to artificially introduce noise and increase the size of our dataset. If you want more detail on image augmentation, I suggest this article:

**[Augmenting Images for Deep Learning](https://towardsdatascience.com/augmenting-images-for-deep-learning-3f1ea92a891c" \t "_blank)**

**[Using Python to augment data by flipping, adjusting brightness, color jitter and random noise](https://towardsdatascience.com/augmenting-images-for-deep-learning-3f1ea92a891c" \t "_blank)**

[towardsdatascience.com](https://towardsdatascience.com/augmenting-images-for-deep-learning-3f1ea92a891c" \t "_blank)

In production, a model will need to perform under different conditions. These conditions are determined by variables such as lighting, the angle of a camera, the colour of a room, or objects in the background.

The goal of data augmentation is to create a model that is robust to changes in these conditions. It does this by adding noise that simulates conditions in the real world. For example, changing the brightness of images is similar to collecting data during different times of the day.

By increasing the size of our dataset, augmentation also allows us to train more complicated architectures. In other words, it helps the model parameters converge.

**Feature engineering with image data**

The goals of feature engineering are similar. We want to create a more robust model. Except now, we remove any noise that is not necessary for accurate predictions. In other words, we remove the variables that will change with different conditions.

By extracting the most important aspect of an image, we are also simplifying the problem. This allows us to rely on simpler model architectures. This means we can use smaller datasets to find a mapping between the input and target.

An important difference is how the approaches are treated in production. Your model will not make predictions on augmented images. Yet, with feature engineering, your model will need to make predictions on the same features it was trained on. This means you must be able to do the feature engineering in production.

**Image feature engineering with Python**

Okay, with all that in mind, let’s move on to feature engineering. We’ll go over the code and you can also find the project on [GitHub](https://github.com/conorosully/medium-articles/blob/master/src/image_tools/image_features.ipynb).

To start, we’ll use the imports below. We have some standard packages (lines 2–3). **Glob** is used to handle file paths (line 5). We also have some packages used to work with images (lines 7–8).

#Imports   
import numpy as np  
import matplotlib.pyplot as plt  
  
import glob  
  
from PIL import Image  
import cv2

As mentioned, we’ll be working with images used to power an automated car. You can find examples of these on [Kaggle](https://www.kaggle.com/datasets/conorsully1/jatracer-images?select=object_detection). We load one of these images with the code below. We start by loading the file paths of all the images (lines 2–3). Then load (line 8) and display (line 9) the image at the first path. You can see this image in **Figure 1**.

#Load image paths  
read\_path = "../../data/direction/"  
img\_paths = glob.glob(read\_path + "\*.jpg")  
  
fig = plt.figure(figsize=(10,10))  
  
#Display image  
img = Image.open(img\_paths[0])  
plt.imshow(img)



Figure 1: example of track image (source: author)

**Cropping**

A simple method is to crop images to remove unwanted outer areas. The aim is to only remove parts of the image that are not necessary for predictions. For our automated car, we could remove pixels from the background.

To do this, we load an image (line 2). We then convert this image to an array (line 5). This array will have dimensions 224 x 224 x 3. The height and width of the image are 224 pixels and each pixel has an R G B channel. To crop the image we select only the pixels from position 25 onwards on the y-axis (line 8). You can see the result in **Figure 2**.

#Load image  
img = Image.open(img\_paths[609])  
  
#Covert to array  
img = np.array(img)  
  
#Simple crop  
crop\_img = img[25:,]

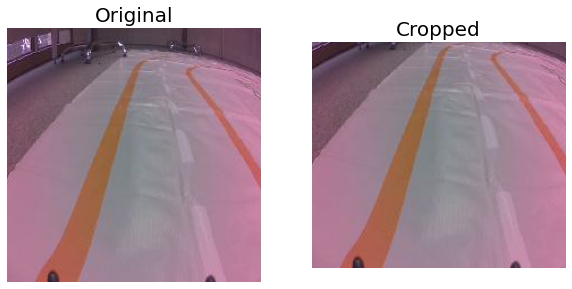


Figure 2: cropping an image (source: author)

You might want to maintain the aspect ratio. In this case, you can achieve similar results by turning the unwanted pixels black (line 3).

