Self-Driving Car Lab
PROFESSOR MICHAEL LITTMAN

SDC Lab
Lane Sharing through Social Reinforcement Learning

—

By Jacob Beck

In collaboration with:

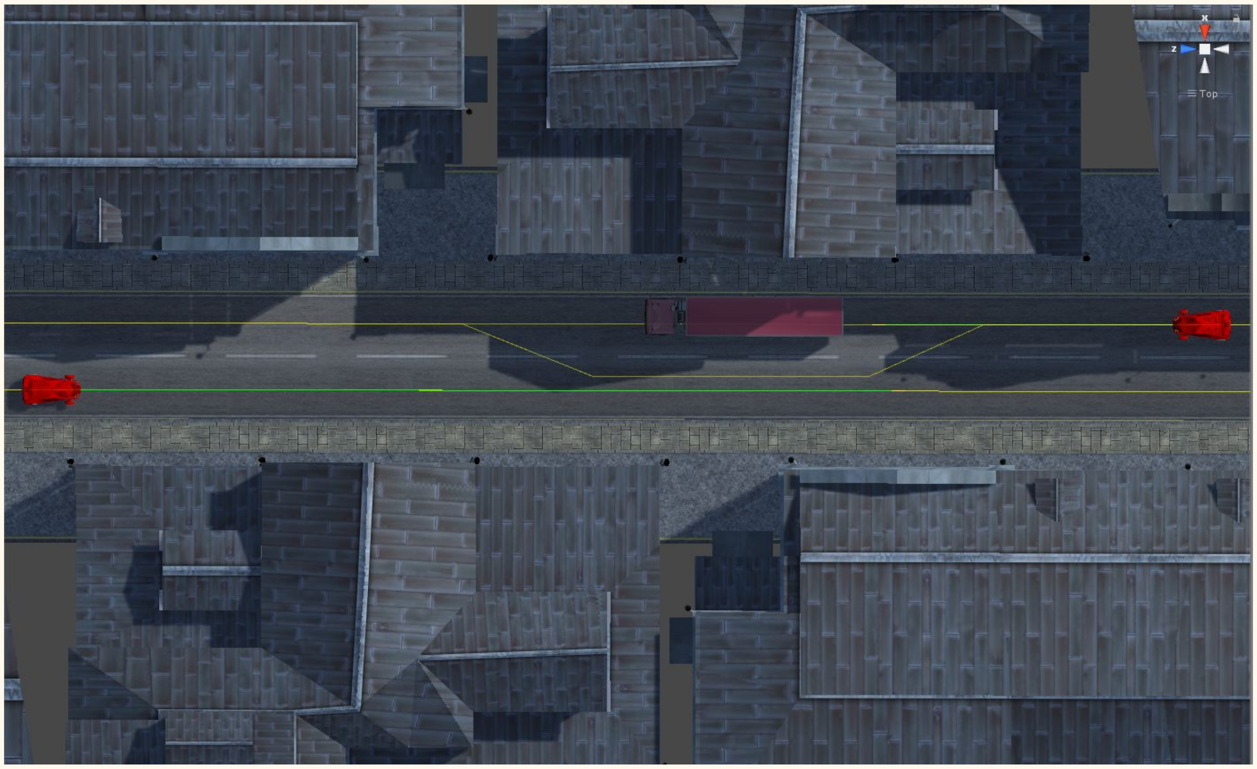
Zoë Papakipos, Matt Cooper, Michael Gillett, JD Fishman, Jerome Ramos

Abstract

For our research, we focussed on autonomous car control in Unity. To get situated with the platform, and the socket IO between it and our python-based deep learning models, we started by each making our own neural net to do behavioral cloning. That is, we each collected about 30 minutes of driving on our Unity-based simulator, and then we trained neural nets to mimic our actions. (Our behavioral cloning networks were CNN's that mapped a state – raw pixel values – to a continuous steering angle.)

After getting situated, we moved into reinforcement learning. We decided that an interesting task would be to model and navigate complex social interactions between cars, instead of just treating them as objects in the environment with a constant trajectory. We decided that this would be an important area to explore to prevent self-driving cars deployed in the real world from being bullied into undesirable conditions. For example, if self-driving cars did not engage in social interactions, humans drivers or pedestrians could always cut in front of the self-driving car and force it to remain stationary indefinitely. After reading about similar work done on learning to navigate merging and stopping at intersections [2], we decided to focus on a similar ares: the specific task we picked was lane sharing.

