Introducing SADS and HAPPS: Deep-Learning Approach to Autonomous Vehicles

Bomin Kim[1]
[1] *University of Nottingham, School of Psychology, Nottingham, United Kingdom*

Recent advances in artificial intelligence have led to a rapid explosion in self-driving cars based on deep-learning algorithms. Indeed, many major car industries, including Tesla, Ford, BMW, and GM, are actively testing and have demonstrated these cars can successfully drive millions of miles without any human intervention. Leveraging deep-learning methods (e.g., LaneNET, ENet, YOLOv3), we design, implement, and evaluate Simple AuDeep Driving System (SADS) and Helped AuDeep Premium Piloting System (HAPPS). Both models are based on NVIDIA's end-to-end approach for autonomous driving system to map raw pixels from input images to make steering commands for a car. In the end, we compete in two competitions: kaggle board and live-testing in which a car based on the SADS model succesfully navigated across three realistic driving circuits.

Keywords: Deep-Learning algorithms, autonomous vehicles, computer vision, object detection, CNN

INTRODUCTION

With the advent of deep-learning technology, self-driving cars have made a substantial development, going from discrete scientific prototypes to industrial applications with the ability to autonomously drive within urban areas and highways for an extended time period. Research on autonomous cars can be dated as far back when Pomerleau built the Autonomous Land Vehicle in a Neural Network (ALVINN), which consists of a fully-connected network. Despite its simplicity, ALVINN laid the foundation to end-to-end autonomous navigation [1].

At a conceptual level, there are two paradigms for autonomous driving systems: mediated perception and end-to-end. Mediated perception is a popular approach used in the industry which involves multiple sub-components (e.g., radar and sensors) when making steering commands [2, 3]. Conversely, the end-to-end approach focuses on building an artificial intelligence trained on human actions on the road that continuously makes steering directions by instantaneously processing images [4, 5]. Chen and colleagues utilized the Caffe framework to develop a CNN to process TORCS (The Open Racing Car Simulator) dataset. To address generalizability, Su et al. also developed an end-to-end learning model which was trained on a large dataset with varying road conditions, utilizing CNN for object detection and classification [6].

The aim of the current study was to develop a deep learning self-driving algorithm and compete in a competition where a real car would autonomously navigate around 3 realistic test circuits making appropriate steering commands where necessary. The competition consisted of a kaggle hosted challenge and life-testing challenge.

MATERIALS AND METHODS

Dataset

Our data comprised 13,800 training images and 1,000 test images that captured road conditions via Raspberry Pi (RPi). Specifically, each image was labeled with the binary velocity ($stop = 0$; $go = 1$) and steering angle ($range = 0 - 1$), in which an angle value of 0.5 corresponded to 90° (i.e., drive straight), and a value smaller than 0.5 corresponds to left turn, a value larger than 0.5 corresponds to right turn (see Fig.1).

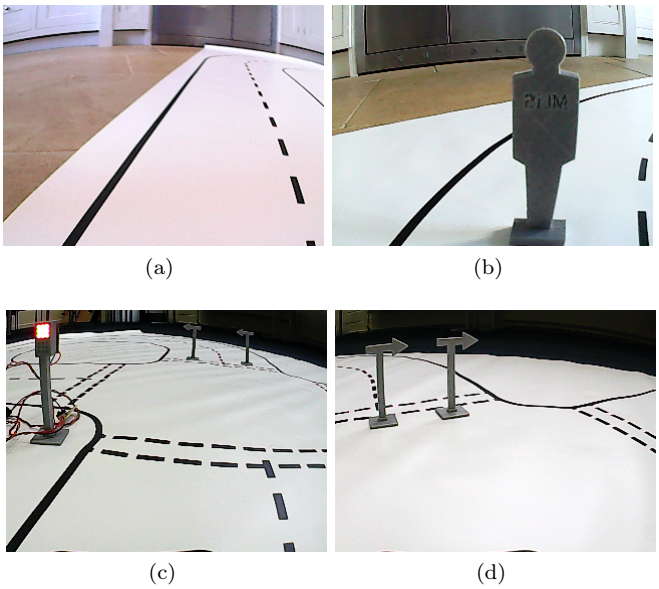


Figure 1. Examples of driving scenes: (a) no specific object on a straight road ($speed = 1$, angle ≈ 0.5), (b) an obstacle in the road ($speed = 0$), (c) red traffic light ($speed = 0$), (d) right turn sign ($angle \approx 1$).

