MASTERING ATARI PONG USING DEEP REINFORCEMENT LEARNING

TEAM MEMBERS

Arpit Saxena
Kritti Sharma
Shreyas Changothia
Shubham Lohiya

INSTRUCTOR

Prof. Abir De
Department of Computer Science and Engineering, IIT Bombay

Contents

1 Introduction

Controlling agents with high dimensional sensory inputs is a long-standing challenge of Reinforcement Learning (RL). Deep Learning (DL) has actually made it possible to extract high-level features from raw sensory data by utilizing a wide range of neural network architectures (convolutional networks, multi-layer perceptrons, restricted Boltzmann machines and recurrent neural networks). Implementation of similar techniques in RL, although potentially very beneficial when dealing with sensory data, can be quite challenging from a DL perspective. Couple of these challenges are listed below:

- Typically, DL requires large amounts of hand-labelled data whereas RL requires us to to learn from a scalar reward signal which can be sparse, noisy and delayed. The delay between actions and rewards can be quite long, as compared to the direct instantaneous association between inputs and targets in conventional DL applications.
- Another major concern is that the data in DL is almost always assumed to be independent, whereas in RL, the states can be highly correlated. Moreover, this data distribution is subject to change as our RL algorithm learns, whereas DL assumes a fixed underlying data distribution.

The paper (Mnih et al., 2013) demonstrates how all of the above listed challenges can be overcome using a convolutional neural network to learn successful control policies from sensory input data in complex reinforcement learning enviroments. This convolutional neural network is trained with a variant of Q-learning algorithm and stochastic gradient descent is used to update the weights. An experience replay mechanism is implemented to take into account the correlations in the data. The methods are applied primarily to the Atari 2600 game, Pong. Results are also obtained on Atari games such as Breakout and Boxing.

2 Background

In Reinforcement Learning (RL), the agent interacts with the environment in a sequence of actions, observations and rewards. At each time-step, the agent chooses an action a_t from the set of legal game actions, $A = \{1, ..., K\}$, which modifies the internal state of the emulator. This is observed in the form of an image x_t, which is a vector of raw pixel values representing the current screen. In addition it receives a reward r_t representing the change in game score.

Since it is impossible to fully understand the current situation from only the current screen x_t, we consider sequences of actions and observations, $s_t = (x_1, a_1, x_2, ..., a_{t-1}, x_t)$, and learn game strategies that depend upon these sequences. Each of these sequences are assumed to terminate in a finite number of time steps. This formalism leads to a large Markov Decision Process (MDP) problem where the goal of the agent is to learn the actions which maximizes the future rewards.

The future discounted return at time step t with a discounting factor γ per time-step, is defined as:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

where T is the time step at which the game terminates.

The optimal Action value function, defined as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a, is given by:

$$Q^*(s,a) = max_\pi \, \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

where π is a policy mapping sequences to actions.

In RL problems such as learning to play Atari games, where the input state space is computationally very large, it is not feasible to maintain a tabular estimate for the action value function for each state-action pair. Thus, a linear function approximator is used to estimate the action value function as $Q(s,a,\theta) \approx Q^*(s,a)$. A neural network function approximator (referred to as a *Q-network*) is trained by minimising a sequence of loss functions $L_i(\theta_i)$ that change at each iteration i:

$$L_i(\theta_i) = \mathbb{E}_{s,a \ \rho(.)}[(y_i - Q(s,a;\theta_i))^2]$$

where θ_i are the weights, y_i is the target for the ith iteration and $\rho(s,a)$ is the probability distribution over the sequences and actions.

This loss function is optimised using stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution and the emulator, the algorithm is known as ***Q-learning***. This algorithm is well known to be *model-free* and *off-policy*.

