Towards Robot Navigation with Deep Reinforcement Learning

Reintegrating AI Final Project

Izzy Brand, Josh Roy, Nate Umbanhowar

Abstract

In this project, we explored machine learning in an embodied setting in order to investigate the unique challenges and opportunities presented by placing a learning agent in the real world. We developed a robot, ExplorerBot, capable of navigating the environment and avoiding obstacles. ExplorerBot, a two-wheeled robot constructed from 3D-printed parts and standard low-cost electronics, served as a test-bed for two primary control policies: the first was a hand-coded policy based on values from ExplorerBot's time-of-flight (ToF) distance sensors; the second was a learned policy based on deep Q-learning. We tested deep Q-learning on input from a forward facing camera, but after extensive training this approach failed to produce desirable navigation behavior. We hypothesize that this is due to the fact that typical deep Q-networks (DQN) require on the order of millions of samples, which we were unable to collect due to limitations of sampling frequency in the real world. To validate our DQN implementation, we then trained on input from the ToF sensors, a fundamentally easier problem. Although this DQN-ToF policy was not optimal in a reward-collection test, it allowed ExplorerBot to reliably navigate in its environment without crashing.

Introduction

The major goal of this project was to develop an end-to-end neural network to control ExplorerBot, demonstrating a fusion of connectionist and embodied approaches. Deep Q-networks have recently made breakthroughs in learning control policies from high-dimensional input in a wide variety of tasks such as Atari 2600 games, which are designed to be difficult for humans, so using a DQN to control ExplorerBot was a natural choice [9, 10]. While deep Q-learning has been successful in a variety of simulated domains, there is a paucity of Q-learning on real-world agents. Deep Q-learning for autonomous rovers has been attempted several times in simulation [6, 5], and policies trained in simulation have been able to transfer to the real-world with limited success [3, 15]. In order for a policy to transfer from simulation to the real world, the simulation must be sufficiently representative of the real environment. Creating such a simulation is difficult and time intensive, and therefore not always practical. To explore the feasibility of training without simulation, we aimed to train a DQN to drive and avoid obstacles using only real-world data.

System Design

ExplorerBot

ExplorerBot is built from readily available components. The chassis and wheels were designed in CAD and printed in PLA. These parts took roughly 6 hours to print. We put rubber bands around the wheels to serve as tires. Each wheel is driven by a continuous rotation servo, enabling precise velocity control. The robot can move forward and backwards at up to 1m/s by driving both wheels in the same direction, and turn by driving the wheels at different speeds.

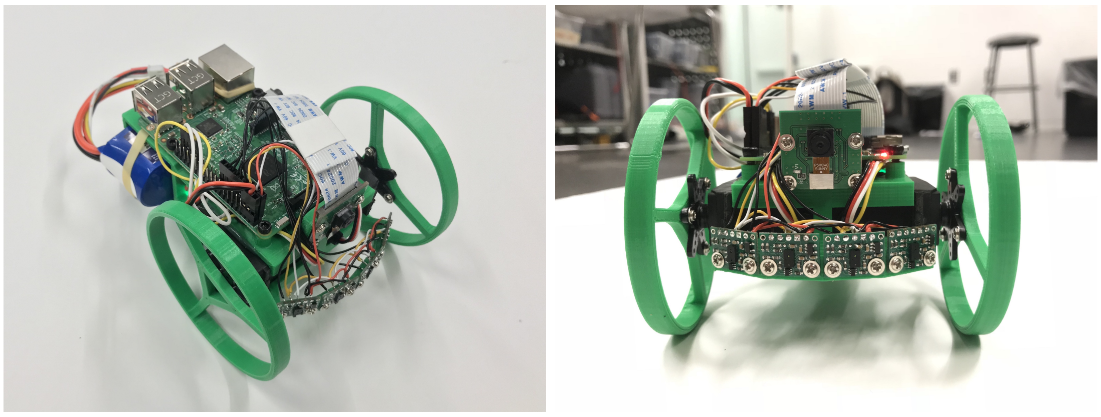

Figure 1: The assembled ExplorerBot. The right image shows a clear view of the camera, as well as the four ToFs arrayed below.