RELATED MODELS ON SELF-DRIVING CARS

NVIDIA [4]

One of the most seminal research has been accomplished by the researchers in NVIDIA, where they proposed an innovative end-to-end CNN that takes in image images and outputs to the steering commands for a self-driving car. The model consists of 9 total layers, including a normalization layer, 5 convolutional layers for feature extraction and 3 fully connected layers to predict steering demands. Surprisingly accurate, this framework is successful in simple real-world environments such as highway lanes and additionally flat, obstacle-free routes. The processing speed for this model is also very quick,

processing 30 frames per second. Due to its efficiency, we rely on NVIDIA's framework to build our models (see below).

LaneNET [7] and ENet [8]

In general, existing lane detection models are typically based on two distinct categories: hand-crafted low-level features (i.e., application of Hough transform and Kalman filter to low-level features, such as colors, edges, and shapes) [9] and convolutional neural networks (CNNs) . LaneNET is a deep neural network (DNN) based method which works in two steps (see Fig. 2). In the first step, an encoder-decoder designer is utilized in which each pixel is given a binary value corresponding to whether it belongs to a lane, proposing lane-edge estimates. In particular, a "stacked depthwise separable convolution" and 1x1 convolution layers are used to encode the lane features and "non-parametric decoding layers" for decoding feature resolution. The decoded map is then converted to lane-edge coordinates and is provided to the second step. In the second step, a separate DNN, which consists of a "point-feature encoder" and "LSTM decoder", is used to precisely localize lanes based on the lane edge estimates from the first step.

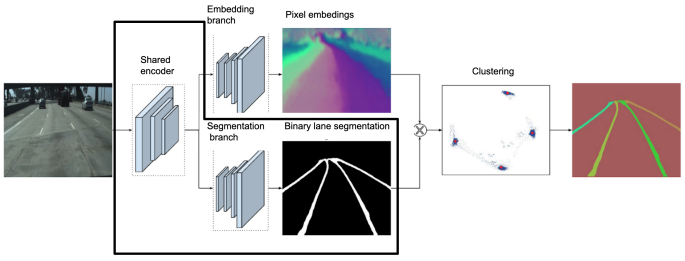


Figure 2. LaneNET architecture. It consists of two branches: segmentation branch (binary lane masking), embedding branch (N-dimensional embedding per lane pixel). Black outlines denote the process we employed into our model.

Despite its high accuracy even in complex scenarios, LaneNET has a complex training procedure and warrants extensive labeling, hindering its usability. ENet, a much simpler and faster lane detection algorithm, bypasses the aforementioned limitation yet detects lanes with adequate accuracy. ENet performs image semantic segmentation using encoder-decoder architecture. Each training example requires two labels: 1) true label, which contains the class for every pixel; 2) imprecise label, which is the less accurate label for every pixel. The loss is the weighted sum of categorical cross-entropies, so that the contribution of a pixel with label c_i is multiplied by:

$$w(c_i) = \frac{1}{\ln\left(p(c_i) + c\right)} \tag{1}$$

where $p(c_i)$ is the probability of occurrence of the class across the dataset (number of pixels labelled as c_i divided by the total number of pixels) and c is a constant (c = 1.02).

YOLOv3 ("You-Only-Look-Once") [10]

YOLOv3 is a real-time object detection algorithm, which utilizes a single deep CNN to predict an object's class and identify the corresponding object location by drawing a box around its extent (see Fig. 3).

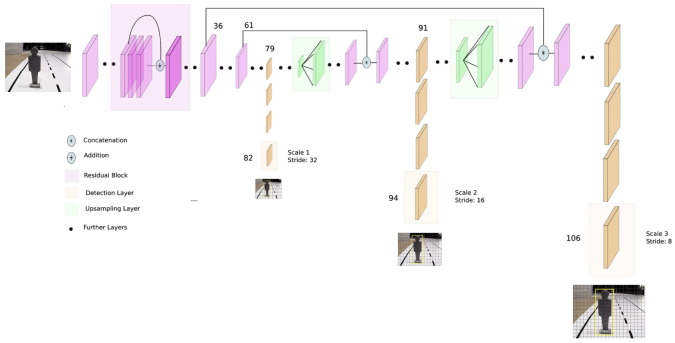


Figure 3. YOLOv3 architecture. Leveraging Darknet-53 CNN, the model incorporates shortcut residual connections alongside better object detectors with feature map upsampling using concatenation. Together with Darknet-53, it is additionally stacked with 53 more layers, ultimately yielding 106 fully-connected convolutional layers

