Neural Networks and Deep Learning

Homework III, Simone Trevisan, 1238612

January 23, 2021

1 Deep Q-Learning: quantitative input

The aim of this exercise is to develop a deep Q-learning network, able to solve a simple Atari game imported using OpenAI Gym toolkit using quantitative state parameters.

1.1 Cartpole-v1

Cartpole game consists in preventing the fall of a pole attached to a moving cart (reverse pendulum). The state of the environment are cart position, cart velocity, pole angle and pole angular velocity. The game ends when the pole reaches a critical angle from where it is impossible to recover or when the cart exits from the screen. Possible actions are push the cart right or left. Reward is 1 at each step and the game is considered solved after 500 steps. The input of the network are the state values, the output are the q-values for each possible action.

1.1.1 Layout

This deep Q-learning implementation uses two identical neural networks: the policy network, that is in charge to pick the next move, and the target network that keeps the target fixed allowing the policy to learn a given strategy before moving on with something (hopefully) better.

The network layout is very simple: a 4 length input layer (environment state), two dense layers with 128 neurons and $Tanh()$ activation function. The output are the two q values corresponding to the possible actions. SGD optimizer is used for adaptive learning rate. Loss function is a Huber loss that uses a squared term if the absolute element-wise error falls below a threshold and an L1 term otherwise. This prevents exploding gradients and makes the network less sensitive to outliers. Moreover, the gradient is clipped at 2 to improve stability.

A replay memory is implemented to sample from past experiences independent training data.

1.1.2 Parameters and exploration profiles

Deep Q-learning adds some extra parameters to tune and it is also very hard to keep stable. Since the problem is quite easy to solve with the parameters provided during lab, different exploration profiles are tested to better understand how they impact the learning.

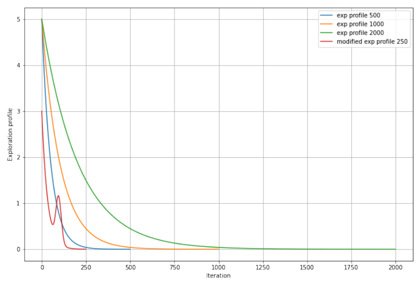
Moreover, a penalty based on cart position is added to reward, in order to push it at the center of the frame. This gives the network some prior knowledge about the environment. Prior knowledge is something that also humans always exploit during tasks: e.g. in this game we do not need to learn physics laws from scratch, they are something that we can already predict since episode 1.

Four different exploration profiles are tested: a 2000 episodes exponential decreasing profile, a 1000 episodes exponential decreasing profile, a 500 episodes exponential decreasing profile and a 250 episodes profile composed by an exponential decreasing function and a Gaussian function that adds a second period for exploration to the learning phase (fig.1a).

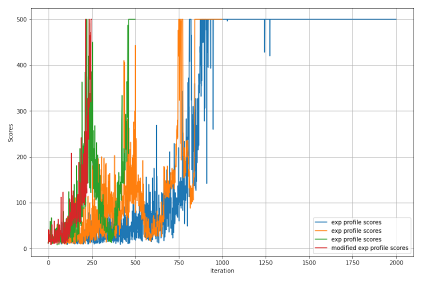
All the profiles are tested with the given parameters but the last one, that required to halve the step before target update and double the memory size. This fastened the learning and allowed the network to exploit older samples.

1.1.3 Results

All the implementations are able to reach convergence. However, they differ in the number of episodes needed (fig.1b). In the longer profile, the network start to learn at episode 750 reaching stability around episode 1000. The 1000-profile performances are quite similar, but it is possible to note that it started to learn earlier but around episode 500 it experienced what is called catastrophic forgetting: the network suddenly forget past information trying to learn new ones. This might suggest that it is possible to reach convergence also after 500 episodes, and so happened with the third profile. However, this time the catastrophic forgetting is even more evident at episode 250. With the these changes to improve stability and exploration, the network is finally able to converge and systematically win the game after 250 episodes.



(a) Exploration profiles.



(b) Training scores with different profiles.

Figure 1

1.2 LunarLander-v2

In the LunarLander environment the goal is to land a spaceship on the lunar soil. State vector has 8 components: horizontal and vertical position, horizontal and vertical velocity, angle and angular velocity, and left and right leg contact. Possible actions are: do nothing, fire main engine, fire left engine, fire right engine. Landing pad is always at coordinates (0,0). Reward for moving from the top of the screen to landing pad and zero speed is about 100/140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. The game is considered solved with a score higher than 200.

1.2.1 Layout and parameters

The network is very similar to the one that solved CartPole with some minor changes on layout and hyperparameters to match the higher complexity of the game. Hidden layers have now 150 and 120 neurons, and they are activated by ReLU function. The exploration profile is an exponentially decaying of 500 episodes. Gamma parameter for the long term reward passes from 0.97 to 0.99 (landing successfully provides many points). Due to the higher complexity, higher memory size to store different states is provided (replay memory size = 100000). Learning rate is decreased to 0.001 and Adam optimizer without weight decay is used to apply momentum and speedup convergence. Target network is updated at every episode and gradient clip is set to 5.

