CARMA: A Deep Reinforcement Learning Approach to Autonomous Driving

Matt Vitelli
mvitelli@stanford.edu

Aran Nayebi
anayebi@stanford.edu

Abstract

We created a deep Q-network (DQN) agent to perform the task of autonomous car driving from raw sensory inputs. We evaluated our agent's performance against several standard agents in a racing simulation environment. Our results indicate that our DQN agent is capable of successfully controlling a car to navigate around a simulation environment.

1 Introduction

Controlling an agent from high dimensional sensory inputs is a key challenge for any reinforcement learning application. One approach to this problem is to use hand-crafted features in conjunction with a linear parametrization of the policy or value function. The downside of this approach therefore is that success is strongly dependent on the quality of these feature representations. Neural networks offer one such solution, as they are able to learn salient features from raw sensory inputs. Mnih et al. [6] brought the approach of combining deep learning with reinforcement learning to center-stage by demonstrating a convolutional neural network (CNN), trained with a variant of Q-learning, that can learn successful control policies from raw video data in order to play Atari. We wanted to scale up this deep Q-learning approach to the more challenging reinforcement learning problem of driving a car autonomously in a 3D simulation environment.

2 Prior Work

The task of driving a car autonomously around a race track was previously approached from the perspective of neuroevolution by Koutnik et al. [4] to control a car in the TORCS racing simulation environment using vision from the driver's perspective. In order to address the issue of high dimensional inputs, Koutnik et al. [4] first transformed the visual input into a compact feature vector using a deep, max pooling convolutional neural network (MPCNN) that was evolved to discriminate between differing images. Next, this feature vector was used to separately evolve an extremely small recurrent neural network (RNN) controller, which would control the car's movements. Instead of using neuroevolution, we will use a neural network for state estimation, in conjunction with a more traditional reinforcement learning algorithm such as Q-learning.

3 Problem Formulation

Our problem is formulated as a Markov Decision Process (MDP) in which at each timestep t, an agent observes a state s_t and performs an action a_t that leads it to a new state s_{t+1} and observes a corresponding reward $r_t = R(s_t, a_t)$. Throughout the paper, we only make use of model-free reinforcement learning approaches and thus do not assume a transition model $T(s_t, a_t, s_{t+1})$ between the states s_t and the next states at s_{t+1}.

State Space Representation

We formulate our state-space as follows: At each timestep t, the agent receives an image I_t of the environment, an estimate of the agent's speed v_t, an estimate of the agent's distance to the center of the road d_t, and an estimate of the angle between the agent's forward vector and the center of the road ϕ_t. Thus, the state s_t is represented as the tuple s_t = (I_t, v_t, d_t, ϕ_t).

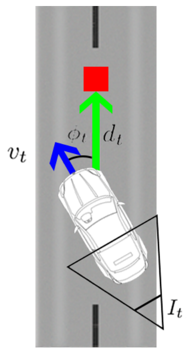

Figure 1: A visualization of s_t

Action Space Representation

At each timestep t, the agent can perform a discrete action $a_t = (acc_t, steer_t)$. We define $acc_t = (Brake, DoNothing, Accelerate)$ and $steer_t = (TurnLeft, DoNothing, TurnRight)$. Thus, there are a total of 9 possible actions the car can take at each timestep.

Reward Function

We based the reward function on simple logic for a hand-crafted car controller defined below. The hand-crafted car controller was designed to safely navigate around a track created in our simulation environment and the corresponding v_{max}, v_{brake}, and $\phi_{threshold}$ values were set accordingly so as to ensure the hand-crafted car controller did not depart from the road at any point along the track.

Algorithm 1 Hand-Crafted Car Controller

```
1:  function GetCarPolicy($s_t$)                ▷ Returns a policy for the given state.
2:      $v_{brake} \leftarrow 20$               ▷ Defined in meters/second.
3:      $v_{max} \leftarrow 18$                 ▷ Defined in meters/second.
4:      $\phi_{threshold} \leftarrow 0.05$      ▷ Defined in radians.
5:      $acc \leftarrow DoNothing$
6:      $steer \leftarrow DoNothing$
7:      if $v_t > v_{brake}$ then
8:          $acc \leftarrow Brake$
9:      end if
10:     if $v_t < v_{max}$ then
11:         $acc \leftarrow Accelerate$
12:     end if
13:     if $\phi_t > \phi_{threshold}$ then
14:         $steer \leftarrow TurnLeft$
15:     end if
16:     if $\phi_t < -\phi_{threshold}$ then
17:         $steer \leftarrow TurnRight$
18:     end if
19:     return ($acc$, $steer$)
20: end function
```

