Reinforcement Learning for a Simple Racing Game

Pablo Aldape
Department of Statistics
Stanford University
paldape@stanford.edu

Samuel Sowell
Department of Electrical Engineering
Stanford University
alex4936@stanford.edu

December 8, 2018

1 Background

OpenAI Gym is a popular open-source repository of reinforcement learning (RL) environments and development tools. Its curated set of problems and ease of use have made it a standard benchmarking tool for RL algorithms. Furthermore, its challenges are designed to be computationally tractable on modern consumer-grade hardware.

We saw OpenAI Gym as an ideal tool for venturing deeper into RL. Our objective was to conquer an RL problem far closer to real-world use cases than the relatively clean examples found in *DMU* or homework assignments, and in particular one with a continuous action space and very high-dimensional state space. The `CarRacing-v0` environment provided exactly this, with a dash of video game excitement to boot.

As an online learning task involving the learning of in-game actions from pixels, `CarRacing` is a prime target for deep reinforcement learning. Much of our methodology is due to Mnih et al.'s landmark exploration of deep reinforcement learning for Atari gameplay [2], and the course notes which clarified and expanded upon it [3]. More recently double and dueling deep Q-networks have seen success in solving similar tasks [5].

2 Problem description

2.1 Gameplay and reward

The objective of `CarRacing-v0` is to pilot a car through a randomly generated two-dimensional world of racetrack, grass, and boundaries, reaching the end of the track in as little time as possible. Predictably this encourages driving on the road and avoiding boundaries; from playing the game ourselves we know that the car tends to spin out of control on the grass, causing much time to be wasted during the herky-jerky journey back to the road.

The published implementation of `CarRacing-v0` scores a completed run as follows:

$$\texttt{score}(t) = 1000 - \frac{t}{10},$$

where t is the number of in-game frames it takes the car to reach the end of the track. The documentation claims that an algorithm which routinely achieves scores above 900, i.e. directs the car to the finish in at most 1000 frames, is sufficiently fit.

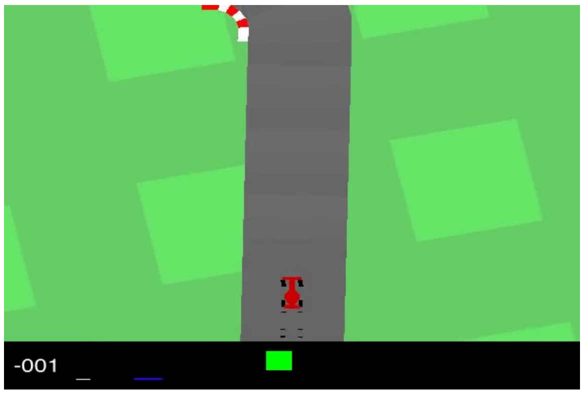


Figure 1: A snapshot of the in-game screen.

2.2 State space

In accordance with how a human would learn to play the game using the arrow keys, the input to our learning algorithm is the sequence of frames for the in-game screen, each represented as a $96 \times 96 \times 3$ grid of RGB values. In the absence of simplifying assumptions about the screen, then, the size of our state space is a staggering $256^{3 \cdot 96^2}$. Of course we can use our domain knowledge to preprocess the screen in a way that drastically reduces the size of our state space while sacrificing little to no information relevant to the "driver." Some methods for doing so are listed here:

(i) Resizing and cropping the image; say, halving the resolution and trimming down the sides. This can reduce the number of possible screens to around $256^{3 \times 48 \times 36}$.[1]

(ii) Classifying pixels only as "road" (1) or "not road" (0) using their color. This reduces the base of the exponential to 2; in conjunction with (i) our cardinality would be about 10^{1560}.

Clearly these simple methods do not get us anywhere near a manageable state space. Additionally, all preprocessing strategies come with a computational tradeoff. Since our learner receives frames in real time from the environment, any preprocessing subroutine must be executed at each timestep. In some implementations this can dramatically slow training. Furthermore, as we refine our preprocessor to use more and more sophisticated methods to label states[2] we drift further away from generalizability, which is a goal of our project and in general a Good Thing.