#Change pixels to black  
crop\_img = np.array(img)  
crop\_img[:25,] = [0,0,0]

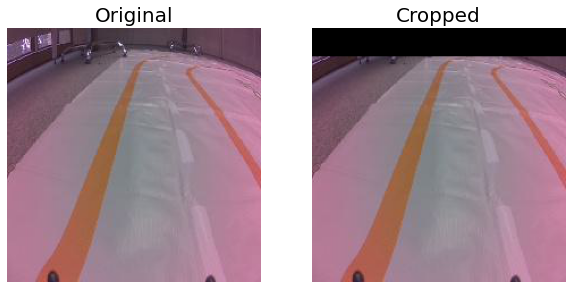


Figure 3: cropping by changing pixel colour (source: author)

With cropping, we are removing unnecessary pixels. We can also avoid the model overfitting to the training data. For example, the chairs in the background may be present at all left turns. As a result, the model could associate these with the prediction to turn left.

Looking at the above image, you may be tempted to crop it further. That is you could crop the left side of the image without removing any of the track. However, in **Figure 4**, you can see for other images we would remove important parts of the track.

#Additional cropping   
crop\_img = np.array(img)  
crop\_img[:25,] = [0,0,0]  
crop\_img[:,:40] = [0,0,0]

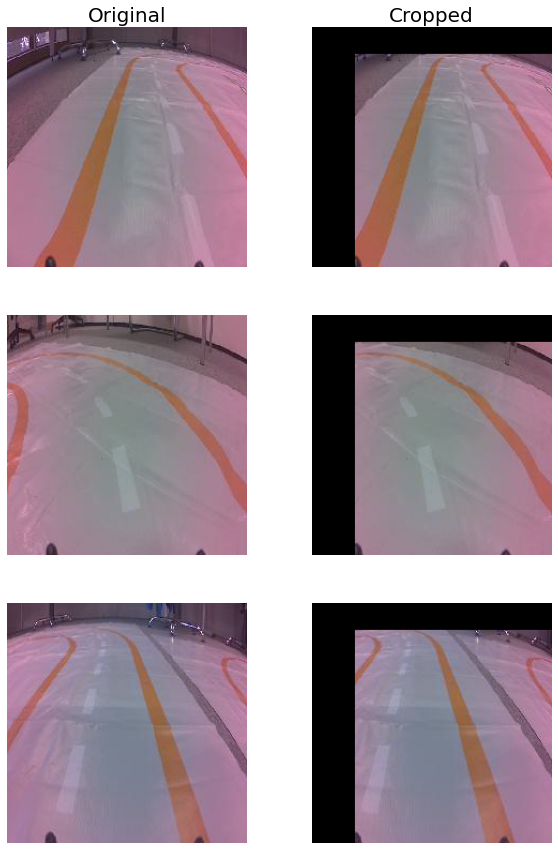


Figure 4: example of bad cropping (source: author)

That circles back to point that feature engineering will need to be done in production. Here you do not know what image will be shown to the model at what time. This means the same cropping function will need to be applied to all images. You need to ensure that it will never remove important parts of the image.

**Grayscale**

For some applications, the colour of the image is not important. In this case, we can grayscale the image. We do this with the **cvtColor** function from OpenCV (line 2).

#Gray scale  
gray\_img = cv2.cvtColor(img,cv2.COLOR\_RGB2GRAY)

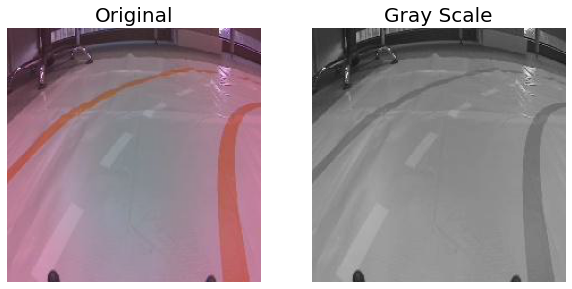


Figure 5: gray scale of image (source: author)

Grayscaling works by capturing the colour intensity in an image. It does this by taking a weighted average of the RGB channels. Specifically, the function above uses this formula:

**Y** = 0.299\***R** + 0.587\***G** + 0.114\***B**

We can understand the benefits of this by looking at the number of input values for each image. If we used all RGB channels it would consist of **150,528** values (224\*224\*3). For grayscale images, we now only have **50,176** values (224\*224). The simpler input means we need less data and simpler models.

**RGB channels**

One of the channels may be more important. Instead of grayscaling, we can use only that channel. Below, we select the R (line 6), G (line 7) and B (line 8) channels. Each of the resulting arrays will be of dimension 224 x 224. You can see the respective images in **Figure 6**.

