AUTONOMOUS VEHICLE CODE

DATA CAPTURE, TRAINING THE MODEL, DEPLOY

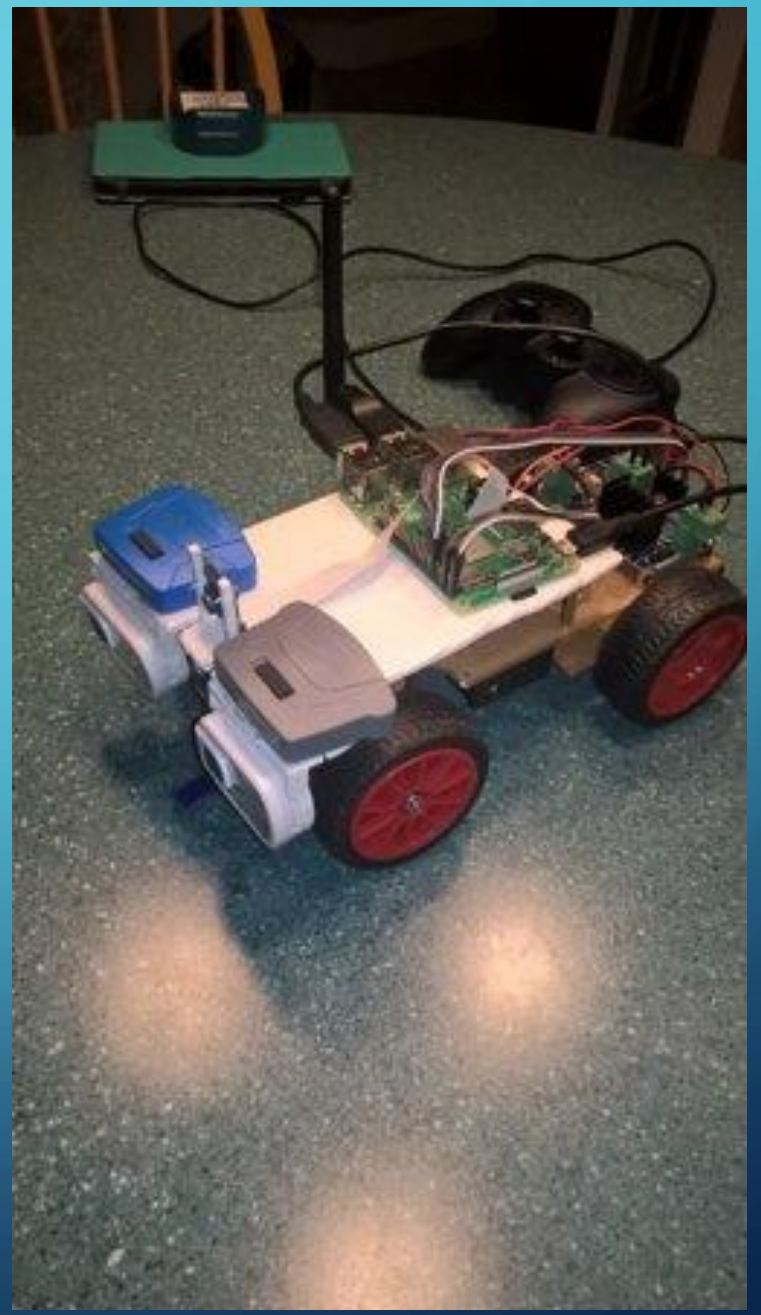
Gene Olafsen

CAPTURE

- Car and Gamepad

THE PLATFORM

- Car
- Gamepad



IO LIBRARY AND GAMEPAD CONTROLLER

- The EVDEV package provides bindings to the generic input event interface in Linux. The evdev interface serves the purpose of passing events generated in the kernel directly to userspace through character devices that are typically located in /dev/input/.

```
from evdev import InputDevice, categorize, ecodes

gamepad = InputDevice('/dev/input/event0')
```

SETUP COMMON

- A call is made to the common library to initialize the servo library and establish camera parameters.

```
def setup():
    global pwm
    pwm = servo.PCA9685()
    pwm.set_pwm_freq(60)
    global camera
    camera = PiCamera(resolution=(FRAME_W,FRAME_H))
    camera.video_stabilization = False
    camera.framerate = 30
```