A Raspberry Pi 3 serves as the onboard computer and WiFi relay. The Raspberry Pi is a small, ARM-based development board that runs Debian. Although the Pi is surprisingly fast, our initial test of training a small CNN MNIST classifier onboard confirmed our hypothesis that we would need to run neural network computation on a more powerful computer. Fortunately, the Pi has built in WiFi capabilities, so we were able to send sensor data offboard.

A fixed, forward-facing camera collects RGB video frames. We were able to efficiently estimate optical flow by exploiting the Raspberry Pi's GPU H.264 video encoder [14]. The H.264 encoding pipeline uses a motion estimation block, which calculates the most likely motion of 16x16 pixel macro-blocks from frame to frame. The built-in `RaspiVid` tools allow the user to access these motion vectors, which approximate optical flow. The H.264 video encoding takes place almost entirely on the GPU, so these motion vectors can be used with almost no CPU overhead.

In addition to the camera, the robot is equipped with an array of four time-of-flight (ToF) distance sensors on the front of the chassis, splayed out slightly such that the outer two sensors can detect objects diagonally in front of robot. Each ToF sensor has millimeter resolution and a range of about about 20cm. The Pi communicates with the sensors via I^2C protocol.

Item	Description	Cost	Quantity	Total
Raspberry Pi 3	Onboard Computer	\$35	1	\$35
Raspberry Pi Camera	Onboard Camera	\$15	1	\$15
VL6180X	Distance Sensor	\$8.50	4	\$34
High Speed Continuous Rotation Servo	Motors	\$20	2	\$40
5V 3A BEC	Onboard Power Supply	\$3.50	2	\$7
64GB MicroSD	Onboard Storage	\$13	1	\$13
1500mAh 3S 20C Lipo	Battery	\$12	1	\$12
Total	Total Cost of the Robot			\$156

Figure 2: Robot Bill of Materials

Software Architecture

The ExplorerBot software and the DQN are written primarily in Python, and make use of Robot Operating System (ROS) and TensorFlow respectively [12, 1]. We chose Python to allow for rapid prototyping. The computationally intensive tasks are handled by TensorFlow and NumPy, which are written in C.

As mentioned, we experimentally determined that training the DQN onboard the Raspberry Pi would be impractical by timing the training of a CNN MNIST classifier. As such, we ran the DQN on an offboard computer with a GPU. The offboard computer runs a Flask webserver which support two operations: ExplorerBot can send a state which gets added to the DQN memory and receives an action from the Q-network in return, and the robot can request an iteration of backpropagation to be run on the Q-network.

Onboard, the robot runs a script which collects data asynchronously from the camera and the ToF sensors using ROS. At each iteration the script sends the current sensor readings to the server, executes the received action by writing to the servos, and makes a request for a backpropagation step. This can be run at a fixed rate or as fast as the server can handle.

Deep Q-Learning

We model the robot's interaction with its environment as a Markov decision process (MDP), which is a 5-tuple (S, A, R, T, γ). For our problem context, each element of the MDP is defined as follows: S is the set of possible states of the robot in the environment as perceived by the agent's sensors. A is a finite set of ExplorerBot's available actions, which we specified first as possible motor-speed deltas and later altered to setting the motor speeds directly to discrete values. $R = R(s, a, s')$ describes the reward received by taking action a in state s and transitioning to state s'. Several different reward functions were tried, and are described in the experiments section. $T = T(s, a, s')$ describes the transition probability of transitioning to state s' when taking action a in state s. In our problem setting, this is not known and is implicitly learned as the agent learns to estimate the expected future reward when taking action a from state s. Finally, γ, the discount factor, is a hyperparameter that specifies the degree to which the agent values current rewards over future rewards.

Q-learning is a reinforcement learning method that aims to learn an optimal action-value function $Q(s, a)$ which describes the maximum expected reward achievable after taking action a in state s. This optimal function Q satisfies an important relation called the Bellman equation, which states that

$$Q(s,a) = \mathbb{E}_{s' \sim T(s,a,\cdot)}[r + \gamma \max_{a'} Q(s', a')|s, a]$$

Intuitively, this holds because the total reward received by taking action a in state s and thereafter following an optimal policy is equivalent to the total reward achieved by receiving reward r for taking action a in s, ending up in s', and following the optimal policy thereafter.