The process of YOLOv3 object detection includes first splitting an input image into grid cells (SxS grid cells), in which each cell then predicts a number of boxes in the image (i.e., predicted location and dimensions) and objectness score. YOLOv3 makes predictions at three separate scales, far outperforming older versions when detecting smaller objects. Furthermore, Yolov3 uses predefined 9 "anchor boxes", 3 boxes at different scales, which represent ideal location and the dimension of the object. Object score denotes the confidence that the anchor boxes contain a detectable object and the probability of the object to a specific class. That is, it describes how likely an object corresponds to a particular class inside the predicted bounding box. The confidence score is computed by using logistic regression as this linear regression of offset prediction yields a lower mean average precision (mAP) value. The class confidence is calculated using multiple independent logistic classifiers instead of a single softmax layer. This is advantageous as an image can have multiple labels in which labels are not always mutually exclusive (e.g., a person and a woman). YOLOv3 predicts multiple bounding boxes, but using non-maximum suppression, most are reduced to a single box with the highest overlap between ground truth box and predicted box divided by non-overlap (i.e., Intersection over Union (IoU)).

YOLOv3 loss function consists of three separate components: classification, localization, confidence loss, given by:

$$\mathcal{L}_{YOLO} = \mathcal{L}_{classification} + \mathcal{L}_{localization} + \mathcal{L}_{confidence} \tag{2}$$

Firstly, if an object is detected, the classification loss at each cell is simply the squared error of the class conditional

probabilities per class, depicted as:

$$\mathcal{L}_{classification} = \sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \in classes} \big(p_i(c) - \hat{p}_i(c)\big)^2 \quad (3)$$

where 1_i^{obj} has value 1 if cell i is predicted to contain an object and 0 otherwise. $p_i(c)$ and $\hat{p}_i$(c) are the conditional (with prior cell i containing an object) true and predicted probabilities of the object belonging to class c.

The second component is the localization loss, which illustrates the error between the predicted boundary box and the ground truth, depicted in the following equation:.

$$\mathcal{L}_{localization} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \Big[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2\Big]$$
$$+\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \Big[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2\Big] \quad (4)$$

where (x, y) $[(w, h)]$ is the true bounding box center position [height and width] and $(\hat{x}, \hat{y})$ $[(\hat{w}, \hat{h})]$ the corresponding prediction. The model predicts the square root of the width and height of the bounding box, instead of absolute values because absolute errors in large and small boxes should not be weighed equally (i.e., 2-pixel error in a small box is not equivalent to a big box).

The final part is the loss of the confidence score for each bounding box predictor, depicted:

$$\mathcal{L}_{confidence} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \big(C_i - \hat{C}_i\big)^2$$
$$+\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} \big(C_i - \hat{C}_i\big)^2 \quad (5)$$

where C is the confidence score and $\hat{C}_i$ is IoU of the predicted bounding box over the labeled box. It is important to note that most boxes do not contain any objects, that is, numerous grid cells correspond to non-object. Consequently, there is a class imbalance problem in which the model is trained to detect background more often than detecting objects. This problem is mitigated by setting different values of λ, in which the highest penalty is for coordinate predictions ($\lambda_{coord} = 5$) and the lowest for confidence predictions when no object is present ($\lambda_{noobj} = 0.5$)

DETR (DEtection TRansformer) [11]

Recently, transformers have gained much popularity for not only its simplicity but also its powerful mechanism called attention. Transformer's self-attention mechanism allows selective focus, weighing influence on certain parts of the input, thereby reasoning more efficiently. Proposed by Facebook AI researchers, DETR is the first model to successfully integrate a pretrained CNN backbone with Transformers as a central building block to perform object detection (see Fig. 4). DETR is a set-based object detector which makes direct prediction over N boxes, in which each box is assigned to a class, and a vector of a center, width and height of the bounding boxes.

Firstly, CNN takes the input image and outputs a low-resolution activation map that is later projected into a lower channel dimension (i.e., flattened to a single unit). The transformer-encoder applies a self-attention mechanism to find pixels that are positively correlated to each other. before moving to the transformer. Then, the decoder is fed with a small fixed number of object queries (i.e., learned positional embeddings) alongside the output from the encoder, where another attention layer is applied. These transformed-object queries are passed independently through the shared feed-forward network to predict either detection (class and corresponding bounding boxes) or "no object" class. That is, the model predicts the number of objects in an image, so that N_O corresponds to objects, and N - N_O corresponds to the no-object class boxes.

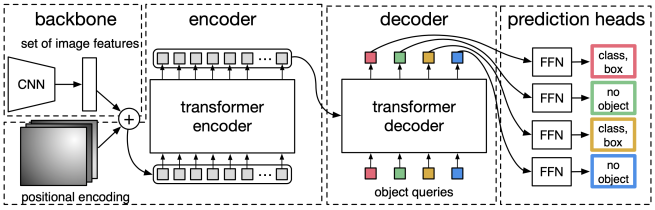


Figure 4. DETR architecture. Combination of traditional CNN with a transformer, DETR directly predicts a set of object detection. During training, bipartite matching assigns model's prediction with ground truth boxes during training.