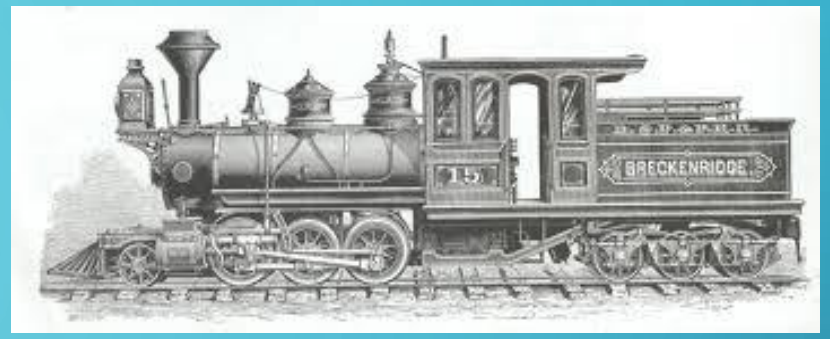
NAME THE TRIAL

- A naming convention is important to organize the captured test and validation data.
- R#CW
- R#CW_V
- R#CCW
- R#CCW_V

MORE SETUP

```
def setup_capture():
    os.system('mkdir ' + CAPTURE_DIR)
    os.system('mkdir ' + CAPTURE_DIR + '/' + str(TRIAL))
    os.system('mkdir ' + CAPTURE_DIR + '/' + str(TRIAL) + '/image')
    os.system('mkdir ' + CAPTURE_DIR + '/' + str(TRIAL) + '/steer')

def setupMotor(busnum=None):
    pwm.frequency = 60
    forward0 = 'True'
    forward1 = 'True'
    GPIO.setwarnings(False)
    GPIO.setmode(GPIO.BOARD)
```

TRAIN



- Machine learning works by finding a relationship between its features (input) and a label (output). The model is 'shown' examples from a dataset. Each example helps define how each feature affects the label.

KERAS TRAINING

- Keras provides three functions that can be used to train deep learning models:
  - .fit
  - .fit_generator
  - .train_on_batch

FIT

- Trains the model for a given number of epochs (iterations on a dataset).
- This command looks similar to the syntax used in scikit learn library
- Accepts training data (x) and label information (y)
- Defaults are shown below:

```
fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
validation_split=0.0, validation_data=None, shuffle=True,
class_weight=None, sample_weight=None, initial_epoch=0,
steps_per_epoch=None, validation_steps=None)
```

LIMITATIONS TO BEING 'FIT'

- The call to .fit makes two primary assumptions:
  - The *entire* training set can fit into RAM
  - There is *no* data augmentation going on (i.e., there is no support for Keras generators)
  - Why is data augmentation important?... Data augmentation is a form of regularization and it helps avoid overfitting and increases the ability of the model to generalize.

ADVANTAGES OF BEING 'FIT'

- The perfect choice for small data sets.
- Batching of data is handled automatically by 'fit' at the cost of having to hold the training set in memory.

FIT_GENERATOR

- Trains the model on data generated batch-by-batch by a Python generator (or an instance of Keras.utils.Sequence).

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1,
callbacks=None, validation_data=None, validation_steps=None,
class_weight=None, max_queue_size=10, workers=1,
use_multiprocessing=False, shuffle=True, initial_epoch=0)
```

ADVANTAGE FIT_GENERATOR

- The generator is run in parallel to the model, for efficiency. For instance, this allows for real-time data augmentation on images on CPU in parallel to training your model on GPU.
- The use of keras.utils.Sequence guarantees the ordering and guarantees the single use of every input per epoch when using use_multiprocessing=True.

FIT_GENERATOR YEILD

- Keep in mind that a Keras data generator is meant to loop infinitely — it should never return or exit.
- Since the function is intended to loop infinitely, Keras has no ability to determine when one epoch starts and a new epoch begins.
- Therefore, compute the steps_per_epoch value as the total number of training data points divided by the batch size. Once Keras hits this step count it knows that it's a new epoch.

TRAIN_ON_BATCH

- Offers fine-grain control over the Keras training process. The function accepts a single batch of data, performs backpropagation, and then updates the model parameters

```
train_on_batch(x, y, sample_weight=None, class_weight=None)
```

EPOCH, BATCH, ITERATION

- Epoch - Is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE.
- Batch Size - Total number of training examples present in a single batch.
- Iteration - The number of batches needed to complete one epoch.

IMAGAUG

- A python library that helps with augmenting images for machine learning projects. It converts a set of input images into a new, much larger set of slightly altered images.
- https://github.com/aleju/imgaug

IMAGAUG CODE