#Load image  
img = Image.open(img\_paths[700])  
img = np.array(img)  
  
#Get rgb channels  
r\_img = img[:, :, 0]  
g\_img = img[:, :, 1]  
b\_img = img[:, :, 2]

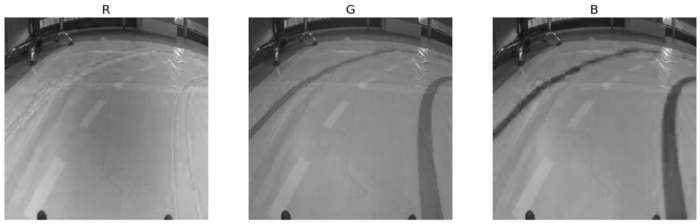


Figure 6: RGB channels (source: author)

You can also use the **channel\_filter** function. Here the channel parameter (c) will take on the values 0,1 or 2 depending on which channel you want. Keep in mind that some packages will load channels in different orders. We are using PIL which is RGB. However, if you load images with cv2.imread() the channels will be in BGR order.

def channel\_filter(img,c=0):  
 """Returns given channel from image pixels"""  
 img = np.array(img)  
 c\_img = img[:, :, c]  
  
 return c\_img

With these transformations, you need to think about whether you are removing important information from the image. For our application, the track is orange. In other words, the colour of the track can be helpful in distinguishing it from the rest of the image.

**Intensity threshold**

With grayscaling, each pixel will have a value between 0 and 255. We can simplify the input even further by transforming it into binary values. If the grayscale value is above a cutoff the pixel value is 1 otherwise it is 0. We call this cutoff an intensity threshold.

The function below is used to apply this threshold. We first grayscale the image (line 5). If the pixel is above the cutoff it is given a value of 1000 (line 8). If we set the pixel to 1 it would be below the cutoff. In other words, all the pixels would be set to 0 in the next step (line 9). Finally, we scale all the pixels so they take on a value of either 0 or 1 (line 11).

def threshold(img,cutoff=80):  
 """Apply intesity thresholding"""  
   
 img = np.array(img)  
 img = cv2.cvtColor(img,cv2.COLOR\_RGB2GRAY)  
   
 #Apply cutoff  
 img[img>cutoff] = 1000 #black  
 img[img<=cutoff] = 0 #white  
   
 img = img/1000  
  
 return img

A part of the automated car project was to avoid obstacles. These were tins painted black. In **Figure 7**, you can see how applying the intensity threshold function we can isolate the tin from the rest of the image. This is only possible because the tin is black. In other words, its intensity is greater than the rest of the image.

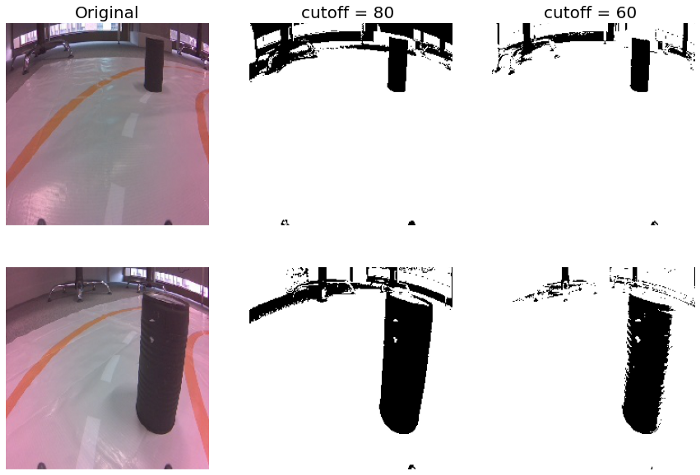


Figure 7: feature engineering with an intensity threshold (source: author)

The cutoff can be treated as a hyperparameter. Looking at Figure 7, a larger cutoff means we include less background noise. The downside is we capture less of the tin can.

**Edge detection**

If we want to isolate the track, we could use canny edge detection. This is a multiple-stage algorithm used to detect edges in images. If you want to understand how it works I suggest reading

[Sofiane Sahir](https://medium.com/u/4c917479ce84?source=post_page-----14fe4fcd4353--------------------------------)

’s article on [canny edge detection](https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123).