Our reward function is defined as follows:

Algorithm 2 Reward Function

```
function REWARD($s_t$, $a_t$)
    $v_{min} \leftarrow 5$                                              ▷ Defined in meters/second.
    $v_{brake} \leftarrow 20$                                           ▷ Defined in meters/second.
    $v_{max} \leftarrow 18$                                             ▷ Defined in meters/second.
    $\phi_{threshold} \leftarrow 0.05$                                  ▷ Defined in radians.
    $reward \leftarrow 0$
    if $v_t > v_{brake}$ and $acc_t = Brake$ then
        $reward \leftarrow reward + 1$
    end if
    if $v_t > v_{brake}$ and $acc_t = Accelerate$ then
        $reward \leftarrow reward - 1$
    end if
    if $v_t < v_{max}$ and $acc_t = Accelerate$ then
        $reward \leftarrow reward + 1$
    end if
    if $v_t < v_{max}$ and $acc_t = Brake$ then
        $reward \leftarrow reward - 1$
    end if
    if $\phi_t < -\phi_{threshold}$ and $steer_t = TurnRight$ then
        $reward \leftarrow reward + 1$
    end if
    if $\phi_t < -\phi_{threshold}$ and $steer_t = TurnLeft$ then
        $reward \leftarrow reward - 1$
    end if
    if $\phi_t > \phi_{threshold}$ and $steer_t = TurnLeft$ then
        $reward \leftarrow reward + 1$
    end if
    if $\phi_t > \phi_{threshold}$ and $steer_t = TurnRight$ then
        $reward \leftarrow reward - 1$
    end if
    if $v_t < v_{min}$ and $acc_t \neq Accelerate$ then         ▷ Encourage forward motion at low speeds.
        $reward \leftarrow reward - 1$
    end if
    return $reward$
end function
```

It is important to note that while the reward function defined above was designed to reflect the hand-crafted car controller's logic, it shares several key differences with the hand-crafted controller. Namely, an agent is only penalized if its policy is precisely the opposite of what the hand-crafted controller would do. In other words, there is no penalty associated with choosing a $DoNothing$ action with both the $steer_t$ and acc_t terms of a given action. As will be shown in subsequent sections, this small but important caveat to the reward function allows our agents to exploit the discounted cumulative rewards inherent to the reward structure and achieve higher average cumulative rewards than the hand-crafted controller was capable of achieving.

4 Modeling Approach

We experimented with various agents, ranging from a discrete agent to a variety of model architectures for the deep Q-network agents.

Discrete Q-learning Agent

Our discrete agent utilizes Q-learning on a discretized version of the state space. For this agent, the states consist of the car's speed, angle from the center of the track, and distance from the center of the track. We discretized the state space into 3 angular bins, 4 speed bins, and 4 distance bins for a total of 48 bins. We distributed the bin values linearly and chose the minimum and maximum bin values to correspond roughly to the threshold values used in our hand-crafted car controller.

Deep Q-Network Agents

CNN Agent

The CNN agent was provided a 32×32 pixel image of the track from the player's perspective. The CNN architecture that we found that worked best was 7 layers deep, consisting of 5 consecutive convolutional layers and two fully connected (dense) layers. As to our choice of nonlinearity between each layer, we initially started with the standard ReLU activation, defined as:

$$relu(x) = \max\{0, x\}.$$

However, we unfortunately found that this type of activation lead to many neurons being inactive and therefore unused. As a result, we ended up using leaky ReLU nonlinearities between each layer, where a leaky ReLU activation is defined for any constant $a \geq 0$ as:

$$leakyrelu(x) = \begin{cases} x \text{ if } x > 0 \\ ax \text{ otherwise.} \end{cases}$$

The motivation behind using leaky ReLU activations is to allow for a small, non-zero gradient when the neuron is not active [5]. The architecture of this agent, with the filter sizes specified for each convolutional layer, is given in Figure 2 below:

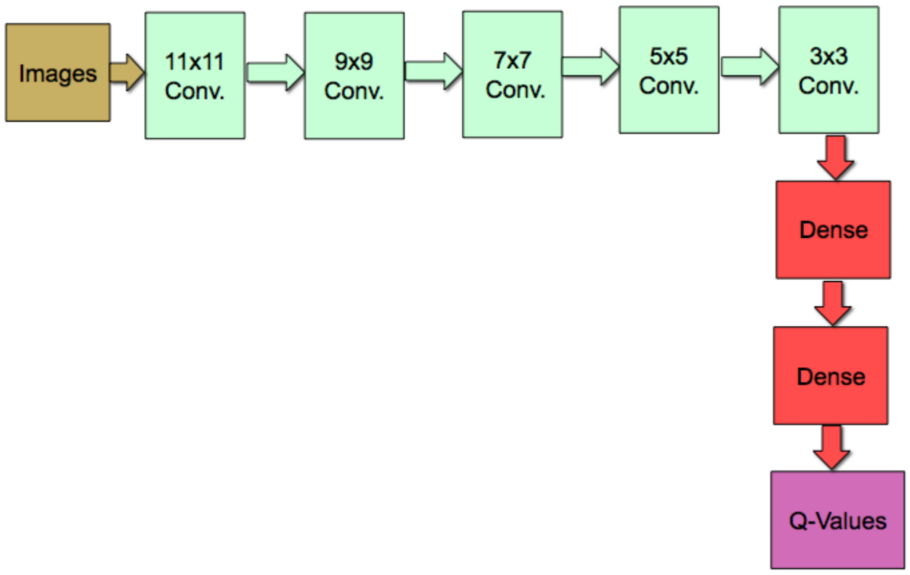

Figure 2: Architecture of the CNN agent.

Unfortunately, while we found that the above architecture worked the best among the CNN agents, it still was not even able to make a complete turn, which motivated us to try an RNN agent.

RNN Agent

The RNN agent was provided as input a vector consisting of the car's speed, angle from the center of the track, and distance from the track. We used a Long Short-Term Memory (LSTM) network [1][2] for the RNN as this type of network utilizes gating units to overcome the issue of vanishing gradients that are common when training RNNs. The hidden states of the LSTM consisted of 32 units and the LSTM maintained a history of 50 timesteps, which allowed us to train the RNN agent in real-time. The architecture of this agent is given in Figure 3 below:

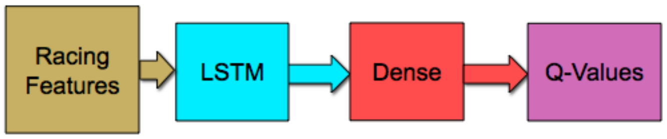

Figure 3: Architecture of the RNN agent.

We found that the RNN agent drove smoothly, but could only complete around a half a lap of the track on average. As a result, this motivated the DQN agent we finally settled on, namely, a hybrid CNN-RNN architecture.

Hybrid CNN-RNN Agent

The hybrid CNN-RNN agent combined both the CNN and RNN architectures in a novel way. Rather than having our agent take in a single input, we created a new agent that takes in two inputs. The first input consists of the 32×32 pixel image of the track from the player's perspective, and the second input consists of the vector containing the car's speed, angle from the center of the track, and distance from the track. Next, the outputs of the CNN agent and the RNN agent (namely, their feature vectors) are concatenated and passed through two fully connected (dense) layers. The architecture of this agent is given in Figure 4 below:

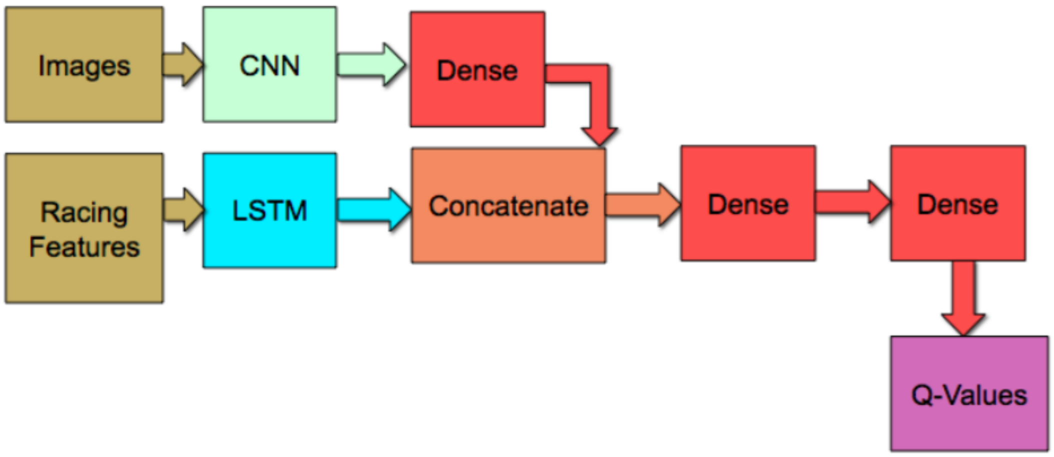

Figure 4: Architecture of the hybrid CNN-RNN agent.