```
sometime = lambda aug: iaa.Sometimes(0.3, aug)
sequence = iaa.Sequential([
    sometime(iaa.GaussianBlur((0, 1.5))),
    sometime(iaa.Sharpen(alpha=(0, 1.0), lightness=(0.75, 1.5))),
    sometime(iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 3.), per_channel=0.5)),
    sometime(iaa.Dropout((0.0, 0.1))),
    sometime(iaa.CoarseDropout((0.10, 0.30), size_percent=(0.02, 0.05), per_channel=0.2)),
    sometime(iaa.Add((-10, 10), per_channel=0.5)),
    ],
    random_order=True # do all of the above in random order
    )
```

NORMALIZE/EQUALIZE

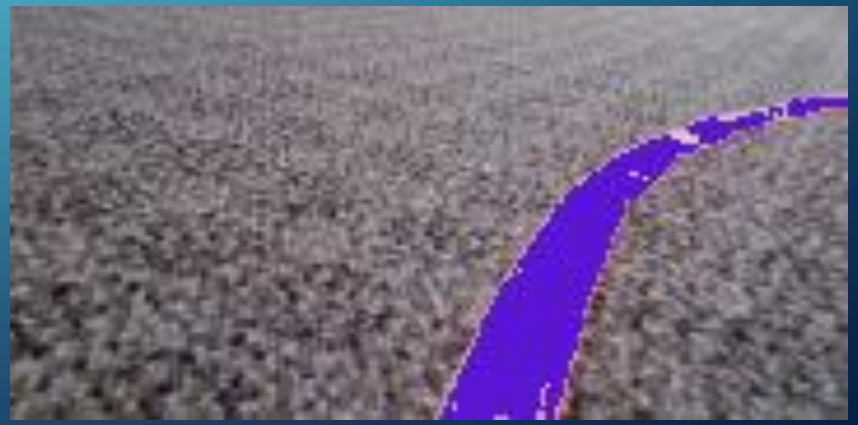
- Typically improves the image contrast by spreading pixel values across the full range of the picture.
- https://docs.opencv.org/3.2.0/d5/daf/tutorial_py_histogram_equalization.html

NORMALIZE/EQUALIZE CODE

```
def equalize(image):
    norm=np.zeros((image.shape), np.float32)
    norm[:,:,0]=cv2.equalizeHist(image[:,:,0])
    norm[:,:,1]=cv2.equalizeHist(image[:,:,1])
    norm[:,:,2]=cv2.equalizeHist(image[:,:,2])
    return norm

def normalize(image):
    image = image - np.mean(image, axis=(0,1))
    image = image / np.std( image, axis=(0,1))
    return image
```

RECOLOR LINE

- Well this is quite a training-data specific data augmentation routine! By sampling the RGB values that represent the 'training line color' the routine re-colors 'most' of the line with a random color.
- Note: It could be better if it varied the target color in some way proportional to the characteristics of the underlying color.



RECOLOR CODE

- 70% of the time the image is altered with this code.

```
def recolor_line(data):
    color = list(np.random.choice(range(256), size=3))
    # generate a random replacement color
    blue, green, red = data[:,:,0], data[:,:,1], data[:,:,2]
    mask = (red > 200) & (red < 230) & (green > 150 ) & (green < 195) & (blue
> 150) & (blue < 195)
    data[:,:,:3][mask] = color
    return(data)
```

IMAGE SHIFT

- 50% of the time, the image is altered with this code.

```
    if np.random.random() > 0.5:
        shift = np.random.randint(-2,2)

        if shift > 0:
            image[-shift:,:,:] = 0
            image[:-shift,:,:] = image[shift:,:,:]
        elif shift < 0:
            image[:-shift,:,:] = 0
            image[-shift:,:,:] = image[:shift,:,:]
```

IMAGE FLIP

- 50% of the time, the image is altered with this code.

```
    if np.random.random() > 0.5:
        image = cv2.flip(image, 1)
        steer = -steer
```

COMPILE THE MODEL

- Model is instructed to use a custom loss function.

```
minimizer = SGD(lr=0.0001)

#model.compile(loss=custom_loss, optimizer=minimizer)
model.compile(loss=custom_loss, optimizer='adam')
#model.compile(loss=losses.mean_squared_error, optimizer='adam')
```

CUSTOM LOSS FUNCTION(S)

```
def custom_loss(y_true, y_pred):

    # custom loss 1
    #y_pred = tf.clip_by_value(y_pred, -1+1e-7, 1-1e-7)
    #loss = -((1. - y_true) * tf.log(1. - y_pred) + (1. + y_true) * tf.log(1.
+ y_pred))
    #loss = tf.reduce_mean(loss)

