A Deep Reinforcement Learning approach for a classic control problem

Ricardo Yamamoto Abe

January 19, 2017

Abstract

This work presents a Q-learning framework implementation based on deep neural networks, called deep Q-learning. An agent was trained to solve the *OpenAI Gym* [1] implementation of the classic *CartPole* control problem [2].

1 Introduction

The problem we will try to solve is based on the cart-pole problem described in [2].Figure 1 shows a screenshot from the OpenAi's implementation *CartPole-v0*. A pole is attached by an un-actuated joint to a cart, which moves along a friction-less track. The system is controlled by applying a force to the left (action 0) or right (action 1). The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time-step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. The cart-pole model has four variables that form an state:

- χ – position of the cart on the track
- θ – angle of the pole with the vertical
- $\dot{\chi}$ – cart velocity
- $\dot{\theta}$ – rate of change of the angle

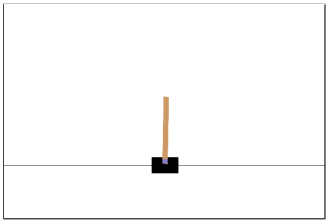


Figure 1: A screenshot from CartPole-v0.

This problem was selected for being relatively simple, having 4 dimensions for the input and 2 actions, but not feasible in a scenario which Q-learning uses tables to store data, as each input variable is not discrete.

We will use a reinforcement learning [3] based implementation, following a deep-learning [4] approach.

2 Data visualization

A sample of states from CartPole-v0 was generated. Analyzing figure 2, generated from a sample of 300 states, we note that each one of the four dimensions is a real number and were normalized to let feature values lie between $[-0.5, 0.5]$. No data preprocessing appears to be needed, as there is no noise generated by the OpenAI environment and input values are constrained.

Besides, there is no apparent correlation between each dimension.

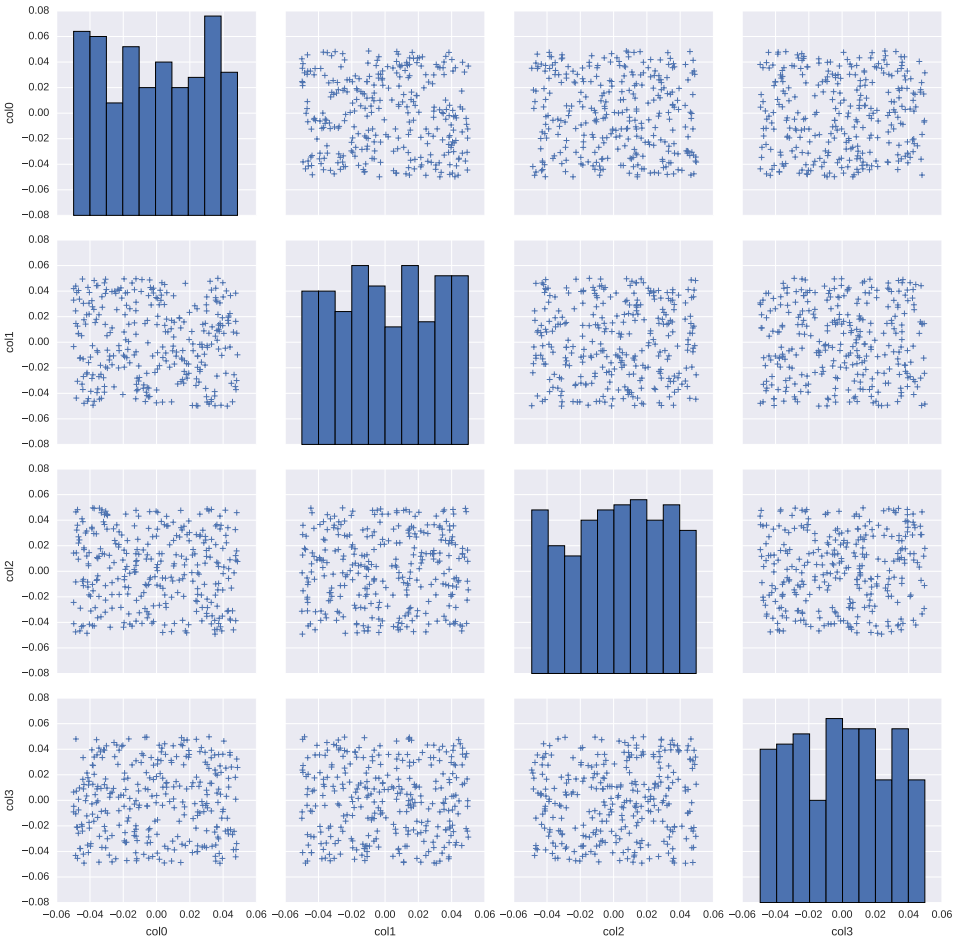


Figure 2: States from CartPole-v0.