2.3 Action space

The action space is the set of triples $(s, a, d) \in [-1, 1] \times [0, 1] \times [0, 1]$, where the steering coefficient s ranges from hard left to hard right, the acceleration a ranges from none to full

[1] We know from playing the game ourselves that "zooming in on" the car too much is a poor idea, as it moves fast enough that bends in the road several car-lengths away must be planned for in advance.

[2] An example we considered is explicitly sorting states into categories like `road/straightaway`, `road/sharpLeftTurn` or `grass/roadOnRight`, similarly as a human might describe the game state in language.

steam ahead, and the deceleration d ranges from none to slamming the brakes. Reading the source code reveals that in the human version of the game (controlled with arrow keys) the actions are discretized, with the left arrow indicating $s = -1$, down indicating $d = 1$, etc. Since we were able to achieve good results with the arrow keys and since many implementations are simplified by a discrete action space, we made the early decision to restrict the output of our learner to

$$\mathcal{A} := \{\texttt{left}, \texttt{right}, \texttt{accelerate}, \texttt{decelerate}, \texttt{nothing}\}.$$

Conceptually this simplifies the problem somewhat by loosely reframing it as a classification task from images to action labels. The comparison is not precise, however, since contextual data outside the frame (e.g. velocity) should inform the choice of action as well.

3 Solution approaches

3.1 Classical reinforcement learning

The size of our state space immediately rules out several categories of RL algorithms:

- Maximum likelihood-based methods are out of the question since at any given timestep it is overwhelmingly likely that we have never before observed precisely the same screen, meaning that the vast majority of our T and R estimates would be undefined.
- Table-based model-free methods such as Q-learning and Sarsa are also ruled out, as the space of state-action pairs (s, a) is far too large to be stored in tabular form, even given all the memory in the world. Using sparse matrix representations for Q-tables is no help since it does not address the underlying issue that the huge majority of $Q(s, a)$ values will never see an update.
- Linear approximation of the form $Q(s, a) := \theta^T \beta(s, a)$ does not admit clear choices of basis function. Generally speaking, compressing an image s into vector form is a task ill-suited to manual, problem-specific implementation.

What, then, to do? In order to proceed we need a low-dimensional representation of the game screen which will allow us to approximate $Q(s, a)$ for unseen s. Deep Q-learning is up to the task.

3.2 Deep Q-learning

The essence of deep Q-learning is the estimation of $Q^*(s, a)$ using a type of neural network called a Deep Q-Network (DQN) parametrized by a vector θ. At training iteration i we write the network parameters as θ_i. The loss at this step is given by the temporal difference error

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s'}\left[\left(R(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)^2\right].$$

Differentiating with respect to θ_i, we find that loss is minimized when the Bellman equation is satisfied, i.e.

$$Q(s, a; \theta_i) = \mathbb{E}_{s'}\left[R(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})\right].$$

This in conjunction with the fact that neural networks are universal function approximators implies that, given sufficient training data, a DQN will learn the optimal values of Q.

Deep Q-learning needs some tweaks to succeed in gaming contexts. The first involves decorrelating successive training samples. We do this using experience replay. With experience replay all "experiences" (s, a, r, s') observed in training are stored in a database $\mathcal{D}$, and at each iteration a random sample of experiences is drawn to train the network. For simplicity we trained on only one random experience in $\mathcal{D}$ at each timestep. The second addresses the issue of a target value which changes in time with the network parameters θ_i, using an auxiliary network called a target network. We did not implement target networks in our code. Further discussion can be found in [2].

4 Models

We implemented three different classes of neural network, to varying degrees of success.

4.1 Fully connected neural network

The first of the three was a simple fully connected architecture where the first layer consisted of 9216 nodes (one node for every pixel in the 96x96 grid) and each node took in just the green channel of its respective pixel. The reasoning behind choosing only the green channel was that the different features of the environment (gray road, green grass) differed significantly in their green pixel value, so shaving the R and B channels sacrificed no information. The first layer was fully connected to the second layer which consisted of 5 nodes (one for each action). There was no activation function for either layer. We used an exploration rate of $\gamma = 0.1$ and a learning rate of $\lambda = 0.01$.