    # custom loss 2
    loss = tf.square(y_true - y_pred)
    loss = .5 * tf.reduce_mean(loss)

    # custom loss 3
    #loss = tf.abs(y_true - y_pred)
    #loss = tf.reduce_mean(loss)

    return loss
```

FIT_GENERATOR

- assdf

```
history = model.fit_generator(generator = gen_train.get_gen(),
                    steps_per_epoch = num_train,
                    epochs  = 10,
                    verbose = 1,
                    validation_data = gen_valid.get_gen(),
                    validation_steps = num_valid,
                    callbacks = [early_stop, checkpoint, tensorboard],
                    max_q_size = 3)
```

THE MODEL

```
image_inp = Input(shape=(FRAME_H, FRAME_W, 3))

x = Conv2D(filters=8,  kernel_size=(5, 5), activation='relu')(image_inp)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(filters=16, kernel_size=(5, 5), activation='relu')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(filters=32, kernel_size=(5, 5), activation='relu')(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)
x = Dropout(.5)(x)
x = Dense(128, activation = 'relu')(x)
x = Dropout(.5)(x)
x = Dense(1, activation='tanh')(x)

angle_out = x
```

KERNEL SIZE

- Best practices for selecting kernel size

FILTER SELECTION

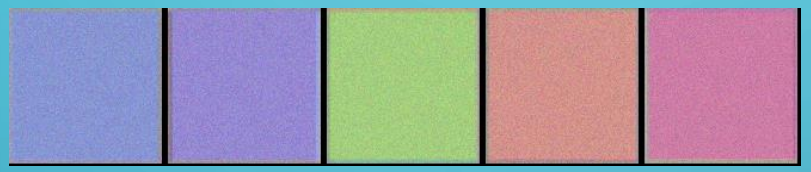
- Best practices for selecting filter parameters

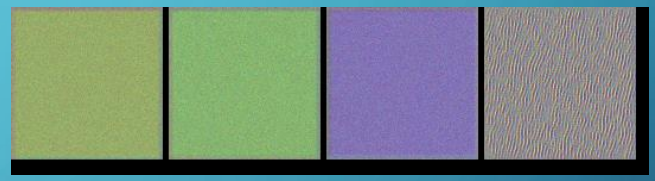
FILTER IMAGES

Training without Re-color

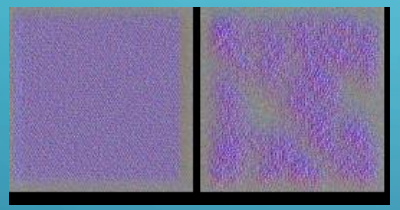
Re-color Training

- CONV2D – 1

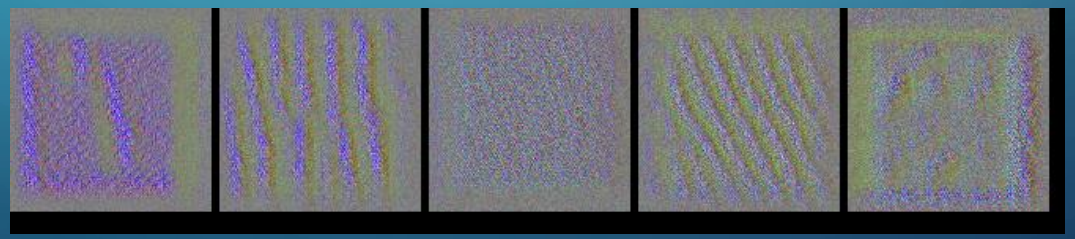




- CONV2D – 2



- CONV2D – 3



