Deep Q-Learning to play Snake

Daniele Grattarola

August 1, 2016

Abstract

This article describes the application of deep learning and Q-learning to play the famous '90s videogame *Snake.*

I applied deep convolutional learning to approximate the game's action-value function, and used the approximation to control an agent by choosing, at each state of the game, the action with the highest Q-value.

Due to the great number of states of the environment, the strong generalization capabilities of deep neural networks allow learning the Q function in a significantly more compact way than the tabular representation usually used in Q-learning.

I ran different experiments to evaluate the model's performance with respect to different environments (with different reward functions and discount factors) and obtained good results, with the agen being able to constantly increase the reward obtained at each episode, albeit with a strange behavior.

1 Introduction

Snake is a 1976 videogame by Sega which was widely popularized by its inclusion in Nokia phones in the 1990s. In the game, the player maneuvers a line (the snake) in an empty box, with the borders of the box and the line itself being obstacles. The goal of the line is to eat[1] as many apples[2] as possible, under the constraint that each time that the snake eats an apple it grows in length. Apples are randomly placed inside of the box, and the position of an apple remains unchanged until the snake eats it.

The simplicity of the game makes is a perfect candidate to demonstrate reinforcement learning techniques but, since the number of states is very big, it can also be challenging to learn it efficiently.

2 Problem formulation

There exist many implementations of the game with different rules; the one used in this article has the following characteristics:

[1] i.e. make the head of the line collide with

[2] Dots as big as the snake's head

- The box environment is divided in a square grid with a fixed size and the snake can only move in a finite number of positions;
- The player can choose four different actions (*right*, *left*, *up*, *down*) which cause the snake to change its direction accordingly, unless the selected action is a complete inversion of the snake's direction (e.g. choose *up* when the snake is going *down);*
- The snake is divided into discreet contiguous units as big as one slot of the grid: apples fill exactly one unit and each time the snake eats an apple, it grows by one unit;
- There is exactly one apple in the box at any given time, which stays in the same position until the snake eats it.

I modeled the game environment with a discreet Markov decision process (MDP) $M =< S, A, P, R, \gamma, \mu >$ as follows:

- S is the finite set of all possible game states. A state is a tuple

  $$< \text{snake units positions, snake direction, snake length, apple position} >$$

  which does not violate the game constraints.

  For practical purposes, states were represented as two consecutive screenshots of the game similarly to Mnih et al. [4], in order to contain all the necessary information.

  The game ends when the snake collides with itself or the wall, and any such state is a final state of the MDP.

- A is the finite set of actions that the agent can choose:

  $$A = \{right, left, up, down\}$$

- P is the state transition probability matrix: state transitions are deterministic whenever the snake is not eating an apple, and are uniformly distributed over a subset of states when the snake eats an apple[3].

- R is the reward function. For every state-action pair:

  $$R(s,a) = \begin{cases} l, & \text{if the snake does not die} \\ a, & \text{if the snake eats an apple} \\ d, & \text{if the snake dies} \end{cases}$$

  Different values for l, a and d were used to evaluate the model's performance in section 4.

[3]The probabilities are uniformly distributed over all states in which the snake position has changed deterministically and the apple position has changed randomly, with uniform distribution over the empty units.

- γ is the discount factor; as above, I used different values for γ to evaluate the model's performance in section 4.
- μ is the set of initial probabilities of the states and is uniformly distributed over all states in which the snake is 5 units long, has the head in the middle of the box and is vertically aligned. The uniform distribution is due to the random position that the apple can assume (same as when the snake eats an apple).

The goal of this work was to create a model which could efficiently approximate the optimal action-value function Q of the MDP, so that the agent could play the game with a greedy policy:

$$\pi(s) = \underset{a \in A}{argmax}\, Q(s, a)$$

without having to explicitly compute the Q function for each state-action pair.

It is also interesting to notice that the only information that the agent has to learn are the same information that human players have when they play the game; no *artificial* representation of the states is given and the agent has to learn the Q function by *looking* at the game.

3 Methodology

3.1 Deep Q-learning

Due to the exponential size of the game states (bound by $O(3^{|G|})$, where $|G|$ is the number of units in the game field) the traditional lookup-table formulation of the action-value function might be expensive to compute.

In the reinforcement learning community, the approximation of Q in such cases is typically done with a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. A neural network function approximator is typically called a Q-network [4].

Furthermore, recent advances in computer vision have seen the rise of deep learning and convolutional neural networks as very powerful techniques with excellent generalization and adaptation capabilities to different problems.