3 Reinforcement learning

Reinforcement learning is learning how to map situations to actions so as to maximize a numerical reward signal. The learner discovers which actions yield the most reward by trying them, having to account not only the immediate reward, but future ones too. An usual reinforcement learning technique is Q-learning.

The Q-learning algorithm can be used to find an optimal action-selection policy for any given finite Markov decision process. It consists of:

- S – set of states in the environment.
- A – the set of actions available to the agent.
- $r : S \times A \to \mathbb{R}$ – the reward function of the task: it returns the real number reward provided to the agent for taking an action $a \in A$ from a given state $s \in S$.
- $\gamma \in [0, 1]$ – the discount factor in the expected long-term reward; it trades off the importance of sooner versus later rewards.

The algorithm has a function that calculates the quantity of a state-action combination:

$$Q : S \times A \to \mathbb{R} \tag{1}$$

and an update rule:

$$Q(s, a) := Q(s, a) + \alpha\big[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big] \tag{2}$$

Q-learning at its simplest uses tables to store data, which is unfeasible as the size of state/action space increases. Having continuous space inputs, that is a concern for the cart-pole problem, so a Q-function approximation using a neural network was chosen to tackle it.

3.1 Deep Reinforcement Learning

Deep learning [4] is a machine learning method successfully applied in different research fields, like speech recognition, computational vision, object detection. Mnih et al. [5] presented the Deep Q-learning model, a neural network implementation used to aproximate Q-function for Atari games, having surpassed a human expert in some of them. This project applies their ideas to solve the cart-pole problem.

The deep neural network architecture is shown in 3. A rectified linear unit (ReLU) [6] function is applied to both hidden layers. The output layer has no

activation function. The training algorithm is shown in 1.

Algorithm 1: Deep reinforcement learning with experience replay and target updates

Initialize the experience replay memory D
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta' = \theta$
for *training iterations = 1, N* **do**
  for *episode = 1, M* **do**
    s_0 = environment's first state
    for *t = 1, T* **do**
      With probability ϵ select a random action a_t
      otherwise select $a_t = argmax_a Q(s_t, a; \theta)$
      Execute action a_t and observe reward r, state s_{t+1}
      Store (s_t, a_t, r_t, s_{t+1}) in D
      Generate mini-batch sample $(s_j, a_j, r_j, s_{j+1}$ from D

$$y_j = \begin{cases} r_j, \text{if } s_{j+1} \text{ is terminal} \\ r_J + \gamma max_{a'} \hat{Q}(s_{j+1}, a_j; \theta'), \text{otherwise} \end{cases}$$

      Perform a gradient descent on $\big((y_j - Q(s_j, a_j; \theta)\big)^2$ w.r.t. θ
      Every C steps reset $\hat{Q} = Q$
    end
  end
end

4 Methodology

4.1 Experience replay

An usual technique for computing the gradient used in the each weight iteration when updating a neural network is called *mini-batch*: compute the gradient against more than one training example. However, most mini-batch optimization algorithms are based on the assumption of independent and identically distributed data. As the control process is a sequence of a state, action and a new state, repeatedly until its end, learning from these samples would violate this i.i.d. assumption.

To avoid this problem, we implemented the *experience replay* method [7]. Each tuple (s, a, r, s') is cached in a limited-size queue (when the queue is full, old items are removed when new ones are inserted). In every weight update iteration, a mechanism randomly samples a fixed amount of tuples from the queue, and the gradient descent training step is processed using these items as input.

4.2 Periodic target Q-value updates

Aiming at improving the learning stability with neural networks, a separate network is used for generating target values y_i in the Q-learning update [5]. Every C updates, a clone of the network Q is created to obtain a target network

