Car Racing using Reinforcement Learning

M. A. Farhan Khan
SUNetID: mfkhan
mfkhan@stanford.edu

Oğuz H. Elibol
SUNetID: elibol
elibol@stanford.edu

Abstract

The aim of this project is to develop a generalizable system that can learn using signals generated only from its environment to optimize actions in the Car Racing game. Only pixel values and rewards provided by the environment are used to learn an optimal policy with a generalizable solution. Here we report on our work using deep convolutional neural nets and asynchronous advantage actor-critic methods to tackle this problem. Using these methods, we were able obtain an average reward of 652.29 over 100 episodes. For reference, the oracle for this project achieves an average reward of 837.76 over 100 episodes. Our results (oguzelibol's algorithm) and the solution we consider the oracle (ceobillionaire's algorithm) can be viewed at `https://gym.openai.com/envs/CarRacing-v0`.

1 Introduction and Motivation

Systems that can learn by interacting with the environment with minimal explicit feedback from humans are of great interest and importance. Means for machines to perform useful tasks without being explicitly programmed alleviates the cost and effort of training them in a supervised fashion which requires labeled datasets or experts to teach the specific task. Besides, potentially higher performance may be obtained this way. Reinforcement learning takes place as the agent (machine) interacts with the environment which requires minimal (or no) domain knowledge and no explicit programming. As the agent takes actions in the environment it receives feedback from the environment through rewards and state observations. Through this feedback loop between the agent and the environment the goal is for the agent to act optimally (receive maximum rewards) given the environment.

Owing to the advancements in algorithms, models and computing power, there has been great progress being made in the field of reinforcement learning in the recent years [1]. Games have been ideal frameworks to demonstrate and measure the performance of developed reinforcement learning systems as they provide a way to quickly iterate on the algorithms and models in a controlled and reproducible environment, while also potentially relating to real world applications without the additional complexity.

Autonomous driving has also been a rapidly progressing recently and also is a very exciting field, although way too complex to tackle in a class project. Taking inspiration from these developments we solve the greatly simplified problem (compared to autonomous driving) of controlling a race car using only a top down view by using a controller model that is generalizable to other tasks that produce pixel level observation and rewards. The aim of this project is to develop a generalizable system that can learn using signals generated only from its environment to optimize actions in the Car Racing game. Only pixel values and rewards provided by the environment are be used to learn an optimal policy with a generalizable solution.

2 Input-Output Behavior (Simulation Environment)

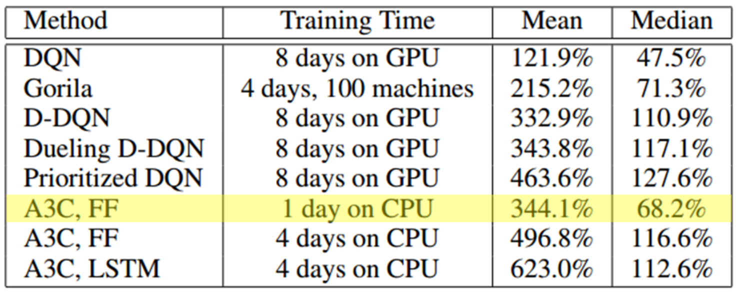
We use the Open AI gym environment [2] to develop our model, specifically the Box2D module with the race car environment [3]. The observation states are 96x96 pixels with 3 color channels for each pixel and 256 magnitude (8 bit) values for each color, representing the view from the top.

For actions, there are 3 continuous values that the agent can choose: For the direction of wheel it can choose a real value from -1(left) to 1(right), for acceleration it can choose a real value between 0 and 1, and also for brake it can choose a real value between 0 and 1. For every frame the reward is set to -0.1 and for every tile that the car clears on the track it gets a reward of $1000/N$ where N is the number of tiles on the track (typically approximately 300 tiles/track). There is no penalty for going off the track other than the frame cost (and the indirect cost of losing stability at hight speeds): the car is no longer on the track and not collecting the rewards associated with visiting track tiles, but getting negative rewards for each frame.