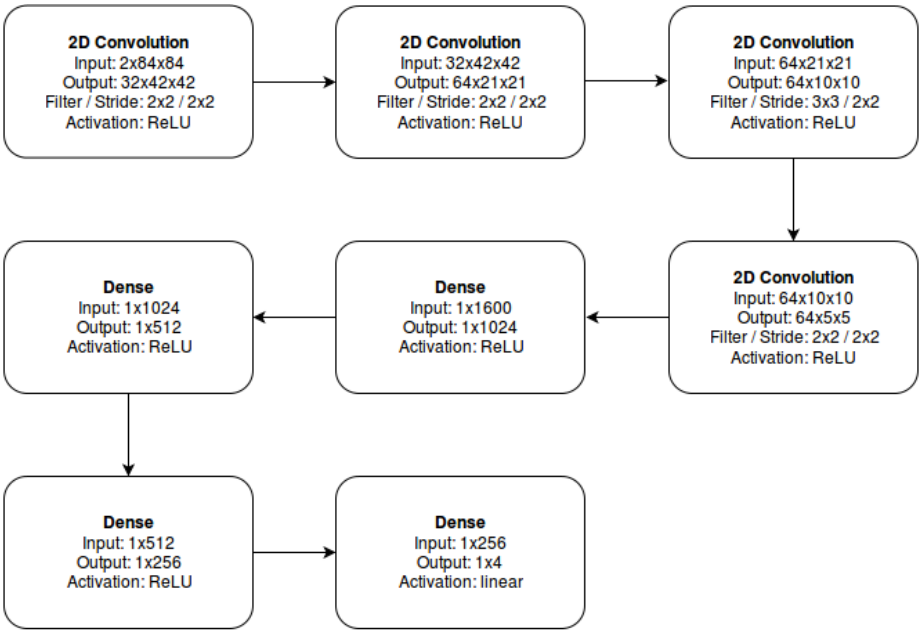
In order to maximize the generalization skills of the model, and make it so that the same model could be applied to a variety of similar problems[4] by simply tuning the model's hyperparameters, I used a deep convolutional neural network as approximator for Q.

The whole procedure of Q-learning with a deep neural network is usually referred to as deep Q-learning.

[4] e.g. other 2d games with finite state space and actions

3.2 Convolutional neural network

The convolutional neural network that I used has the following structure:



There are some differences from the standard CNN architecture, namely:

1. There are no pooling layers; this is done because the main purpose of pooling during subsequent convolutions is to provide translation invariance to the feature extraction, but in this case the exact position of the snake and apple in the environment are important data which cannot be omitted.

2. The activation function of the last layer is linear; this is necessary because the network needs to output big values (so the usual *tanh* is not an option) of varying sign (so *ReLU* is not an option, too).

The input of the network is a tuple of two consecutive screenshots of the game, down-scaled to 84 by 84 pixels and mapped to the 8 bit greyscale color space (0-255), so that the input tensor has a shape of 84x84x2. This ensures that all the elements of the state tuple defined in section 2 are present.

The output of the network is an approximation of the Q function evaluated at the input state over all possible actions and is a 4 dimensional tensor.

The network uses dropout regularization [6] in the fully connected layers and batch normalization [1] between all layers.

3.3 Learning procedure

The learning procedure for the agent is roughly as follows:

Algorithm 1 High level training procedure

```
While not quit:
        While training condition not met:
                Experience episodes
                Add experiences to dataset
        Train deep Q-network
        If finished training:
                quit = True
```

A few observations must be done on this algorithm:

1. During training, the agent picks its actions with an ϵ-greedy policy over the output of the neural network.

   This means that the action that is performed by the agent is either the one proposed by the network (with probability $1 - \epsilon$) or a random one (with probability ϵ).

   The ϵ coefficient is initialized at 1 to maximize exploration (the agent will always pick a random action) and is decreased after every training session (the agent starts to exploit more and more the knowledge learned by the network).

   During the testing phase, ϵ is set to 0 in order to evaluate the exact capabilities of the agent without the random factor.

2. The training set for the network is composed of tuples $< s, a, r, s' >$ [5] where:

   - s is the starting state of the experience
   - a is the action taken by the agent in state s
   - r is the reward for a ($R(s, a)$)
   - s' is the stated reached after taking a

3. Training of the neural network is done by calculating the targets for the network as [3]:

Algorithm 2 Deep Q-Network target calculation

```
Sample random experiences <s, a, r, s'> from the dataset
Calculate the target t for each datapoint:
        if s' is a terminal state then t[a] = r
        else t[a] = r + gamma * max(network_output(s))
```

By doing so, the Q-network will update the Q function towards the optimal Q function similarly to the standard Q-learning update rule:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

1. The optimizer algorithm used in training is the Adam optimizer, which automatically updates its learning rate and momentum and performs well without fine-tuning [2].