We apply the algorithm with the **cv2.Canny**() function. The **threshold1** and **threshold2** parameters are for the hysteresis procedure. This is the final process of the edge detection algorithm and it is used to decide which lines are actually edges.

#Apply canny edge detection  
edge\_img = cv2.Canny(img,threshold1 = 50, threshold2 = 80)

You can see some examples in **Figure 8**. Like with intensity thresholding, we are left with a binary map — white for edges and black otherwise. The hope is that the track is now easier to distinguish from the rest of the image. However, you can see that edges in the background are also detected.

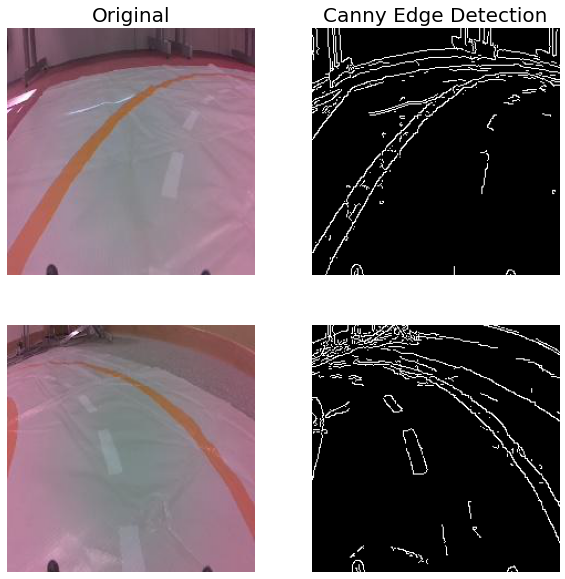


Figure 8: canny edge detection (source: author)

**Colour filter**

We may have better luck by isolating the track using pixel colour. We do this using the **pixel\_filter** function below. Using **cv2.inRange**() we convert the image to a binary map (line 10). This function checks if a pixel falls within the range given by the lower (line 5) and upper (line 6) lists. Specifically, each RGB channel must fall within the respective range (e.g. 134-t ≤ R ≤ 194+t).

def pixel\_filter(img, t=0):  
   
 """Filter pixels within range"""  
   
 lower = [134-t,84-t,55-t]  
 upper = [192+t,121+t,101+t]  
  
 img = np.array(img)  
 orange\_thresh = 255 - cv2.inRange(img, np.array(lower), np.array(upper))  
  
 return orange\_thresh

Simply, the function determines if the pixel colour is close enough to the orange colour of the track. You can see the results in **Figure 9**. The t parameter introduces some flexibility. With higher values, we can capture more of the track but retain more noise. This is because pixels in the background will fall within the range.

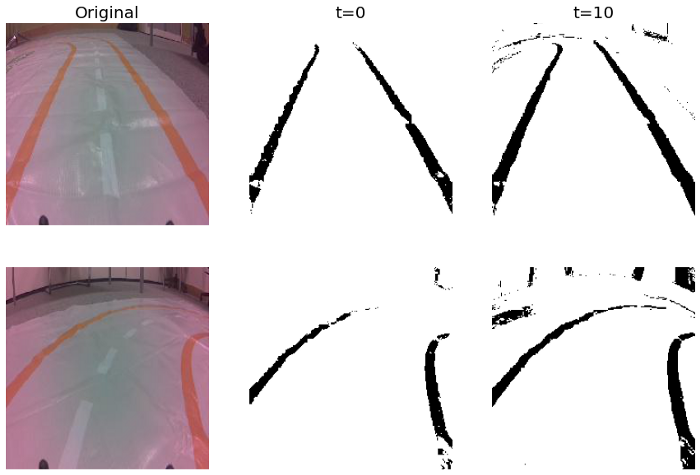


Figure 9: filtering orange pixels (source: author)

You may be wondering where we get the lower and upper bounds from. That is how do we know the channels will fall between [134, 84, 55] and [192, 121, 101]? Well, we use a colour-picker created with Python. We outline how this is created in the article below.

**[Building a Color Picker with Python](https://towardsdatascience.com/building-a-color-picker-with-python-55e8357539e7" \t "_blank)**

**[Creating a tool that selects the RGB channels from image pixels](https://towardsdatascience.com/building-a-color-picker-with-python-55e8357539e7" \t "_blank)**

[towardsdatascience.com](https://towardsdatascience.com/building-a-color-picker-with-python-55e8357539e7" \t "_blank)

In **Figure 10**, you can see the picker in action. We select pixels from multiple images and try to select them at different locations on the track. This is so we get a full range of pixel values across different conditions.