4.2 Convolutional neural network

Convolutional layers are particularly well-suited for image recognition and feature extraction, so it was natural to use them as the first layers in a neural network. We used TensorFlow's `Conv2d` to assemble a network with the following architecture, again using only the green channel as input:

1. 32 8x8 filters using ReLU activation.
2. 64 5x5 filters using ReLU activation.
3. 64 3x3 filters using ReLU activation.
4. Flattening layer.
5. Dense layer of 64 nodes, linear activation.

6. Dense output layer of 5 nodes (corresponding to our action labels).

We kept the same learning and exploration rate of 0.1 and 0.01 respectively.

4.3 Transfer learning with VGG16

One issue with using untrained custom CNNs is that training the convolutional layers can significantly slow down the model, while the goal of our project is not feature extraction but the prediction of Q values from these features. To address this we incorporated a pre-trained CNN called VGG16 into our model. VGG16 is a 19-layer network composed of `Conv2D` and `MaxPooling` layers trained on ImageNet (a set of 1.2 million images with 1000 categories), and a top performer in localization and classification competitions for that dataset [4]. We concatenated dense Q-predicting layers to this network and marked all VGG16 layers as non-trainable in TensorFlow. Training neural networks with this architecture is known as transfer learning [6]. The full architecture is below:

1. VGG16 (22 layers) (NON-TRAINABLE)

2. Flattening layer.

3. Dense layer of 32 nodes, ReLU activation.

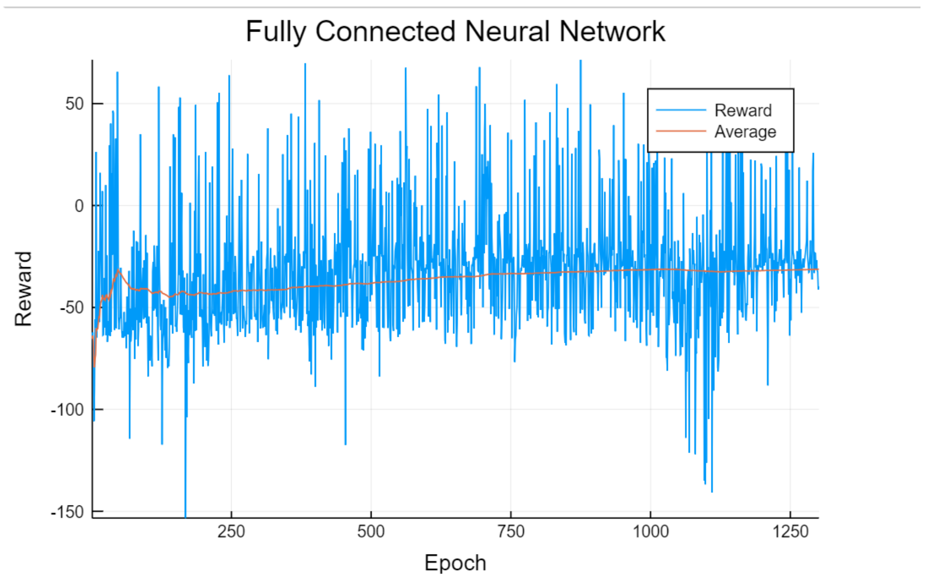
4. Dense output layer of 5 nodes (corresponding to action labels), softmax activation.

5 Model evaluaton

Overall we are not very content with the performance of our models. However, we must recognize that reinforcement learners require many epochs to reach a decent solution, and we believe that we were an order of magnitude off in the number of simulations that we ran on each of our models. What we are happy about is the clear upward trend in our rewards as the epochs rolled by. Knowing this, we are confident that our best model would have reached a reasonable solution given ample time and computational power (i.e. GPUs/TPUs).