The controller of the game takes continuous values as mentioned [steer,gas,brake] in the range of [[-1 to 1],[0 to 1],[0 to 1]]. First, we discretized the controller inputs with 9 distinct actions (keyboard play implementation of the game): [+1, 0, 0], [-1, 0, 0], [0,0,0], [+1,+1,0], [-1,+1,0], [0,+1,0], [+1,0,0.8], [-1, 0,0.8], [0,0,0.8]. The discrete value for braking was chosen to be 0.8 to prevent the tires from fully locking while braking (for greater stability). Later, because of the limited time for training and limited computational resources, we constrained the action space further to converge on reasonable results. This was a decision made after running with all the 9 actions as well as a truncated 7 action version. Thus, we trained our model by limiting the action space to 4, keeping steer left [-1, 0, 0], steer right [+1, 0, 0], accelerate [0,+1,0] and brake [0,0,0.8].

3 Model

After exploring the literature on deep reinforcement learning strategies, we got more aware of the advantages and disadvantages of various models. DQN methods that had large success on Atari games [1] rely on experience replay memory that reduce the non-stationarity of the data and decorrelates the updates, however it comes at a cost of large memory requirements and specialized hardware and long training times. Recently a method called asynchronous advantage actor critic (A3C) was shown to perform well using a standard multi-core CPU [4]. In our project we chose to implement this model for the Car Racing game. Given that training and tuning times are usually the bottleneck for deep networks, this model helped us in alleviating those concerns and train faster than DQN (Figure 1).



Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

Figure 1: Comparison of different reinforcement methods from [4] (not our work). Scores were obtained on Atari games in that work. We have chosen to implement A3C, FF due to high relative performance and low training time and resources

3.1 Asynchronous Advantage Actor-Critic Model

In the actor critic model, both the value function and the policy function is learned at the same time as shown in 2. In the model, the advantage function $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$ is optimized. The actor can be thought as the policy maximizing $Q(a_t, s_t)$ and the critic optimizing the value prediction of each state $V(s_t)$ to eventually improve the advantage function. Specifically, both the policy ($\pi(a_t|s_t; \theta)$) and the value function ($V(s_t; \theta_v)$) are parametrized, and both are learned. As suggested in [4], we share the same convolutional neural net to estimate both, only differing at the projection layer after the last hidden layer. The output $\pi(a_t|s_t;\theta)$ is estimated by softmax function and $V(s_t;\theta_v)$ is estimated with a linear output as described below in the Implementation section in more detail.

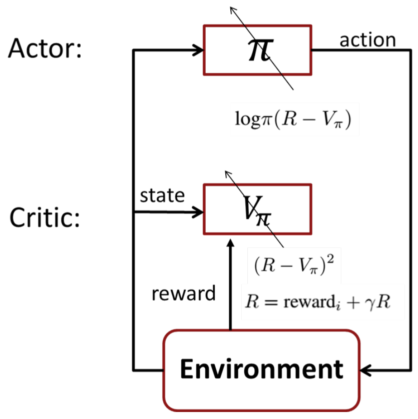


Figure 2: Schematic representation of the actor-critic framework.

