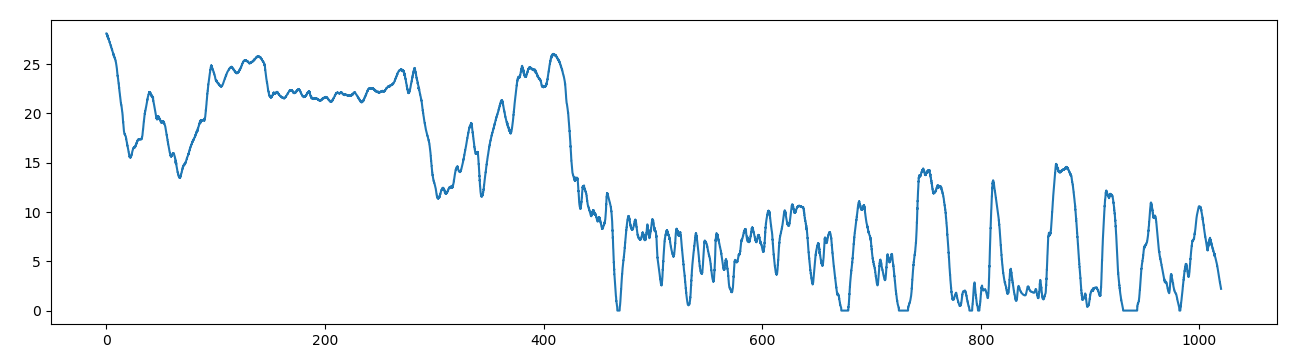
# Speed data

Watching the videos for train and test, I found that the distribution of data between the two is not identical. Seems like test has more instances of stopped car seeing cars drive across its view. This could be trouble for classic train>test scenario. My plan, view the test video, find sections which are very different from train, manually annotate, and train on that data, this is most important for the stopped car is fast cars crossing its view, so the 0mph cases, which are also easy to annotate.

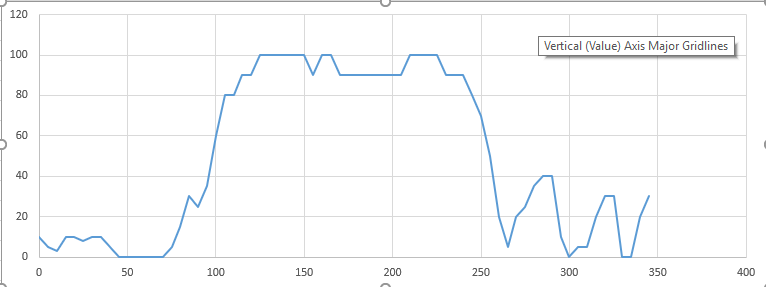
## Train

MPH



## Test

Several trials of watching the video and writing down the speeds estimated. Km/h



the

# data preparation

batch shuffling from <https://chatbotslife.com/autonomous-vehicle-speed-estimation-from-dashboard-cam-ca96c24120e4>

# Top level model

A regressor that looks at 2 sequential images and predicts the ground speed of the vehicle taking the images.

Initial concept: gray scaleCrop>edge detection>concatenate (previous frame, current frame)>2D Conv>(maybe some stuff like pooling or other>(maybe more conv)>dense layer>speed value.

Then after some quick search for reasonable 2dConv net architectures as a starting point, I found “Vehicle Motion Detection using CNN”, by Yaqi Zhang, which led me to cv2.calcOpticalFlowFarneback, which I will use in place of the edge detection.

Concept 2: grey scale>crop>edge detection>cv2.calcOpticalFlowFarneback>2dConv> dense layer>speed value. This is very simple but I gotta start somewhere.

I’ll setup the keras as follows. Custom layer>standard layer>standard layer

# Custom layer

## Cropping frames



The bottom of the image is the dash, it’s a waste of compute resources to run that through a train and test workflow, so crop the bottom off.

