Simulated Transfer Learning Through Deep Reinforcement Learning

William Doan WILLRD9@VT.EDU
Griffin Jarmin GAJARMIN@VT.EDU

Abstract

This paper encapsulates the use reinforcement learning on raw images provided by a simulation to produce a partially trained network. Before training is continued, this partially trained network is fed different raw images that are more tightly coupled with a richer representation of the non-simulated environment. The use of transfer learning allows for the model to adjust to this richer representation of the environment and the network eventually exhibits desired behaviours in the real world. This is due to iteratively training the network on gradually more accurate simulated representations.

1. Introduction

Control theory has traditionally been used and extensively studied in the field of robotics. The field of control theory has developed many useful approaches to solving robotic control such as (M.). While these methods have seen great success in a number of practical applications, they have shortcomings. Some examples of said shortcomings are control algorithms' reliance on noisy sensor data, inability to handle non-linear environments well, and difficulty modelling the stochasticity of the environment.

In this paper we present a different method of robotic control. The method used in (Mnih et al., 2013) provided a good foundation for applying reinforcement learning to high dimensional input data. The generalization power of the convolutional neural network that is used in the deep Q-learning algorithm alleviates the concern surrounding noisy sensor data. Also, rather than hand engineering features the modeller deems important, we allow the learning algorithm to decide what features are important.

As shown in (Mnih et al., 2013), a large number of training iterations are needed in order to successfully train a model. It is impractical to obtain a sufficient number of real world examples to achieve convergence. This requirement introduces the need to simulate the environment. Using this simulation we can train on the simulation-produced examples rather than real examples. The goal of this paper is to show that by using standard training methods on a deep q-network in simulation, and gradually introducing data more similar to what is produced by the real environment, we can obtain desired results.

Virginia Tech CSx824/ECEx424 final project report. Copyright 2016 by the author(s).

2. Related Work

2.1. Reinforcement Learning in Robotics

Using reinforcement learning as a technique to obtain desired behaviour from a robotic vehicle is not a novel concept. As (Mahadevan & Connell, 1991) shows, the use of reinforcement learning in order to program behaviour based robots can be reasonably achieved. Furthermore, the (Mahadevan & Connell, 1991) also utilizes the concept of Q-learning to program desired behaviours. This paper, based off of (Mnih et al., 2013), uses a function appropriator to represent the action-value function which will be discussed in a later section. However, (Mahadevan & Connell, 1991) utilized the vanilla Q-learning function. (Mahadevan & Connell, 1991) mentions that prior attempts to implement Q-learning with respect to robots were monolithic, meaning that a single action or behaviour was desired from the robot. (Mahadevan & Connell, 1991) presents a robot that is manually programmed to select between multiple Q-functions dictated by temporal conditions. The robot being implemented in this work has access to a multitude of actions and desired behaviours that are dependent on conditions that the algorithm learns, rather than designer-defined conditions.

2.2. Transfer Learning

Q-learning requires a series of example situations in order to explore the state space of the environment. This essentially samples state sequences to gain information about the environment, actions, and rewards. The authors of (Mahadevan & Connell, 1991) actuated 2000 iterations to pro-

vide the Q-function with a sufficient amount of information about the state space. We thought that this was a very inefficient way to train our robots, especially since the state space and the available actions in our set-up was vastly larger. This was a great opportunity to take advantage of transfer learning. Transfer learning, as defined in (Torren & Shavlik, 2009), is the improvement of learning in a new task through the transfer of knowledge from a related task that has already been learned. In this work, it was decided to create a rough simulation of the expected environment. The deep Q-network network was to be trained on the rough simulation, transferred to a more realistic simulation, trained some more, and finally tested on real-time images from the non-simulated environment. As (Yosinski et al., 2014) studies, the first-layers of a convolutional neural network are essentially generalized edge detectors. The latter layers of the neural network learn more complex and non-linear features of the data. The inspiration to use transfer learning came from the expectation that the first layers of the deep Q-network would learn the basic edges of the images. Just as the more pronounced features came to activate the deeper layers, new images would be fed into the network. This would hopefully result in the network learning a very simple, yet rough interpretation of the expected environment. As the real environment came to be, the input images would be modified. We would have to ensure that the Q-learning algorithm will have sufficient explorative ability during theses representation transitions.