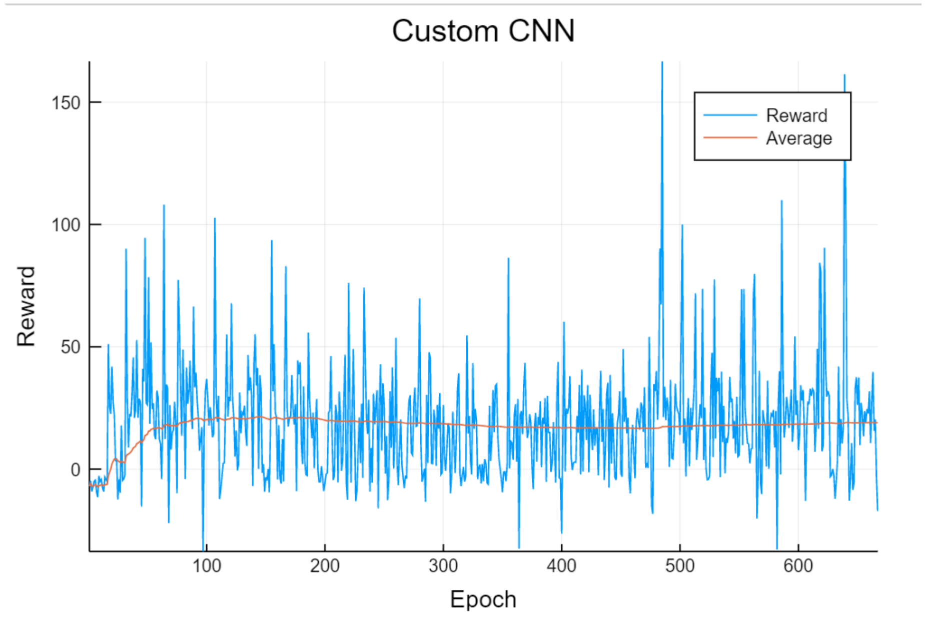
5.1 Fully connected neural network

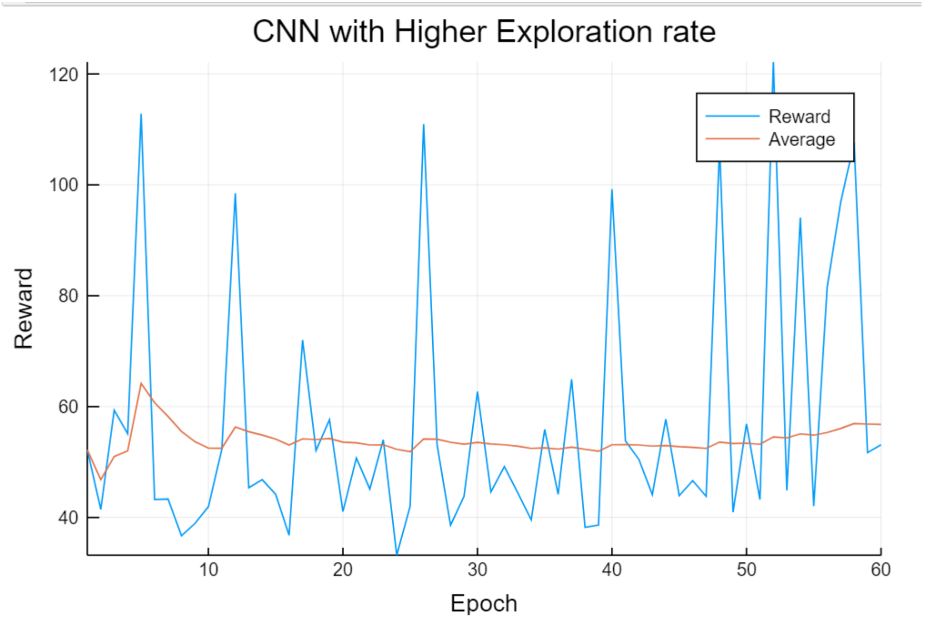
Predictably, our results with the vanilla network were unexciting. We ran the network for about 1300 epochs and saw some noticeable improvement, but not the rapid improvement we would have liked to see. There is a faint upward trend, but after starting around -100 our rewards still ended in the negatives after about 1300 epochs. Perhaps varying our learning rate from the static 0.01 value would have resulted in faster reward increase, but this everyday neural network was intended to be a baseline against we could compare the others, not a top performer.