1.2.2 Results

The game requires the network to understand how gravity works to win. And this happens after few episodes, but the spaceship just levitates to avoid crash until the max number of steps is reached, losing a lot of points due to fuel consumption. Then it understand that landing somewhere on the moon soil is a better choice and finally it discover that landing in the right place provides even more points. In figure 2 it is possible to see that it achieve an average score of more than 200 points. However it is not stable and time to time it fails the game. Indeed, on 100 random test trials it achieve an average of 182 points.

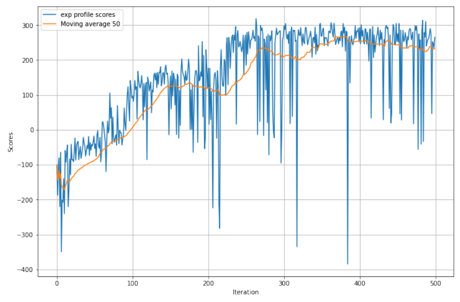


Figure 2: Training scores and moving average

2 Deep Q-Learning: visual input

The aim of this exercise is again to develop a Deep Q-Network able to solve CartPole game. However, this time the input is represented by the frame and not by quantitative values. Thus the network has to learn how to extract valuable information from the screen and choose the proper action.

2.1 Preprocessing

Since the original frame contains much more details than necessary to extract information, it is preprocessed to decrease the network complexity. First, it is converted into gray scale (single channel) and rescaled to 160x240, then the values are normalized and thresholded to obtain a binary image. The action decision should be mostly based on the pole angle, so a square 80x80 portion of the image centered on the cart is used as input. The position of the cart is taken from quantitative state information for simplicity, but it could be easily extracted from the frame by standard computer vision solutions. Again a sort of prior knowledge is given to the network, but as discussed before it could be considered fair since also humans generalize previous experience when face new tasks.

2.2 Layout

The input of the network consists in 4 subsequent 80x80 frames, and the output are the q-values of the possible actions. The first block is a depthwise separable convolutional neural network with 3 layers. Each layer processes two couples of frames separately and then along the entire depth through a pointwise convolution. After each convolution batch normalization and ReLU activation function are applied. The second block is a dense feed forward neural network with 3 hidden linear layers and ReLU activation function. The resulting network has 8.66 millions parameters.

During the developing, different network architecture has been explored: Double Deep Q-Network (DDQN), Double Dueling Deep Q-Network (D3QN), D3QN with Prioritize Experience Replay (D3QN + PER). However my implementations of them are outperformed by a simple DQN with a particular training strategy described below.

2.3 Parameters and training

The exploration profile is again an exponential decay of 2000 episodes. Adam optimizer without weight decay is picked to apply momentum. Huber loss function calculate the error. Gradient clipping is set to 5 and the other hyperparameters are listed below:

- Learning rate: 1e-4
- Long term reward: 0.97
- Replay memory capacity: 50000
- Step before target net update: 2
- Batch size: 64
- Min samples for training: 1000
- Bad state penality: 0

During training, each action is repeated twice. This was actually designed to speedup processing skipping a frame. However, the separable layers encourage the idea of skipping just the action selection, continuing to give all the frames at input. This cancel the effectiveness in terms of computational efficiency (bottleneck is environment rendering), but in practice allows the network to achieve higher score.

2.4 Results

The training scores are shown in figure 3: when the temperature goes down, the network starts to learn how to play and reaches an average score of 300 at episode 1250. Then, the average score drops, even if the network is still able to win in some episodes.

Despite the good results in training, the test performed using the same strategy of repeating the action is not that good (4a), as happened in LunarLander. It is hard to give an explanation to this behaviour, but it might be that when we are not able to reach convergence, having an evolving policy is better than a non optimal fixed one.

Finally the network is tested giving it the ability to control the action at each frame, and it turned out that it outperforms any other solution in this way: the histogram in figure 4b is shown that it reaches a score between 400 and 500 in more than half of the trials (44% of perfect scores). Again, it is hard to interpret neural networks behaviour, but those performances might be the result of having trained the network in an environment where it is forced to be very conservative in the action selection.

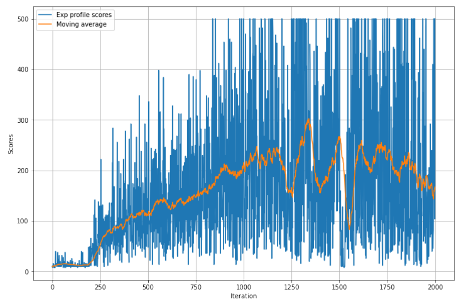
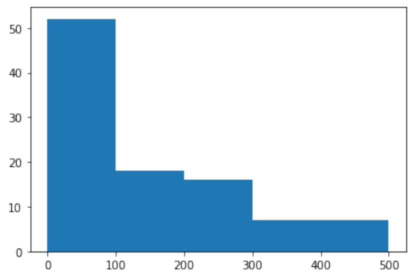
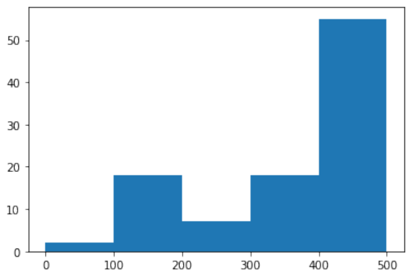


Figure 3: Training scores and moving average



(a) Histogram of scores with double frame action.



(b) Histogram of scores with single frame action.

Figure 4