Typical RL frameworks estimate Q by the Bellman equation in a process called value iteration, which iteratively performs the update $Q_{t+1}(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q_t(s', a')|s, a]$. This provably converges to the true function Q [8]. In deep Q-learning, the Q function is parameterized as a neural network $Q(s, a) = Q(s, a; \theta)$, where the network (with parameters θ) takes in a state s and outputs a vector of Q-values for all possible actions, and action a indexes its corresponding Q-value.

We implemented deep Q-learning with experience replay [10, 13]. In this formulation, value iteration using the Bellman equation is replaced by stochastic gradient descent of the loss function

$$L(\theta_t) = \mathbb{E}[(y_t - Q(s, a; \theta_t))^2]$$

where $y_t = r + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \theta^-)$. Experience replay chooses random batches of memories from a fixed size buffer of past (s_j, a_j, r_j, s_{j+1}) tuples to update the network weights to minimize $L(\theta_t)$. This helps decorrelate successive states, improving generalization of the learned Q-function. $\hat{Q}$ is a target Q-network whose parameters, θ^-, are updated periodically to be the same as that of the current Q-network, Q, with parameters θ_t. Updating the target network periodically as opposed to after each iteration provides a more stable training target [10].

Algorithm 1 Deep Q-learning with experience replay [10, 13]

