Autonomous driving in TORCS

Matej Kosec
Department of Aeronautics and Astronautics
Stanford University
`mkosec@stanford.edu`

Abstract

Autonomous driving is one of the most exciting applications of deep learning methods, with companies around the world competing to be the first to enter the market with a fully autonomous vehicle. This project represents an introduction to working within the autonomous driving space, in particular getting used to the different data-types (sonar, image, speed). A modular approach is presented which teaches a RL policy gradient agent (CS234 portion) to drive from low-dimensional sonar input, and then a second module is introduced (CS230 portion) which can map from visual inputs to the sonar inputs. The interplay of these two modules is analysed and enhanced.

1 Introduction

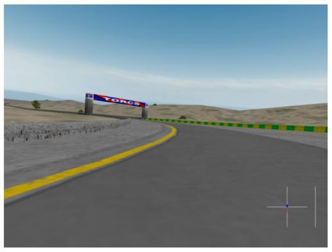
Autonomous driving is one of the most demanding, promising, and interesting applications of deep learning methods. It is incredible data and compute intensive, and thus difficult to scratch the surface of in a single course project. Thus this project has two portions: a CS234 portions which applies reinforcement learning techniques for teach a agent to drive from low-dimensional inputs, and a CS230 component which explores how high-dimensional inputs can successfully be mapped to the low-dimensional space in which the agent already knows how to drive. The TORCS driving simulator is used due to its low resource requirements and the ability to train on low-dimensional inputs in over-time (sped up).
The interested reader is referred to related work which inspired this project. In particular: [3], [5], [4], [2]. Additionally, interfacing with the racing simulator TORCS is achieved through modifying and adding to the gym_torcs environment provided at `https://github.com/ugo-nama-kun/gym_torcs`.
To view the project code please follow the invitation link [1] for the code on GitHub repository [2].
The report for the CS234 portion of the project is available in pdf from the footnote[3].

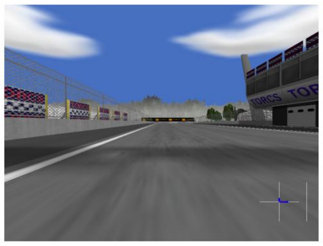
2 Dataset and Features

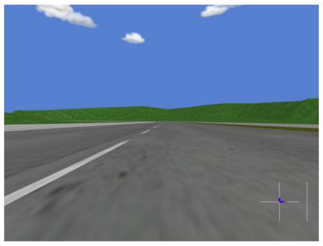
In autonomous driving we are hoping to train our vehicle on some small subset of the global road network and then hopefully see it generalize well to unseen examples. Thus it is sensible that train/dev/test datasets should be collected using different tracks. While this may seem to produce datasets from different distributions, it does accurately reflect the level of generalization we expect our model to achieve.
The training set consists of a sequence of 30k frames collected during the course of this project on the train course (see figure 1d). Similarly, the test and dev sets contain 3k examples sampled from the

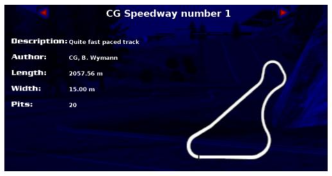
[1]GitHub Invite: `https://github.com/MatejKosec/cs230_Autnomous_Driving/invitations`
[2]CODE: `https://github.com/MatejKosec/cs230_Autnomous_Driving.git`
[3]CS234 report: `https://drive.google.com/open?id=19Jh9W6KbAjasdpJKZIOxfteB5f48Q9_-`

dev and test tracks, respectively. The dev track is shown in figure 1e and the test track in 1f.
The choice of these tracks is quite deliberate. They are quite wide at 15m, and have have strong lane markings (unlike dirt tracks). Additionally, the train track starts with a sweeping left turn while the test and dev tracks start with a straight. Thus if our model over-fits to the training set we will be able to see it trying to make a left turn on an a straight (does occur in practice). This choice of train/dev tracks makes spotting poor generalization much faster than if the all datasets started with a straight.


(a) The train track visuals

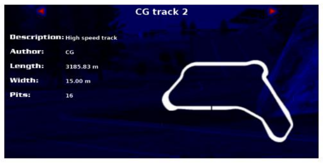

(b) The dev track visuals


(c) The test track visuals


CG Speedway number 1

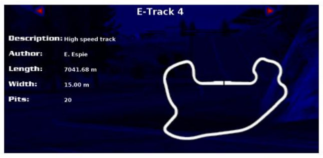
Description:	Quite fast paced track
Author:	CG, B. Wymann
Length:	2057.56 m
Width:	15.00 m
Pits:	20

(d) Train track (starts with bend)


CG track 2

Description:	High speed track
Author:	CG
Length:	3185.83 m
Width:	15.00 m
Pits:	16

(e) Dev track (starts with straight)