3.2 Implementation

We have implemented the model in Python using the TensorFlow library [5]. We have based our model and the hyper-parameters on the cited literature. Specifically, the input image is pre-processed to gray scale images to reduce the dimensionality before feeding in to the network. We stack last 4 states to form a tensor of depth 4, so the state contains information about the immediate past states. This allows the network to extract information about the velocity and acceleration of the car. Using a larger depth may work better in this case, however will come at a computational cost. Our model uses 2 convolutional layers (layer 1: with 16 kernels, size 8x8, stride 4, and layer 2: with 32 kernels size 4x4, stride 2) with rectified linear unit (ReLU) activations, followed by a dense layer to output 256 values . We have tried both resizing the image to 84x84 as done in literature and keeping these values, and also feeding a 96x96 and adjusting the second stage filter size to 3 (from 4) to meet the size requirements. These 256 nodes are inputs to two different projection to generate the action probabilities and state values. Later in the paper we report on results obtained on 84x84 architecture, although 96x96 architecture also produces similar results. Figure 3 shows the architecture used for the model. To generate the action probabilities ($\pi(a_t|s_t;\theta)$) we use a softmax layer (so that the outputs are in the range of 0 to 1) with 4 outputs corresponding to each action that the car can take at that state. To generate the value estimate ($V(s_t;\theta_v)$) of the state we use a dense network that maps to a single scalar output using a linear activation unit (because we want the continuous real value). Thus, most of the parameters are shared between the action network and the value network. The specific architecture and hyper-parameters were chosen based on the literature and existing models as a starting point.

During learning we clip the reward attained in each frame (max of 1) to stabilize the learning. Also we used gradient clipping to 40 to avoid exploding gradient problems. As suggested in literature we used rmsprop as the optimizer, this gives faster and more stable convergence compared to vanilla stochastic gradient descent. We anneal the learning rate linearly to reach 0 at the the maximum allowed learning iterations. We tuned the learning rate (0.0001) and maximum iterations (4 million) to obtain a stable configuration. For all training we used 8 parallel sessions updating the parameters asynchronously.

We also terminate an episode if the current score is decreases below maximum score - 5. This allows for faster training when the car wanders off to the field. Thus the minimum score that can be achieved in each episode is 0.

Loss functions were implemented as outlined in Algorithm S3 of [4] . Specifically, for the action network the loss is $\log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$ and for the value network it is $(R - V(s_i;\theta'_v))^2$ where $R = \text{reward}_i + \gamma R$ and the total loss to optimize is the sum of them. The losses are calculated

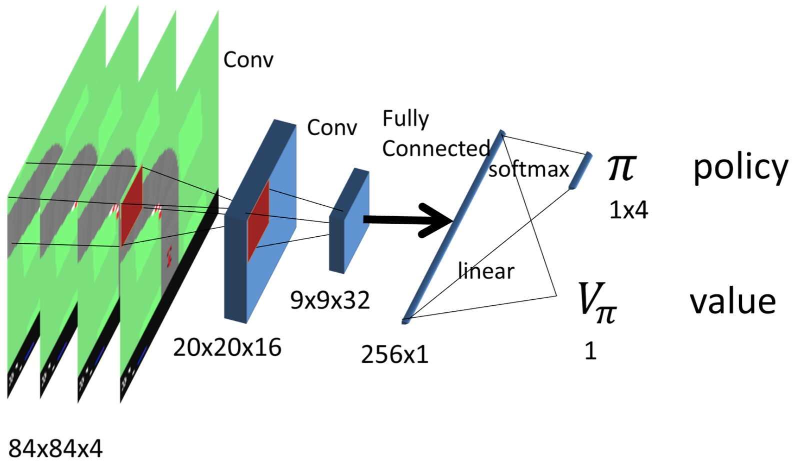


Figure 3: Schematic of the network used for learning. The convolutional neural net contains the shared parameters for the value and the action estimators. After projecting to a 256 dimensional vector the last convolutional layer using a fully connected network, this 256 dimensional hidden layer is used to estimate the value and action at each state. The action is estimated using a softmax layer, while the value is predicted using a full connected network with a linear output

at the thread level (θ'), but the global values are optimized (θ) the global level. We had 8 threads running in parallel for training.