Our goal for the semester was to train an RL agent to optimally share a lane with oncoming traffic. Here, we define optimal in terms of getting to the destination quickly and without an accident. In our setup, the RL car is faced with an opposing car who is forced to merge into the RL car's lane, do to a truck obstructing its path. In our experiments, we assume the task of perception is done for us. The RL car is given as input its own velocity and position, as well as the opponent's speed and position. The RL car follows a predefined path and must choose the correct speed at every given time step. To do so, we used a deep Q-network, or DQN, based on the paper that introduced them [1].



Implementation Details

Backround

In reinforcement learning there are states, actions, rewards, and transitions. The goal is to find a policy, or a set of instructions for which action to take in every possible state. More exactly, the goals is to find the optimal policy, or the policy that will maximize the amount of reward you will get. (Note that typically we actually do not maximize total reward. Rather, we have some constant fraction "gamma," and for each time step into the future, the weight on a reward is multiplied by gamma, making it count less by a factor of gamma. We do this because we care more about the present... and also it makes the math work out nicer for infinite sums).

In order to find the best action for a particular state, we need some numerical representation of how good that action is in that state. This is known as the "Q-value." given a policy, Q(s,a) is the expected reward you will achieve after taking action "a" in state "s". (Generally, when it is not specified, this Q-value is calculated for the optimal policy. This is denoted by Q*.)

In our case, we define the state to be the velocity and position of both cars. We define the actions to be accelerate to 0, 10, 20, or 30 mph (action 0, 1, 2, and 3 respectively). We define reward to be -1 for a transition in which you did not reach the end of the street (to encourage speed), -30 for crashing, and 30 for reaching the end of the street). The transition function, or the function takes you from one state to the next, is simply abstracted away for us by our Unity simulator. The simulator restarts when you crash, get to the end of the street, or have taken too long.

The Agent

Our RL network takes in a state and predicts the Q-value of all possible actions in that state. If our network can give us these the Q-values, then we know the optimal policy: the action to take is simply the one with the maximum Q-value.

In order to learn these Q-values, our network takes actions in the simulator: Unity gives our network a state (s), our network choices an action (a), and then Unity returns the resulting reward (r) that action achieved, in addition to the new state we wound up in (s'). The state, reward, and next state are all out of our agent's control. All the agent gets to pick is "a". (Note that although the agent thinks it is best to take the action with the maximum Q-value, we actually take the next action according to epsilon-greedy sampling: some fraction (epsilon) of the time we act randomly, and the rest of the time we greedily take the action with the best Q-value. The point of such sampling is to encourage exploration and prevent premature convergence to a suboptimal policy, biased by how we chose to explore. We decay epsilon over training in order to encourage exploitation later on and manage the exploration/exploitation tradeoff.)

Every time we see a (state, action, reward, next state) pair, we group these four together in what is known as a SARS. Once we get a SARS, we immediately record it in a history of SARS's. (This history is actually a buffer and an old SARS is thrown out every time we see a new one once the buffer is full.) Training now becomes a much easier problem. Everytime we want to train, we grab a batch of data randomly from our experience buffer and train on that. (This is known as experience replay and ensures that we both can still make good predictions on old data, and also prevents only similar states from winding up in the same batch.)

Now that we have our training data, how do we learn to predict the Q-value for one SARS? First, we run the state (s) through our network this produces a Q-value for every possible action. (The current architecture consists of two fully connected hidden layers, with an output later equal to the number of actions). Knowing which action (a) we took, we can look at the prediction for Q(a). How do we get a loss for this Q(a) that we predicted? We simply use the 1-step lookahead. That is, we set the label for what Q(a) should have been equal to the reward (r) plus our discounted prediction for the value of the next state (r + max[Q(s')]). Then, treating this label as a constant, we can do backpropagation through Q(a). More specifically, we minimize the mean squared error between the predicted Q(a) and the label we calculated. That is, we minimize the temporal difference error.