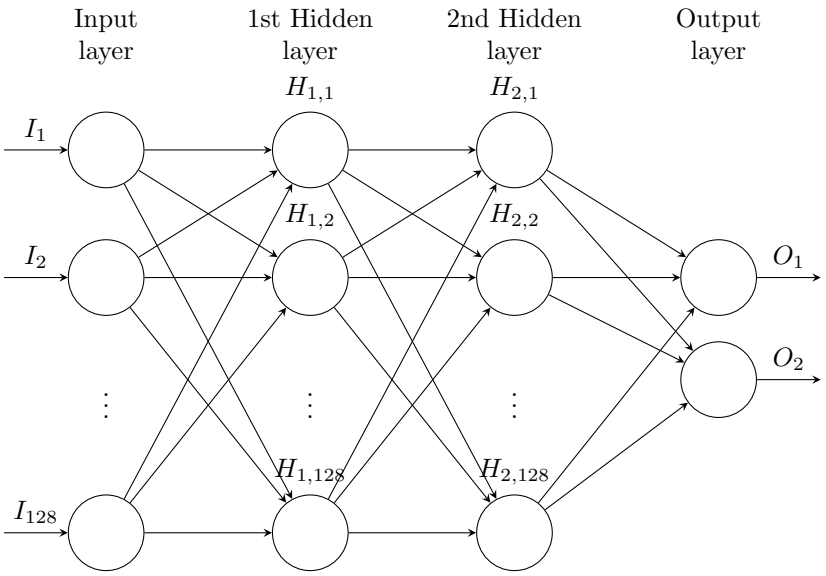


Figure 3: Architecture of the neural network.

$\hat{Q}$ and use $\hat{Q}$ for generating the Q-learning targets y_i for the following C updates to Q.

The main idea behind this method is that in standard Q-learning, an update that increases $Q(s_t, a_t)$ often increases $Q(s_{t+1}, a), \forall a$ and hence also increases the target y_j, leading to oscillations or divergence of the policy.

4.3 Metrics

The main metric is the average reward over 100 consecutive episodes, running an ϵ-greedy policy with $\epsilon = 0$, without updating the weights from the neural network approximation of Q. In the *OpenAI* page for *CartPole-v0*, it is defined that a learner solves the problem if it achieves a score of 195.0 or better.

Another metric is the policy's estimated action-value function Q, which provides an estimate of the discounted reward that can be obtained by following its policy from any given state [8]. Before the training starts, a fixed set of states is collected and the $\max Q$ average for all of them is calculated after each training epoch.

4.4 System specifications

This project was processed in an Intel Core i7-3770K CPU, 16 GB of ram, NVIDIA GeForce GTX 980 Ti GPU, running Linux Mint 17.1, kernel version 3.13.0-37, CUDA 7.5.

5 Results

To better comprehend the results, a baseline was defined and each subsection will explain the effects of changing one or more parameters. As each episode

was capped to 200 iterations, the maximum average reward is 200.

Each training lasted for 100 iterations and each one of them processed 2000 actions.

5.1 Baseline

Graphic (a) from figure 4 shows the result using the baseline parameterization:

- mini-batch size = 10
- maximum iterations per episode = 200
- learning rate = 0.00025
- target Q-value updates = 1 (basically, it does not use the technique)
- $\epsilon = 1.0$
- ϵ update = -0.1, after each training iteration.
- minimum $\epsilon = 0.1$
- discount factor $\gamma = 0.95$
- reward after reaching terminal state = 1

At this configuration, it reaches the 200 average reward.

5.2 Learning rate

Graphic (b) from figure 4 shows that lowering the learning rate to 0.0001 did not gave better results. However, it is noticeable that allowing it to run more episodes probably would be good, as the average reward function had an increasing tendency.

5.3 Target Q-value updates

Freezing the target Q-function gave worse results for $C = 100$ (figure 4 (c)), but slightly better ones for $C = 200$. Figure 4 (d) shows that, after reaching the 200 average reward, the function keeps stabilized.

5.4 Discount factor γ

This parameter showed extreme effects to the model. Graphic (e) from 4 shows no sign of learning at all for $\gamma = 0.5$; both average reward and average q-value functions are flat. However, increasing γ to 0.99 in graphic (f) allowed the model to reach a best average score of 200.0 in just 18 epochs; the average q-value was affected too, stabilizing around 100.

5.5 Terminal state reward

The environment continuously gives a +1 reward while the the pole is within 15 degrees from vertical and the cart does not move more than 2.4 units from the center; no other reward value is returned. So, in an attempt to make the agent learn faster, a negative reward is used when the environment arrives at a terminal state, and it was very effective.

Graphics (a), (b) and (c) from 5 shows the results of setting the reward to -1, -10 and -100, respectively, at terminal states. As this parameter value decreases, the results got better.

The sequence of pairs (average reward, average q-value) were: (195.2, 17.4), (200.0, 20.9), (200.0, 20.6). Setting -10 or -100 generated models that solved the problem.

5.6 Changing more than one parameter

After seeing the effects of each parameter, a new parameterization was used: terminal state reward = -100, $\gamma = 0.99$, target q-value update = 200 and learning rate = 0.0001. Figure 5 (d) shows that, although it reached the average reward of 200.0 and q-value near 100.0, the functions behavior were not stable. The training size could be a problem, so changing the learning rate back to 0.00025 was tried and brought better results, as seen in (e).