$$\log\pi(R - V_\pi) \tag{1}$$

$$(R - V_\pi)^2 \tag{2}$$

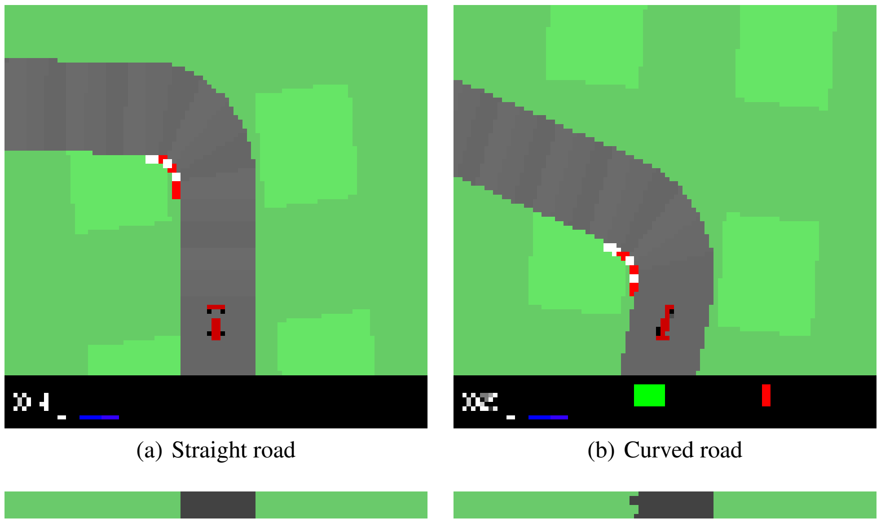
$$R = \text{reward}_i + \gamma R \tag{3}$$

4 Results

4.1 Baseline

We have implemented a baseline for this project using a simple approach without any learning and using our domain knowledge, with different states shown in Figure 4. The idea is to remain on the road as much as possible to gain the maximum reward. We do this by taking a 96x5 pixel observation of each frame directly in front of the car, which is pretty simple because the camera centers on the car. Each element of this matrix stores 1 or 0 representing the presence or absence of a grey pixel (the road). This 96x5 matrix is then reduced into a 48x5 matrix by discarding two quarters from either end, which is then further reduced into a 48x1 vector by adding the 5 rows of pixels column wise. We then compute the sum of the two halves of this vector and make a decision on going left or right depending on which half has a larger sum. Intuitively, if the road turns left or the car leans towards the right, there will be more gray pixels on the left half of our observation window and the required action in this case would be to turn left to remain on the road.

This simplistic approach works well and is able to complete the entire track in 3000-6000 iterations with average rewards varying between 400 to 600. However, the Gym environment's monitoring framework limits each episode to 1000 iterations and using this framework, we were able to achieve an average reward of 145.42 over 100 episodes, which is a decent lower bound for our approach. The distribution of rewards achieved by the car is shown in Figure 5.



(a) Straight road (b) Curved road

(c) Straight window (d) Curved window

Figure 4: Different states captured from baseline implementation

4.2 Advantage Actor Critic

Using the outlined approach (see Implementation), we were able get an average reward of 652.29 for 100 episodes. For reference, the oracle for this project achieves an average reward of 837.76 over 100 episodes. Both our results (oguzelibol's algorithm) and the oracle (ceobillionaire's algorithm) can be viewed at `https://gym.openai.com/envs/CarRacing-v0`. Also, the results comparing the baseline and the actor-critic method are shown in Figure 5:

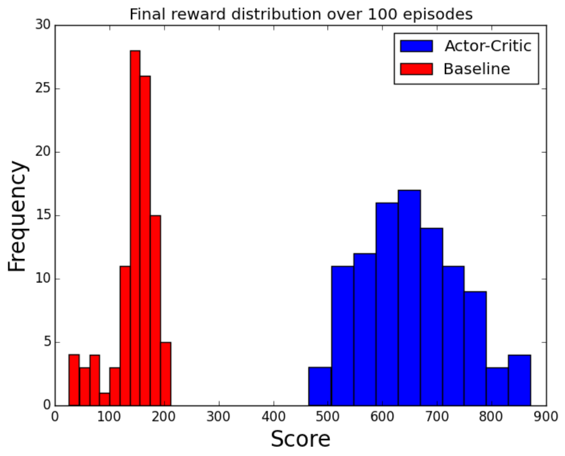


Figure 5: Episode reward histograms for Baseline (red) and Actor-Critic (blue) models for 100 episodes

5 Analysis

Although we were not able to exceed the oracle and surpass the threshold of 900 point to have the problem considered solved in the openAI environment with this work, we were able to come pretty close to the oracle and exceed the baseline by a large margin. In this section we further investigate the behavior of the algorithm and understand what can be changed to improve the results.

