CS230

Learning to manage a system of bridges subject to seismic hazard using deep-Q networks

Gitanjali Bhattacharjee
Department of Civil and Environmental Engineering
Stanford University
gjee@stanford.edu

Abstract

Optimizing the management of highway bridges subject to uncertain seismic hazard remains an open problem in civil engineering. The problem can be cast as a Markov decision process (MDP), with management actions comprising doing nothing, retrofitting, or repairing bridges in the system. Model-based solution techniques become intractable for larger numbers of bridges. We therefore investigate the use of deep Q-learning – a model-free reinforcement learning (RL) technique – to approximate the action-value function, $Q(s, a)$, for a three-bridge system. We develop an architecture for a deep Q-network (DQN) and apply it to one- and ten-year planning horizons, with mixed results. While the DQN performs well in the one-year episodes, further work is required to understand the RL agent's performance over the decade-long horizon.

1 Introduction

Optimizing the management of spatially distributed infrastructure systems subject to uncertain hazard remains an open problem in civil engineering. Improving the management of highway bridges subject to seismic hazard is one subset of this broader class of problems. Highway bridges at seismic risk are of particular interest because their failure during earthquakes can lead to the disruption of transport systems at a regional scale, leading to large indirect costs (i.e., costs beyond those of repairing the bridges) [1]. In this project, we use a small subset ($b = 3$) of a larger test bed of bridges ($B = 1743$) owned and managed by the California Department of Transportation (CalTrans). All bridges are located in the San Francisco Bay Area, and therefore subject to substantial seismic hazard.

Like other structures, a highway bridge's capacity to resist ground shaking can be described using a fragility function. Fragility functions are typically created for classes of structures and calibrated for individual structures. Fragility functions map from a measure of ground-shaking to the probability that the structure has equaled or exceeded some level of damage. Damage states d are often described as minor, moderate, extensive, or complete (collapsed), in ascending order of severity.

Fragility functions are typically modeled using lognormal cumulative distribution functions (CDFs), which are parameterized by a median asset capacity, μ , and logarithmic standard deviation, β. If X denotes the measure of ground shaking and d_i the damage state of interest, the probability that a structure is in the damage state of interest or a more severe damage state is

$$P(d \geq d_i | X = x) = \Phi(\frac{ln(x) - \mu}{\beta})$$

Thus, the larger μ is, the greater the bridge's resistance to ground-shaking.

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

Here we model the damage state of a bridge as binary. A designation of "damaged" indicates that the bridge has at least extensive damage. A damaged bridge is non-functional. If undamaged, the bridge has some capacity μ. If damaged, the bridge has no capacity to resist ground-shaking. Here we account only for damage due to earthquakes, i.e. we do not account for other sources of structural deterioration.

The management of a highway bridge can be modeled with three actions: doing nothing, retrofitting, or repairing. We can model the effect of each action on the capacity of the bridge to resist ground-shaking by modifying μ accordingly. Doing nothing has no effect on μ, retrofitting increases μ by a multiplicative factor (here, by a factor of two), and repairing restores the bridge's original capacity.

The management of a system of highway bridges can then be cast as a Markov decision process (MDP), where b indicates the number of bridges of interest. The state, s, is described by a vector with 2b elements, since each bridge has a damage state d and a retrofit state r: $[d_0, r_0, \ldots, d_b, r_b]$. (For clarity, the aforementioned state will be referred to as the environment state.) The agent can choose one of the three aforementioned actions at each bridge. Thus, the action is described by a vector of b elements: $[a_0, \ldots, a_b]$. We use a simple rewards function,

$$r(s,a) = \frac{1}{b}\sum_{i=1}^{b} F_i, \text{ where } F_i = \begin{cases} 1 & \text{if bridge } i \text{ is functional} \\ -1 & \text{if bridge } i \text{ is not functional} \end{cases}$$

as a proxy for system performance. The reward has a range of $[-1, 1]$. All actions are free – i.e., retrofits and repairs cost nothing.