E-Track 4

Description:	High speed track
Author:	E. Espie
Length:	7041.68 m
Width:	15.00 m
Pits:	20

(f) Test track (starts with straight)

Figure 1: The test, dev, and train tracks

The model itself takes as input three grayscale (pixel values from 0 to 1) images of 64x64. The three images are the last three frames seen by the vehicle. This is hoped to enable the network to get a sense of how the environment is moving with respect to the car. Thus the input is of shape [batch size, 64,64,3] where the three image channels represent the three different time-steps.
The model is expected to output a series of 19 sonar sensor measurements. The TORCS environment provides these sonar measurements (without noise) and it is known that they are arranged in equally spaced arcs from left to right as shown in figure 3.
The TORCS environment is also able to provide the speed of the car, which is used as an input to the reinforcement learning agent (part of CS234 project).

3 Methods

The primary methodology of this project is to split the work into three parts: 1) train a RL agent low-dimensional sonar inputs (CS234), 2) use supervised learning to train a CNN to map from image inputs to low-dimensional sonar (CS230), 3) combine the two models together (plug-and-play or retraining) (CS230)
Using such a modular approach is necessary in order to improve the data efficiency of the reinforcement learning task. Additionally, TORCS does not allow collecting data at faster than real-time with visual inputs turned on. However, when only using sonar we are able to collect data 200 times faster (simulation time and real time are decoupled). Leading to a much more efficient work flow and debugging situation. In fact, with such high computational costs the RL problem may indeed become untractable.

3.1 Model architecture

The model architecture is split into two portions. A dense network trained in CS234 to drive on dense sonar inputs (see figure 2 right) and a CNN based on the segnet [1] architecture that maps from 64x64x3 images to the sonar inputs. Note that the output of the CNN is activated with a sigmoid as all sonar inputs are normalized distances between 0 and 1 (0 meters to 200 or more). Additionally, leaky ReLU activations are used as in RL settings it is desirable to be able to un-learn bad decisions. The design of the network was explained more in-depth in the milestone report.

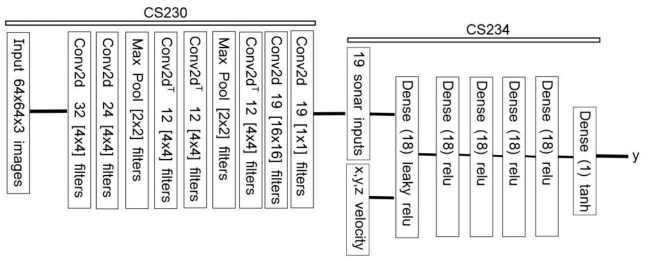


Figure 2: Modular architecture of the implemented deep neural network

3.2 Sample Input Output Pair

Figure 3 shows a sample input to the CNN and the corresponding predicted sonar input as well as the true sonar input and the steering direction. Note the attempt to steer along the line

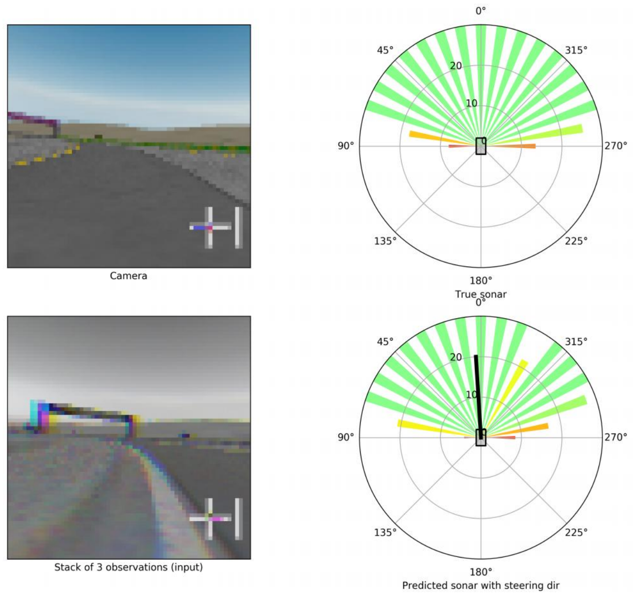


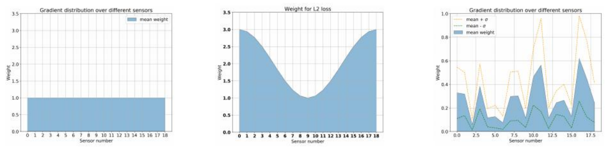
Figure 3: Top left: current camera state, Top right: true sonar state, Bottom left: stacked last 3 frames, Bottom right: predicted sonar and steering direction

3.3 Choosing the loss function

The loss function is L2 norm between the predicted and the true sonar values. To improve performance these were first heuristically weighted to emphasize the sonars closer to the side of the road (0 and 19). Later we decided to weight the sonars based on how much they influence the final decision of the RL agent. This was done by weighing the sonars based on the gradient of the stering direction with respect to each sonar. The different weighing schemes are shown in 4. Interestingly, as shown in figure 4c the gradient weighing is not similar to the heuristic but and is very noisy.