5.7 Relaxing the limit of iterations per episode

The last training experiment used a configuration with increased iterations per episode (1000) over 500 training iterations. The other parameters were:

- Discount factor $\gamma = 0.99$
- Reward = -100 if reached a terminal state
- Target Q-value update = 200
- Learning rate = 0.0001

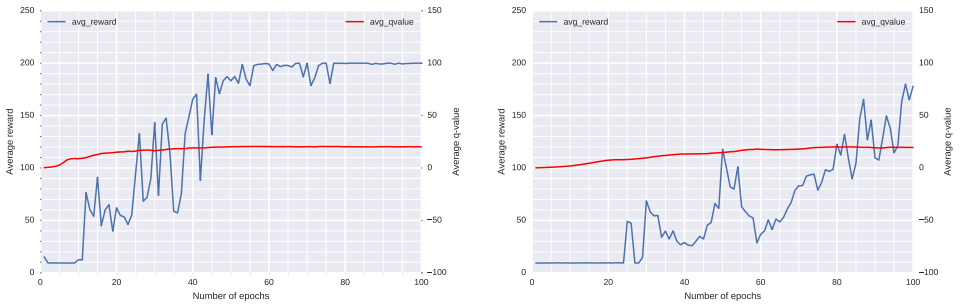
As the number of training iterations was greater than the default, a smaller learning rate was chosen. Figure 6 shows its result.

The trained agent was capable of running a full episode capped at 5 million iterations. Considering that each action is executed over an average of 3 frames and the refresh rate is 60Hz, rendering it would take more than 69 hours. The file **best_model.h5** contains the neural network weights for this model.

6 Conclusion

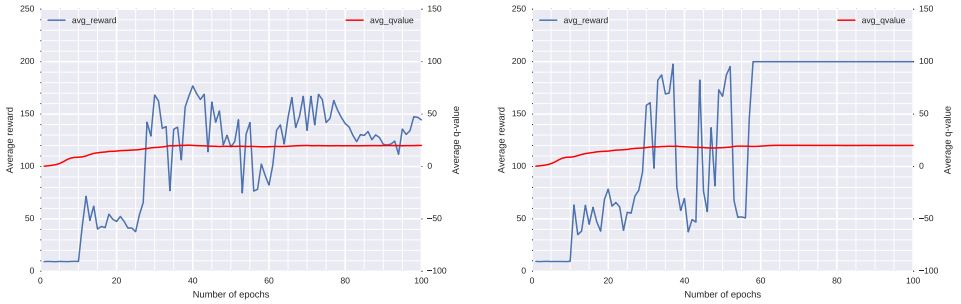
The results found were convincing about the model's ability to solve the cart-pole problem. Even without a negative reward when reaching a terminal state, which would show a failure in controlling the cart, the algorithm achieved the 200.0 average reward, as seen in graphic (f) from figure 4.

One of the most intriguing features of this algorithm lays in the fact that no domain knowledge was needed: the meaning of each one of the 4 dimensions in the input is not needed for the learning process.



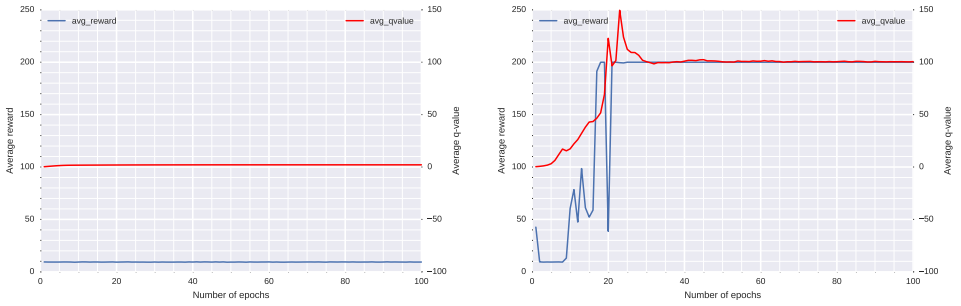
(a)

(b)



(c)