3. Background

The novelty presented by (Mnih et al., 2013) was to use a convolutional neural network (CNN) as a function appropriator in the traditional Q-learning algorithm. The goal of Q-learning is to find the optimal policy π that maps sequences to actions. We look at a task where an agent interacts with an environment $\mathcal{E}$, in a series of actions and rewards. As is true in most reinforcement learning settings, at each time-step the agent selects and action a_t which is part of the set of all legal actions $A = \{1, ..., K\}$. This action is then executed in the environment. The execution of this action results in a modification of the state $x_t \in \mathbb{R}^d$, which in our case is an image, and a scalar reward r_t. Due to the fact that the current state is not fully represented by the current image x_t we consider sequences of states $s_t = x_1, a_1, ..., , a_{t-1}, x_t$ and learn from these. The job of π is to chose the action that will maximize the future reward. The need for the function appropriator arises from the difficulty of evaluating future reward. The number of states to explore increases by a factor of K for each time step you estimate in the future. This makes it necessary to approximate each action's future reward. In traditional Q-learning the future reward is referred to as the Q-value and is given by the Q-function, defined as

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q^*(s^{'}, a^{'})|s,a] \tag{1}$$

Where γ is the rate at which we discount future rewards. However, now we examine what this algorithm looks like with a function appropriator. The output of your CNN is defined as $Q(s; a; \theta)$. Using this we can define a loss function

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim p(\cdot)}[(y_i - Q(s,a;\theta_i))^2] \tag{2}$$

$$y_i = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q(s^{'}, a^{'}; \theta_{i-1})|s,a] \tag{3}$$

where y_i is the target for iteration i and $p(s, a)$ is a probability distribution over sequences and actions.

The architecture proposed by (Mnih et al., 2013) uses an image of the current state as input to a CNN and has the output be the Q-value for each legal action. In order to get some temporal dependence when making a decision the last n state images are concatenated as input, $x = x_t, x_{t-1}, ..., x_{t-n}$. In order to allow for a more vast exploration of the state space, actions are chosen on an ϵ-greedy policy that selects the maximizing move with probability $1 - \epsilon$, otherwise it chooses a random move. As the number of training iterations increase ϵ is slowly decreased.

4. Simulated Transfer Learning

We hypothesize that a deep Q-network trained on a simulation of an environment could be used as a starting point for training on data collected from the real environment. This would allow you to apply the deep Q-learning algorithm to robotics. As stated previously, it would not be feasible to perform the number of iterations needed to train a deep Q-network with data examples collected solely from the real world. If you used a simulation-trained model as a starting point for real world training, then you could potentially decrease the number of real world training iterations needed by several orders of magnitude.

Unfortunately, at this point in time we do not have the infrastructure needed to apply transfer learning in real life. In order to test our hypothesis, we will simulate the transfer learning. To do this we train a model using one image representation of our environment, then after some number of iterations we switch the image representation. The idea behind this is that our simulation won't perfectly capture the environment and if our model is able to start out better than random after the switch is made, and then continue learning, then the real life transfer learning would most likely be successful as well.

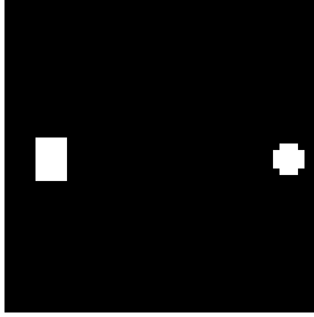

Figure 1. Image of what the input to the CNN looks like initially.

