Report

Alessandro Valentino

1 Introduction

The aim of the home task is to train an agent to play the game Car Racing[1] by OpenAI gym using imitation learning.
As by the OpenAI documentation, the environment description is as follows

"State consists of 96×96 *pixels. Reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles in track.[...]Episode finishes when all tiles are visited."*

Imitation learning is a reinforcement learning technique based on showing the agent *trajectories* (i.e. collections of pairs (state, action)) taken from an expert behaviour, which are considered to be samples from the optimal policy for the given environment. Supervised learning is then applied so that the agent can learn to approximate the optimal policy.

2 Expert data

The first part of the home task requires to collect expert data to train the agent. For this part, I played the role of the expert, namely I generated the samples by playing the game manually, and recorded the images representing the states and the actions via the script `expert_traing.py`.
I collected 33333 samples through 16 episodes (a playing time of roughly 20 minutes), scoring an average total reward of 788.8 (+/- 52.3).

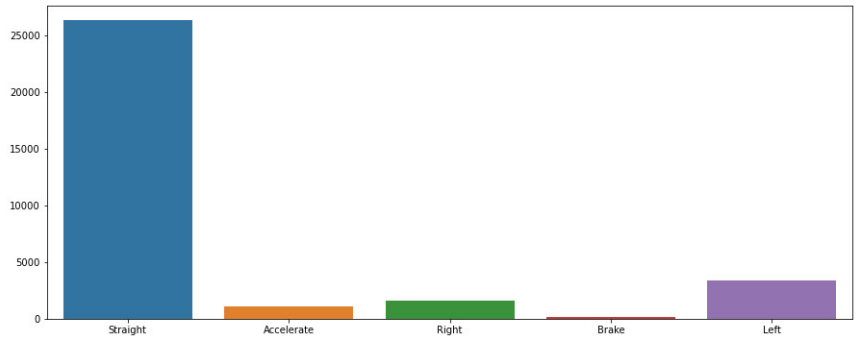
3 Data analysis and pre-processing

After collecting the samples, I performed some data analysis and pre-processing, whose outcome I will briefly describe. More details are available in the Jupyter Notebook `DA expert data` attached.

[1] `http//gym.openai.com/envs/CarRacing-v0/`

3.1 Actions

The environment accepts an action in the form of a 3d vector a.
The first thing that I did was to classify an action into one of the following five categories: Straight, Accelerate, Left, Brake, Right. This was done by noticing that the possible values in the steering component (namely, $a[0]$) of the action vector a are 0.0, -1.0, 1.0 corresponding to keeping the actual velocity (Straight), turning left (Left) or turning right (Right), respectively.
After discarding the samples containing invalid actions, i.e. those that do not fit in the above categories due to some multiple keypressing while playing the game (a total number of 13), I looked at the distribution of actions



which showed that the dataset is heavily unbalanced towards the Straight action. This is due to the fact that keeping the car's velocity without keeping accelerating is quite a common action to take while playing the game. I then expected the learning algorithm would learn this action "too much". One possible strategy to reduce the relevance of this issue could be to discard some of the samples containing the Straight action (say 50% of them), in order to have a more balanced sample distribution. Nevertheless, this would have entailed a heavy loss of samples, and since it has not shown in practice any improvement of the learning model, I decided not to follow this strategy.
Finally, I *one-hot encoded* the actions to create the target labels for the learning algorithm, and saved them to a npy array.