One trick that the DQN paper employs to aid in learning this prediction is to calculate our labels using a "stale" net [1]. This means that every so often, we write our a copy of our Q-net that we do not update. We never change this stale net, only replace it every so on with new copies of our "fresh" net. The idea is that by using a stale net, the labels we are trying to match do not move around as much; it is easier to make progress without a moving target. The exact loss equation is depicted below:

$$L_{fresh}(s,a) = (r + \gamma * Q_{stale}(s,a) - Q_{fresh}(s,a))^2$$

Intuitively, what this loss is doing is moving what our fresh network predicted for Q(a) towards what it wound up being, with the rest of the history beyond 1 timestep predicted by our stale network. One of the reasons we use a 1-step lookahead for calculating our TD error is that it allows us to do experience replay correctly. If we tried to use a further look-ahead, and record discounted "r" over several steps into the future, our labels that we use to train the action A would be based on the steps taken by an old policy that we are no longer using.

The Opponent

Due to time constraints, we decided to train the RL agent against a programmed AI instead of humans. The opponent has 5 aggression levels (0 to 4 inclusive). At "agro" level 0, the opponent always stops. At agro level 4, the agent always goes, no matter if it will crash or not. In between, the opponent will stop or go depending on how close the RL agent is to the truck obstructing the road. (This allows for the RL car to learn to come up to the back of the truck quickly to try to cause the other car to stay still.) In addition to these agro levels, we trained our RL agent against 3 different agro distributions. We had a mellow distribution, which had a ⅓ chance of having aggression level 4 and a ⅔ chance split among agro levels 0 to 3. With this distribution, if the RL agent guns the gas pedal, and "bullies" the opponent, there will be a crash ⅓ of the time. We also had a neutral distribution, with a ½ chance of agro level 4 which would result in a collision when the RL agent bullies the opponent. And we had an aggressive distribution, where there is a ⅔ chance of agro level 4 which would result in a collision when the RL agent bullies the opponent.

Results

In order to see how well our agent learned, we tested it on each of the three agro distributions mentioned above. The agent learned against the aggressive distribution for 202 episodes, against the neutral for 256 episodes, and against the mellow for 463 episodes. (Note: we cut it off early for the first two strictly due to time limitations. We would have liked to let it run further.) In addition, we compared it to human performance and the performance of a random agent. For the human trials, each of us played against one of the three distributions (at random) for 20 episodes (or runs) each. The results are graphed below:

Graphs

Figure 1: My Results

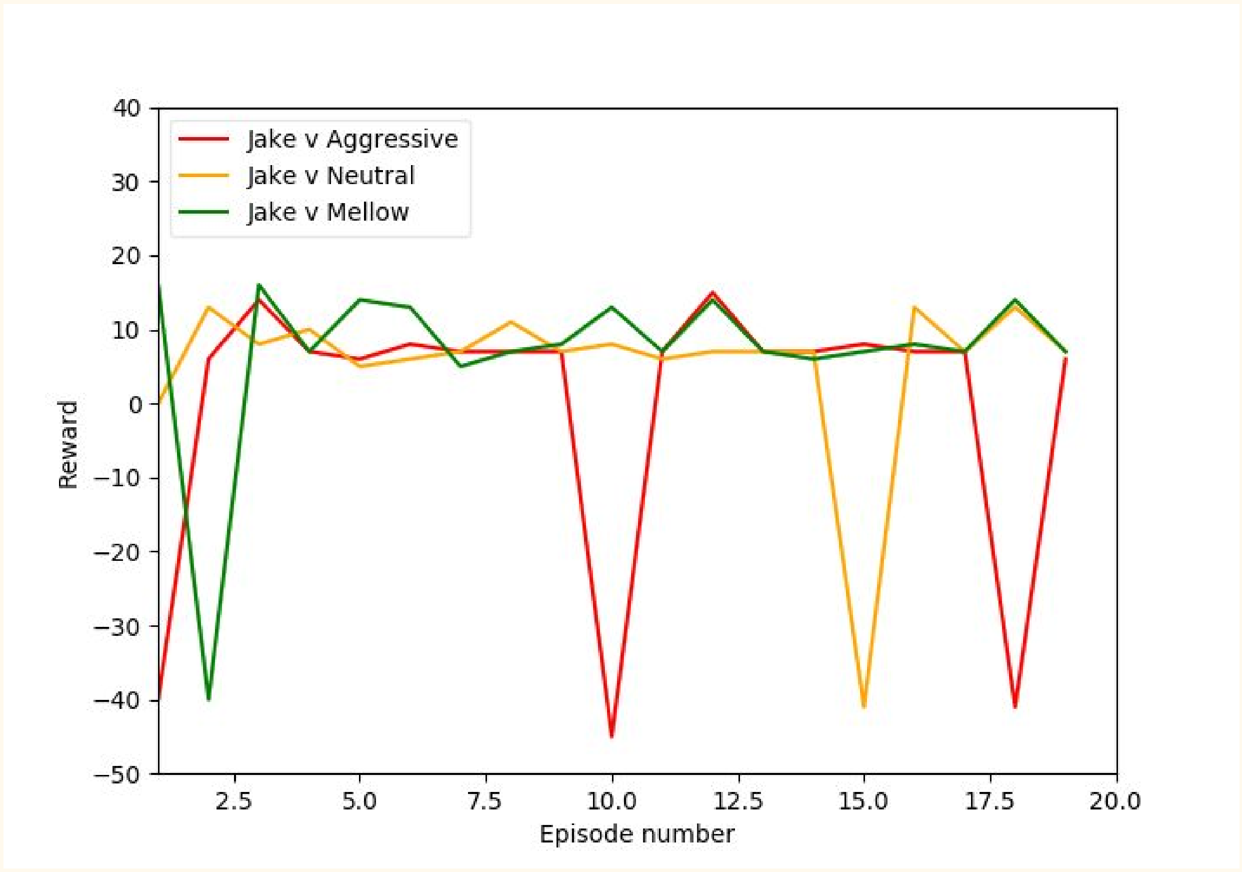


Figure 2: Average Human Results (with 1 standard deviation)

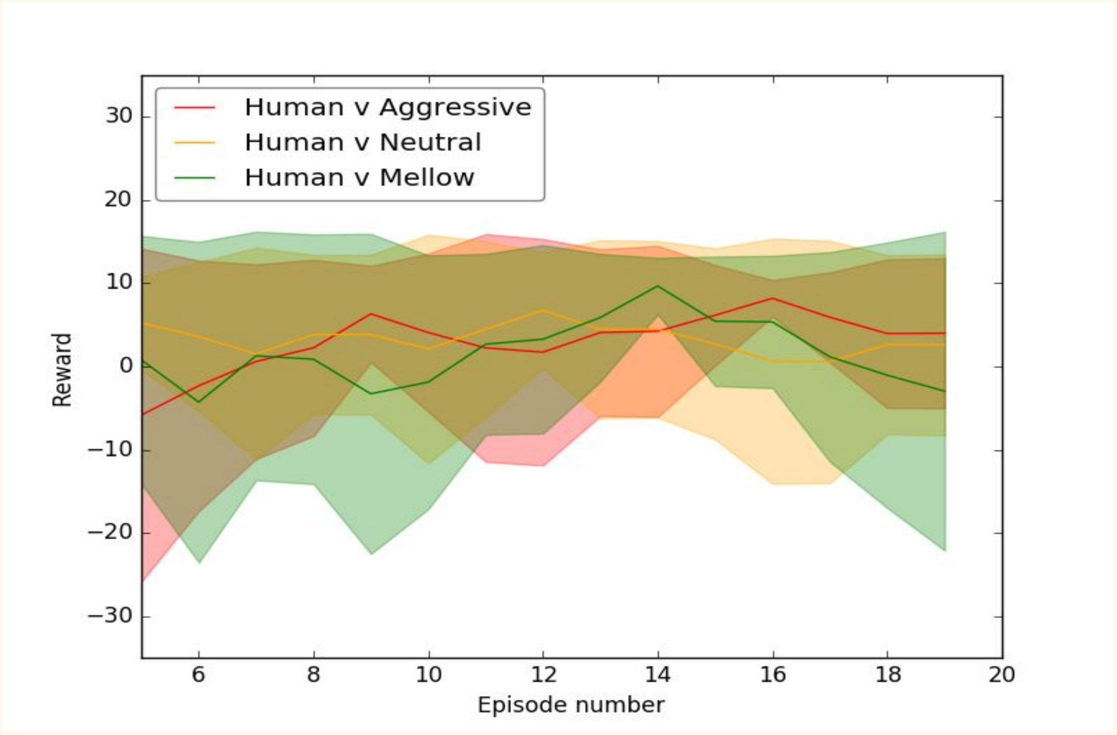
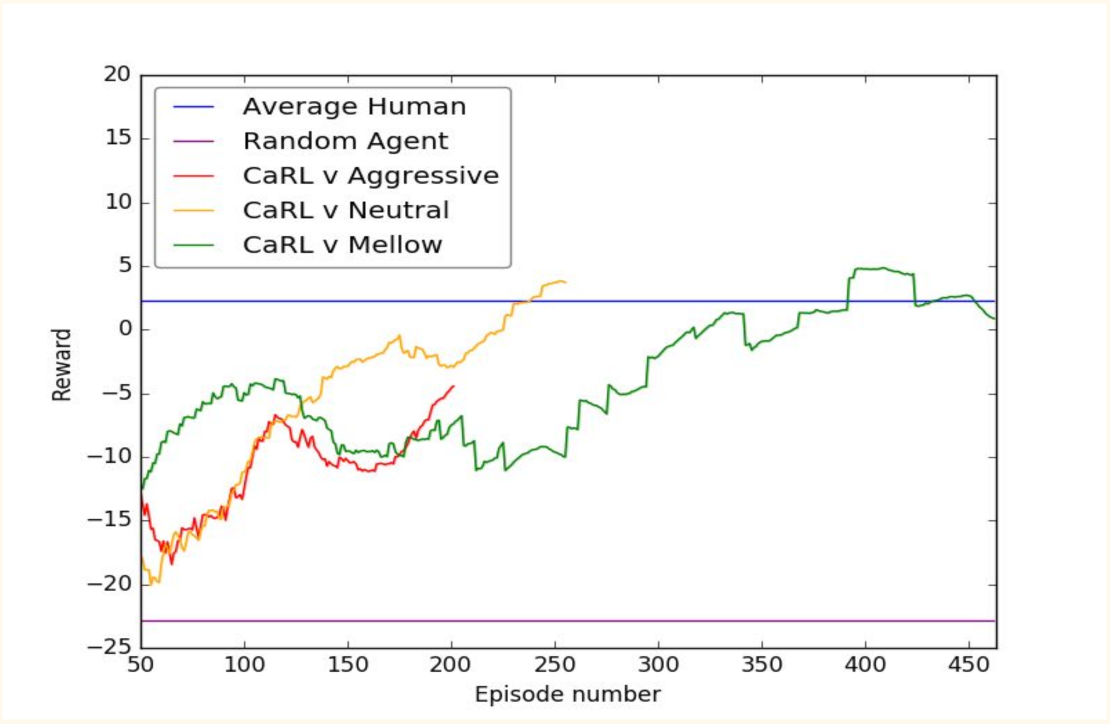