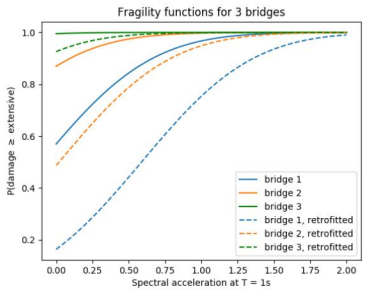

Figure 1: Retrofitting a structure shifts its fragility function to the right.

While model-based techniques can be used to solve the above MDP if b is small, they become intractable for larger numbers of bridges as tabular representations become computationally onerous. Therefore, we investigate the use of deep Q-learning to approximate the action-value function, as detailed in *Methods* [2]. The input to the deep Q-network (DQN) is a state described by a vector with b elements, $[\mu_1, \ldots, \mu_b]$, each of which is the median resistance to ground-shaking, μ, of the associated bridge. (This will be referred to as the agent state.) The DQN outputs a vector of 3^b elements, each of which is the $Q(s,a)$ value associated with a particular action vector.

2 Related work

Deep Q-learning has not, to the best of our knowledge, been applied to physical infrastructure management. However, MDPs have been used as models for infrastructure management in various contexts, most notably in the management of individual bridges [3]. More recently, partially-observable MDPs (POMDPs) have been used to model the management of the components of a single bridge subject to deterioration and under uncertainty [4,5,6].

Many papers in the literature address the distinct but related challenge of optimizing the management of a system of component structures whose performances are additive. For example, minimizing the operations and maintenance costs of a wind farm comprising multiple turbines can be reframed as minimizing the O&M costs of each turbine [7].

However, the relationship between the state of a single bridge in the highway network and the performance of the whole network, as measured by various traffic metrics, is not known analytically. The relationship between the state of the whole bridge network and the traffic network performance requires simulation of multiple systems [1]. Furthermore, assumptions of independent behavior cannot reasonably be made for a portfolio of structures subject to seismic hazard [8].

3 Dataset and Features

In addition to the fragility function parameters of each bridge, datasets required to build the environment simulator included the ground motions associated with each earthquake in the Uniform

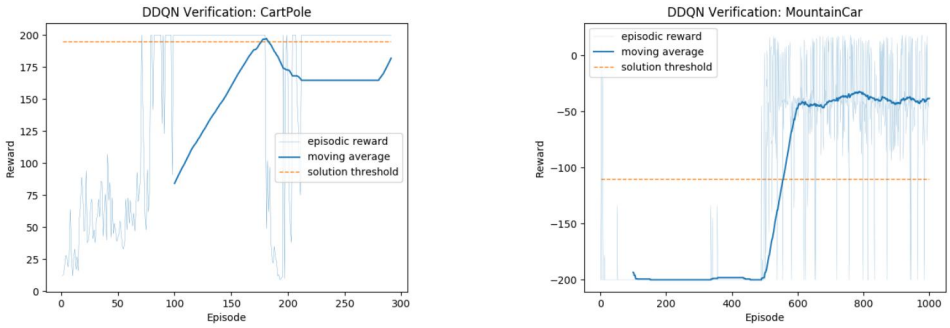

Figure 2: Performance of DDQN agents playing CartPole (left) and MountainCar (right).

California Earthquake Rupture Forecast, Version 2 (UCERF2) and the annual frequency of each of the ruptures in UCERF2. These datasets can be obtained as part of Miller's traffic model [9].

4 Methods

Deep Q-learning is an off-policy model-free reinforcement learning (RL) method that uses a deep Q-network (DQN) to approximate the action-value function, $Q(s, a)$, of an MDP [2]. The deep Q-network takes as input a state vector and produces as output a vector in which each element is an estimate of the Q-value of taking a particular action from the input state [2]. The parameters of the DQN, θ, are updated via gradient descent on the loss, computed for a single sample as:

$$loss = (y - Q(s, a; \theta))^2, \text{ where } y = \begin{cases} r(s, a) & \text{if } s \text{ is terminal} \\ r(s, a) + \gamma * \max_{a'} Q(s', a'; \theta) & \text{if } s \text{ is not terminal} \end{cases}$$

However, learning solely online from consecutive samples, has three principal drawbacks: (1) it is data-inefficient, as each sample is used only once (2) it is inefficient due to the strong correlations between the samples (3) it increases the risk of parameters getting stuck in poor local minima, or even diverging [2]. Instead of this standard online Q-learning approach, Mnih et al suggest that the agent should also accelerate learning through experience replay [2]. To do so, the agent stores observations of the state, action, reward, and next state – or (s, a, r, s') tuples – in memory. Then, the agent randomly selects a minibatch of observations from memory, computes the loss on each sample in the minibatch, and updates the parameters of the DQN via minibatch gradient descent on the loss [2].