Another unique aspect of DETR is the proposed loss, which is highly related to the direct set prediction approach. Specifically, given N_O object labels, the final set is null-padded to have N items in the end. To apply the loss function, the model performs bipartite matching which finds a one to one pair between prediction and ground truth label based on 'matching cost' (see Eq. 6). DETR uses the Hungarian algorithm, which sums up the loss for all matched pairs determined from the previous matching stage. It is important to note that predicting multiple boxes for the same box will yield an increase in the loss for only one of the predicted boxes will match with a counterpartner in the label set. Therefore, DETR needs to learn and reason efficiently the approximate relation between a point and its relation to the rest of the image by indicating the likelihood of the surrounding areas to be positively related to that point.

The main difference between conventional object detection algorithms and DETR is the removal of many hand-designed components (e.g., non-maximum suppression and predefined anchor boxes) that explicitly embody prior knowledge, hence making the process less computationally expensive. The permutation of the predictions that

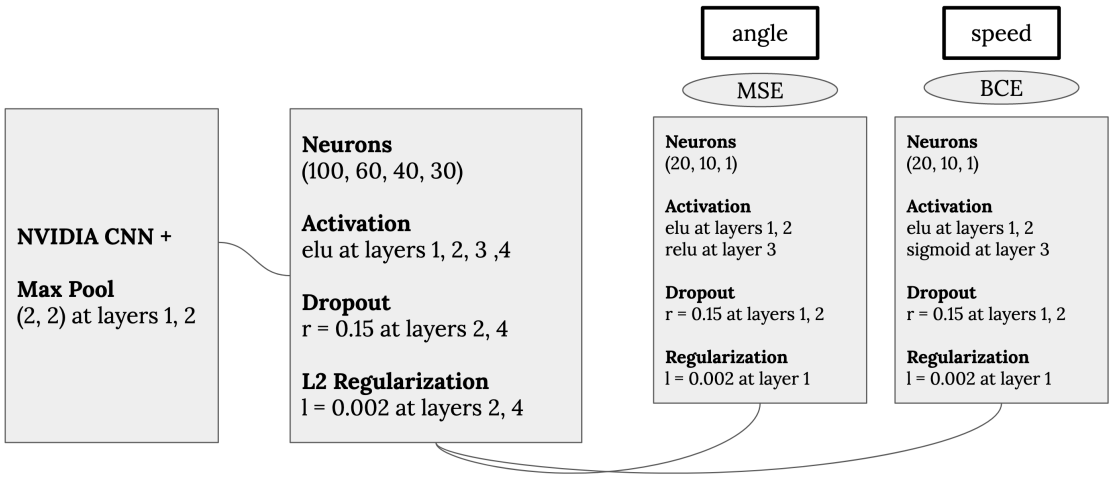


Figure 5. SADS architecture. Extending NVIDIA model, max pool layer, dropout and L2 Regularizations were added. ELU activation functions were used to introduce non-linearity. The loss was split into two independent units, MSE for angle and BCE for velocity.

contributes to the loss can be formally expressed as:

$$\hat{\sigma} = argmin_{\sigma \in \mathfrak{S}_N} \sum_{i}^{N} \mathcal{L}_{match}(y_i, \hat{y}_{\sigma(i)}) \tag{6}$$

where σ are the different permutations of N elements (set denoted by $\mathfrak{S}_N$, y_i are the labeled tuples of classes and boxes and $\hat{y}_{\sigma(i)}$ are the i^{th} predictions of the model according to σ.

A key component aspect of DETR loss is application of generalized intersection over the union (GIoU) which penalizes boxes that are farther away from each other. The GIoU loss can be expressed as:

$$\mathcal{L}_{GIoU}(b_i, \hat{b}_{\sigma\hat{(i)}}) = 1 - \left(\frac{|b_i \bigcap \hat{b}_{\hat{\sigma}(i)}|}{|b_i \bigcup \hat{b}_{\hat{\sigma}(i)}|} - \frac{|B(b_i, \hat{b}_{\hat{\sigma}(i)}) \setminus b_i \bigcap \hat{b}_{\hat{\sigma}(i)}|}{|B(b_i, \hat{b}_{\hat{\sigma}(i)})|} \right) \tag{7}$$

where $b_i \bigcap \hat{b}_{\hat{\sigma}}(i)$ and $b_i \bigcup \hat{b}_{\hat{\sigma}(i)}$ denote (respectively) the intersection and the union of the regions defined by b_i and $\hat{b}_{\hat{\sigma}(i)}$, $|r|$ is the area of r, $B(a, b)$ is the smallest convex hull that encloses a and b and $A \setminus B$ represents the elements of A not contained in B. This loss is identical to classic IoU but adds an extra penalization for boxes that, apart from having 0 intersection, are apart from each other.