DETERMINING THE FILTER PARAMETER

- There is no easy way to choose the number of filters. I'd like to share one of my favorite sanity checks on the number of filters. It takes 2 easy steps:
- Try to overfit at 500 random images with regularization.
- Try to overfit at the whole dataset without any regularization.
- Usually - if the number of filters is too low (in general) - these two tests will show you that. If - during your training process - with regularization - your network severely overfits - this is a clear indicator that your network has way too many filters.

https://stackoverflow.com/questions/48243360/how-to-determine-the-filter-parameter-in-the-keras-conv2d-function

REGULARIZATION

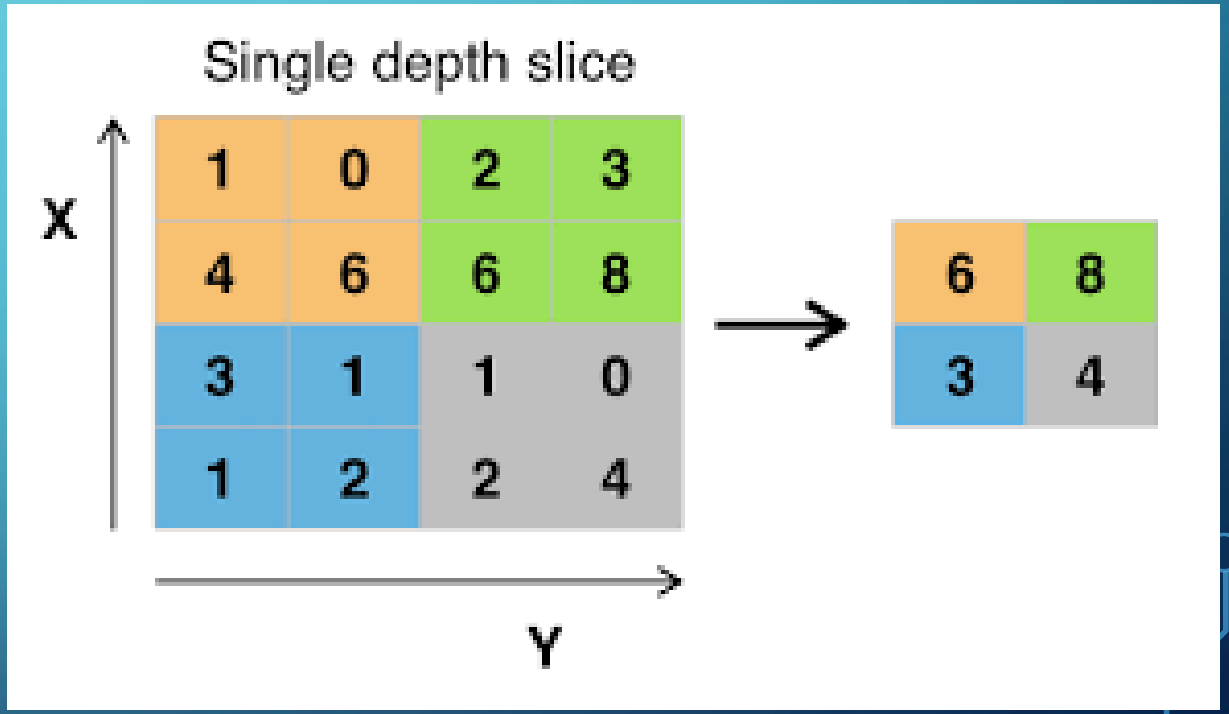
- Kera Regularization options

POOLING LAYERS

- The Keras.layers.MaxPool2d method is used.
- Reduces the dimensionality by a factor of 4.
- The pooling layer is a method to tell the neural-network to expect some invariance in the position of features within the instance (video frame) that is being processed.

WHERE TO PUT A POOLING LAYER

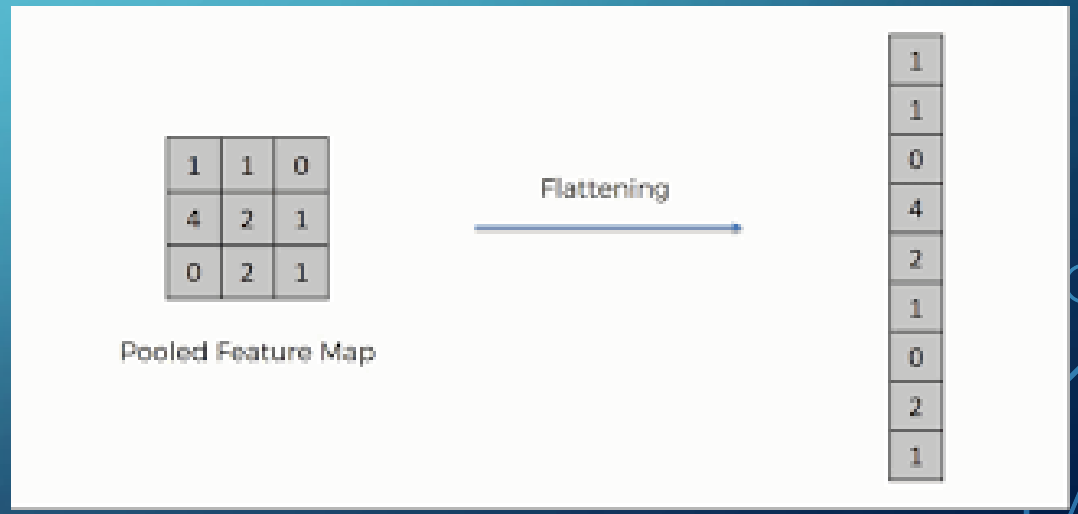
- the common way the pooling layer is used is after a convolution layer. In this case, the convolution layer tries to identify some important structure in some region of the data while the pooling layer obscures the exact location of this structure.



FLATTEN LATER

- Keras.layers.Flatten reduces the tensor to a single dimension whose shape is equal to the number of elements of the tensor.
- ImageH * ImageW * rgb = 90*180*3 = 48,600