Deep Q-learning uses the same DQN to select and evaluate actions, which can result in overestimation of Q-values [10]. Those overestimations may lead to "overoptimism" and suboptimal policies [10]. Van Hasselt et al proposed double deep Q-learning, a refinement in which the $max_{a'}$ operation is decomposed into action selection and action evaluation [10]. In brief, two distinct DQNs with the same architecture are instantiated: a main DQN and a target DQN. During experience replay, the main DQN is used to select the action that will maximize the Q-value of the next state while the target DQN is used to estimate what that Q-value is. Gradient descent is used to update the parameters of only the main DQN. At some update interval (a hyperparameter specified by the user), the weights of the main DQN are copied to the target DQN. Double deep Q-learning has been shown to significantly improve the performance of an agent and was tested in this problem setting as well, with good results [10] .

5 Experiments, Results, and Discussion

Validation of the DDQN implementation was performed by solving, as shown in Figure 2, two classic OpenAI Gym environments: CartPole-v0 and MountainCar-v0. The solution criterion for CartPole is to achieve an average score of at least 195.0 over 100 consecutive episodes [11]. The DDQN agent solved CartPole in 176 episodes using an NN with two hidden layers (HLs) of 8 and 4 nodes, a learning rate of 0.001, and a batch size of 32 for experience replay. The solution criterion for

MountainCar is to achieve an average score of at least -110.0 over 100 consecutive episodes [12]. The DDQN agent solved MountainCar in 556 episodes using an NN with three HLs of 256, 128, and 64 nodes, a learning rate of 0.0001, and a batch size of 64 for experience replay. When playing MountainCar, the DDQN agent was allowed to observe (and store) 1000 episodes before starting to train. All hidden units used ReLU activation functions with He uniform variance scaling initialization, while the units in the output layer used a linear activation function.

For this project, $b = 3$ for computational tractability, and the size of the action space $|A| = 27$. As this is a non-standard deep Q-learning problem, identifying a suitable NN architecture – i.e., one that allowed the accurate prediction of Q-values proved a necessary first step. To identify such an architecture, we compared the ability of various NNs to predict the returns of one-step episodes. In an episode with multiple time-steps,

$$Q(s, a) = r(s, a) + \gamma \text{argmax}_{a'} Q(s', a')$$

However, in an episode with just one time-step, this equation reduces to

$$Q(s, a) = r(s, a)$$

which makes evaluation of an NN's predictive power straightforward and relatively efficient.

The NNs' predictive power was evaluated every 100 training episodes by measuring the loss (mean-squared error) recorded on a batch of 64 one-step episodes randomly selected from memory. Multiple NNs were tested, with two to five HLs, each with five to 500 hidden units. All HLs had ReLU activation functions with He uniform variance scaling initialization, while the output layer used a linear activation function. All NNs used the Adam optimization algorithm, and their losses were measured as a mean-squared error. All NNs had minibatch sizes of 64. The choice of hyperparameters – in particular, learning rate and minibatch size – was predicated on a combination of two factors: first, observing that a learning rate of 0.0001 and a minibatch size of 64 resulted in both relatively speedy and good performance on the MountainCar task and second, brief trial and error with learning rates of 0.001 and 0.00001 proving less successful.

The best-performing NN had two hidden layers, each with 300 hidden units, and a learning rate of 0.0001. As expected, the DDQN agent outperformed the DQN agent, with a minimum loss of 1.1 after 80000 episodes, compared to 2.7 for the DQN agent. For the DDQN agent, the target network was updated every 20 steps, a hyperparameter choice based on trial and error. The magnitude of the performance difference between the two agents indicates that using DDQN would be useful for longer episodes, as expected per [10].

