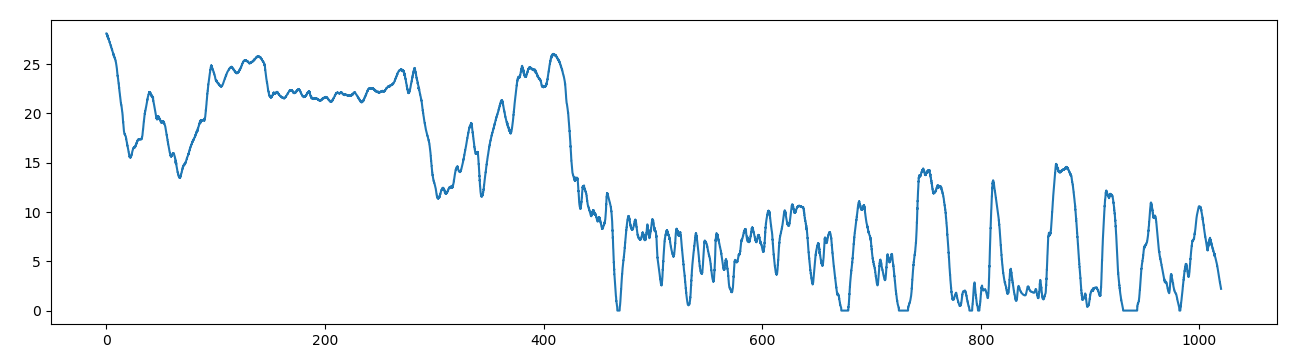
# Speed data

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.

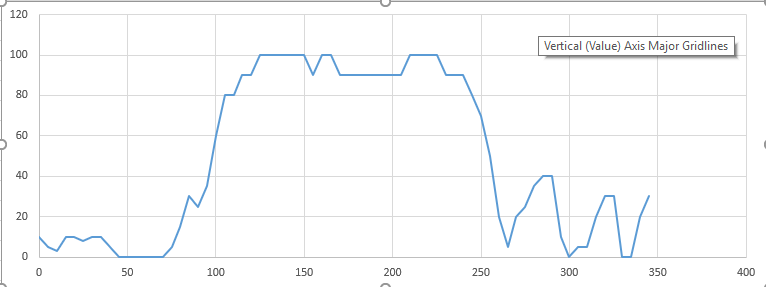
## Train

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>cv2.calcOpticalFlowFarneback>2dConv> dense layer>speed value. This is very simple but I gotta start somewhere.

## 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]



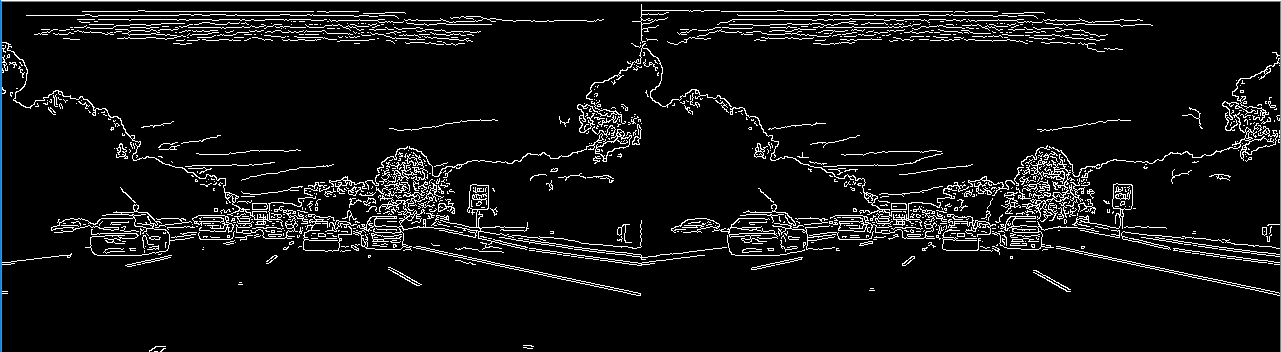
We’ll try this.

## 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.



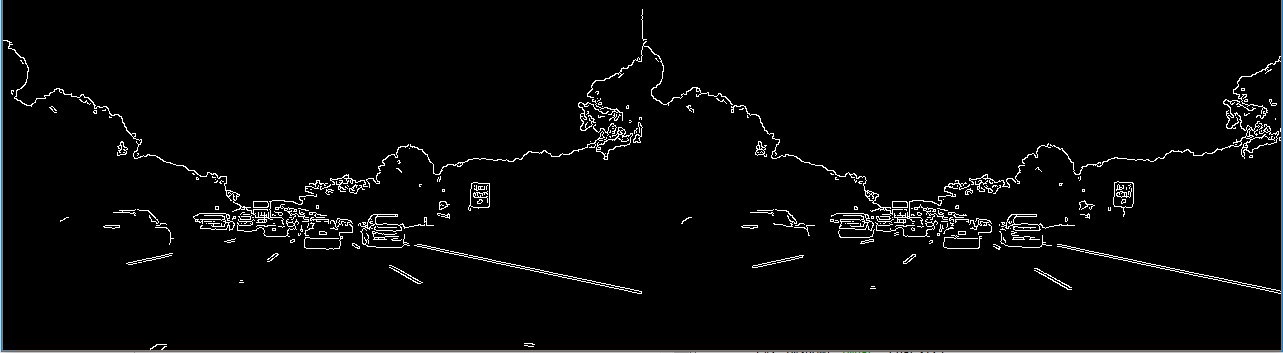
Using 50,100. Not bad, getting road side features.



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.

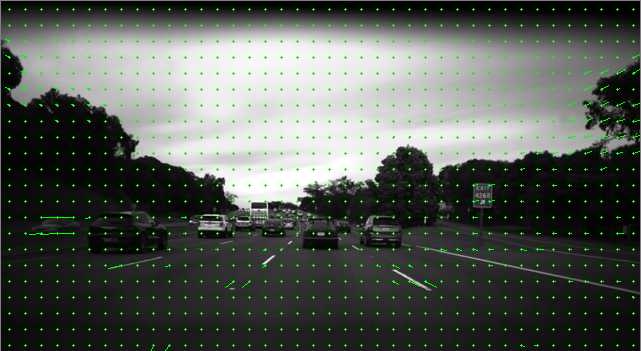
Might need to adjust these parameters based on image lighting or estimated speed, or maybe even edge pixel density (i.e. I make a target “white pixel density” of say 3%, and adjust parameters to approach this).

### HED

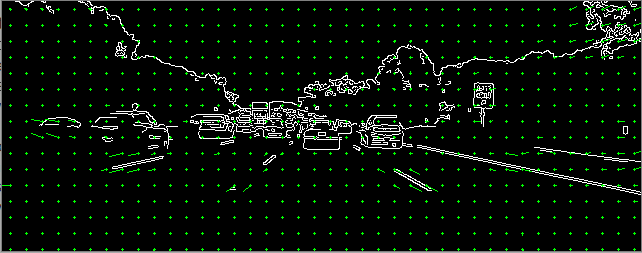
Toolchain not installed yet.

## 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