Deep Q-Network Demo

Rishikesh Vaishnav

July 4, 2018

Basic Implementation

Code

Manual Implementation

- The code for this project is available at: `https://github.com/rish987/Reinforcement-Learning/blob/master/demos/deep_q_network/code/deep_q_network.py`.

Keras Implementation

- The code for this project is available at: .

Implementation Details

- Unlike the Atari gameplay environment described by Mnih et. al., the pole-cart environment is not perceptually aliased. That is, the current observation of the state is theoretically all that is needed to determine an optimal value. Therefore, the current state can be equated with the current observation, without taking into account past observations and actions.
- Because the observation space of the Atari gameplay environment is much larger than the pole-cart environment, it should suffice to use a smaller ANN model.
- Because the observation space of the pole-cart environment is small and not spatially correlated, it is not helpful to use a convolutional neural network.

Manual Implementation

- The model is a simple vanilla neural network with one hidden layer:
  - Let M be the number of nodes in the hidden layer.
  - Let K be the number of output nodes (i.e., number of actions).
  - Let $\sigma(x)$ be the sigmoid activation function $\frac{1}{1+e^{-x}}$.
  - The hidden layer is calculated as:

$$Z_m = \sigma(\alpha_{0m}^T + \alpha_m^T x(s)), m = 1, \ldots, M$$

- The output layer is calculated as:

$$\hat{q}(s, a_i; \theta) = \beta_{0i} + \beta_i^T Z, i = 1, \ldots, K$$

- Solving for the gradient of the sample error:

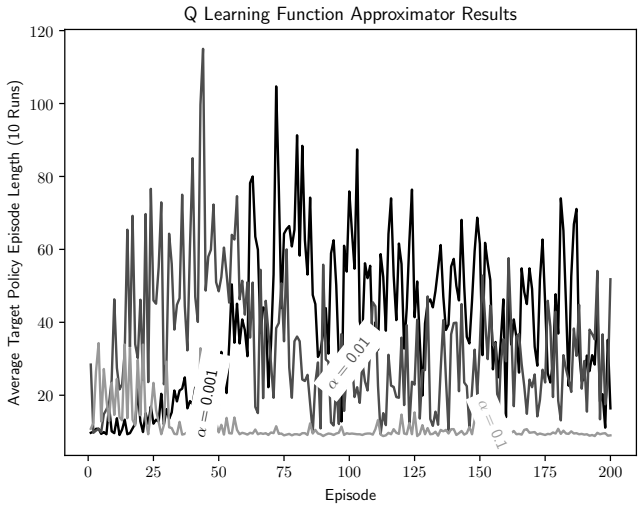
$$\nabla_\theta J_t(\theta) = -2(y_t - \hat{q}(s_t, a_t, \theta))\nabla_\theta \hat{q}(s_t, a_t, \theta)$$

$$\frac{d}{d\beta_{jk}}\hat{q}(s_t, a_t, \theta) = \begin{cases} 0 & k \neq a_t \\ \begin{cases} 1 & j = 0 \\ \sigma(\alpha_{0j}^T + \alpha_j^T x(s_t)) & j > 0 \end{cases} & k = a_t \end{cases}, k = 1, \ldots, K$$

$$\frac{d}{d\alpha_{im}}\hat{q}(s_t, a_t, \theta) = \beta_{mk}\sigma'(\alpha_{0m}^T + \alpha_m^T x(s_t)) \begin{cases} 1 & i = 0 \\ x(s_t)_i & i > 0 \end{cases}$$

- A decaying ϵ was used, which started at 1.0 and decayed to a minimum value of 0.01.
- A constant decay rate was used for the learning rate α.

Results



- The results for different α can be summarized as follows:
  - The largest α learned initially, but worsened and failed to find a policy that was capable of passing the episode at all. This suggests that, in starting with a larger α, it could be useful to decay by a larger factor.
  - The middle α learned very well initially, but worsened and failed to find a policy that was capable of consistently passing the episode. This also suggests that, in starting with a larger α, it could be useful to decay by a larger factor.

  - The small α learned slowly, but eventually leveled off at a better policy than those of the other αs. This may mean that the decay rate was too large relative to this α, causing it to cease improving after α became negligibly small.

- These results suggest that this learner is feasable, and in order to improve this learner, it is necessary to more carefully control how α decays over time. It may also be useful to increase the number of hidden layers in the model.

- To verify that this was the case, I move on to replacing my manual implementation with a Keras implementation that handles training on its own and with which I can easily change the ANN parameters.