Proposed Models: SADS and HAPPS

In the current study, we propose two models: Simple AuDeep Driving System (SADS) and Helped AuDeep Premium Piloting System (HAPPS).

For the SADS model, we took the original NVIDIA convolutional backbone and connected it to 4 linear layers (see Fig. 5). The first common layer is then later split into two units that perform regression for the angle and classification for the speed (i.e., angle loss = the Mean Squared Error (MSE), speed loss = Binary Cross Entropy (BCE)). To take care of the vanishing gradient problem, we also made further modifications to the model's parameters by using Exponential Linear Unit (ELU) activation for every layer to introduce non-linearity.

Furthermore, to address the problem of overfitting and eventually improve the loss rate and accuracy, we added regularizations and dropout layers after convolutional layers to prevent co-adaptation of learned hidden features. In addition, we train all models with the ADAM exponential-decay optimizer, with initial learning rate of = 0.003 and decay rate of 0.02 for every 10000 steps. Opposed to the original framework, alongside inclusion of a max pooling layer, we apply data augmentation including salt-and-pepper noise (SD= 0.2), random translation, contrast, and brightness. It is important to note that the angle has been multiplied by π to address the vanishing gradient problem and all our corresponding results depict angles scaled up by π.

Extending from the SADS model, we designed a semantically sensible color dictionary for distinct objects (see Table 1) and painted object boxes into the binary output of the lane detection model accordingly. The HAPPS model takes in input images that have been pre-processed (including colored boxes and lines), presumably helping the model to learn and make decisions more efficiently. HAPPS is further divided into three categories: HAPPS-2L, HAPPS-3L, and HAPPS-3R (see Fig. 6)

Table I. Painting Algorithm Dictionary

Channel	Obstacles	Right	Left	Red	Lane
R	100	0	0	255	255
G	0	100	0	0	255
B	0	0	100	0	100

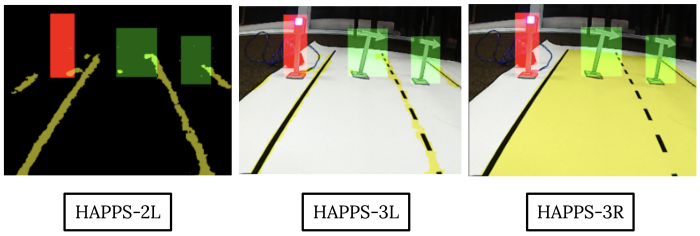


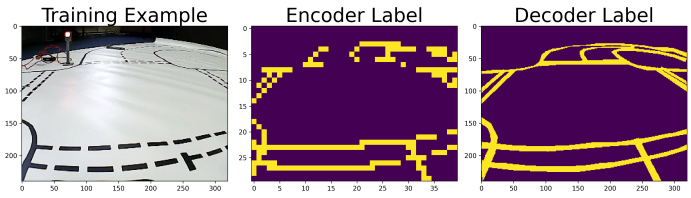
Figure 6. Different versions of HAPPS. HAPPS-2L corresponds to overlay of YOLOv3 colored boxes on lane detection via ENet. HAPPS-3L is the overlay of HAPPS-2L on the input image. HAPPS-3R differs from HAPPS-3L in that instead of lane detection, ENet performs road detection.

RESULTS

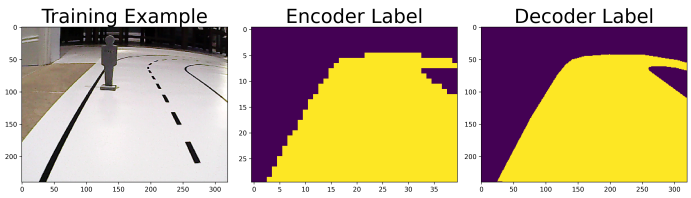
Lane detection: LaneNET and ENet

The ENet training procedure consisted of two steps: 2) freeze the decoder and train the encoder 2) freeze the encoder and train the decoder. Furthermore, we trained the model with an adam optimizer with learning rate of $=$ 0.005, modified the last layer to sigmoid and used BCE to calculate the loss. Interestingly, we found that the optimal weight for the minority class was approximately twice the value calculated with equation 1. Additionally, it is important to note that we have two different lane detection tasks: lane (ENet-L) and road (ENet-R) (see Fig. 7).