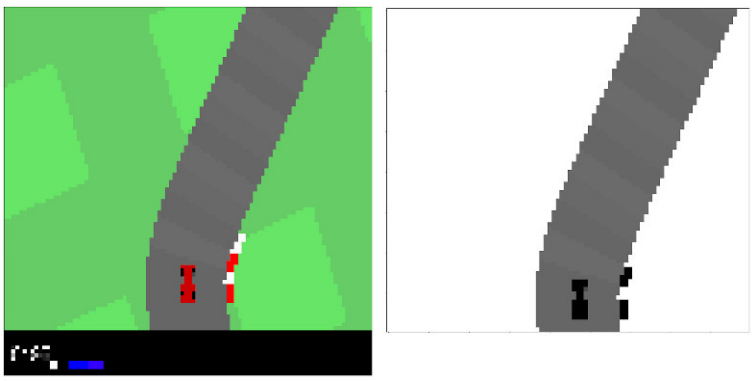
3.2 Images

Each state is represented by a 96x96 RGB image. I first decided to discard the first 50 frames (hence the first 50 samples) from each collected episode trajectory, since this is a "zooming scene", and does not contain any useful information for the agent. After some experimentation, for each image I then applied the following pre-processing

- Recolored the green patches (see figure below) with the same color;
- Rescaled the pixels' range to the interval [0.0, 1.0];

- Cropped the image to 80x90 pixels to exclude the lower status bar, and reduce size;
- Kept only the green channel, and inverted the colors.

The effect of the pre-processing is as follows



The pre-processed images were finally saved to a npy array.