5.2 Convolutional neural network

Our CNN was a better performer right from the get go compared to our fully connected neural net. Surely this is due to the CNN's superior performance in interpreting the image input. Similar to the fully connected neural network, the CNN started in the negatives for the first couple runs of the simulations, but once it figured out to go forward initially we can see a clear increase in the average reward right around the 50th epoch mark. The rest of the time was spent trying to learn how to turn, and successfully in a couple runs, where our reward was more than 100. After observing some runs, we saw that it had learned to accelerate hard in the beginning, but could not slow down enough to make a significant turn, so the car would start turning and drift into the grass. We believe the runs of 100 or more had very slight turns in the beginning and the car was able to negotiate the turn at breakneck speeds.





We then experimented with the exploration rate of our CNN to see how it affected performance. We may have gotten lucky in this run, but increasing the exploration rate to 0.3 allowed for the model to immediately learn to go accelerate initially and was able to achieve a reward of 100 within the first 5 epochs. Unfortunately, we did not have the time or computational power to explore this option further; however, we did include exploration drop off as the average reward increased with every epoch. We suspect that with the larger exploration rate we would have randomly chosen more turning actions and may have learned to turn faster as well.

5.3 VGG16 transfer network

Our rationale for trying transfer learning was that it might speed up the training process, but sadly we were wrong in this regard. Since the network is so deep, passing 1000 screenshots through it each epoch proved a bottleneck for our puny CPUs, and we had neither time nor patience to let the transfer network train for sufficient time. In preliminary tests, however, the VGG16 layers did do an excellent job in grouping similar screen images.

5.4 Batch Learning

We also attempted to learn off of iterations of game playing done by ourselves. We played the game, trying to get as good of a score as we could, and recorded all (s, a, r, s') experiences while we played. Again, we encoded our state to just include the pixel value of the green channel. We then wrote the tuples to a text file and read them during training. Although the idea has proven to be useful in other domains, we didn't get results even worthy of showing in this paper. Based on our technique of playing the game, the best action in many cases was to do nothing. This was because we would accelerate at the beginning to a slow, but steady speed, then coast as we finished (coasting as in not accelerating again). Our model interpreted our strategy as doing nothing every time, so it would just sit there. We attribute this to our mistake of not including a speed component in either the state or reward function so our model did not know how fast it was going. Basically, our model would interpret the a given state the same way if the car wasn't moving, or was moving really fast.

6 Conclusions

Neither of us had experience in implementing machine learning algorithms prior to this project, and unfortunately much of our research time was spent trying to make up for this. We spent significant time on learning the peculiarities of TensorFlow, and after that on trying to get our algorithms to run on a GPU instance in Google Cloud.[3] Of course we would have preferred to use this time refining our methodology to achieve more convincing results. We could have done a better job making decisions under the uncertainty of how difficult learning this technology would be.

Since so much of our time was spent trying to train our neural networks, we did not have the chance to experiment with alternative discretizations of the action space or leaving it continuous. In our proposal we also planned to fork the game code to include a more sophisticated reward function, train learners on both, and compare their performance both quantitatively and qualitatively. Unfortunately this would have been unproductive given the performance of our base networks.

Though our results didn't live up to our expectations, we learned a lot in the process. Our original proposal for this project was to do something similar to what we did for project 2 and use Q-Learning and Sarsa to achieve our goals for the project. It wasn't until the status update when we found out our problem was much harder than we first expected. This forced us to do a lot of our own research on DQNs and be able to somewhat successfully

[3] We gave up on this because of a combination of software dependency headaches and a lack of funds.

implement it with limited resources. If it were smooth sailing the whole way through, though, we wouldn't have had as much fun.

References

[1] Kochenderfer, M. Decision making under uncertainty. MIT Lincoln Laboratory Series, 2015.

[2] Mnih, V. et al. Playing Atari with deep reinforcement learning. DeepMind Technologies, 2013.

[3] Morton, J. Deep reinforcement learning: course notes. AA 228, Stanford University, 2018.

[4] Simonyan, K. & Zisserman, A. Very deep convolutional networks for large-scale image recognition. ArXiv, 2015.

[5] Juliani, A. Simple reinforcement learning with TensorFlow. Medium, 2016.

[6] Karpathy, A. Transfer learning. CS 231N course notes, Stanford University, 2018.