Deep Learning Lab – Assingment 3

Guilherme Alessandri Alvim Miotto – 4653503

December 4, 2018

On this assignment, we had to use imitation learning to train an agent to play the game Car Racing from OpenAI Gym. The model used to mimic the behaviour of the expert was a convolutional neural network. My agent obtained an average **test reward of 864.8 (std 79.9)**. A video[1] of the test run is available at:

`https://youtu.be/oCVkiLBZb24`

1 Expert data

The first part of the project was to play the game manually in order to generate samples to train the neural network. I introduced two changes in the controls to make it easier to control the car:

1. Increased break strength. This allowed me to generally drive faster and reduce the velocity just before turns.
2. Implement logic that automatically cuts gas when the turning. It largely reduced drifting.

Of course, this kind of settings is something very personal. It was something that worked for me, but can't be seen as an absolute improvement.

Twelve episodes were recorded, which gave a total of 21.318 samples. The average reward was 897.2 (std 23.1).

2 Data preprocessing

Before using the expert data to train the network, some data preprocessing was made.

[1]The video starts with some screen flickering, but after 1:10 it goes away.

2.1 Sample rejection

Not all samples from the expert dataset were used for training. I started by discarding the first 50 samples of each episode. Later on I will explain why those samples are not necessary. After that, I discarded samples in order to have a more balanced distribution of actions. Figure 1 shows the original distribution of actions in the dataset. The id of each action is shown on Table 1. As we can see, there is a major predominance of the action *accelerate*. This may cause problems while training the network, because it may converge to the solution of always outputting *accelerate*. So I randomly removed 50% of the samples labeled as *accelerate*. The resulting data distribution is shown on Figure 2. After these two evictions, the dataset reduced from 21.318 to 14.213 samples; a decrease of 33.4%.

2.2 Image editing

The samples that remained underwent a state (image) preprocessing. The aim here was to reduce the number of features of the images trying to leave out only those that are indeed important for the agent. This tends to make the neural network training easier. A sequence of image editions were performed:

1. Hide the reward counter;
2. Recolored the status bar with gray. This made the velocity indicator more visible.
3. Recolored grass with a single shade of green.
4. Recolored road with a single shade of gray.
5. Recolored curbs with the same gray used on the road;
6. Converted the image to grayscale.

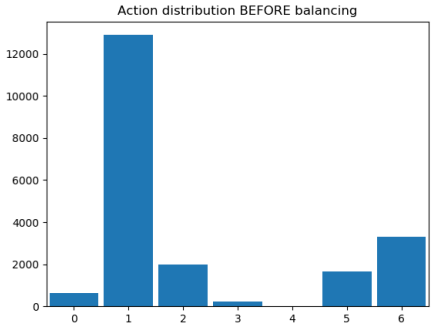

Figure 1: Action histogram before balancing

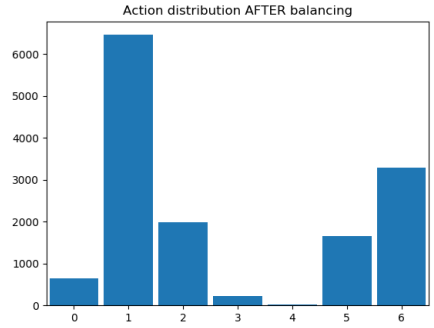

Figure 2: Action histogram after balancing

Id	Action
0	Drive straight
1	Accelerate
2	Turn right
3	Turn right braking
4	Brake
5	Turn left braking
6	Turn left

Table 1: Action encoding

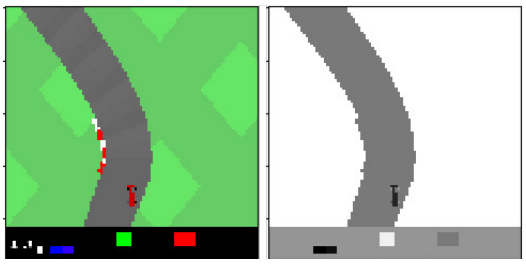

Figure 3: State preprocessing. Before (left) and after (right)

The final result is shown on Figure 3. At the left-hand side is a state before preprocessing and at the right-hand side is the same state after it.

2.3 History of states

The code is able to work with states that are composed by a stack of subsequent states, i.e. history. I've tested values for history length ranging from 1 to 5. Nevertheless, the best final outcome (driving performance) was obtained with history length equals 1. Therefore, this was the value used.

3 The agent

The agent's decision making is governed by two mechanisms: a neural network, and an overwrite logic.

3.1 Neural network

The most important part of the agent is the neural network. As requested by the tutors, it was totally written in raw Tensorflow, i.e. low-level API. Nevertheless, I wrapped my code in a package I named (for lack of creativity) *my_neural_network*. It works like a high-level API, analogous to Keras or PyTorch's Sequential. This package allowed me to be more agile while experimenting with different architectures.

Table 2 shows the architecture used. I used Adam with learningg rate of 5×10^{-4} as the optimizer for minimizing a cross-entropy loss. The network was trained for 200.000 steps with batches of 100 samples. Figure 4 shows the evolution the loss throughout the steps. As we can see, the loss stabilizes early on, however, I decided to train longer because the validation accuracy was still steadily improving (Figure 5), finally converging around 69%. The training accuracy (Figure 6) quickly converged to 68% and noisily oscillated around that value throughout the training.