3 Literature Survey

3.1 TD-Gammon Neural Network

Temporal difference methods are based on the concept that learning relies on the difference between temporally successive predictions, with the goal of making the learner's predictions for the current input, match closely the ones at the subsequent time step. The TD-Gammon Neural Network (Tesauro, 1994) works with little knowledge of the game backgammon and learns to play at superhuman levels. Organised as a standard multi-layer perceptron architecture, TD Gammon was designed to learn complex non-linear functions. The neural network observes a sequence of board positions which are fed in as input vectors. For each of these input vectors, the neural network generates an output vector indicating the exected outcome of that particular input vector. The weight update at each time step is done using the TD(λ) algorithm as follows:

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

where α is the learning rate, w is the weights vector, Y_k is the output of the network and $\nabla_w Y_k$ is the gradient of network output with respect to the weights. When the game ends, a reward r is assigned and the weights are updated using $(r - Y_x)$ instead of $(Y_{t+1} - Y_t)$.

3.2 Neural Fitted Q-Iteration

The Neural Fitted Q-Learning algorithm (NFQ) (Riedmiller, 2005) belongs to the family of fitted value iteration algorithms and is a special form of experience replay technique, implemented as a multi-layer perceptron. The updates are performed off-line by considering an entire set of transition experiences of the form (s, a, s'), where s is the original state, a is the action and s' is the transitioned state. The supervised learning method for batch learning, RPROP was used. The target is calculated as the sum of the cost of transition and the expected minimal path cost for the state s', which is computed on the basis of the current estimate of the Q-function. This memory based method of training by storing and reusing all transition experiences makes the neural learning process quite data efficient and reliable.

4 Method

In this project, we use the algorithm, ***Deep Q-Learning with Experience Replay***, for learning to play Atari Games. We store the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set $D = \{e_1, ...e_N\}$, pooled over many episodes into a *replay memory*. During the inner loop of the algorithm, we apply Q-learning minibatch updates to samples, $e \sim D$, drawn at random from the pool of stored samples. After performing experience replay, the agent executes an action according to an ϵ-greedy policy. The Q-function works on fixed length representation of histories produced by a function ϕ.

Algorithm: Deep Q-Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = 1 to M do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for t = 1 to T do
        With probability ϵ select a random action a_t
        otherwise select $a_t = max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t
        and image x_{t+1}
        Set $s_{t+1} = (s_t, a_t, x_{t+1})$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = \begin{cases} r_j, & \text{for terminal } \phi_{j+1}, \\ r_j + \gamma max_{a'}\{Q(\phi_{j+1}, a'; \theta)\}, & \text{for non-terminal } \phi_{j+1}. \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

5 Model Architecture

Raw Atari frames are 210×160 pixel images with a 128 color palette. These raw frames are preprocessed by gray-scaling, down-sampling and cropping into an (84×84) image. We have used Open AI Gym Environment: PongNoFrameskip-v4. For the experiments here, this preprocessing is performed on the last 4 frames of a history and stacked together to produce the $(4 \times 84 \times 84)$ input to the Q-network.

The way we have parameterized the Q-function is by using an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual action for the input state.

The $(4 \times 84 \times 84)$ input image to the neural network is acted upon by the first hidden layer, consisting of 32 (8×8) convolutional filters with stride 4, followed by the application of a rectifier nonlinearity. The second hidden layer convolves 64 (4×4) filters with stride 2, again followed by a rectifier nonlinearity. The next hidden layer convolves 64 (3×3) filters with stride 1, again followed by ReLU. The final hidden layer is fully-connected, consisting of $(512 \times (64 \times 7 \times 7))$ linear weights and 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The valid actions in the case of Pong were the following 6 actions: ['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']