```
keras.layers.Flatten(data_format=None)
```

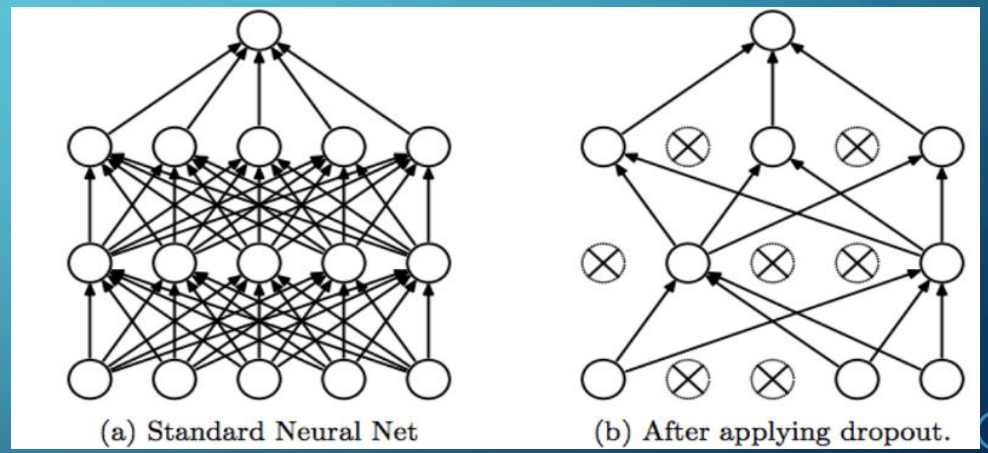


FLATTEN'S POSITION IN THE MODEL

- The last stage of a Convolution Neural Network (CNN) is a classifier.
- Flatten takes the output of the CNN and 'flattens' its structure to create a single long feature vector. This dense representation is the input to the Artificial Neural Network (ANN) classifier.
- The flattening step is needed to make use of fully connected layers after the convolutional layers. Fully connected layers don't have a 'local' limitation like convolutional layers (which only observe some local part of an image by using convolutional filters)

DROPOUT LAYER

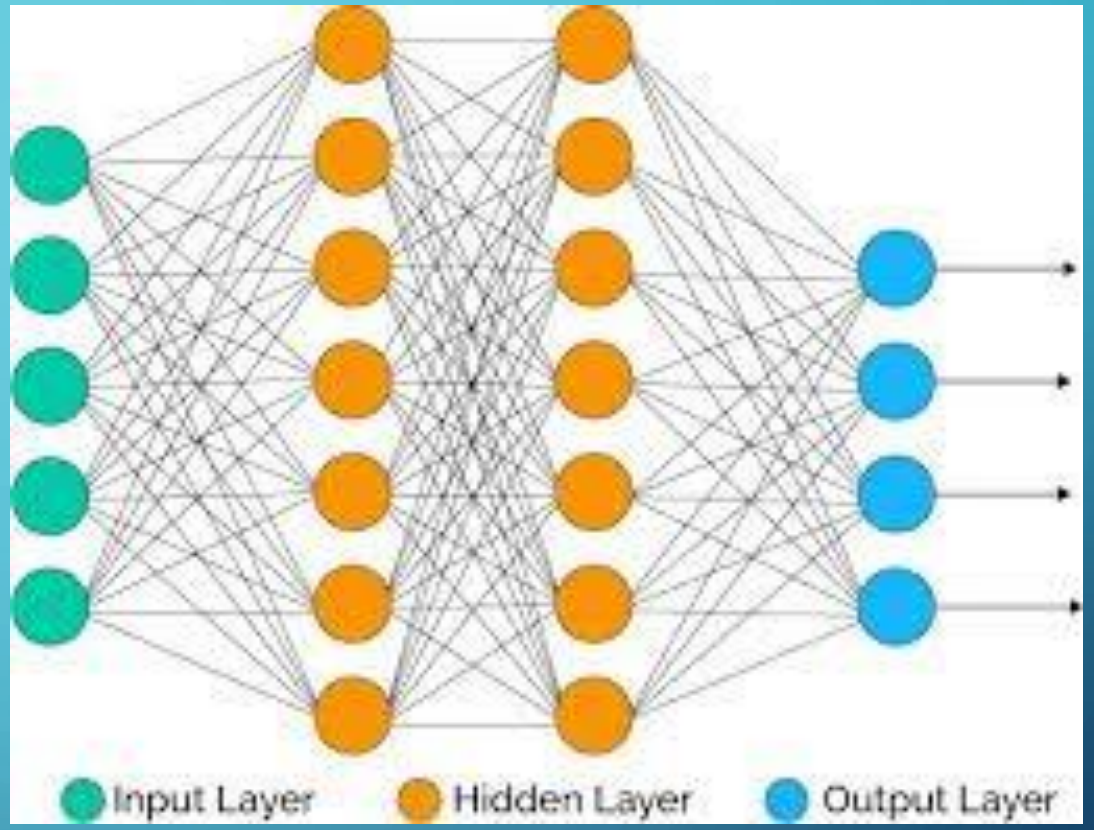
- Keras.layers.Dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random.
- Dropout is a regularization technique, which aims to reduce the complexity of the model with the goal to prevent overfitting
- Dropout curbs the individual power of each neuron. It helps reduce a co-dependency relationship between neurons.