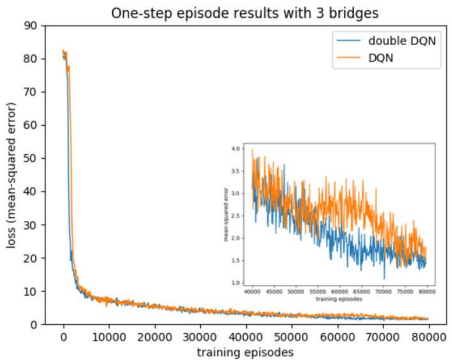

Figure 3: Loss over time of one-step return predictions;inset shows discrepancy between DQN and DDQN results during last half of training.

One key change that made predicting the one-step episode returns possible was changing the DQN input from the environment state to the agent state. The rationale for this change was that a structural engineer could reason that a set of bridges with larger fragility function parameters will probably perform better in an earthquake than a set of bridges with smaller fragility function parameters, and thus lead to larger rewards. However, even a structural engineer would find it more difficult to see this relationship if presented with environment state representations, which are tuples of binary variables. Thus, we could reason that an NN might more easily learn the relationship between fragility function parameters and overall performance from the agent state.

Having identified an NN architecture capable of learning a $Q(s, a)$ function for one-step episodes, we then used the same architecture for a DDQN agent and extended the length of the episodes to 10 timesteps (10 years). This longer horizon reflects the length of real planning horizons in infrastructure management, which may extend further to 30 or even 50 years. The choice of hyperparameters was based on those that worked well

for the one-step episodes: a learning rate of 0.0001, a target network update interval of 20 steps, and a minibatch size of 64.

The undiscounted return for a ten-year episode ranges from -10 to 10. The agent was evaluated every 100 training episodes; its average undiscounted episodic return over 10 decade-long evaluation episodes (with random starting states) is plotted in Figure 4. The maximum average undiscounted return was 6.4 after 150000 training episodes. However, this maximum was not consistently achieved. The DDQN agent does learn to repair and retrofit bridges, improving its average episodic return with training. However, the improvement is slight and the moving average over the average undiscounted return does not reach or exceed 0.0. As repairs and retrofits are free, the agent was expected to achieve the near-optimal return during evaluation, but did not. This is a somewhat disappointing result, as this problem can be solved in a straightforward manner using simple model-based techniques, like value iteration.

This result may also be due in part to the uncommon fragility of bridge 3, as shown in Figure 1. The agent does not learn to retrofit this bridge, likely because there is little chance of benefit from said retrofit – that is, the bridge is so vulnerable to ground shaking to begin with that retrofitting per the scheme described previously has little effect on the chance it will become damaged in an earthquake. However, the agent should learn that there is some benefit, however small, of retrofitting this bridge since the probability of damage does decrease, however slightly.

The agent's failure to learn may be due more to a lack of training time than other problems. The agent trained for a maximum of 150000 episodes in an action space of size 27. In sharp contrast, Mnih et al. trained their agent for 10 million episodes, though it acted in much smaller action spaces (of size 4 to 18) [2]. This strongly suggests that training was too short. In addition, increasing the NN depth from two to five HLs did not improve results over 50000 training episodes (figure not shown). Though these pieces of evidence are not conclusive, taken together they suggest that increasing training time may be of more concern than potential under-fitting.

6 Conclusion/Future Work

Useful findings from this work include:(1) a continuous representation of the state of the bridges in the system proves a better input for the DQN than a vector of binary variables (2) a shallow and relatively large DQN seems able to approximate the Q-values of a small system of bridges well.

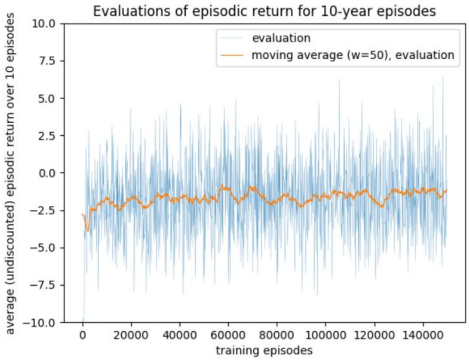

Figure 4: The performance of a DDQN agent on 10-year episodes was evaluated every 100 training episodes and shows some improvement over time.