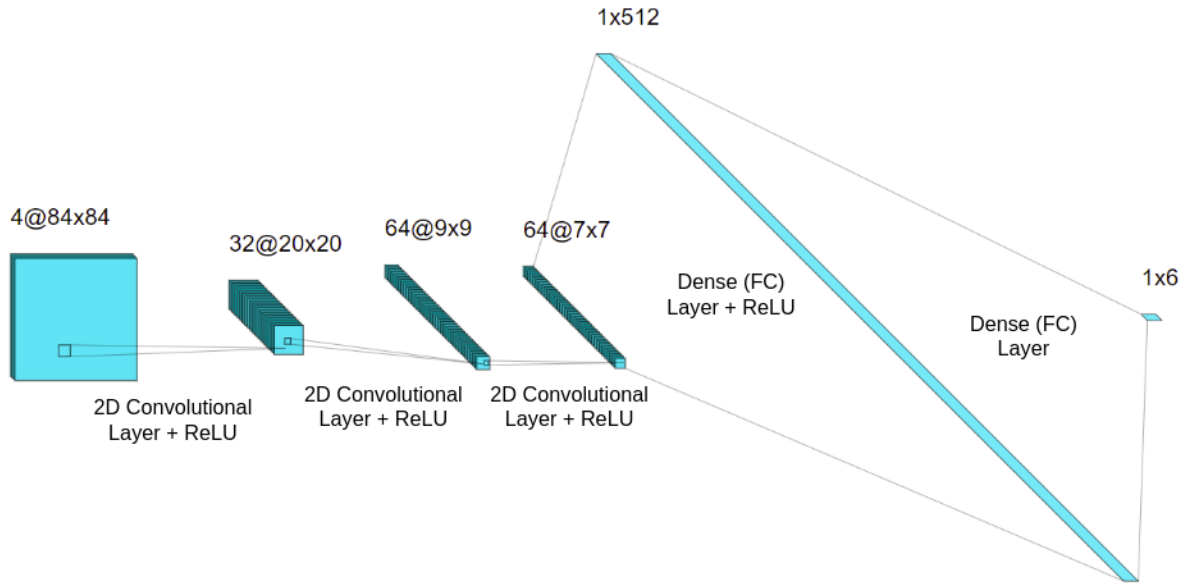

Figure 1: Deep Q-Network (DQN) Architecture

6 Modifications

Q-Learning algorithms suffer from substantial over-estimations of Q-values because they include a maximization step over estimated Q-values. These over-estimations, not only increase the number of episodes of training required to reach optimal policies, but can even lead to sub-optimal policies asymptotically (Thrun and Schwartz, 1993). We propose using a Double Deep Q-Learning Network (DDQN) to tackle this problem. DDQN uses two separate Q-value estimators (2 neural networks). A target model(Q') for action selection and a primary model (Q) for action evaluation. Using these independent estimators, we can avoid the maximization bias by disentangling our updates from biased estimates.

For updating model Q, we use the following target values:

$$Q^*(s_t, a_t) \approx r_t + \gamma Q(s_{t+1}, argmax_{a'}\{Q'(s_{t+1}, a')\})$$

And then perform gradient descent on Q using:

$$Loss = (Q^*(s_t, a_t) - Q((s_t, a_t)))^2$$

To update the target model, we periodically set the weights equal to the weights of the primary model (Q)

Algorithm: Double Deep Q-Learning

Initialize primary network Q_θ, target network $Q_{\theta'}$,replay buffer D, $\tau << 1$
for each iteration do
    for each environment step do
        Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
        Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
        Store (s_t, a_t, r_t, s_{t+1}) in replay buffer D
    for each update step do
        sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim D$
        Compute target Q value:
            $Q^*(s_t, a_t) \sim r_t, +\gamma Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a')$
        Perform gradient descent step on $(Q^*(s_t, a_t) - Q_{\theta'}(s_{t+1}, a'))$
        Update target network parameters:
            $\theta' \longleftarrow \tau * \theta + (1 - \tau) * \theta'$

---

7 Experiments and Results

7.1 Pong

Both DQN and DDQN are able to achieve SOTA score of 20 with an average reward over the last 10 epsisodes being 19.6 for DQN and 18.6 for DDQN. DQN takes around 600 episodes of training to achieve peak performance while DDQN achieves the same in 200 episodes. Playing at various model checkpoints reveals how good our agent after a certain number of episodes.