Trial and error led to imgc0 = img[0:350, 0:639],



The top of the image is typically full of sky or other stuff we don’t need.

imgc0 = img[100:350, 0:639]



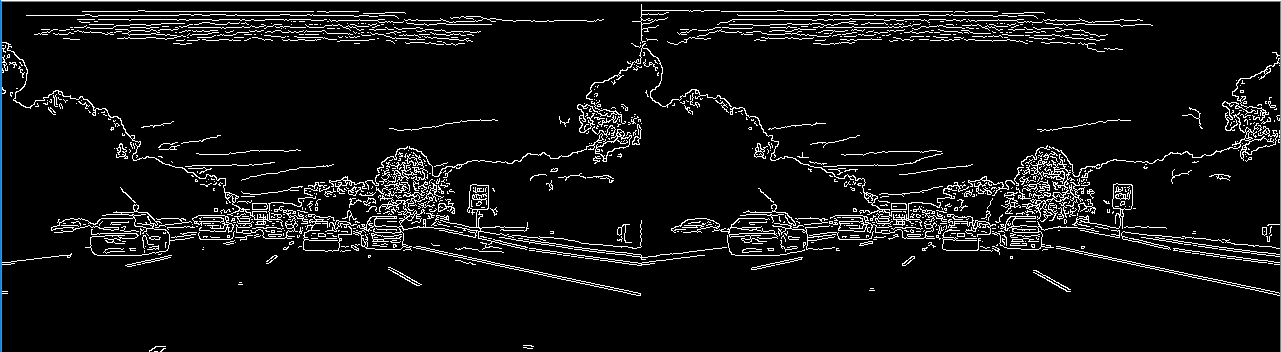
We’ll try this.

## Edge detection

### Canny

2 parameters. Could be hyperparameters, but I think with a bit of trial and human review, I can get a reasonable starting point.

Using 20,50. We’re getting sky, which is useless.



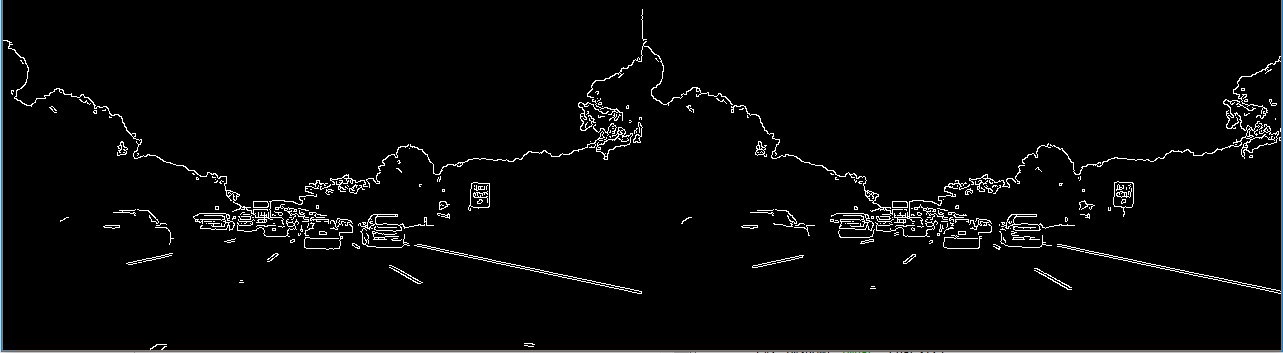
Using 50,100. Not bad, getting road side features.



75,150. Happy medium, lets start with this.



100,200. Not bad either, but less road side features, might be a good thing?



200,400. Not enough stuff left. If lane lines were always present, this might do, but I can rely on that.



I’ll start with 75, 150.