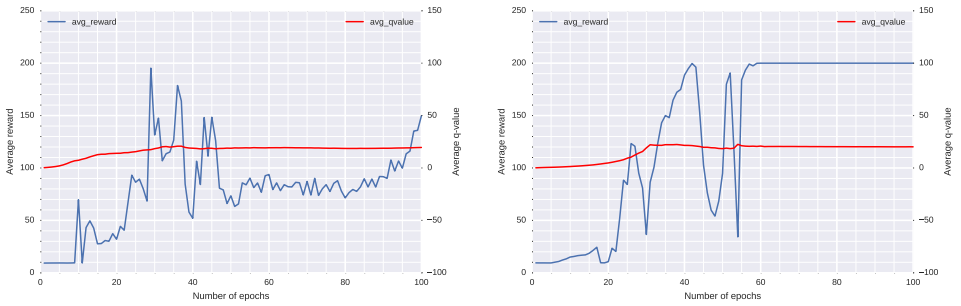
(d)



(e)

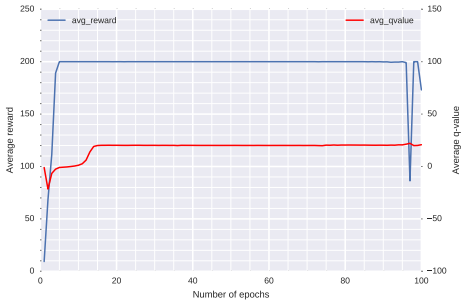
(f)

Figure 4: (a) Base model (b) Learning rate = 0.0001 (c) Target Q-value update = 100 (d) Target Q-value update = 200 (e) Discount factor $\gamma = 0.5$ (f) Discount factor $\gamma = 0.99$.

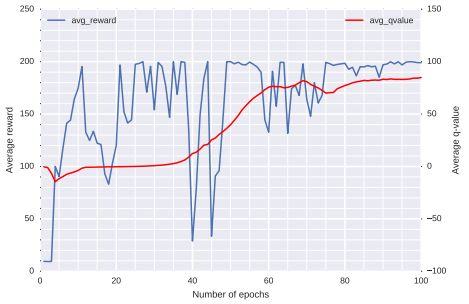


(a)

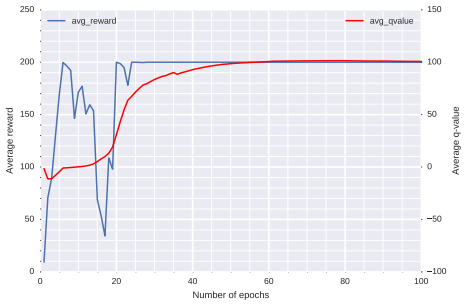
(b)



(c)



(d)



(e)

Figure 5: (a) Reward = -1 at episode end (b) Reward = -10 at episode end (c) Reward = -100 at episode end (d) First combination of parameters (e) Second combination of parameters.

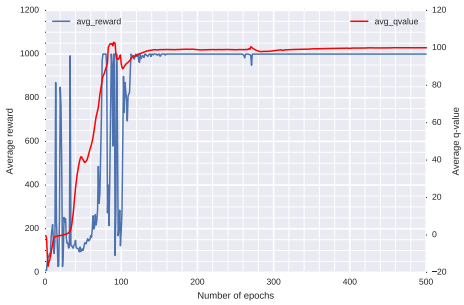


Figure 6: Model based on best parameters and limiting the number of iterations in an episode to 1000. Notice that the value range for both functions is different than that used in figures 4 and 5.

6.1 Reflections

In a broader view, this project was very satisfying, both research and coding wise, but not without problems:

- The main issue: this project is the second attempt to finish the capstone. The first one was a chess AI based on deep learning, very slow and – as I sadly discovered later – a bit difficult to validate. The initial metrics were not good enough and, after reviewing other possibilities, I found it was better trying to solve another problem.
- Insisting on the same path of frustration: even after quitting the first project, I did try to play with a problem that, according to some papers, would take several days – more like two whole weeks – to converge. At least this time I did not spent two and a half months before figuring out it was not a good idea.
- LaTeX generates beautiful reports, but it was hard to use it after not writing any LaTeX-related thing in years.

On the positive side, all the time I spent on the chess project brought me an advantage: I could focus on studying the theory, instead of switching between theory and coding related topics. As I already had experienced the libraries, I had knowledge of what to expect from them, feature and performance wise.

6.2 Improvements

The deep Q-learning implemented was based on [5]. Since its publication, many others researched optimizations for it and new techniques were published, like Double Q-learning [9]: two value functions are learned by randomly updating on of two sets of neural network weights, θ and θ'. A comparison against the old model could be made to see which one is faster to reach the 195.0 average reward and more stable.

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[2] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *EEE Transactions on Systems, Man, and Cybernetics*, 1983.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2012.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 02 2015.

[6] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines,"

[7] T. de Bruin, J. Kober, K. Tyuls, and R. Babuska, "The importance of experience replay database composition in deep reinforcement learning," 2015.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[9] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015.