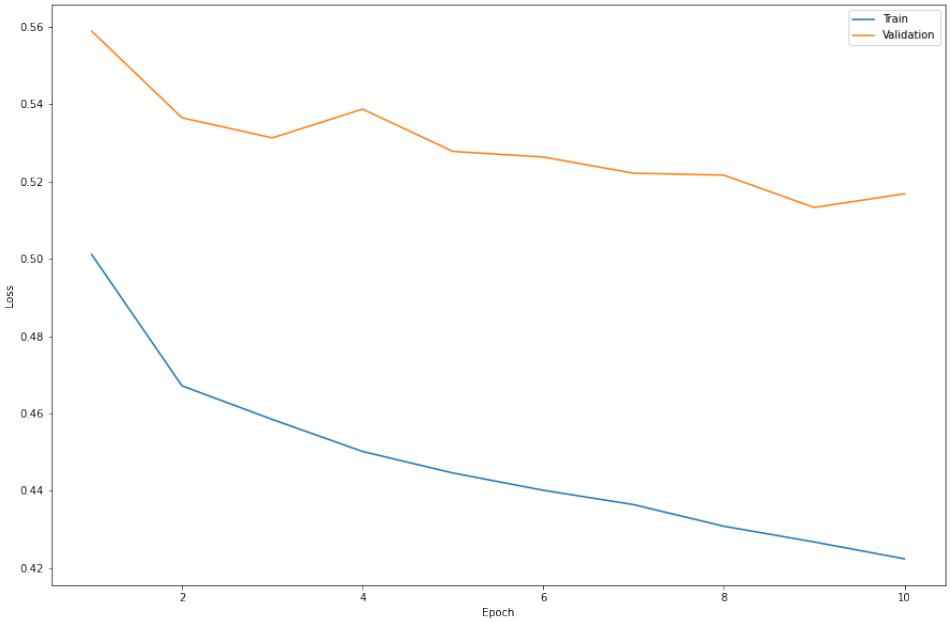
4 The agent

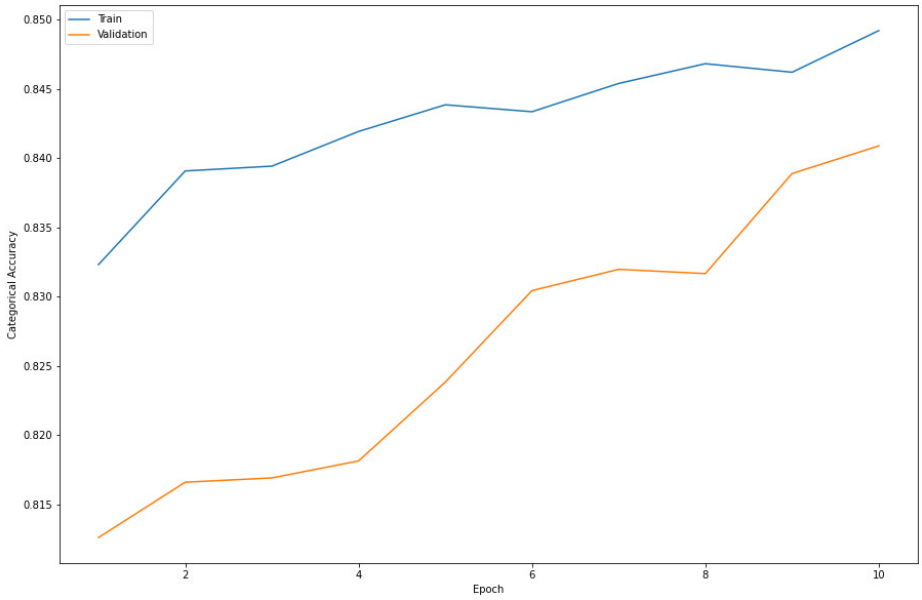
The agent's main decision making is based on a simple (deep) convolutional neural network, which is overwritten in some few situations (see section 4.1).
The neural network architecture is given by the following layers:

- A 2D Convolutional layer made of 16 filters of kernel size 5x5 with 4x4 strides and ReLu activation function;
- A Dropout layer with 50% drop;
- A 2D Convolutional layer made of 32 filters of kernel size 3x3 with 2x2 strides and ReLu activation function;
- A Dropout layer with 50% drop;
- A Dense layer with 128 units and linear activation function;
- A Dense layer with 5 units and softmax activation function.

The loss function is given by the categorical crossentropy, and I used the Adam optimizer.
I implemented the network in Tensorflow/Keras and trained it on Google Colab[2] using GPU.
After experimenting with the various hyperparameters, I decided to use a learning rate of $5\text{x}10^{-4}$, and train the network for 10 epochs with a 64 batch size.
The train and validation losses for the network are given by

[2]Tensorflow.ver = 2.4.1, Keras.ver = 2.4.0





From the plots one can see that the training loss slowly reaches 0.43, while the validation loss is starting to raise just above 0.52 in the last epoch. Since continuing the training would decrease the training loss, but make the validation loss oscillate around this value, I decided 10 epochs were enough, and

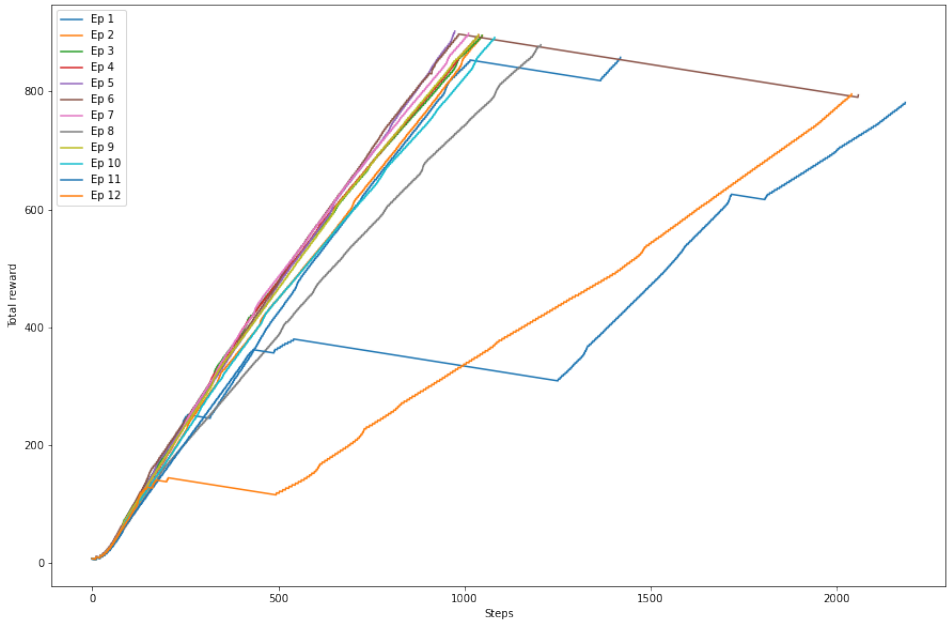
stopped the training. Moreover, one can see that the categorical accuracy keeps increasing and there is not so much overfitting present.
The (relatively) high value of the categorical accuracy needs some discussion. By looking at the confusion matrix over the validation set, one can see that most of the accuracy comes from correctly identifying the Straight action. Since the dataset is heavily unbalanced towards this action, this is expected. At the same time, one can see that in some non-negligible cases the network "misunderstands" the other actions (in particular Acceleration) by being too confident about the Straight action. I then decided to reduce the confidence of the network regarding the Straight action by recalibrating the probability vector produced by the softmax layer with a set of weights. This is quite a usual machine learning technique when dealing with unbalanced datasets and probabilistic models. After some educated guesses driven by the confusion matrix, I managed to obtain a satisfying choice of weights.