5.1 Learning Curve Analysis

We find that the training becomes unstable under for long training times, the maximum score fluctuates. Thus we have resorted to saving and stopping the training when the score reaches above a preset threshold. The training curve is shown below if Figure 6. We get to 1.4 million iterations in about 11 hours on a laptop. We have also observed that during training actions values would favor just steering left and right without accelerating and this corresponds to the 0 reward plateau seen in Figure 6 between iterations 0.7 and 1 million. This possibly happens because the model jumps from one local minima to another. To alleviate the problem we tried decreasing the learning rate, however did not find a suitable rate to solve the problem. Another issue may be in our implementation of the multiple threads, which may cause shared model not being updated appropriately, this is something we will have to investigate.

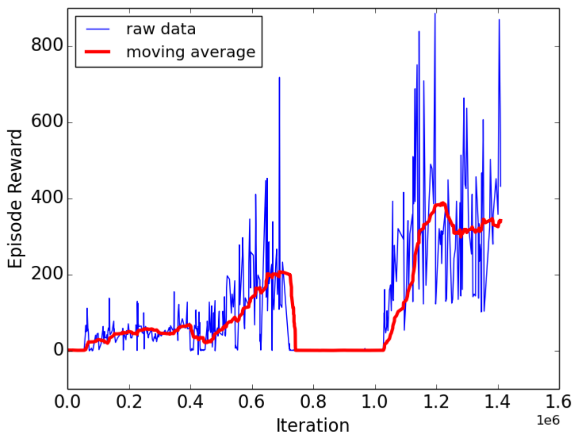


Figure 6: Learning curve for the result presented in this work. Both the subsample raw data (blue) and the smoothing moving average (red) capturing the trend is shown. The checkpoint at 1.4 Million iterations was used as it have the best results. Continuing the training beyond this point results in a average rewards dropping back to 0.

5.2 State Value Analysis

We further analyzed the internal state of model during operation by observing the predicted value, action and states as shown in Figure 7. We see that the state values predicted by the network highly correlates with the difficulty of the state. For example when a tight curve is observed the expected value drops sharply, where as when the car takes the turn the expected value shoots up. Thus we conclude that the value network is learning a reasonable estimator for the value of the state.

5.3 Action Prediction Analysis

In order to gain more insight into the model, in this section we analyze the predicted actions as a function of state. Similar to the previous section, we run an episode and record the action probabilities as a function of the state. Figure 8 shows the instantaneous probabilities of each action as a function of the frame number in the episode (for frames 0 to 200). We see that there are heavy fluctuations in the probabilities from state to state. Also we notice that the probability of braking is nearly 0 for all frames.

To get a cleaner picture of the internal state and the associated states we plot the action probability curves by using a moving average smoothing function and analyze the respective actions as shown in Figure 9

Model settles on a solution which braking is almost never used and due to the lack of a "nothing" state (corresponding to [0,0,0]), it chooses to do left,right actions instead of accelerating or braking. This suggests that the performance of the car can be greatly improved by allowing more actions. Ideally the car should be accelerating during stretches and braking before a corner. However instead