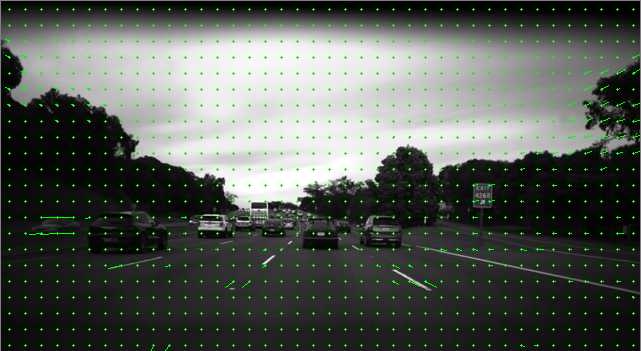
Might need to adjust these parameters based on image lighting or estimated speed, or maybe even edge pixel density (i.e. I make a target “white pixel density” of say 3%, and adjust parameters to approach this).

### HED

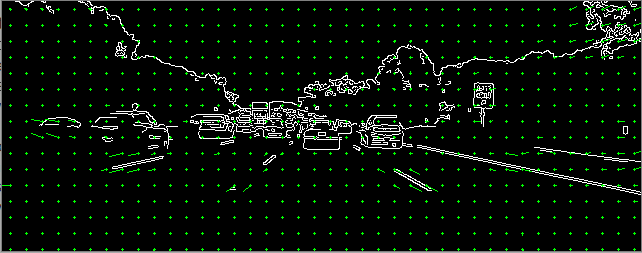
Toolchain not installed yet.

## Optical Flow

Default settings from opencv example opt\_flow.py



Maybe this also works well with images of canny edges? Looks like it does



## Developing the functions which make up the custom keras layer

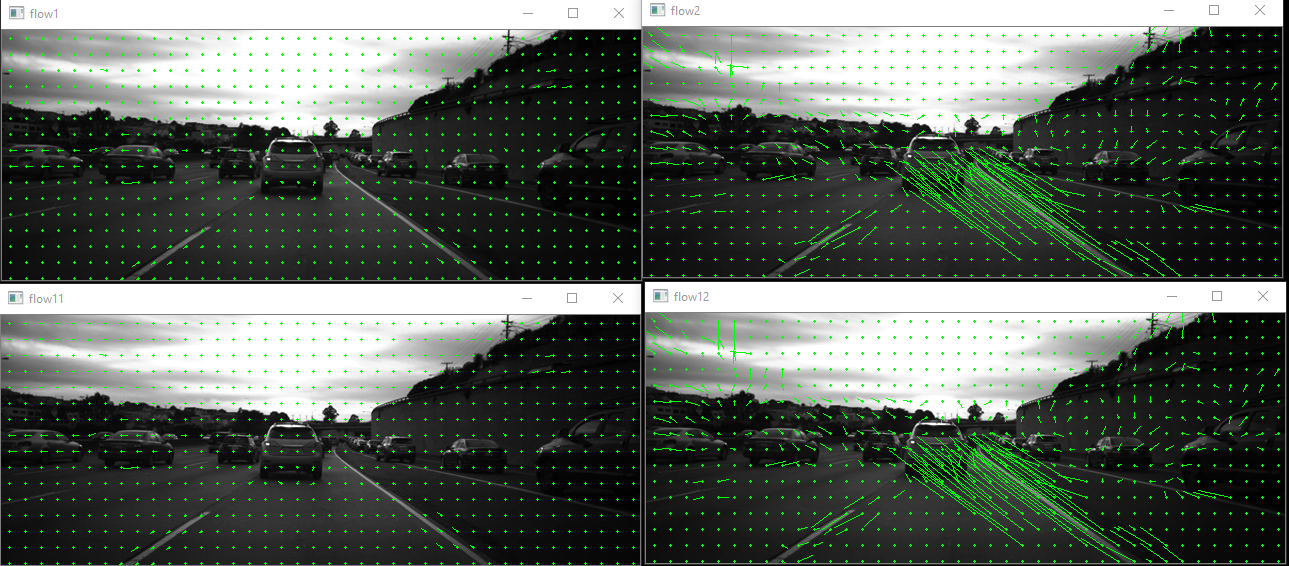
Stack looks like this:

* CustomLayer
  + image\_tensor\_func
    - cv2\_preprocess\_tnsr\_fcn

I have test function for cv2\_preprocess\_tnsr\_fcn. These were last seen here: 51c3423ccdb6a99f883b4bb6dd8f81e3e938f4db

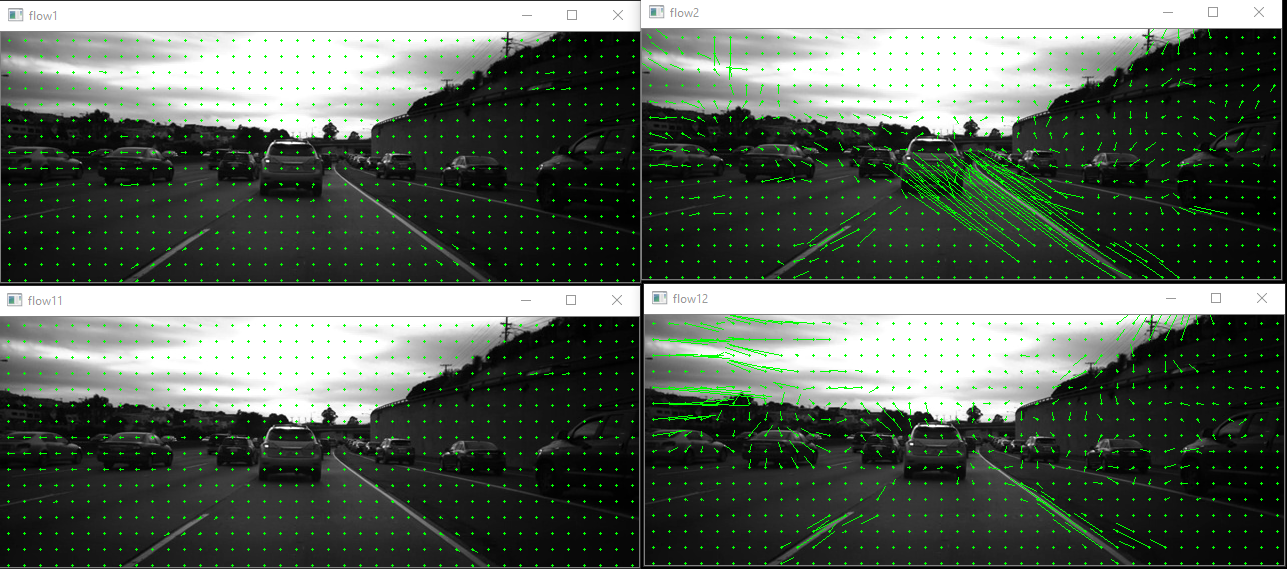
I then removed cv2\_preprocess\_old and this section of main, 3] DEEVELOPING PREPROCESSING TECHNIQUE at: f38c3085a5165b5100f35159aa1b94882cc9b8f1

At this point, I’m getting this output:



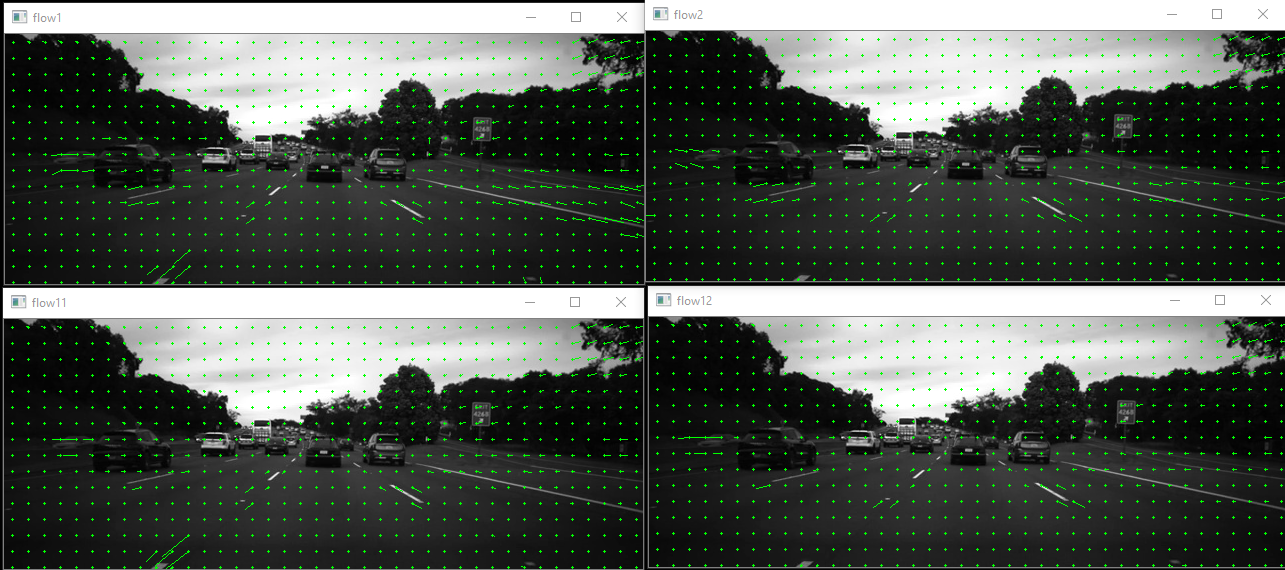
This doesn’t exude confidence that the preprocessing pipeline is working as desired, but at least the 2 versions of it are doing the same thing, so at least I have some control of what’s happening.

Try removing the Canny Edge step from the function creating the lower row of images.



I think it’s fair to say that the lower set without edge detection upstream of flow detection is better, well for these 2 image sets at least. Could be that the edge detection was poorly tuned for these image sets. Whats more troubling is that the flow in each column is very different. The left column is frame10898.jpg and frame10899.jpg, while the right column is frame10899.jpg, frame109.jpg. so there is the cause of the issue. Need to figure out why the original dataframe is returning some rows with sequential images, and some rows with non-sequential images.

Fixed that.



Now, we can see that its fair to say the upper row shows more instance of speed relative optical flow. Lets start with this.

709459ef7ca053433b51cecdef1aae40b042d760

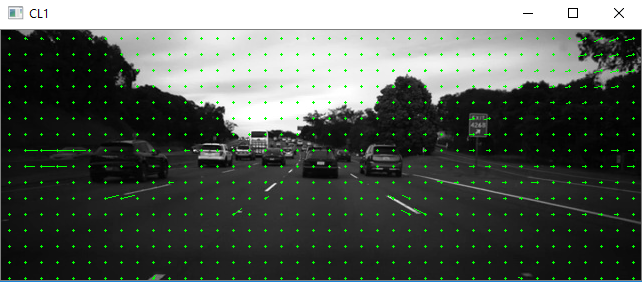
Tests with custom layer resulted in input output size mismatch. Input expected size =[dim1,dim2,dim3,dim4]:

* dim1=sequences of images or images sets [1:inf]
* dim2 = image num pixels height, 250 for this project
* dim3 = image num pixels width, 639 for this project
* dim4 = image depth/layers(e.g. RBG, hsv, optical flow) in this project, the 2 layers will be grey scale image now, and grey scale image previous.

Input expected type = float64.

Clearly the dataframe doesn’t have this shape or type. Solution: make a generator which produces a input of this shape and type from the dataframe.

Made a prototype naive generator, with a test feature for batch\_size==2, and ran test on custom layer. Cv.Canny function was giving error. The below image is without Canny edge dection.



Below image produced using generator prototype with batch\_size=3