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta_1$
Initialize target action-value function $\hat{Q}$ with weights $\theta_1^- = \theta_1$
**for** episode 1, $M$ **do** Initialize sequence $\bar{s}_1 = \{s_1\}$ and preprocessed sequence $\phi_1 = \phi(\bar{s}_1)$
    **for** $t = 1, T$ **do**
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \arg\max_a Q(\phi(\bar{s}_t), a; \theta_t)$
        Execute action $a_t$ in the emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $\bar{s}_{t+1} = \bar{s}_t, a_t, s_{t+1}$ and preprocess $\phi_{t+1} = \phi(\bar{s}_{t+1})$
        Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta_t^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta_t))^2$ with respect to the weights $\theta_t$
        Every $C$ steps reset $\hat{Q} = Q$, setting the weights $\theta_t^- = \theta_t$
    **end for**
**end for**
```

Algorithm 1 details the deep-Q learning process with experience replay, specifically as presented in *Human-level Control Through Deep Reinforcement Learning* [10]. Note that the states in this algorithm are formulated as sequences $\bar{s}_t = s_0, a_0, s_1, a_1, ..., a_{t-1}, s_t$, where s_t represents the state at time t. However, the preprocessing function $\phi_t = \phi(\bar{s}_t)$ returns the concatenation of the last h states in the sequence, where h is a history length hyperparameter. Including a brief history of states as part of the input to the Q-network, relaxes the model to be h-Markov, and gives the agent some ability to execute longer action-sequences. We implemented this algorithm with $h = 1$.

For ExplorerBot, the state, s_t, at a time t is a tuple of the sensor data observed at that time

- $s_t = (\text{img}_t, \text{flo}_t, \text{mot}_t, \text{tof}_t)$
- The image, img_t, is $160 \times 240 \times 3$ array of pixels.
- flo_t is $10 \times 16 \times 3$ array of the corresponding flow vector components.
- mot_t is a tuple of the motor speeds, $(\text{mot}_t^{(1)}, \text{mot}_t^{(2)})$, where the speed of each motor ranges from from -1 to 1.
- tof_t is a tuple of the ToF distance sensor reading, $(\text{tof}_t^{(1)}, \text{tof}_t^{(2)}, \text{tof}_t^{(3)}, \text{tof}_t^{(4)})$, where each sensor ranges from 0 to 1.

Experiments

Before beginning Q-learning on the robot, we attempted to handcraft our own policy to generate desirable behaviors. This proved useful both as a benchmark, and as a way to collect a variety of data without the robot crashing into obstacles in environment too often. Our handcrafted policy calculates a linear combination of the ToF distance readings to determine a speed for each motor. With some tuning, we were able to get the agent to drive around quickly and turn to avoid objects.

In creating this hand-crafted policy, we observed that it was possible to achieve good behavior without retaining a history sequence of states, as the original DeepMind paper did [9]. With this and mind, and considering that removing the history would greatly reduce the number of parameters in our Q-network, we decided to initially set our history length to $h = 1$, effectively making the Markov assumption about our environment.

Learning to Navigate with the Camera

Our initial network architecture consisted of four convolutional layers on both the RGB frames and the optical flow frames. The convolution outputs were concatenated with the current motor values and fed through two more fully connected layers to produce the Q values. Importantly, the network was not given access to the ToF sensor data.

The reward function was

$$R(s_t, a_t, s_{t+1}) = \alpha\big(2 \times \min(\mathrm{tof}_{t+1}) - 1\big) + (1-\alpha)\begin{cases} \mathrm{mot}^{(1)}_{t+1}\mathrm{mot}^{(2)}_{t+1} & \text{if } \ \mathrm{mot}^{(1)}_{t+1} > 0 \vee \mathrm{mot}^{(2)}_{t+1} > 0 \\ -\mathrm{mot}^{(1)}_{t+1}\mathrm{mot}^{(2)}_{t+1} & \text{otherwise} \end{cases}$$

This function rewards two aspects which we determined to be desirable in the next state: the ToF sensors do not detect objects close by, and the motors are both spinning forward quickly. We used $\alpha = 0.75$ so that the robot would prioritize having open space in front of its sensor over moving forward quickly.

Our first action space allowed the robot to change the speed of the motors by a specified amount. For each wheel, the robot could choose to increase the speed, stay the same speed, or decrease the speed. Three choices for each of the two motors meant there were nine possible actions.

We used a hand-crafted policy capable of driving the robot without bumping into objects too often to collect 10,000 memories. We initialized the Q-network running Algorithm 1 for 50,000 batches on this bank of memories without taking actions or adding new memories.

Thus initialized, we began training with the robot in the loop. Behavior was seemingly random, though the training loss fell much more quickly when training on the robot than on our hand-collected data. We believe this is because the robot refilled its memory buffer with memories where it took actions according to its own Q-function rather than according a handcrafted policy. When training on the hand-collected memories, the actions it took in those memories would not necessarily agree with what Q would have selected, and accordingly Q struggled to estimate the future reward, $r + \gamma\hat{Q}$.