DQN

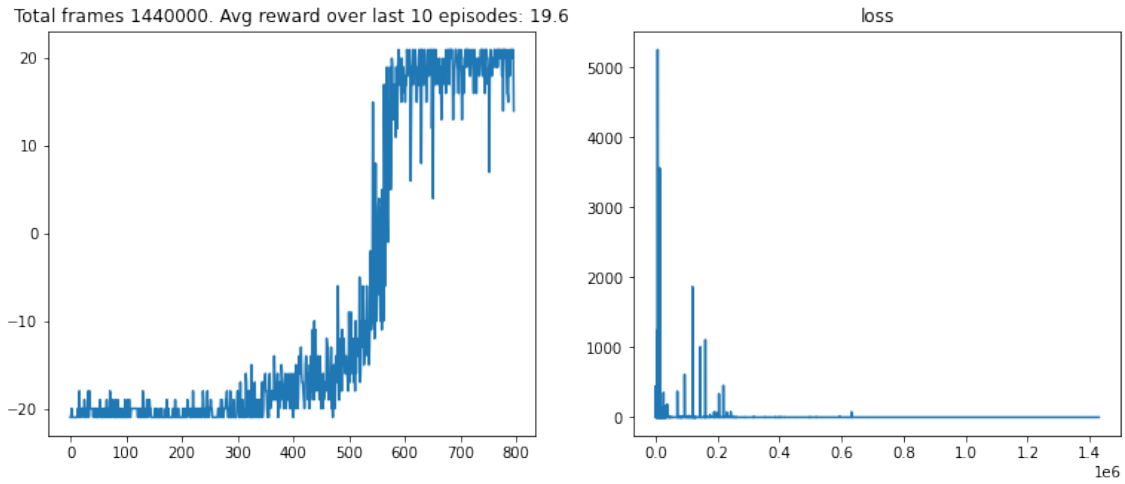

Figure 2: Pong-DQN.

DDQN

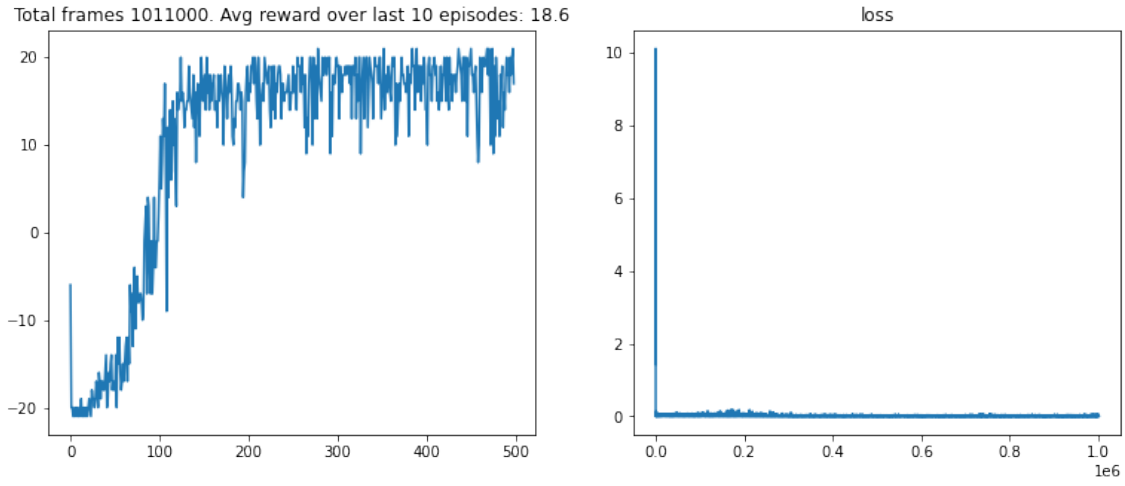

Figure 3: Pong-DDQN.