Figure 3: Agent vs Average Human vs Random Agent



Interpretation and Comments.

1. Humans vs. Cars: what do the graphs show?

The human results were very surprising. On average, we humans scored approximately 0 reward, which is worse than the RL agent winds up doing. Even though the optimal policy is fairly obvious (i.e. just stay right behind truck), it appears that we humans are not good at systemically executing the optimal policy. Even though the controls were not difficult to get used to, we interpret this to mean that humans are risky and aggressive – at least in simulation. (However, it should be noted that most of the episodes achieved positive reward, and the only reason that the mean was 0 was because of occasional crashes. Since the humans only had 20 episodes to learn from, it is possible we were still "exploring" a couple times throughout the 20 episodes.)

2. Reward Shaping: comments on behavior

Because of how negative the reward was for crashing (-30), there was an incentive to act very cautiously, and so the car did not learn any interesting behaviors. Perhaps this is reasonable for the real world, but it would have been interesting to see the RL car be more aggressive, while also not crashing. It could have learned a behavior such as: try to time it so that when the opponent slides back into their own lane, you pass by with as little space as possible and going as fast as possible. For example, since we know the exact location the opponent re-enters their lane, and it never changes, we could stop farther back and then start accelerating before they are back in their lane. This would allow us to slide by with momentum, and already be up to speed when the other car drives by. (This could also be done by slowing down, but not coming to a complete stop, as is often done in real life.) In order to elicit these behaviors, we could have waited time as being more important (i.e. a lower cost for crashing or a more negative cost for time passing.) Or we could have made acceleration take a lot longer.