We also observed that when the learning rate was set very high, the robot would choose one action and repeat it until the target-Q network updated, at which point it would usually repeat a different action. In general we found learning rates between 1e-5 and 1e-8 to be most effective for our implementation.

After roughly 5,000 iterations, the robot converged on a pattern of spinning rapidly in place. This policy could perhaps be justified by the argument that spinning rapidly in place prevented the agent from experiencing the negative reward of getting too close to a wall while sacrificing the reward it would have earned from driving forward. However, more often than not the robot would simply start spinning very close to a wall.

Even after training for several hours with various hyperparameters, this behavior still dominated, so we decided to reduce the size of the network to 3 convolutional layers on the image, and two convolutional layers on the optical flow. We trained this architecture for an hour without running the pre-training on hand-collected memories, and the behavior converged yet again on the rapid spinning in place.

To test yet another initialization, we trained the smaller network architecture to imitate the hand-crafted policy. The loss function for this training was the cross entropy loss between the softmaxed output of the Q-network evaluated on s_t and a one-hot vector of the action, a_t chosen by the hand-crafted policy from state s_t

$$L_{imitation} = -\big\langle \text{onehot}(a_t), \log(softmax(Q(s_t)))\big\rangle$$

where $\langle u, v\rangle$ is the dot product of vectors u and v. After training the imitation loss for 20,000 iterations, we used this network as the initialization for another Q-learning session. Initially, ExplorerBot exhibited an interesting behavior: it would drive roughly straight until it hit a wall, then begin spinning in place to the left. This spinning would push it away from the wall, and it would gradually spiral out of the spin until it hit another wall, and the pattern would repeat. After several hundred iterations of online Q-learning, this behavior faded and the robot resumed its usual policy of spinning rapidly in place.

Confused by this often recurring behavior, we considered the kinds of behaviors that our action space would tend to produce from a mostly random agent. Since six out of the nine possible actions coded for accelerating into a turn, we reasoned it would be nontrivial for the agent to even drive in a straight line. To reduce the amount of learning necessary to attain even the trivial behavior of driving straight forward, we decided to redesign our action space.

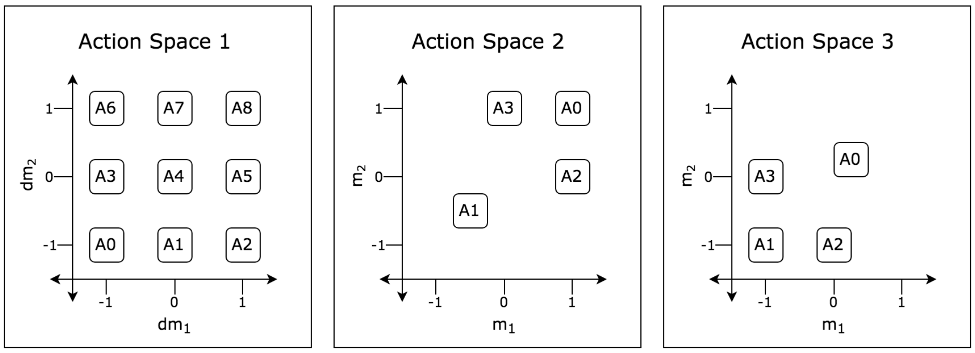

Figure 3: Our first, second, and third action spaces. The first action space allowed the robot to increase or decrease motor speeds by 10% (note the dm_i in the circles). The second action space set the motor speeds directly to one of four presets, where driving backwards (action 1), was half the speed of driving forward (action 0). Action space three changed so that driving backwards was full speed, driving forward was quarter-speed, and turning was achieved by backing up one of the wheels at full speed rather than driving one wheel forward.

Our second action space allowed the robot to choose from one of four actions: drive forward, drive backward, turn left, and turn right. The actions no longer coded for changing the motor speed by a fixed amount, but rather for setting the motors to one of four presets.

After training the Q-network in the new action space for two hours and 12,000 iterations on the robot, we compared its average reward with that of a completely random policy and our handcrafted policy, as shown in Figure 4. To run this comparison, we placed ExplorerBot in each of five different starting locations and ran each policy for one minute, calculating the average reward over that time. Based on these results, we can see that our learned policy does not behave significantly better than a random policy.

Trial Number	Learned (2Hz)	Learned (10Hz)	Random (2Hz)	Hand Coded (2hz)
1	0.0451	0.03235	-0.0407	0.194
2	-0.216	0.09963	0.06704	0.1504
3	-0.0326	0.3566	0.1912	0.2339
4	0.2423	-0.30188	-0.00036	0.3003
5	-0.1286	-0.03383	-0.1252	0.2968
Average	-0.01796	0.0305	0.018	0.218775

Figure 4: We tested each policy on five one-minute trials and collected the average reward over each trial. The first policy is the learned policy running at 2Hz, to match the frequency it ran at while training with backpropagation. The second policy is the learned camera-based policy running at approximately 10Hz without running backpropagation on the server. The next policy is the random policy. The final policy is the hand coded policy artificially limited to run at 2Hz.

In conducting this experiment, we found that our hand-crafted policy performed far worse when running at 2Hz than it had when running at full speed (50Hz). This led us to believe that the problem of navigating with an update frequency of 2Hz was quite difficult, and it was perhaps unsurprising that the Q-learning agent had struggled in this task. We profiled our pipeline and found that the backpropagation on the CNN Q-network was responsible for the majority of the slowdown. With such a slow loop-frequency, training the DQN to produce desirable behavior would take far too long.