7.2 Breakout

Both DQN and DDQN show steady improvement in incurred reward as they train. However, due to lack of time and computational resources, we were not able to achieve SOTA score. The test performance clearly shows the ability of the agent to play well.

DQN

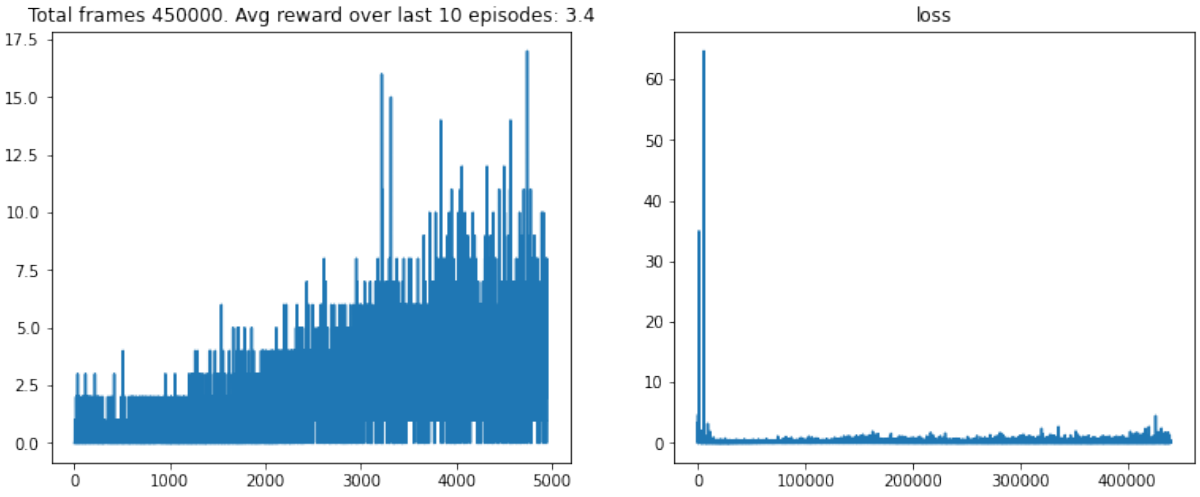

Figure 4: Breakout-DQN.

DDQN

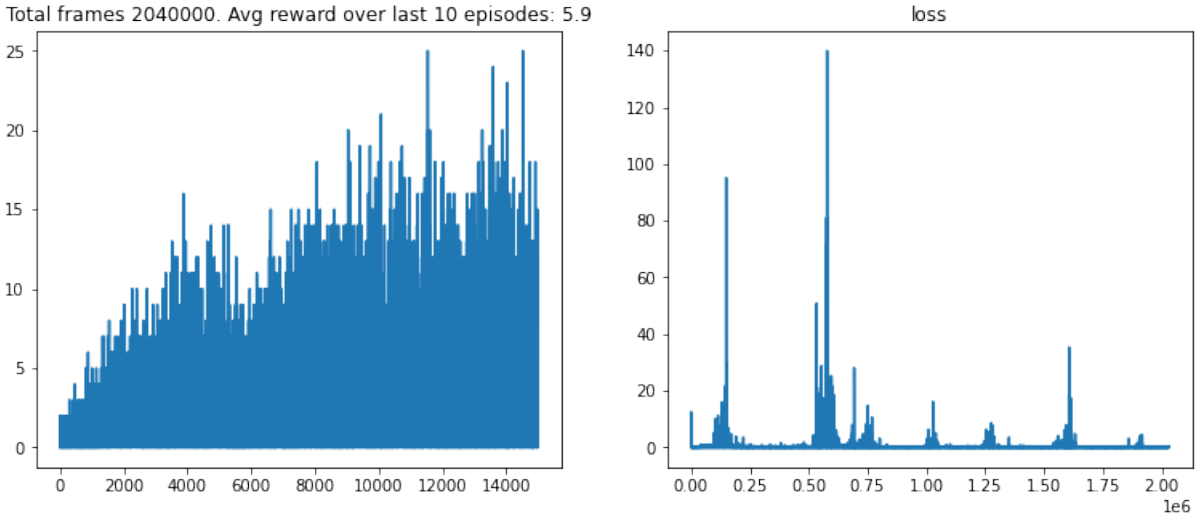

Figure 5: Breakout-DDQN.