4 Experiments

I run many experiments on different MDPs (different reward functions, different discount factors); I chose to report the three most significative ones which are summarized below.

Some technical notes must be taken into consideration:

1. Episodes were forced to terminate after $100 \cdot \textit{snake lenght}$ steps, in order to explicitly terminate those episodes in which the agent is looping indefinetely.

2. The dataset was created by adding <s,a,r,s'> tuples to the experience replay buffer only if the final score of the episode was at least 1 and the episode was at least 10 steps long; this was done to prevent a "suicidal" behavior that the agent demonstrated in the first runs of the algorithm.

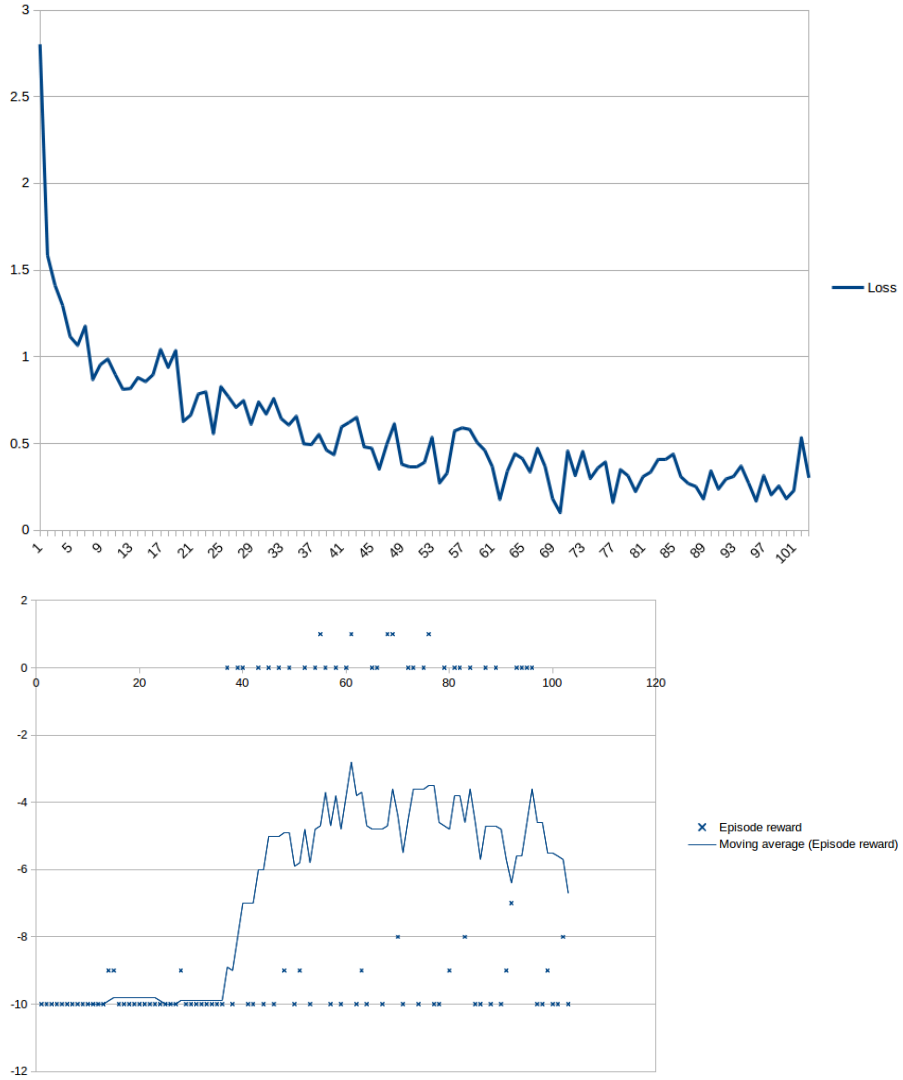
In the following sections, the figures of merit used to evaluate performance will be:

1. Training loss of the Q-network

2. Running mean of the rewards obtained in the testing episodes

4.1 Experiment 1

Apple reward	Death reward	Life reward	Discount factor
$+1$	-10	0	0.95



Notes

It is possible to identify two distinct sets of "Episode rewards" in the second graph: the ones around -9 are the "normal" testing episodes, whereas the one around 0 are episodes in which the snake got stuck in loops, chasing its own tail in order to get as much *life reward* as possible and avoiding the *death reward*.