Learning to Navigate with the ToFs

After discovering that the policy trained on camera input did not perform significantly better than a random policy, we decided to conduct a simpler test in order to verify the behavior of the DQN algorithm on ExplorerBot. We gave the network access to the ToF sensor data and changed the state to

$$s_t = (\text{mot}_t, \text{tof}_t)$$

The new Q-network concatenates the motor and ToF data, then runs them through a hidden fully connected layer of size 20 and a hidden fully connected layer of size 6 before producing an output of Q-values for each of the four actions. This is a vastly simpler problem than learning to navigate with the camera since the network has far fewer parameters and does not have to learn image understanding. Instead, it must only learn to identify and avoid obstacles based on the ToF sensors, and move forward when possible. We expected the network to learn this problem relatively quickly, since our handcrafted policy with ToF sensors and motors was simple, and the training iterations could be run at a much higher frequency due to the smaller network architecture.

In this experiment, we initially used the second action space, as described above. Additionally, we changed the reward function to

$$r_t = \alpha\big(2\min(\text{tof}_{t+1}) - 1\big) + (1-\alpha)\begin{cases}\text{mot}_{t+1}^{(1)} + \text{mot}_{t+1}^{(2)} & \text{if } \text{mot}_{t+1}^{(1)} + \text{mot}_{t+1}^{(2)} > 0 \\ 2(\text{mot}_{t+1}^{(1)} + \text{mot}_{t+1}^{(2)}) & \text{otherwise}\end{cases}$$

This new reward function seeks to punish backing up more strongly than turning, to encourage the robot to take evasive maneuvers before it hits the obstacle. We ran multiple trials with $\alpha = 0.75$ so ExplorerBot would prioritize not running into walls over motor speed. When it still ran into walls, we changed the α to 0.9 to further emphasize obstacle avoidance.

We trained ExplorerBot for several hours (100,000 iterations) and found that the robot developed a tendency to run into obstacles and continue to spin the wheels forward. Once the robot settled on this policy, the Q-network loss began to drop rapidly, even though the behavior did not appear to change. We experimented with increasing ϵ, the probability of selecting a random action, to get the robot to break out of this behavior, but ExplorerBot persistently continued to run into the wall. Increasing γ to place more value on future rewards also failed to discourage this behavior. The sharp drop in the loss could perhaps be explained by the policy having converged on a local maximum in the reward. Although the reward function punished running into objects, it rewarded driving forward, and if both Q and Q^- always chose the "drive forward" action, the expected future rewards would agree and the loss would fall.

At this point we decided to restart the training from scratch. We also changed the history length h from one to four, meaning the input to the Q-network was now the concatenation of the current state and the three previous states. This was the same history length used by DeepMind, and we hoped that the inclusion of a history would allow the agent to learn longer sequences of actions.

After training the Q-network in the loop with the robot for 80,000 steps, we observed that the robot would run into walls, attempt some combination of moving forward and turning, and eventually back up. After backing up, ExplorerBot would either choose to turn or to drive straight back into the wall. More often that not, it drove back into the wall. Furthermore, while it was driving into the wall, it mostly chose the "drive forward" action but also occasionally chose to turn. This continued even when setting $\alpha = 0.9$, suggesting either that the network was not able to learn enough in 80,000 iterations or that it had somehow determined that driving into the wall was a locally optimal policy. However, if the network had decided that driving into the wall and spinning the wheels forward was the optimal policy, it should not have attempted to turn while driving pressed into wall, since that provides lower reward than driving the wheels forward.

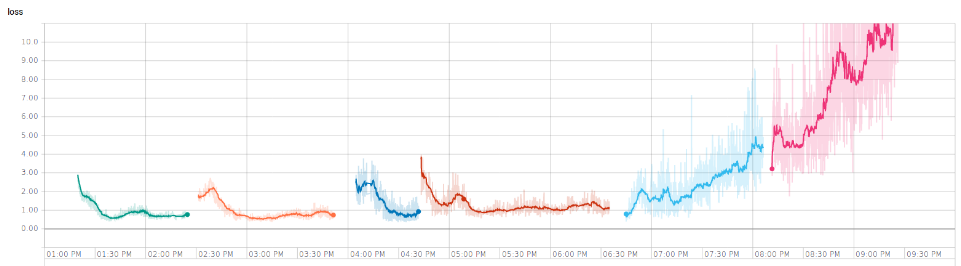

Figure 5: ToF Q-Network loss plotted throughout training. The breaks in the graph are when we changed ExplorerBot's battery. At 6:30pm, around 150,000 iterations, the robot displayed desirable behavior, driving forward and turning to avoid walls. As it kept training, the loss diverged. When tested at 9:30pm in the reward test the final policy performed worse than the policy at 6:30pm.

As ExplorerBot kept training, we changed a few parameters. Most importantly, we changed our action space to the third action space shown in Figure 3. This meant the robot would move forward at quarter-speed, backward at full speed, and turn by rotating one wheel backward. We hoped this change would allow ExplorerBot to spend less time facing walls because turning would now pull it away from the wall, and the decreased forward speed would give it more time to react before running into obstacles.