7.3 Boxing

We did not achieve significant breakthroughs in just 400 episodes, although it maintains an average score close to 0, which implies it has learned to play at least as good as the opponent. In the few tests that we ran, the agent beat the opponent.

DQN

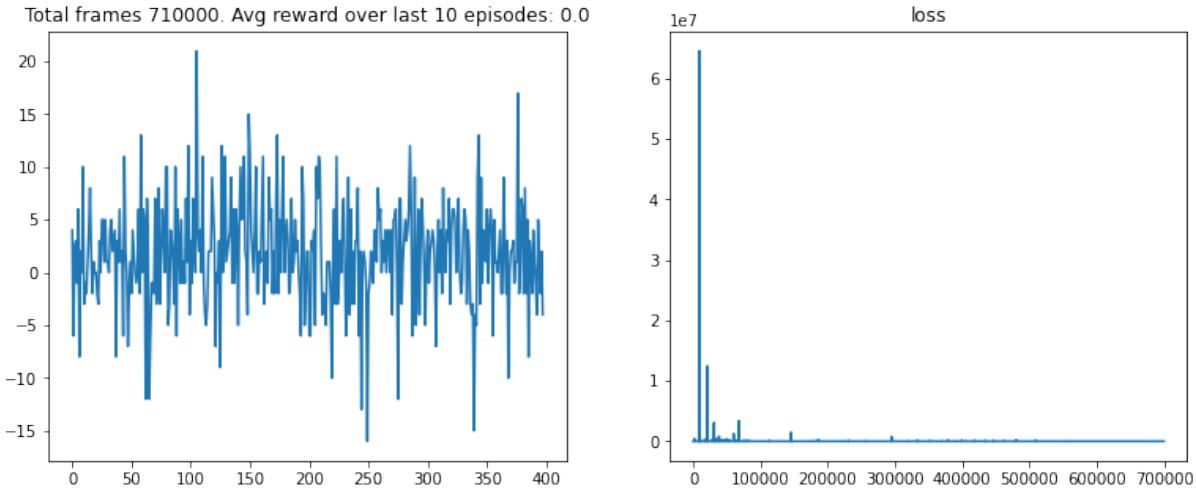

Figure 6: Boxing-DQN.

8 Discussion and Future Work

- Using a Deep Q-Learning model with experience replay memory with only raw pixels as input, we were able to achieve SOTA score in Pong Atari.
- The over-optimistic nature of Q-Learning models can affect their performance. A Double Deep Q-Learning model was used to successfully reduce this overoptimism, resulting in more stable and faster learning with no adjustment of the architecture or hyper-parameters.
- The same DQN and DDQN networks were used in Atari games Breakout and Boxing, with decent results considering the time and computational resources.
- Further improvements can be made by using Prioritized Replay as opposed to random sampling from the memory buffer. Experiences (transitions) that have a higher TD-loss, and thus had a higher impact on learning will have a higher chance of being sampled. This will make the training faster and more efficient

References

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, art. arXiv:1312.5602, Dec. 2013.

M. Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge, and L. Torgo, editors, *Machine Learning: ECML 2005*, pages 317–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31692-3.

G. Tesauro. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, 6(2):215–219, 03 1994. ISSN 0899-7667. doi: 10.1162/neco.1994.6.2.215. URL https://doi.org/10.1162/neco.1994.6.2.215.

S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In D. T. J. E. M. Mozer, P. Smolensky and A. Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*. Erlbaum Associates, June 1993.