To label corresponding lanes and roads, we utilized the python package LabelMe [12] to label arbitrary polygons. As depicted in figure 10a, the encoder's loss decreases and the accuracy increases during the first step, meanwhile the decoder's loss and accuracy are static. As soon as we freeze the encoder and train the decoder, the loss and the accuracy of the decoder improves. Due to ENet-L's task of detecting edges, even non-existing edges can also be detected (see Fig. 8).



(a)



(b)

Figure 7. Training examples of ENet-L and ENet-R: (a) for ENet-L (b) for ENet-R. Middle column denotes the output from the encoder and the right column corresponds to the output from the decoder

Therefore, we further extended and developed ENet-R which specializes in finding high-contrast edges by classifying whether pixels belonged either inside or outside of the road (see Fig. 7a). This model has the advantage over ENet-L by helping the model to distinguish obstacles that are outside of the road. ENet-R is trained using the same architecture but with different labels and performs in a similar pattern to ENet-L (see Fig. 7b).

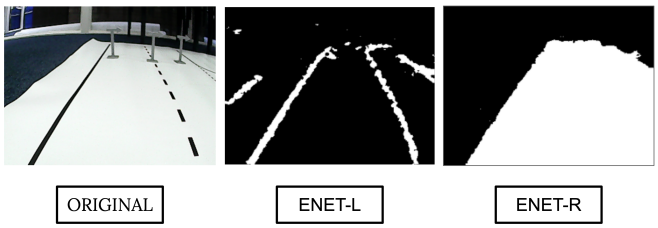


Figure 8. Outputs of unseen data ENet-L and ENet-R. ENet-L specializes in detecting edges , thereby detecting even non-existing lanes. Conversely, ENet-R specializes in detecting high-contrast edges, yielding a much cleaner detection of the entire road.

Object detection: YOLOv3

In the present study, YOLOv3 was trained on 1794 images (80/20 training-validation split) to detect 5 distinct classes (e.g., obstacles, red, green, right, left; see Fig. 9).

It is important to note that besides the data augmentation technique preinstalled in yolo, additional data augmentation of flipping, hue, contrast, translation, and scaling were performed. In the beginning of the training, weights were initialized using the pretrained weights from the darknet53.conv.74. YOLOv3 performance metrics of precision, accuracy, and mean Average Precision (mAP) were computed at a IoU threshold value of 0.5 (see Table 2).

Table II. Performance Metrics of YOLOv3

	Obstacles	Right	Left	Green	Red
TP	814	436	242	84	178
FP	23	0	0	0	0
FN	21	19	24	56	139
Total #	835	455	266	140	317
Precision	0.97	1	1	1	1
Recall	0.97	0.95	0.90	0.6	0.56

As evident from table 2, the recall rate for green and red lights are substantially low compared to other classes presumably due to comparatively lack of traffic light objects in our training dataset. The average loss function value using YOLOv3 is 0.15 and mAP value of 86.1% after completion of training for 10, 000 iterations (see Fig. 10e).