```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

DENSE LAYER

- Keras.layers.Dense - A dense layer is simply a layer where each unit or neuron is connected to each neuron in the next layer




```
keras.layers.Dense(units, activation=None, use_bias=True,
kernel_initializer='glorot_uniform', bias_initializer='zeros',
kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
kernel_constraint=None, bias_constraint=None)
```

INSTANTIATE A MODEL OBJECT

- Define the model input and output.
- Display a summary of the model configuration.

```
model = Model(inputs=[image_inp], outputs=[angle_out])

model.summary()
```

KERAS CALLBACKS

- A callback is a set of functions to be applied at given stages of the training procedure. Use callbacks to get a view on internal states and statistics of the model during training.

```
early_stop  = EarlyStopping(monitor='val_loss', min_delta=0.001, patience=2,
mode='min', verbose=1)

checkpoint  = ModelCheckpoint('weights.hdf5', monitor='val_loss', verbose=1,
save_best_only=True, mode='min', period=1)
```

GENERATORS

- Define a generator for training data, and another for validation.

```
gen_train = BatchGenerator(session_t, batch_num)
gen_valid = BatchGenerator(session_v, batch_num, augment = False)
            # the validation data set does not modify the image
```

DRIVE

- The vehicle can drive autonomously by loading the trained model and weights and then 'fed' video of the line placed on the ground. The model will output the predicted *steering angle*.

LOAD THE MODEL

- Load one of the trained models… some perform better than others!

```
print('begin model load...')
model = keras.models.load_model('weights12-7-18.hdf5',
custom_objects=customObjects)       # well defined model with good training
data
#model = keras.models.load_model('weights11-11-18.hdf5',
custom_objects=customObjects)     # poorly defined model with good training
data
#model = keras.models.load_model('weightsLane11-24-18.hdf5',
custom_objects=customObjects) # poorly defined model with poor training data
print('model loaded')
```

CONFIGURE THE HARDWARE

- Setup the camera and the motors.

```
car.setup()
rawCapture = PiRGBArray(car.camera, size=(car.FRAME_W, car.FRAME_H))
time.sleep(0.1)
counter = 0

print('Calibrated steering - centered now')
car.setupMotor()
car.get_pwm().set_pwm(0, 0, 410)
```

PROMPT FOR THE FORWARD SPEED

- The model is trained with a forward speed of '37'. Any speed much lower than 37 will cause the drive motors to stall... especially on carpet.

```
FORWARD_SPEED = int(input('Enter speed (minimum is about 37):'))
car.setSpeed(FORWARD_SPEED)
car.backward() # yes, the motors are wired in reverse!
```

MODEL PREDICTION LOOP

```
  for frame in car.camera.capture_continuous(rawCapture, format="bgr",
use_video_port=True):

      # Grab the raw NumPy array representing the image
      image = frame.array
      #cv2.imshow("Frame", image) -- if executing from an X-Server this will
show the camera's view

      # Pre-process the image
      net_inp = np.expand_dims(normalize(image[:]), 0)
```

PROCESS MODEL INSTANCE AND SET STEERING ANGLE

```
# Drum roll please... here is where the magic happens!
net_out = model.predict(net_inp)[0][0]
#print('predicted angle ', net_out)

# Send PWM signal to servo
if math.isnan(net_out) == False:
    steer = int(net_out * (STEER_UPPER - STEER_MIDPT) + STEER_MIDPT)
    # print ('Predicted Angle:', steer)
    car.get_pwm().set_pwm(0, 0, steer)
```