5 Deep Q-Network Training

We found that many practical issues become apparent when training DQNs and there are a variety of techniques that can be used to accelerate the DQN learning process. We have broken up the major steps into several sections outlined below.

Learning Parameters

We found that using Adam as the underlying gradient-based optimization technique tended to work well. Adam is a recently published alternative to adapting the learning rate in gradient-based optimization techniques and was first introduced in Kingma and Ba [3]. Apart from this, we do not use any forms of regularization in our cost function and only used the standard L_2 loss between our DQN's predicted Q-values $Q_{\theta_t}(s_t, a_t)$ and the estimated $r + \gamma \max_{a'} Q_{\theta_{t-1}}(s_{t+1}, a^{'})$ from the

DQN's target network. We found that optimizing the DQN for 4 epochs before updating the target network $Q_{\theta_{t-1}}$ tended to provide a good trade-off between fast convergence and numerical stability.

Reservoir Sampling Experience Replays

During training, we keep around a buffer of size K consisting of K different tuples of (s_t, a_t, r_t, s_{t+1}) sampled throughout the training procedure. We use a standard reservoir sampling procedure to replace these tuples over the course of training. This ensures that all samples over the course of training are sampled uniformly at random and helps improve stability over the DQN learning process. After K samples have been observed since the last $Q_{\theta_{t-1}}$ update, we fit the current network using the standard L_2 loss discussed above. In practice, we found that setting $K = 2048$ samples tended to result in a nice balance between updating the DQN reasonably often, while allowing the DQN to observe sufficiently many new samples using its current best policy.

Randomized Restarts within the Simulation Environment

One common problem we ran into during training was that the DQN agents tended to overfit to the first initial segments of the racing track. This is largely due to the fact that by default, whenever the racing simulator is restarted, the racing cars start at the same starting positions and orientations defined in the track. To combat this problem, we modified our simulation environment so that whenever the car was derailed from the track (and hence entered a failure state), the car would be placed at some random position and orientation along the track. This simple modification greatly improved the convergence rate of the DQN agents and helped combat over-fitting policies to best match the start of the track.

Supervised Policy Learning during Early Convergence

Initially, we trained our DQN agents using the standard epsilon greedy procedure outlined in Mnih et al. [6]. However, we found in practice that this often lead to convergence to sub-optimal policies - particularly when navigating the car around in 3D space using image-based features. Instead, we experimented with implementing a simple extension to the learning procedure by introducing the concept of an oracle agent O_{agent} that can be queried for best actions a_t^o at timestep t. The basic concept is simple: before t_{max} timesteps, the DQN always follows the oracle's actions a_t^o using an epsilon-greedy strategy. After t_{max} timesteps, the DQN simply follows its own actions using an epsilon-greedy strategy. The motivation for doing this is that it greatly accelerates the DQN's convergence and empirically, we found that our DQN agents tended to produce sub-optimal policies without the presence of an oracle compared to using an oracle to shape an initial policy. In practice, we found that setting $t_{max} = 40$ epochs tended to produce reasonable results.

6 System Design

Simulation Environment

We used Vdrift (`http://vdrift.net/`) as the simulation environment for our autonomous vehicle. Vdrift is an open-source, cross-platform racing simulator written in C++. We were able to make extensive modifications to the simulator including implementing GPU-accelerated down-sampling of the frame-buffer, disabling the in-game graphics user interface (GUI), computing the distance and angular terms with respect to the road, and integrating an inter-process communication (IPC) system to publish both the states s_t and apply the corresponding actions a_t produced by our agents. We made use of Nanomsg (`http://nanomsg.org/`), an open-source, cross-platform messaging system for the IPC implementation and used Google Protocol Buffers (`https://developers.google.com/protocol-buffers/`) to handle serialization between Vdrift and our agents.

Agent Implementations

Our agents were created in Python using the Keras deep learning framework (`http://keras.io/`), a popular library that supports modular construction of neural networks and serves as a wrapper to several back-end tensor manipulation systems such as Theano and TensorFlow. Our software system was designed so that agents could seamlessly connect to Vdrift and publish commands as needed. In certain circumstances, the car can get derailed from the racing track, so our agents are also able to reset the simulator if needed. We trained our agents using CUDA-based GPU acceleration on commodity hardware. All of the experiments below were run on a Macbook Pro with an Nvidia GeForce GT 750M GPU. At the time of publication, this hardware is relatively modest by most standards and thus is a testament to what relatively shallow DQNs can achieve with limited computational resources.