3.4 Coupling RL agent loss to the CNN model

The RL agent and the CNN are pre-trained separately in two different Tensorflow graphs and sessions. As such, the auto-differentiation framework cannot compute the gradients of the reinforcement



(a) Uniform weighing(b) Heuristic weighing (c) Gradient weighing

Figure 4: Different weighing function for the CNN loss

learning loss with respect to the parameters θ_2 of the CNN. These gradients are required when re-training the coupled model using the pre-trained weights as initializations. Thus, begin able to correctly pass gradients between the two graphs is one of the most challenging aspects of this project (in terms of programming).
The overall execution scheme for a single update step for the 2 coupled models is as follows:

1. Feed a stack of 3 most recent frames into the CNN and run **forward prop** on the CNN
2. Feed the output of the CNN to the RL agent NN and run **forward prop** on the RL agent NN
3. Run **back-prop** on the RL agent NN with respect to the parameters of the RL agent NN
4. Run **back-prop** on the RL agent NN with respect to inputs to the RL agent NN and feed those inputs to the CNN back-prop (next step)
5. Run **back-prop** on the CNN with respect to the CNN parameters using the output of the RL agent NN back-prop as downstream gradients
6. **Update** parameters of the RL agent NN
7. **Update** parameters of the CNN

While this execution scheme is quite a bit more cumbersome than when dealing with a single model, it does conveniently split up the update steps of the CNN and the RL agent NN. This allows different learning rates to be used on the two different network. Unfortunately, picking these two different learning rates turns out to be a difficult hyper-parameter search. For instance, it may seem logical to only slowly change the RL agent policy (as it is known to perform well on sonar inputs), and simply modify the CNN faster. Alternatively, it may be reasoned that the RL agent should change faster as it is a smaller model and should easily adapt to noisy input coming form the CNN. At the same time the CNN is allowed a much higher learning rate. In practice it turns out (for this case) that the second approach consistently produces better results.
To manually pass the gradients between the two models the chain rule should be applied to the RL loss (L_{RL}) which is parametrised by θ_1 (RL agent NN) and θ_2 (CNN). Additionally, the output of the CNN $\hat{y}_2$ is the input to the RL agent NN. Thus the chain rule becomes:

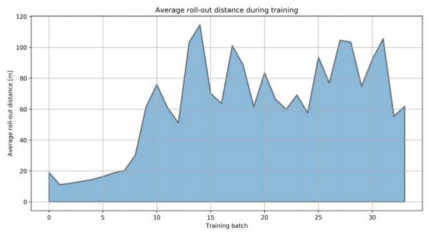
$$dL_{RL} = \frac{\partial L_{RL}}{\partial \theta_1} d\theta_1 + \frac{\partial L_{RL}}{\partial \hat{y}_2} \frac{\partial \hat{y}_2}{\partial d\theta_2} d\theta_2 \tag{1}$$

Note that the term $\frac{\partial L_{RL}}{\partial \hat{y}_2}$ is with respect to the placeholder $\hat{y}_2$ representing the output of the CNN that gets passed to the RL agent NN. This term is computed in Tensorflow using *tf.gradients()* and is then passed as placeholder to the CNN.
Finally, the coupled (RL) loss on the CNN can be defined as (where $\hat{y}_2$ is the sonar prediction of the CNN):

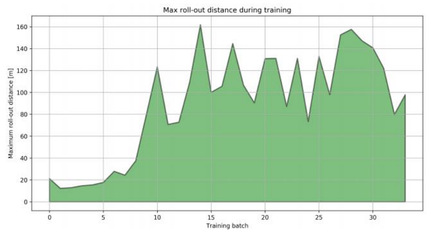
$$L_{\texttt{coupled}}^{CNN} = \overbrace{\left(\frac{\partial L_{RL}}{\partial \hat{y}_2}\right)}^{\texttt{placeholder}} \overbrace{\hat{y}_2}^{trainable} \tag{2}$$

Thus, once the placeholder is passed in correctly from the downstream RL agent NN, the auto-diff will correctly back-prop into the trainable $\hat{y}_2$. The placeholder is treated as a constant by the differentiation. With this adjusted loss function for the CNN, and the correct forward and backward placeholder passing, the coupling between the two graphs is complete.
Figure 5 demonstrates the training performance of the policy gradient algorithm when working with the coupled networks. It can be see that the optimization is able to make significant progress (max

distance increases from 21m to 161m) however the training procedure is highly noisy (Monte Carlo sampling on policy) and could certainly benefit from a larger batch size for policy evaluation. Due to how expensive it is to run the graphical interface this training process is run with an RL sample batch of 200 frames, while the noiseless sensor agent trained successfully with a batch size of 800. However, this would take more than and hour per iteration.
What is interesting however, is that during the 10 steps of the coupled policy gradient updates, the roll-out distance drops slightly at first, and then improves relatively monotonically. As such, the method appears to have been implemented correctly, but requires further hyper-parameter searching to achieve good performance.