This can be explained by looking at how the dataset is built: on average, the number of experiences <s,a,r,s'> in which the snake eats an apple is significantly smaller than the number of experiences in which the snake gets a *life reward*.

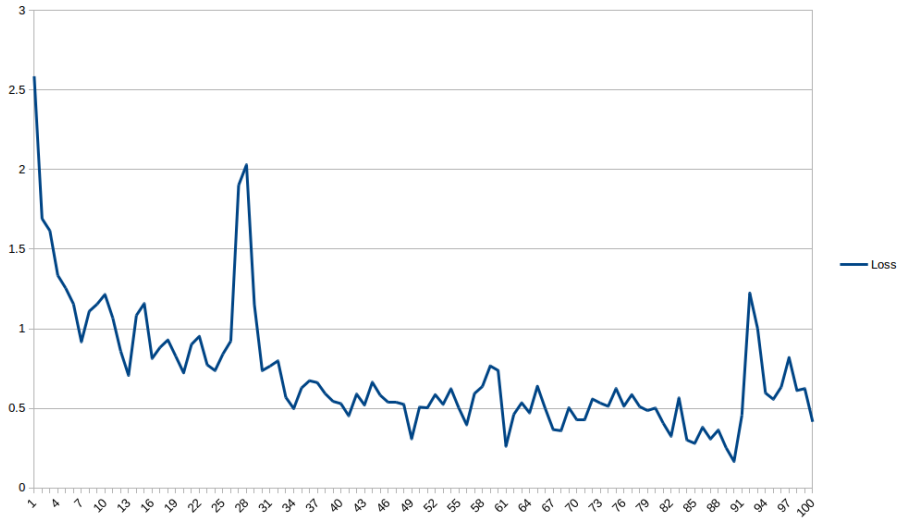
This bias in the dataset is enough to justify the strange behavior of the

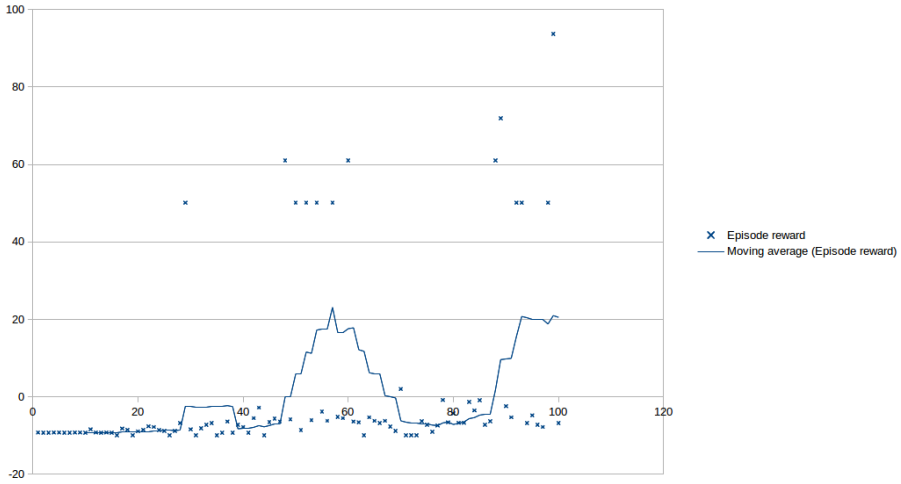
snake.

I chose not to manually fix the bias for both computational reasons (gathering samples took hours) and to stay as close as possible to how humans play the game. It is possible to see that the trend was positive regardless of this problem: "normal" episodes at the end of the experiment had higher average rewards than episodes at the beginning.