7 Experiments & Results

Quantitative Evaluation

We compared our final hybrid CNN-RNN DQN agent against our discrete Q-learning agent, the hand-crafted car controller, and a greedy agent that sought to maximize its immediate reward at each timestep. All agents used the same reward function defined in the Problem Formulation section above. We compared the agents based on three criteria: 1) The average rewards achieved by each agent over the course of its run, 2) the average speed achieved by each agent over the course of its run, and 3) the maximum speed achieved by each agent over the course of its run. All tests were performed on the Vdrift Ruddskogen track and both the hybrid CNN-RNN and discrete Q-learning agents were trained until convergence before measuring the three criteria. Our findings are listed in the table below.

	Average Reward	Average Speed	Max Speed
Hand-Crafted Controller	0.542	17.65 m/s	18.77 m/s
Greedy Agent	1.416	17.53 m/s	18.16 m/s
Discrete Agent	0.635	18.71 m/s	20.21 m/s
DQN Agent	0.915	17.57 m/s	24.71 m/s

Table 1: Performance Comparison between the Different Agents

Several interesting patterns become clear when observing the criteria in the table. First, the hand-crafted controller receives a lower average reward compared to the other agents, but exhibits comparable average speeds to the other agents. This demonstrates that while the reward function is based on the hand-crafted controller's logic, it is not a perfect translation of the controller's logic and allows for some freedom during the learning process. In contrast, we see that the greedy agent consistently outperforms all of the other agents in terms of its average reward, but under performs all other agents in terms of its average and maximum speeds. The reason behind this discrepancy is that the greedy agent tends to seek out high rewards by both accelerating and turning more often. Turning the car in the simulation will cause more friction to the car and hence decreases the car's overall speed.

The DQN agent exhibits the second highest rewards and achieves the highest maximum speed, but has the second lowest average speed. Qualitatively, it seems as though the DQN has converged to a policy where it often accelerates to a point and then uses the Do Nothing action for acceleration to maintain its speed without being penalized. The DQN controller almost never performs braking actions. This tends to lead to great performance in terms of speed, but causes the agent to be more unstable than the other three agents and as a result, it tends to crash more often. Due to the fact that it crashes more often, the average speed tends to be lower as it restarts from a rest state after each crash.

Interestingly enough, our strongest agent seemed to be the discrete Q-learning agent. While the agent's average reward was much lower than both the greedy agent and the DQN agent, its average speeds were much higher and its maximum speed was still competitive with the DQN's maximum speed. Qualitatively, the discrete Q-learning agent maintains a nice compromise between stability and speed and generally only performs braking actions during sharp turns.

Summary

Both reinforcement learning methods exhibited their strengths in navigating autonomous vehicles. While our DQN agent ultimately exhibited more unstable behavior than the traditional discrete agent, we were able to successfully train a DQN to reliably drive around the race track. Both methods outperform the hand designed controller in both their average cumulative rewards and maximum driving speeds.

8 Conclusion

This project has demonstrated the effectiveness of navigating autonomous vehicles using reinforcement learning methods. We have shown that Deep Q-Networks can be an effective means of controlling a vehicle directly from high-dimensional sensory inputs, and we used a novel combination of CNN and RNN networks to achieve this. While currently it seems as though a well-designed, low-dimensional discrete state-space agent is able to more stably control a car compared to a more complex DQN agent, we believe our work could be extended in several ways. Namely, it would be nice to find a better alternative of defining our reward function that still maintains the careful balance between maximizing speed while guaranteeing car stability. Similarly, it would be interesting to generalize our work to continuous action spaces. Despite these limitations, we are still proud that our agents were able to successfully control a car without any explicit notion of the car's underlying dynamics.

References

[1] D. Eck and J. Schmidhuber. "Learning the long-term structure of the blues". *ICANN* (2002): 284-289.

[2] S. Hochreiter and J. Schmidhuber. "Long short-term memory". *Neural Computation* **9** (1997): 1735-1780.

[3] D. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". 3rd International Conference for Learning Representations, San Diego, 2015. `http://arxiv.org/abs/1412.6980`.

[4] J. Koutnik, J. Schmidhuber, and F. Gomez. "Evolving Deep Unsupervised Convolutional Networks for Vision-Based Reinforcement Learning". *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation.* 541-548.

[5] A. L. Maas, A. Y. Hannun, A. Y. Ng. "Rectifier Nonlinearities Improve Neural Network Acoustic Models". ICML 2013.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing Atari with Deep Reinforcement Learning". *Nature* **518** (2015): 529-533.