(a) Uniform weighing

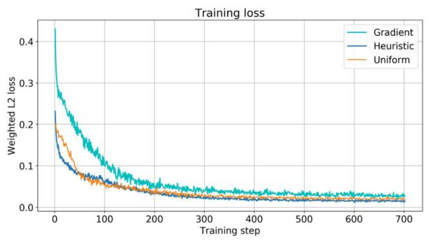


(b) Heuristic weighing

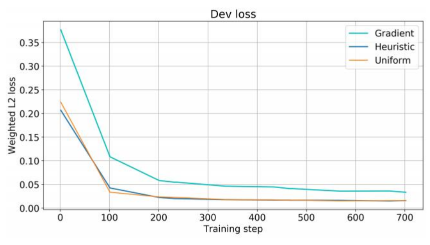
Figure 5: Progress of reinforcement learning on using pretrained CNN and RL agent NN

4 Experiments/Results/Discussion

Figure 6 shows the training and dev loss achieved in the supervised training of the CNN network. Furthermore, the different loss weighing techniques are shown to achieve similar values, however the loss is not indicative of final driving performance as demonstrated in table 1.



(a) L2 loss on the training set



(b) L2 loss on the dev set

Figure 6: Progress of reinforcement learning on using pretrained CNN and RL agent NN

4.1 Quntitative results

Table 1 shows that driving on the low-dimensional inputs remains much better than driving from pixels. Even with retraining (use the pre-trained weights as initialization) does not yield comparable performance. It may however, be seen that the retrained model is over-fitting to the train set (tries to turn on a straight).

4.2 Qualitative results

A video demonstration of driving on noiseless inputs is lined in the footnotes[4]. A less impressive demonstration on driving with pre-trained weights using only 64x64x3 pixels is shown in footnote[5]. Finally, a demo of driving from pixels using pre-trained weights and retraining is available in the last

[4]`https://drive.google.com/open?id=11qnuIjUaLED4k6mpCtTQI45Ul0r98hvJ`

[5]pretrained: `https://drive.google.com/open?id=15hJDCUpOvMVeaVqdbcj-WeKuLh-mL6o5`

Table 1: Roll-out distances for different tested methods

		Roll-out distance [m]				
Track	train/dev/test	Min	Mean	Max	Completed lap	
CG Speedway No. 1	train	41	3057	9499	Yes!	noiseless sonar (CS234)
CG track 2	dev (easy)	589	4221	14642	Yes!	
E-track 4	train (easy)	729	3945	13461	Yes!	
Wheel 2	test (hard)	336	625	1331	No.	
CG Speedway No. 1	train	15	19	24	No.	Un-weighted loss (CS230)
CG track 2	dev (easy)	18	34	51	No.	
CG Speedway No. 1	train	21	26	33	No.	Heuristic weighing (CS230)
CG track 2	dev (easy)	51	178	255	No.	
CG Speedway No. 1	train	23	31	43	No.	Gradient weighing (CS230)
CG track 2	dev (easy)	30	73	203	No.	
CG Speedway No. 1	train	84	130	248	No.	Retraining (CS230)
CG track 2	dev (easy)	26	30	34	No.	
E-track 4	**test!**	**26**	**29**	**33**	**No.**	

footnote[6]. Please contact me if the links are not functioning properly.
Driving from 64x64 pixels seems impossible even for human drivers.

5 Conclusions/Future Work

Important future works should explore better driving simulators such as CARLA and AirSim. The inability to run TORCS in accelerated time with visual inputs has caused development to slow down.

References

References

[1] V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, Dec 2017.

[2] C. Finn, S. Levine, and P. Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. *CoRR*, abs/1603.00448, 2016.

[3] J. Ho, J. K. Gupta, and S. Ermon. Model-free imitation learning with policy optimization. *CoRR*, abs/1605.08478, 2016.

[4] A. Kuefler, J. Morton, T. Wheeler, and M. Kochenderfer. Imitating driver behavior with generative adversarial networks. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 204–211, June 2017.

[5] Y. Li, J. Song, and S. Ermon. Infogail: Interpretable imitation learning from visual demonstrations. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3815–3825. Curran Associates, Inc., 2017.

[6]retraining: https://drive.google.com/open?id=1FwG5sTEvJQn6LyRuiAPeFZkvHlDm2mBv