4.1 Dealing with acceleration and being idle

To deal with the acceleration issue, I decided that for the first 50 frames of the game the agent should not follow the actions determined by the neural network, but rather accelerate. This is consistent with the fact that during the first 50 frames of each episode there is a "zooming scene", which does not contain any useful info, and has been discarded from the sample dataset. This helps the agent to start moving and continue with a certain constant velocity. After that, I decided that the agent should check if its last 100 actions are all equal ("idleing") and do not include acceleration: in this case, the agent gives some little bursts of acceleration. This can be helpful in those parts of the track were the road is straight and long, and allows the agent to get some more velocity, and hence improve performances by obtaining a greater reward.

5 Results and discussion

I tested the agent for 12 episodes, and it was able to achieve an average reward of **865.4 (+/- 44.8)**. It obtained a better result then the expert, which I am very satisfied about.
The following is a plot of the cumulative rewards per episode



We can see that for most of the episodes the agent is able to finish the episode in about 1000 steps. By direct inspection it was also clear that the agent was cloning my behaviour, which in playing the game was quite erratic (occasional "zigzagging", etc.). This mentioned, I was quite satisfied to see the agent being able to bring the car back on track when going off road, sometimes with impressive maneuvers! During the episodes in which the total reward curve decreases before increasing again, the agent still correctly stays on the track and recognizes the curves, but the environment only gives a positive reward if the agent is visiting a new tile. This implies that if the agent misses some tiles, it will not receive any positive reward for all the time spent through the already visited part of the track until it finds the missing tiles.
From the plot we can also deduce that the agent is often choosing not to drive very fast, i.e. not to accelerate often. This was expected, since the Acceleration action is not very present in the training dataset, as a consequence of my playing style.
Some conclusions on working on this task are the following:

1. The rewards depend on having a good expert providing the trajectories, i.e. someone being good at playing the game. The neural network will try to closely mimic the expert, but it is otherwise not aware of the actual rewards that the agent receives from the environment.

2. A bigger dataset of samples would be more appropriate, since it would allow to discard some of the Straight actions which will inevitably appear disproportionally, but still provide enough samples to train the neural network. Also, while collecting the expert data one should maybe consciously

show the acceleration action more often, so that the network can learn it correctly. On the other hand, it would require a longer playing time and at a higher average playing skills.

3. It would be interesting to develop a mechanism to learn the recalibration weights vector. One possibly elegant solution, for my personal taste, would be to do it via an evolutionary algorithm: in a nutshell, one could create many agents with randomly distributed weight vectors, choose those with high fitness per episode (i.e. highest rewards, fastest, etc.), create a new generation of agents by mating, mutating, etc. , and repeating the process. I expect that this could generate agents which try to reduce the steps needed to end the episode.

4. Another implementation I would be interested in carrying on is to apply other forms of imitation learning, like Direct Policy Learning, by providing the agent with expert feedback while rolling out the policy.

5. Image pre-processing is an important step. In this case I considered only the cropped image, though there could be in principle useful information on the status bar that the neural network could exploit.