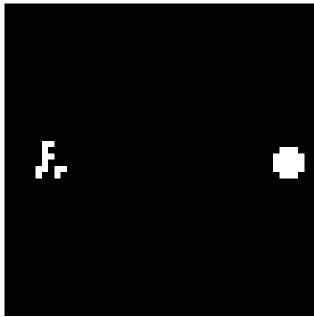

Figure 2. Image of what the input to the CNN looks like after the switch is made.

5. Deployment

5.1. Mapping Actions

The simulation presents a series of actions to the rover that are available for execution during training. The actions the rover may take at any state include moving backwards and forwards at various speeds as well as the option of applying no power to the motors at all.The simulation consists of a one dimensional series of spaces to be occupied by the rover at any one time. The size of this vector space was 38 discrete locations. To ensure consistency of intended moves from the simulation to the true environment, a mapping was required. Of all available actions producing movement, timed experiments were run to ascertain the time required to go from one side of the simulated environment to the other. After measurements were obtained, various motor commands were tested in the non-simulated environment. The motor commands that produced the same times to cross the real environment were selected as a mapping to represent the respective actions in simulation.

5.2. Image processing

As with many hardware sensors used in practical applications, noise is a rather large concern. This factor, coupled with the need to simulate an unseen environment, resulted in many uncertainties of what exactly we expected to observe. For this reason, processing of the obtained images was needed to better match the modelled environment. Initial steps were taken to reduce the burden of processing on the embedded system by placing infra-red LEDs atop the robots, removing the IR filter within the camera sensor, and finally placing a visible light filter in-front of the sensor. This simplified the original image contents. The processing began with a conversion to a mono-chromic bitmap. Then, the image was rotated to be perpendicular with the borders to eliminate any angular discrepancies between the camera and the environment. This rotation was followed by a cropping that produced a square image. This square image was then resized to 50x50 to match the needs of the input layer to the convolutional neural network. The resized image was sent through a filter that removed excess noise and reflections from the borders of the environment.

Figure 3. This is an example of the raw input captured by our camera.

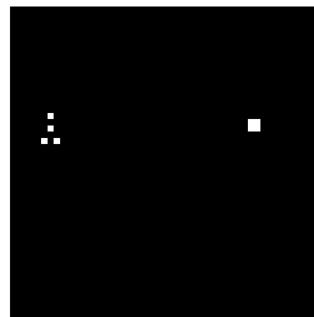

Figure 4. This is an example of an image after pre-processing. As you can see it resembles the images we used in simulation.

6. Experiments

6.1. Model and Simulation Architecture

Our simulation dealt in a 50x50 world. A ball was shot at random angles and speed towards our agent. The agent was rewarded 1 point for coming in contact with the ball, it was penalized 1 point (-1 reward) for allowing a goal. These are the only two states that had a reward, all other states had 0 reward. We used almost the same architecture described in (Mnih et al., 2013), but changed up the filter sizes in our CNN to accommodate for differences in input image size. The input to the CNN consists of an 50x50x4 image, representing the last 4 time steps concatenated into one image. The first hidden layer convolves 16 15x15 filters with stride 4, followed by a non-linearity. The second hidden layer convolves 32 4x4 filters with a stride of 2, again followed by a non-linearity. The final hidden layer is fully-connected and consists of 256 non-linearities. The output layer is a fully connected linear layer with a single output for each valid action.

6.2. Description of Experiment

We trained the deep Q-network for approximately 900,000 iterations using the image representation seen in Figure 1. After that point in time we froze the weights in the CNN and changed the image representation to that seen in Figure 2. We continued training for around 40,000 iterations and observed the results. At the same time, we continued to train using the image representation to compare its results moving forward. We hypothesized that we would see a drop in average Q-value, and a rise in cost when making the switch to the different image representation. However, we expected that they would both rebound as iterations increased.

7. Results

Here we show results from the experiments described above.

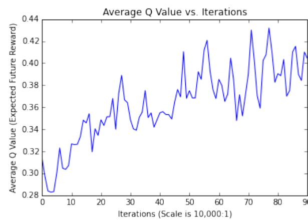

Figure 5. This graph shows our initial training on the images from Figure 1. It shows that through training you are able to increase your average Q-value. This means that the algorithm expects a higher future reward, which is a proxy to overall performance.