CLEANUP AND GET READY TO PROCESS AGAIN

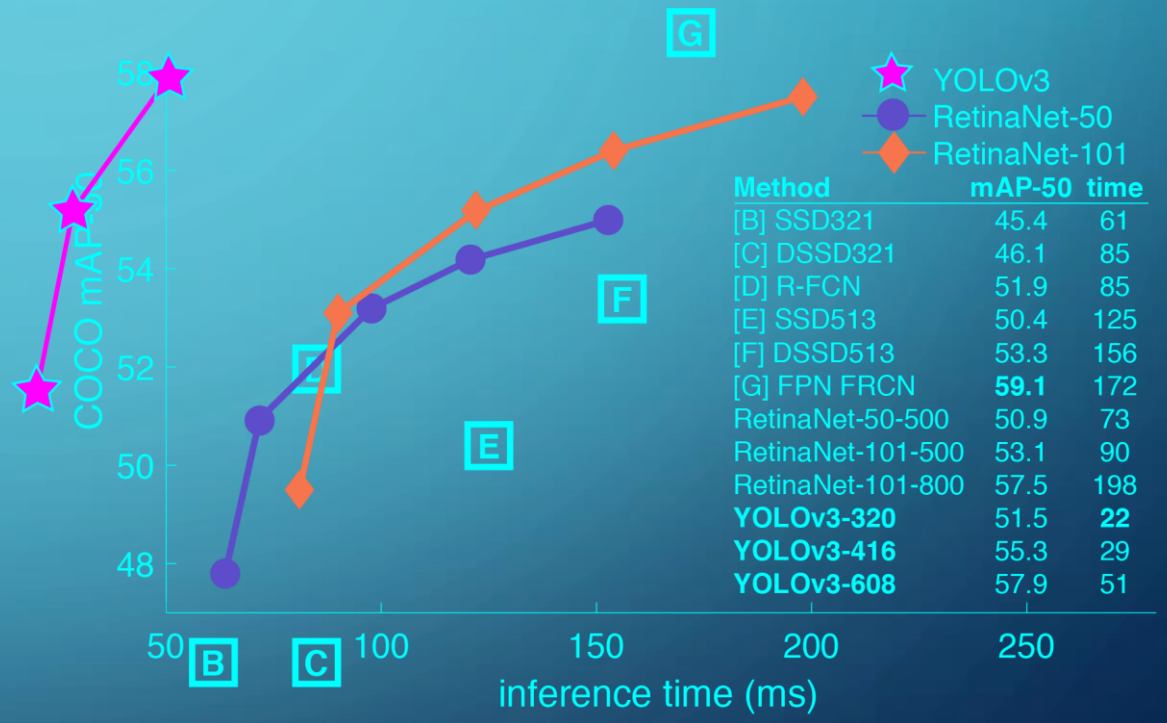
```
        # additional training data can be collected by writing the self-driving
results
        if RECORD:
                cv2.imwrite('demo/image/' + str(counter).zfill(6) + '.png',
image)
                with open('demo/steer/' + str(counter).zfill(6) + '.txt', 'w')
as steer_file:
                    steer_file.write(str(steer*10))
                counter += 1

        # Clear the stream in preparation for the next frame
        rawCapture.truncate(0)

        # If the `q` key was pressed, break from the loop
        key = cv2.waitKey(1) & 0xFF
        if key == ord("q"):
            print('quit - stopping car')
            car.stop()
            break
```

YOLO

- You only look once (YOLO) is a state-of-the-art, real-time object detection system.
- Adding YOLO to the self-driving vehicle to detect people.

COMPARISON TO OTHER DETECTORS

- YOLOv3 is extremely fast and accurate. In mAP measured at .5 IOU YOLOv3 is on par with Focal Loss but about 4x faster. Moreover, you can easily tradeoff between speed and accuracy simply by changing the size of the model, no retraining required!



Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

PROCESSING SPEED ON RASPBERRY PI

- Threading… nope.
- Approx. One frame analyzed every 2.3 seconds

IMAGE RECOGNITION ON LAPTOP

- Web-server on Raspberry PI
- Image is pulled by laptop and analyzed in 'real-time'
- Stop command is given to car. (Pull a 'stop' page)

TRANSFER LEARNING

- Teach YOLO to recognize road signs.

REINFORCEMENT LEARNING AND SELF-DRIVING

- In the reinforcement learning problem, the goal is to find a good behaviour, an action or a label for each particular situation, to maximize the long-term benefits that the agent receives.

AUTONOMOUS DRIVING REWARDS

- Keep the line in the field of view...

CREATE A TRAINING ENVIRONMENT

- Collect images of all the possible turn radius that the platform is capable of turning.
- Collect 'floor/surface' images without a line.
- Render a 'line' on top of floor surface images.
- Simulate 'steering and forward' motion with various steering angles
- Define reward function