4.2 Experiment 2

Apple reward	Death reward	Life reward	Discount factor
+1	−10	0.1	0.95





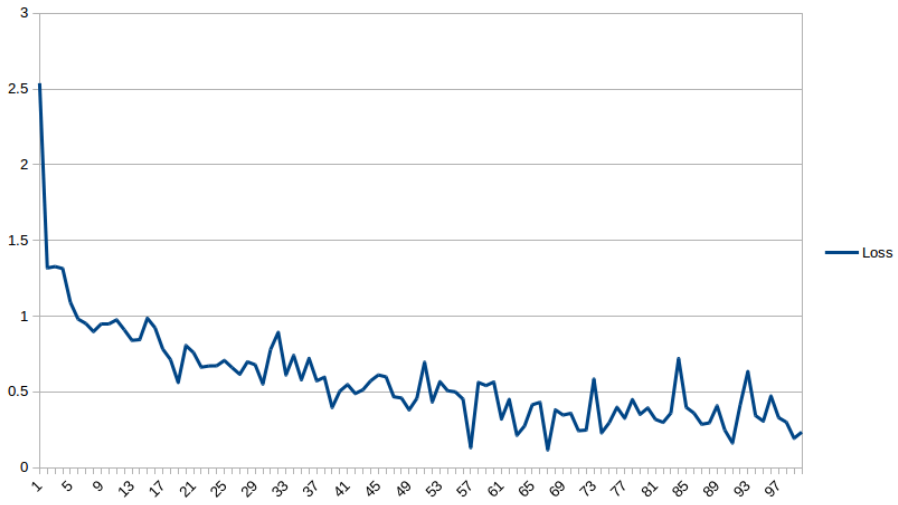
Notes

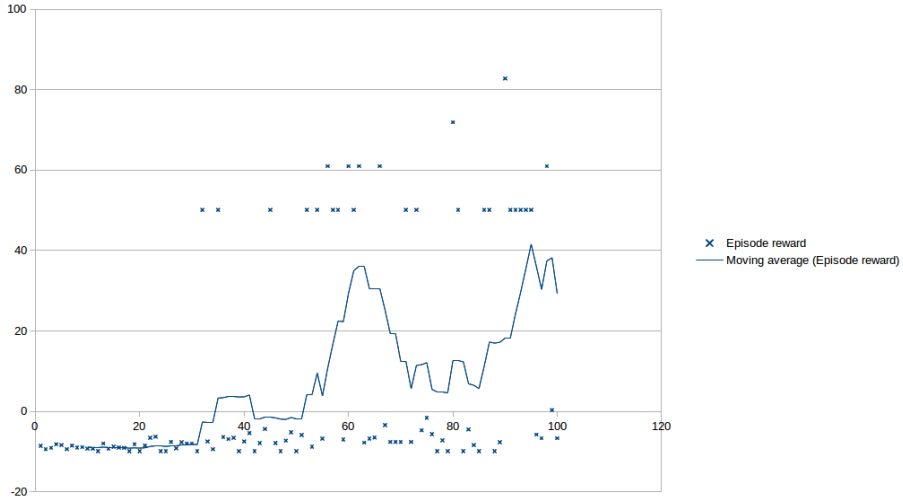
It is possible to see in this case that the average reward dropped in value after the 60th training iteration, and started growing again after a while. This is due to the fact that in that interval the snake never got stuck in infinite loops, thus never getting the high rewards (50 and up). It is still possible to see, however, that the performance on those test episodes was significantly better than before, with rewards getting close and above 0 (meaning that episodes lasted longer and terminated with higher scores).

This shows that the trend would have been positive regardless of the snake getting stuck in loops.

4.3 Experiment 3

Apple reward	Death reward	Life reward	Discount factor
+1	−10	0.1	0.85





Notes

Here, we can notice that the apparent effect of a smaller discount factor was to have more episodes in which the agent got stuck in loops but, since it's not likely that there's a direct correlation between the two phenomena, we might abscribe this to the random initialization of the Q-network or the randomness of the snake's moves due to the ϵ-greedy policy.

In this graph, too, there's a gap due to the agent getting less stuck in loops.

5 Conclusions

The algorithm was run with respect to different MDPs and in each was able to increase the cumulative reward for each episode, but the means through which these results were obtained are peculiar. In each setting, what the agent learnt was that it was more convenient to run in circles forever rather than actively seeking apples.

This type of conservative approach may depend on a number of factors which might not have been taken into account, such as how do humans really percieve the reward function of the game, or which is the real discount factor of the MDP, but ultimately are more likely due to the structure of the dataset and the sampling process.

One could also argue that the problem lies in the amount of exploration performed by the agent but, even by providing explicit adjustments to increase exploration, the results did not change much.

Possible future expansion of this work could be to apply the same learning algorithms to different games and seeing if different results can be obtained.

List of Algorithms

References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[2] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[3] Tambet Matiisen. Demystifying deep reinforcement learning, dec 2015.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

[5] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*, volume 116. Cambridge Univ Press, 1998.

[6] Stefan Wager, Sida Wang, and Percy S Liang. Dropout training as adaptive regularization. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 351–359. Curran Associates, Inc., 2013.