Immediate future work should include a longer training time for the DDQN agent. It should also include removing repairs from the action space; the agency is unlikely to let a damaged bridge remain damaged for a year, as is currently possible in the simulator. Repairs could instead be treated as an automatic feature of the environment and carry a weighty penalty. This reconfiguration would also address another challenge of continuing this work: namely, reducing the state- and/or action-spaces. Without repairs, the size of the action space would decrease from $|A| = 3^b$ to $|A| = 2^b$.

Future work must also focus on ways to make the management of systems with more components computationally tractable. For sufficiently large b – i.e., greater than 10 – the simple rewards function used here can be replaced by the output of a traffic simulator developed for this very test bed by M. Miller [1]. This will result in a truer measure of system performance, with useful output metrics including the total travel time for all trips made on the network. Other opportunities for future work include increasing episode lengths (e.g., to 30 years), and incorporating prioritized experience replay to increase learning efficiency.

Contributions

This project is the work of the sole author. She appreciates the guidance of Jay Whang, the project TA.

Code

Code associated with this project is available in a private GitHub repository at `https://github.com/gbhattacharjee/cs230_dqn`. CS230-stanford has been invited as a collaborator.

References

[1] Miller, M. Seismic risk assessment of complex transportation networks. (2014) Ph.D. dissertation. Dept. of Civil and Environmental Engineering, Stanford University.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. NIPS Deep Learning Workshop.arxiv:1312.5602.

[3] Frangopol, D.M., Kallen Maarten-Jan, & van Noortwijk J.M. (2004) Probabilistic Models for Life-Cycle Performance of Deteriorating Structures: Review and Future Directions. *Progress in Structural Engineering and Materials*, **6**(4):197–212.

[4] Papakonstantinou, K.G. & Shinozuka, M. (2014) Planning structural inspection and maintenance policies via dynamic programming and Markov processes. Part I: Theory. *Reliability Engineering and System Safety*, **130**: 202-213.

[5] Papakonstantinou, K.G. & Shinozuka, M. (2014) Planning structural inspection and maintenance policies via dynamic programming and Markov processes. Part II: POMDP implementation. *Reliability Engineering and System Safety*, **130**: 214-224.

[6] Fereshtehnejad, E. & Shafieezadeh, A. (2017) A Randomized Point-Based Value Iteration POMDP Enhanced with a Counting Process Technique for Optimal Management of Multi-State Multi-Element Systems. *Structural Safety*, **65**:113-125.

[7] Memarzadeh, M., Pozzi, M., & Kolter, J.Z. (2014) Optimal Planning and Learning in Uncertain Environments for the Management of Wind Farms. *Journal of Computing in Civil Engineering*, **29**(5):04014076-1-04014076-10.

[8] Jayaram, N. & Baker, J.W. (2009). Correlation model for spatially distributed ground-motion intensities. *Earthquake Engineering and Structural Dynamics*, **38**:1687–1708.

[9] Miller, M. (2014). Quick traffic model. [Source code] `https://purl.stanford.edu/hx023kk0983`

[10] van Hasselt, H., Guez, A., & Silver, D. (2015) Deep reinforcement learning with double Q-learning. *arXiv preprint*. arXiv:1509.06461.

[11] The CartPole-v0 Environment. OpenAI Gym, `gym.openai.com/envs/CartPole-v0/`.

[12] The MountainCar-v0 Environment. OpenAI Gym, `https://gym.openai.com/envs/MountainCar-v0/`.

Other codes and libraries used

[13] Keras. Chollet, François and others. 2015. `https://keras.io`

[14] Oliphant, T.E. (2006) A guide to NumPy, USA: Trelgol Publishing.

[15] Hunter, J.D. (2007) Matplotlib: A 2D Graphics Environment, *Computing in Science & Engineering*, **9**:90-95.