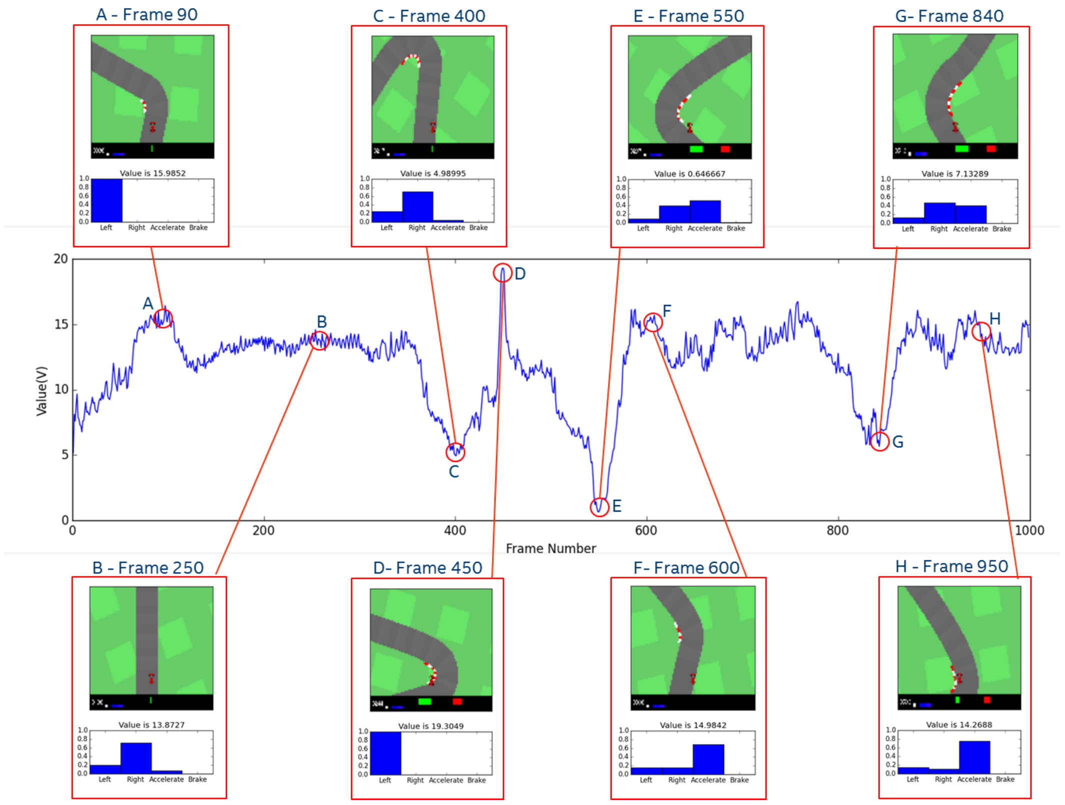


Figure 7: Analysis of the value function and action probabilities at various states. Game begins at Frame 0, only the first 1000 frames are shown for analysis. A - Car is at the first turn and the value function goes up as the car is positioned in a well for the turn. B - With the straight road the value function is also fairly stable. C- Value function takes a dip as a tough turn is detected, perhaps the model has encountered going out of bounds in this situation in the past. D - As soon as car is positioned well in at the turn the expected value of the state goes up. E- As the car is out of bounds for this turn the value dips to a minimum F - Expected value recovers after the car makes the turn (by taking a shortcut through the field). G - Similar to the situation in E the value function dips as the car prepared to take the turn by using a shortcut through the field. H - Fairly steady value prediction as there are no tough turns in the course.

of this it adopts to keep a constant velocity, and only accelerate in situations where it looses velocity (curves). Perhaps improving the implementation, allowing more actions and longer iteration times the model may converge on a more ideal strategy.

6 Conclusion and Next Steps

In conclusion we have successfully shown the application of reinforcement learning to the car racing problem and achieved results close to the oracle. Our analysis shows that there is still plenty of room for improvement as the model settles on an acceptable but suboptimal strategy. Increasing the number of actions, and perhaps even implementing continuous actions will increase the performance of the model. Also improving the multi-threading implementation will allow doing experiments faster to be able tune the learning rate as well as other parameters of the model. We are aware that our implementation is not fully efficient as currently it can't utilize multiple CPUs due to threading used in the implementation and this is an immediate next step that should be addressed to allow quicker iterations for model tuning.