3. Communication and Signaling

Another reason that we did not see interesting behavior is that the Dumb-AI opponent was just too dumb, since we hardcoded the opponent. We did put in some room for signaling and interaction (i.e. by having the opponent take into account your distance from the truck), but not enough to change the optimal policy. If we wanted to experiment with this further, I would suggest training the opponent first using behavioral cloning data from human vs human interactions. If the cloned opponent truly resembles a human, we may see more interesting interactions. These interactions in the real world contain much more communication and are all about signaling intent. (When we see this in the real world, it often resembles a back and forth dance, where the cars start to go at the same time and then need to figure out who should be allowed to go.)

Further Experiments

For future research, there are several different directions we could go in.

1. Smart Opponent

In order to see more interesting interaction, it may be helpful to have a model of how humans drive, and then use this model as our opponent. We could train this via behavioral cloning, and it would it would give us insight into how humans and RL agents would interact on the road. We can learn a policy based on this and show that RL cars can safely avoid being bullied. Moreover, we could play with the reward function to even take into account the opponent's reward and be kind to the opponent. This could be helpful if, for example, we want our RL cars to maximize our own reward, but subject to the constraint that we don't make other humans really unhappy. (This seems to be how many humans drive, especially when considering whether to allow others to merge in front of them.)

This, however, could be problematic: if this policy is rolled out en masse then people may learn clever ways to bully it over repeated experiments. Alternatively, we could forego learning (except for the transition function), and focus on model-based planning. We could learn a transition function by recording the transitions that happen when two random agents are allowed to play each other. Then, using this, we can compute policies. (For example, we learn to pick our next action based on minimax, if we think the world is out to get us.) We could try a number of these policies and show that one of them works well enough to ensure good behavior while not being bullied.

2. Simulator Real-world Translation

Although all of the research on the simulator is interesting, it seems purposeless without a way to take what we learn in the simulator and apply it to the real world. One way we could do this is if we we had more realistic looking simulators. However, since we are not equipped to make the simulator look like the real world, a work-around could be: make the real-world look like the simulator! That is, if we can find a good way to take photorealistic images and make them look like they came from the simulator, then we can run our models learned in the simulator in the real world. Alternatively, we could make both the simulator domain and the real world domain like like a new and arbitrary 3rd domain. We could even learn this third domain using GANs.

3. Efficient Training Without Crashing

One of the main problems we encountered with Unity is that it only runs in real time. Although we could try to switch over to a different simulator or find other ways to get around that, it may be an interesting problem to try to work around. If we ever hope to transition our learning algorithms from the simulator to the real world, they need to be able to be sample-efficient enough to run in real time. Moreover, they need to do so without crashing. There are a couple solutions I would like to try:

1) Teach the agent to act via behavioral cloning with evaluative feedback. One of the problems with behavioral cloning is that you can only show it good examples of actions and the agent has no idea how good they were. One problem with current evaluative feedback algorithms is that they allow the agent to explore as it learns, which is not feasible for a car in the real world, since it would crash and be dangerous. To solve both these problems, we could stick the two together.

2) The agent can learn a model of the transition function and reward function based on this evaluative cloning data. Once it knows the model, it can do offline planning so supplement the evaluative cloning.

3) Instead of learning transitions over states (which can quickly cause impossible states to be predicted in the complex state-space of plausible pictures) we can learn transitions over layers of our network. Moreover, we can learn to weight these layer transitions through gradient descent. (That is, we propagated each forward until some stopping layer, and then take the weighted average across all the activations suggested for this stopping layer.) We can even learn this weighting as a mask that is a function of the current input.

4) We can break the tasks up into smaller behaviors (such as lane following, stopping, speed, safety, smoothness) and learn these separately. We could create a separate neural network for each, then learn a weighted combination of these simple behaviors. We can give evaluative feedback on each of these behaviors and/or on the driving overall. Alternatively, we could learn something similar to options or even give the car sub-reward functions that it can learn to activate or deactivate.

5) add recurrence between layers.

For testing the above, we could have the RL car learn to follow a GPS. That is, given a pointer to your next turn, execute the turn well and get proceed to the next one quickly (allowing the GPS to reroute you if necessary). This is a fairly simple domain, and we already have a lot of the infrastructure setup for recording data in this domain. Additionally, it is a domain that is easily extensible to other driving tasks.

References

[1] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al.

Human- level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[2] D. Sadigh, S. Sastry, S. A. Seshia, and A. D. Dragan:

Planning for autonomous cars that leverages effects on human actions. *Robotics: Science and Systems*, 2016