#	Type	Size	Stride	Activ.
1	Conv2D	5x5x16 filter	4x4	ReLU
2	Dropout	50% drop	—	—
3	Conv2D	3x3x32 filter	2x2	ReLU
4	Dropout	50% drop	—	—
5	Dense	128 units	—	Linear
6	Dense	10 units	—	Softmax

Table 2: Layers of the neural network

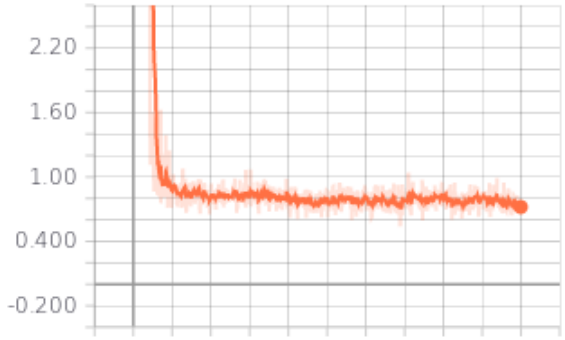

Figure 4: Training loss

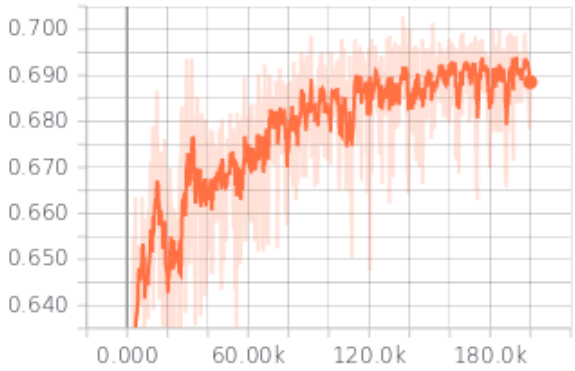

Figure 5: Validation accuracy

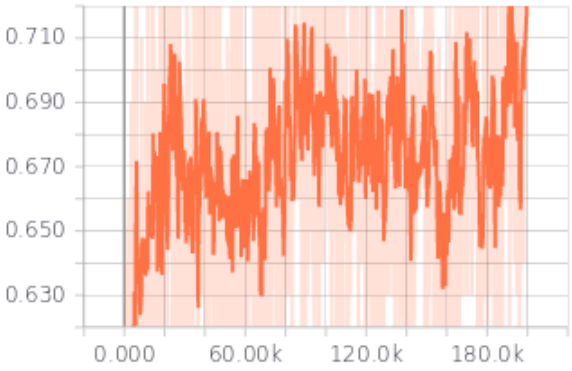

Figure 6: Train accuracy

3.2 Overwrite logic

During the majority of the time, the agent's actions are determined by the neural network. However, there are two situations on which the neural network is overwritten by simple deterministic rules.

The first one is at the beginning of the episode. For the first 50 states, the action is *accelerate*, regardless of what neural network says. Due to this rule, I could, as mentioned earlier, discard the first 50 samples of every episode of the expert dataset. Those samples are peculiar because they show the track from a different zoom level. Therefore, there is a profit, even that small, on removing this heterogeneity from the training dataset. With this rule, the agent will never encounter states with unusual zooming levels.

The second overwrite situation is when the car is stuck. On very sharp turns, the agent will drastically reduce the speed and, on somewhat rare ocasions, for some reason, it will stop and stay put until the end of the episode. This is a clear failure of the neural network that I was not able to solve on the due time. So I decided to implement a simple "anti-freeze" mechanism. The idea is that the agent is constantly monitoring the history of the last 100 actions. If all actions in that history are the same and they are not *accelerate*, the agent overwrites the neural network with a series of short *accelerate* bursts. This has proven effective to "unfreeze" the network, that than is able to take over again.

This "anti-freeze" mechanism had an enormous impact on the average reward of the agent. Those freeze situations, even-tough they are not so frequent, they have a very negative impact on the episode's reward. Once this problem stopped occurring, the average episode reward increased significantly; around 100 points. This overwrite logic can be seen on minute 2:30 of the video I linked at the beginning of this document.

4 Results and discussion

As mentioned earlier, the agent was able to achieve an average reward of 864.8 across 15 episodes. I consider this a very satisfactory result, specially when taking into account that the expert's average is 897.2.

While working on this assignment I reached some conclusions.

1. To get achieve high rewards, it is very important to have a good expert. This sounds obvious but the reason behind it is not that straightforward: The neural network is totally **unaware of rewards**. In other words, in imitation learning the model is just trying to replicate the expert. It's accuracy increases if it does all the mistakes the expert did. It was fun to watch my agent doing the same wrong moves I often did.

2. Validation accuracy is correlated with driving performance, but not so strongly. Sometimes I generated agents with lower validation accuracies, but higher average rewards. The reason for that is the same as mentioned on the item above.

3. State preprocessing matters. Images with less useless features makes convergence faster and allows good results with simpler and faster architectures.