After training for an additional 80,000 steps, we began to observe desirable behavior. Notably, most of the time ExplorerBot would consistently drive forward when in open space, stop before hitting walls and turn to the right, avoiding them. When the obstacle appeared in the left side of the receptive field, it would turn to the right gradually, but when the obstacle appeared in the middle or on the right side it would turn to the right aggressively, spinning almost 180 degrees in place. This was a marked improvement in the policy and further reinforces the hypothesis that the deep Q learning simply takes a long time to converge.

Although the loss was beginning to diverge, we allowed the network to continue to train for an additional 100,000 iterations. The behavior stayed roughly the same, but by the end of the training ExplorerBot was considerably less decisive when turning around, especially if the obstacle appeared in the left side of its receptive field. When monitoring the sensor values, we observed that the leftmost ToF sensor would occasionally drop to zero distance even when there was no obstacle present — an inspection of the robot revealed that there was dust partially obscuring the sensor. We also observed that the robot would continue to drive straight forward even when its left ToF sensor indicated there was an obstacle. It appears that the dust may have been present throughout most of training, and the robot had learned to ignore the values from that sensor. Perhaps this is why it chose to always turn to the right.

Trial Number	Learned (150k iters)	Learned (250k iters)	Random	Hand Coded
1	0.4681	0.2072	0.7295	0.7894
2	0.4927	0.0054	0.6081	0.8156
3	0.4837	0.2268	0.1972	0.7982
4	0.4176	0.2850	0.6201	0.8238
5	0.5528	0.1563	0.6196	0.8063
Average	0.4829	0.1761	0.554	0.8066

Figure 6: Average Rewards. We ran the same test as Figure 4 with different policies and the reward function used in training the ToF based nn. The first policy is the ToF based neural network after 150,000 iterations. The second is the neural network policy after 250,000 iterations. The next policy is the random policy, and the final policy is the hand coded policy running at its max frequency (approx 50Hz).

Discussion

In Figure 6, the random policy accumulated considerably higher reward than either of the trained policies. There were a few factors that made this possible. Since this random policy was tested in action space three, it had an expected tendency to drive backwards. Since the action space backs up faster than driving forward, picking random actions has an expected value of moving backwards. This enabled the random policy to back ExplorerBot into a wall and stay there. In this situation, the ToF sensors were not able to sense any obstacles. Because the reward for not detecting obstacles accounted for 90% of the reward, this resulted in extremely high reward, though the robot continued backing into a wall. This shows that the reward function is easily gamed and should have a lower α in order to give less importance to the ToFs and more to moving the robot forward.

Notably, the neural network did not learn to exploit this flaw in the reward function. We believe this could have happened for two reasons. First, the network follows a greedy policy when selecting actions. The γ value was too small for ExplorerBot to realize that driving forward eventually leads to a wall and therefore lower overall reward than somehow avoiding driving all the way to the wall. Given a discount factor of 0.99, the reward received at a timestep 70 steps (3.5 seconds) in the future will be counted at half the value of the current reward. Considering that the robot takes longer than 4 seconds to reach a wall from the middle of our test environment, we think this is a likely explanation.

Furthermore, it is possible that the Q values that the neural network generated were not sufficiently accurate to the true Q values to accurately predict reward so many timesteps in the future. This problem could be solved by training the network for longer and considering a higher γ value.

Future Work

Because the ToF-DQN successfully learned to drive and avoid obstacles, we conclude that it is possible to train a DQN on an embodied robot in the real world. However, it takes a very large amount of data to train such a network. This is true of other DQNs as well — for example, the DeepMind's Atari 2600-playing network took many millions of iterations to learn each game, and continuous motion generating Q networks regularly train for about 1-1.5 million steps [9, 10, 7, 2].

This amount of training data is difficult to collect in the real world. The DQN required 8 hours and roughly 150,000 iterations before producing any desirable behavior when using ToF sensor data. Based on the amount of training other DQNs have required, training ExplorerBot to navigate with camera might have simply required at least ten times more training data. ExplorerBot receives a very compressed and specialized representation of the world through the ToF sensors — the robot simply sees its distance to objects in four directions, whereas an RGB image would require significantly more processing to extract features useful for navigation.