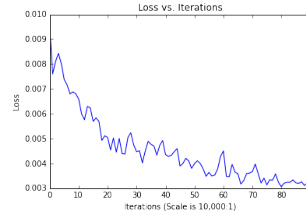

Figure 6. This graph shows the cost function decreasing over time in our initial training. This tells us that our model is learning to predict future reward more accurately over time.

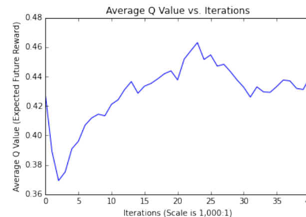

Figure 7. This image shows the average Q-value after we froze the weights from training and changed the input images to those seen in Figure 2. As you can see, the average Q-value drops initially due to the differences in input images, however the algorithm is able to continue training with the new images and recover from the initial drop.

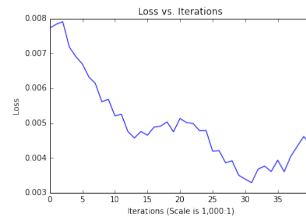

Figure 8. Comparing this graph to the one seen in Figure 6, you can see that the loss starts at a much higher value than what Figure 6 ended at. This is presumably due to the differences in input images. You can see a minor increase in cost that correlates to the decrease of the average Q-value seen in Figure 7. Over time the algorithm is able to continue improving by decreasing the cost, despite the change in input images.

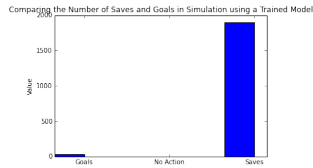

Figure 9. This graph shows the results of each episode when continuing to train our original model for another 40,000 iterations. As you can see at this point, the model is more or less trained. It achieves a save percentage of about 98% over apporximately 2,000 episodes.

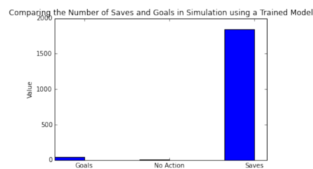

Figure 10. This graph shows the results of each episode when switching the input images to the ones shown in Figure 2. As you can see, the results are comparable. The save percentage when using the different images was approximately 97%. This shows that although there was an initial drop in the average Q-value, the overall performance of the system was not impacted heavily.

8. Conclusion and Future Work

The goal of this paper was to show that by using standard training methods on a deep Q-network in simulation, and gradually introducing data more similar to what is produced by the real environment, we can obtain desired results. In our experiment we simulated this by training on one image representation, then switching to another one while keeping all model parameters constant. Our results showed that while there was an initial drop in performance after the switch, the overall performance was not heavily impacted in the end. We also showed that it is possible to continue training with the new image representation and achieve improved results. We believe this shows that you could train a model in simulation, then use that model as a starting point to test in a real environment. This should reduce the amount of training iterations needed to achieve desired results in the real world.

In future work we would like to test our hypothesis in the real world. As of now, the infrastructure to do learning on data from the real world is not in place. While our simulated transfer learning results are favourable, it would be interesting to see if transfer learning from simulation to the real world improves the performance of the agent in the way we saw in this paper.

References

M., Araki. Pid control.

Mahadevan, Sridhar and Connell, Jonathan. Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings of the 9th National Conferance on Artificial Intelligence (AAAI-91)*, T.J. Watson Research Center, Box 704 Yorktown Heights, NY, 1991.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. Technical report, Deepmind Technologies, December 2013.

Torren, Lisa and Shavlik, Jude. Transfer learning. In Soria, E., Martin, J., Magdalena, R., Martinez, M., and Serrano, A. (eds.), *Hanbook of Research on Machine Learning Applications*. IGI Global, 2009.

Yosinski, Jason, Clune, Jeff, Bengio, Yoshua, and Lipson, Hod. How transferable are features in deep neural networks? Technical report, 2014.