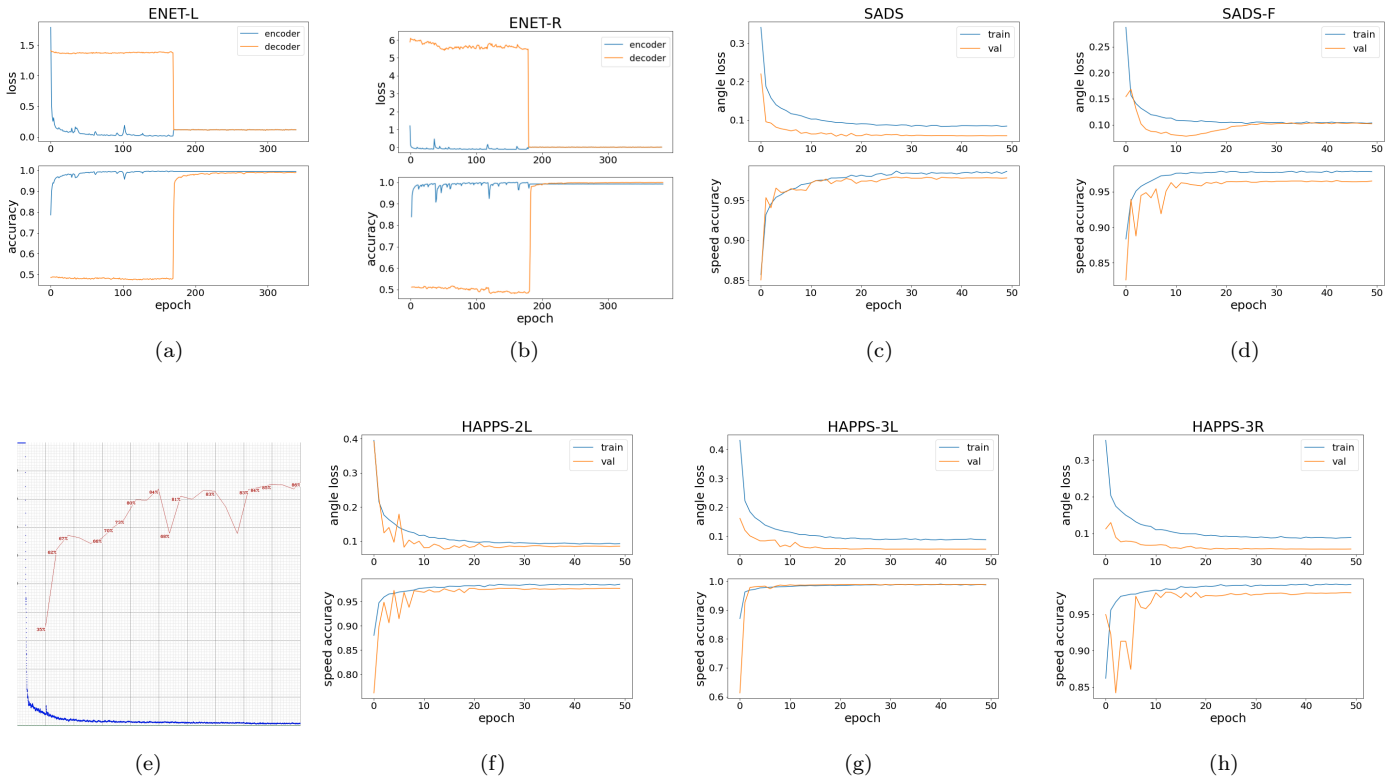


Figure 9. Learning curves for: (a) ENet-L (b) ENet-R (c) SADS (d) SADS-F (e) loss and mAP curve for YOLOv3; blue line depicts the loss curve and the red line denotes the mAP value for every 1000 iterations (f) HAPPS-2L (g) HAPPS-3L (h) HAPPS-3R.The minimum loss and maximum accuracy value for each models are stated in Table 3

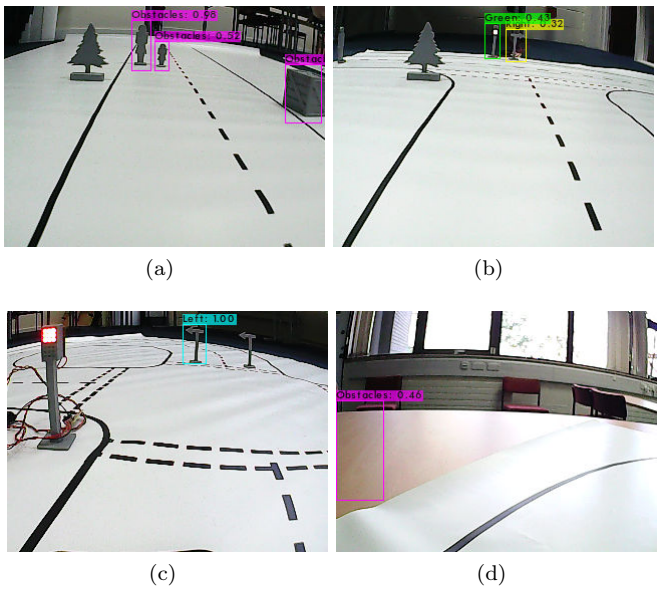


Figure 10. Results from YOLOv3. (a) detection of three obstacles. (b) detection of small images of green light and right turn. (c) Fail red light detection; false negative (d) detection of no-object obstacle; false positive

Object detection: DETR

Despite our continual effort, we were unsuccessful in performing object detection via DETR. We tried modifying the number of object-queries to 5 objects as our task only requires $5 - class$ object detection. Furthermore, we hypertuned other parameters such as the learning rate and trainability of other layers, but all images detected objects as no-class. A potential reason for this is the intrinsic lack of information in our data. That is, DETR thrives in complex object detection (original N of object-queries is 100), but our data could be too simple for DETR to detect objects.