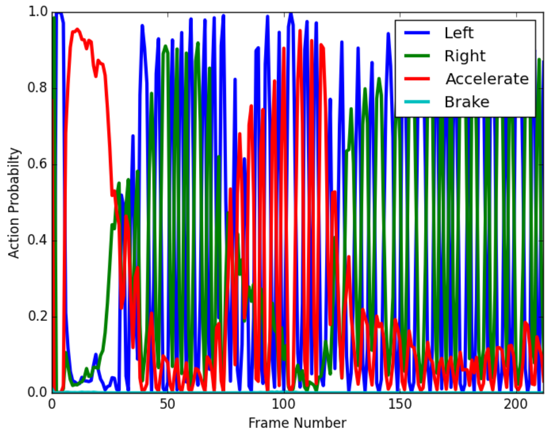


Figure 8: Raw action probabilities for each frame. Notice the heavy fluctuation of the probabilities from frame to fame. Also the probability for braking is close to 0 for all cases.

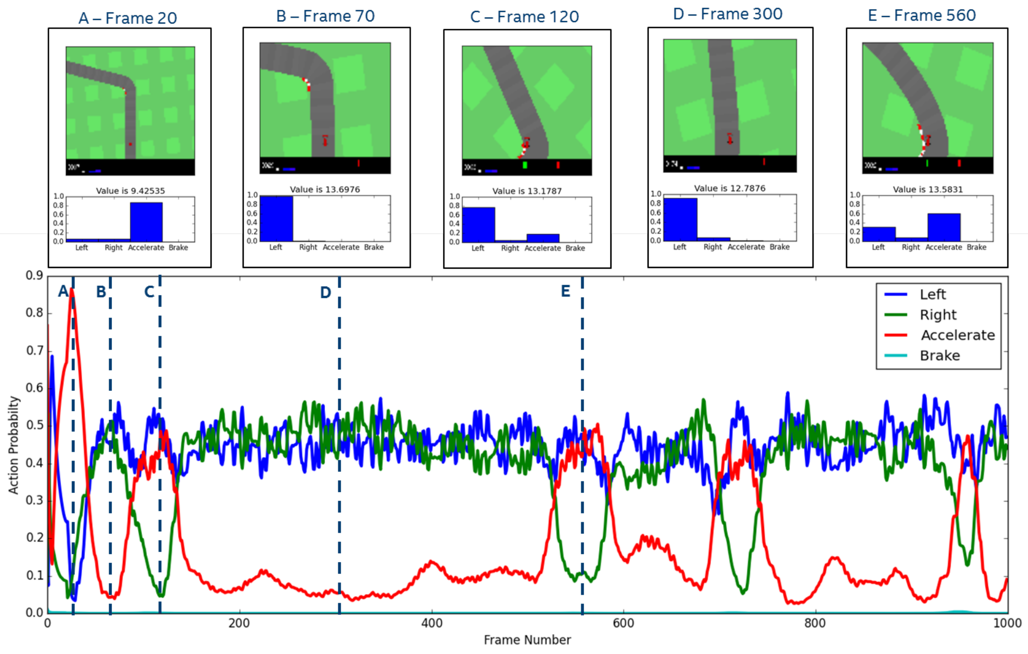


Figure 9: Analysis of the action probability at each frame. Curves were smoothed for better readability. A - Game starts with a zoomed out view of the track and zooms in, during this period acceleration is the highest probability action. B - During a short straight stretch of left and right steer have almost equal average probabilities (although notice that the instantaneous probability at frame B is heavily favors Left, the curve is smoothed out, thus they show equal probability at that frame) C - For a left turn, the action probability is nearly equal for a Left or Accelerate action. D - Similar to B, During a short straight stretch of left and right steer have almost equal average probabilities. E - Similar to C, for a left turn, the action probability is nearly equal for a Left or Accelerate action.

References

[1] V. Minh et al. Human-level control through deep reinforcement learning.

[2] https://gym.openai.com/.

[3] https://gym.openai.com/envs/carracing-v0.

[4] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.