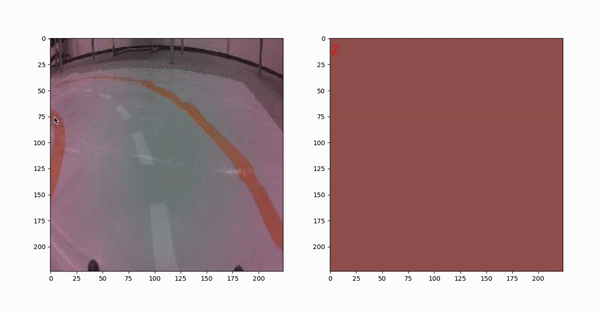


Figure 10: picking colours from the track (source: author)

We selected 60 colours in total. You can see all these in **Figure 11** (with a bonus visual illusion). The RGB channels of all of these colour are stored in a list — “colours”.

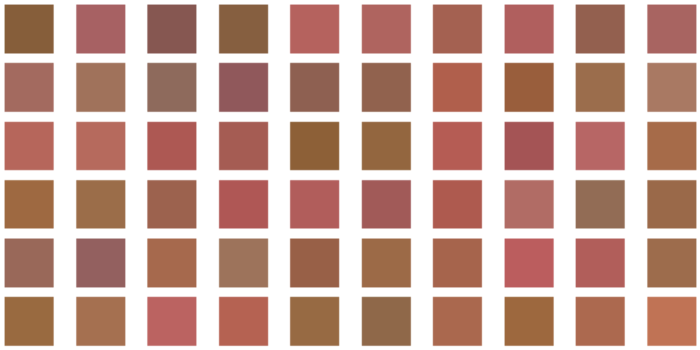


Figure 11: 60 track pixels selected with colour picker

Finally, we take the minimum and maximum values for each of the RGB channels. This gives the lower and upper bounds.

lower = [min(x[0] for x in colours),  
 min(x[1] for x in colours),  
 min(x[2] for x in colours)]  
  
upper = [max(x[0] for x in colours),  
 max(x[1] for x in colours),  
 max(x[2] for x in colours)]

**Limitations of feature engineering**

After all this, you may not be convinced. A major benefit of deep learning is that it can identify complex patterns without the need for feature engineering. This is a good point.

Feature engineering requires critical thinking. You need to figure out what aspects of the images are important. You then need to write code to extract these aspects. The time required to do all of this makes no sense for some applications.

Additionally, for some methods, we’ve seen that we were not able to eliminate all noise. For example, the black background in the intensity thresholding. Counterintuitively, the noise that remains may now be even harder to distinguish from what is important. That is the remaining noise and object pixels have the same value (1).

Really, the benefit comes when dealing with relatively simple computer vision problems. Our track never changes and the objects are always the same colour. For more complex problems, you will need more data. Alternatively, you can fine-tune a pre-trained model on a smaller dataset.

I hope you enjoyed this article! You can support me by becoming one of my [**referred members**](https://conorosullyds.medium.com/membership) **:)**

**[Join Medium with my referral link — Conor O’Sullivan](https://conorosullyds.medium.com/membership" \t "_blank)**

**[As a Medium member, a portion of your membership fee goes to writers you read, and you get full access to every story…](https://conorosullyds.medium.com/membership" \t "_blank)**

[conorosullyds.medium.com](https://conorosullyds.medium.com/membership" \t "_blank)

| [Twitter](https://twitter.com/conorosullyDS) | [YouTube](https://www.youtube.com/channel/UChsoWqJbEjBwrn00Zvghi4w) | [Newsletter](https://mailchi.mp/aa82a5ce1dc0/signup) — sign up for FREE access to a [Python SHAP course](https://adataodyssey.com/courses/shap-with-python/)

**Image Sources**

All images are my own or obtain from [www.flaticon.com](http://www.flaticon.com/). In the case of the latter, I have a “Full license” as defined under their [Premium Plan](https://support.flaticon.com/hc/en-us/articles/202798201-What-are-Flaticon-Premium-licenses-).

**Dataset**

**JatRacer Images** (CC0: Public Domain) <https://www.kaggle.com/datasets/conorsully1/jatracer-images>

**References**

OpenCV, **Color conversions** <https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html>