SADS and HAPPS

In addition to SADS, we also train the model with horizontally flipped dataset, where the velocity is kept constant but the angle is modified to be the absolute value of original angle minus 1 (referred as SADS-F). Interestingly, we observe an initial dip of the validation loss presumably because the SADS-F model is overfitting to the flipped dataset due to the asymmetry of the problem (see Fig. 10d).

Results of SADS and HAPPS illustrate that HAPPS-3L performed the best, supporting our hypothesis that with more aid superimposed on the input image the model becomes more accurate (see Fig. 10; Table 3). Even though HAPPS-3R performed relatively well, We speculate the reason HAPPS-3R not outperforming HAPPS-2L is lack of training time for road detection, resulting in road-overlays to be slightly inaccurate compared to lane-overlays. With more time, we would have trained ENet-R more, fine-tuning the parameters to get optimal results.

Table III. Comparison among models

Model	MSE (angle)	Accuracy (v)
SADS	0.018	0.979
SADS-F	0.035	0.965
HAPPS-2L	0.024	0.977
HAPPS-3L	0.018	0.989
HAPPS-3R	0.018	0.980

MLiS Competition: Kaggle and Live-test

We submitted the different versions of HAPPS model to kaggle, ultimately obtaining a score (MSE) of 0.01827. However, in order to achieve low inference time, we submitted SADS model for the live-test competition. Because the SADS model does not incorporate any object detection models, it failed to detect unclear objects from each other (e.g., traffic lights). Furthermore, despite the car accurately predicting angles for right and left turns, it failed to stay in lane. A potential solution is incorporating object detection model so that the car could detect right and left turns from a distance and react accordingly. Additionally, the car had exceptional cornering ability compared to other teams and this could be because of hyper-tuning of the parameters but also the multiplication of $\pi totheangle$.

DISCUSSION

In the current study, we propose two models to predict steering angles and velocity for a self-autonomous car to navigate without any human-intervention. Through our kaggle and live-test competitions, we show that our model performs with high accuracy and low loss, alongside successful navigation in realistic driving circuits.

However, there are still some limitations and future directions that should be addressed. Firstly, the training dataset contained some mislabeled data, specifically erroneous velocity. Furthermore, our dataset only included discrete images of the track, which substantially limited the scope of potential deep-learning algorithms to use. If our dataset was continuous with recurrent history, we could have used reinforcement-learning and recurrent neural network to build a more complex and accurate self-driving car model. Furthermore, output for an end-to-end algorithm (e.g., SADS and HAPPS) is defined as steering angles and binary velocity value. However, in a more realistic driving environment, the actions a car can perform is much more complicated (e.g., reverse, slowing down, brake). Furthermore, including a more fine-detailed velocity action such as slowing down if a pedestrian is moderately close or even changing lanes to avoid obstacles, is another exciting future step.

ACKNOWLEDGEMENTS

I would like to thank Albert Albesa Gonzalez in working on this project together for the past few months. Wouldn't have been able to finish it with you! Go Team AuDeep! Thank you to Dr. Maggie Lieu and Dr. Adam Moss for organizing this assignment and competition and Simon Dye for assembling the cars and also organizing the competition.

[1] D. A. Pomerleau, *Alvinn: An autonomous land vehicle in a neural network*, Tech. Rep. (CARNEGIE-MELLON UNIV PITTSBURGH PA ARTIFICIAL INTELLIGENCE AND PSYCHOLOGY ..., 1989).

[2] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, in *Proceedings of the IEEE international conference on computer vision* (2015) pp. 2722–2730.

[3] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, The International Journal of Robotics Research **32**, 1231 (2013).

[4] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, arXiv preprint arXiv:1604.07316 (2016).

[5] H. Xu, Y. Gao, F. Yu, and T. Darrell, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017) pp. 2174–2182.

[6] B. T. Nugraha, S.-F. Su, *et al.*, in *2017 2nd International Conference on Automation, Cognitive Science, Optics, Micro Electro-Mechanical System, and Information Technology (ICACOMIT)* (IEEE, 2017) pp. 65–69.

[7] Z. Wang, W. Ren, and Q. Qiu, arXiv preprint arXiv:1807.01726 (2018).

[8] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, arXiv preprint arXiv:1606.02147 (2016).

[9] R. Gopalan, T. Hong, M. Shneier, and R. Chellappa, IEEE Transactions on Intelligent Transportation Systems **13**, 1088 (2012).

[10] J. Redmon and A. Farhadi, arXiv preprint arXiv:1804.02767 (2018).

[11] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, in *European Conference on Computer Vision* (Springer, 2020) pp. 213–229.

[12] B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman, International journal of computer vision **77**, 157 (2008).