There are a couple different ways to combat the amount of data needed to train such a robot. The loop frequency of training the ExplorerBot was approximately 10Hz when using the ToF sensors and 2Hz when using the camera images. We believe that increasing the loop frequency of the robot would decrease the amount of time required to train the network. This would give the network more examples of the spaces that the robot has explored. For example, if the robot needs to drive straight into a white wall for some number of steps in order to learn to avoid white walls, increasing the loop frequency would decrease the amount of real world time required to collect that number of examples. Furthermore, increasing the loop frequency would allow the robot to take more fine grained actions. For example, this would allow the robot to see a wall more times before crashing into it, giving it more opportunities to recognize the obstacle and turn away.

To increase the loop frequency going forward, we would like to optimize the communication between ExplorerBot and the offboard computer. Additionally, we would like to re-train the network with a faster GPU in order to further increase the frequency.

Another approach to combating the data-deficiency problem is transfer learning, which could improve behavior through better image feature extraction. Numerous works have demonstrated that taking weights from a trained network and employing them as an initialization or prepossessing step for a new network in a similar problem domain can significantly reduce the required number of training examples [11]. ExplorerBot's visual Q-network could be initialized using weights from an existing CNN such as ImageNet [4]. Alternatively, an autoencoder neural network trained on data collected from the robot's environment could extract salient features specific to the domain. The encoder portion of such an autoencoder could serve as a useful preprocessing step or initialization for the agent's deep Q-network. Other feature generation systems such as SIFT, or edge-detectors could also be used to preprocess input to the Q-network.

Conclusion

Although end-to-end neural network architectures are capable of learning complex tasks and optimizing every step of the pipeline, and although reinforcement learning has proven to be a powerful tool to enable agents to learn complex behaviors guided only by simple reward functions, a potent marriage of neural networks and reinforcement learning, deep Q-learning, is rarely applied to real robots. This is because DQNs often take an incredibly long time to learn useful policies, and therefore are only practical to train in simulated environments.

In this project we asked the question, *what if you don't have a simulator?* We took a step into the uncharted waters of a training a DQN from scratch in the real world on an embodied agent. Although ExplorerBot failed to learn to use the camera for navigation, given sufficient training time we see no reason why this should be impossible. By simplifying the learning problem, and reducing the number of network parameters, we were able to train a DQN on a robot in the real world without pretraining in simulation. Indeed, we were surprised to find that this smaller network converged on desirable behavior in as few as 150,000 iterations. In it's current form, deep Q-learning on real robots appears to be mostly impractical, but we propose a number of possible method to accelerate training including transfer learning and increasing the loop frequency. With ExplorerBot we have shown that deep Q-learning can produce desirable policies after training exclusively in the real world, and we hope that in the future DQNs continue to migrate from simulations and game environments into real-world embodied agents.

References

[1] 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, Dandelion 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.

[2] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

[3] Gregory Kahn, Adam Villaflor, Bosen Ding, Pieter Abbeel, and Sergey Levine. Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation. *arXiv preprint arXiv:1709.10489*, 2017.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[5] Tai Lei and Liu Ming. Mobile robots exploration through cnn-based reinforcement learning. *Robotics and Biomimetics*, 3(1):24, Dec 2016.

[6] Tai Lei and Liu Ming. A robot exploration strategy based on q-learning network. In *Real-time Computing and Robotics (RCAR), IEEE International Conference on*, pages 57–62. IEEE, 2016.

[7] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[8] Francisco S Melo. Convergence of q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep*, pages 1–4, 2001.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[11] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[12] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[13] Melrose Roderick, James MacGlashan, and Stefanie Tellex. Implementing the deep q-network. *arXiv preprint arXiv:1711.07478*, 2017.

[14] Liz Upton. Vectors from course motion estimation. Accessed: 5 4, 2018. https://www.raspberrypi.org/blog/vectors-from-coarse-motion-estimation/.

[15] Fangyi Zhang, Jürgen Leitner, Michael Milford, Ben Upcroft, and Peter I. Corke. Towards vision-based deep reinforcement learning for robotic motion control. *CoRR*, abs/1511.03791, 2015.