MI-TNN semestral project

Deep Q Networks in reinforcement learning

Hynek Davídek
davidhyn@fel.cvut.cz

June 17, 2019

1 Problem

This semestral project focuses on using neural networks (DQNs) in reinforcement learning environments. These environments are namely `LunarLander-v2` and `CartPole-v0` provided by OpenAI's Gym (`https://gym.openai.com/`) framework. In the project I compare performance of a randomly behaving agent with performance of three agents using various neural networks.

2 Reinforcement learning and DQNs

In order to achieve the desired behavior of an agent that learns from its mistakes and improves its performance, we need to get more familiar with the concept of Reinforcement Learning (RL).

RL is a general concept that can be simply described with an agent that takes actions in an environment in order to maximize its cumulative reward. The underlying idea is very lifelike, where similarly to the humans in real life, agents in RL algorithms are incentivized with punishments for bad actions and rewards for good ones.

One of the common techniquest to solve RL problems is to use so called Q-learning. In the Q-learning the agent decides its next action based on lookup table $Q(s, a)$ which contains Q-score for a given state-action pair. However in many environments (usually continuous like the two aforementioned) it is not feasible for an agent to has such table. There are two main reasons why not, first is given the state space which is continous the table would not simply fit into the memory. The second one is that even if we fit the table in the memory given the nature of the states most of them would

never be updated. Instead we will use neural networks to approximate and learn the Q-values.

In this project I am using DQN with experience replay algorithm. It uses memory (deque in my case) to save tuple $(state, action, reward, nextstate, done)$ to use for training the neural net. For learning it randomly takes batch of N samples from the memory and creates the (states, q-values) tuple which consists of N states and N q-values to update. The q-values formula is following: $Q(s', a) = Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a')) \cdot (1 - terminal)$ where s' is the next state, a denotes action, α is the learning rate, γ is discount factor, a' is the best predicted action and finally $terminal$ is indicator whether the state is terminal or not. When asked for the next move the DQN either answers with random action (with decaying probability ϵ) from the action space or the best predicted action from the neural net. The random action thing takes place because of the exploration of the state space.

3 Environments

As written in the documentation:

3.1 `CartPole-v0`

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep 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.

3.2 `LunarLander-v2`

Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. 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. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

4 Architectures

All architectures are feedforward. Because the observations of the environment do not consist of any visual information it makes sense not to use the CNNs.
The first architecture (called simple) looks as following:

- input (4 values for `CartPole-v0` and 8 for `LunarLander-v2`)
- fully-connected layer (64 neurons)
- leaky relu activation
- fully-connected layer (64 neurons)
- leaky relu activation
- fully-connected output layer (2 values for `CartPole-v0` and 4 for `LunarLander-v2`)

The second architecture (called medium) looks as following:

- input (4 values for `CartPole-v0` and 8 for `LunarLander-v2`)
- fully-connected layer (150 neurons)
- leaky relu activation
- fully-connected layer (120 neurons)
- leaky relu activation
- fully-connected output layer (2 values for `CartPole-v0` and 4 for `LunarLander-v2`)

The third architecture (called advanced) looks as following:

- input (4 values for `CartPole-v0` and 8 for `LunarLander-v2`)
- fully-connected layer (128 neurons)
- leaky relu activation
- batch normalization
- fully-connected layer (128 neurons)
- leaky relu activation
- dropout ($p = 0.2$)

- fully-connected output layer (2 values for `CartPole-v0` and 4 for `LunarLander-v2`)

It need to be pointed out, that advanced in this context means using of batch normalization and dropout techniques to hopefuly prevent overfitting.

Another thing that needs to be pointed out is the using of Leaky ReLUs instead of standard ReLUs. I decided to use the Leaky version as it is resistant to the dying ReLUs problem which prevents gradient from flowing during the training, which is something we do not want for sure.

All architectures use mean squared error as the loss function. The reason is because we predict function values, so it makes sense to use this loss function. As the optimizer I chose Adam, since it is one of the current state-of-the-art optimizers and because I had poor results with others such as SGD.

5 Training

For `CartPole-v0` the number of training episodes is 1000. For `LunarLander-v2` it is 250. The decision on the training episodes number was based solely on time that it takes to process one episode. Both environments are also bounded to maximum of 200 and 1000 steps respectively to prevent from running for too long (that is default behaviour of OpenAI's gym).

Each test is run with the same initial seed (42) to ensure fair environment generation for all agents.

6 Results

The cummulative results over all training episodes for all architectures and environments can be seen in table 1.

	Average cummulative reward	
	`CartPole-v0`	`LunarLander-v2`
simple	169.9	-123.7
medium	181.7	-96.8
advanced	14.5	-518.3
random	22.8	-178.4

Table 1: Average cummulative reward for different architectures and environments.

Now I will discuss results for each agent, except for the random one,

as there is nothing to be discussed. The random agent was used only as a baseline for comparison with other used methods.

6.1 `CartPole-v0`

As it can be seen from 1 and 1 the Simple DQN Agent performed fairly well in the `CartPole-v0` environment. OpenAI claims that score for successfully solving the task is 194.5 over last 100 episodes. Even though the agent doesn't perform that well it shows rising trend of reward per run with occasional spike to 0.

The medium agent has performed quite well as showed in 2. It showed rising trend as well as the Simple version but also it showed much higher average score so we can claim that this agent solves the task successfuly. Even though there are some drops by the end of the training which may suggest some overfitting most likely.

The Advanced agent performed horrendously. It can be clearly seen in 3 that the agent showed falling trend of scoring so we can suppose that is not learning at all for some reason. Most likely that is not because of the number of neuron in layers but because using batch normalization and dropouts. I am not really sure why this happened as I thought that these techniques will help learning by a lot. It should be also told that it performed even worse than the random agent 4.

6.2 `LunarLander-v2`

First of all it should be told that OpenAI doesn't claim any score to be good enough to solve this environment. But on most discussion forum and Reddit people claim that values above -150 are good enough to be considered as solutions.

The Simple agent performed well enough on this much complicated environment as can be seen in 5. Again it shows rising trend with only some minor score falls and convergence around -120 so we can say that the simple agent sucessfuly solves the task.

As can be seen in 6 the Medium agent performes also quite well (much better than the Simple one). It again shows rising trend so we can expect it to learn better if run for longer time. And again we can suppose that the Medium agent solves the task.

The Advanced version performed worse than the Random agent again. This time it even dropped in score as low as -7000 for one episode. The possible reason has already been discussed above.

7 Conclusion

This project showed the capabilities of using Deep Q Networks (DQN) in two OpenAI environments for reinforcement learning. I proposed three architectures, two which were successful and one which was a disaster, most likely because of my low experience in the Neural Networks field. Overall I am satisfied with the results of the project as I was able to use neural networks for other uses than just image classification as showed in most tutorials online.

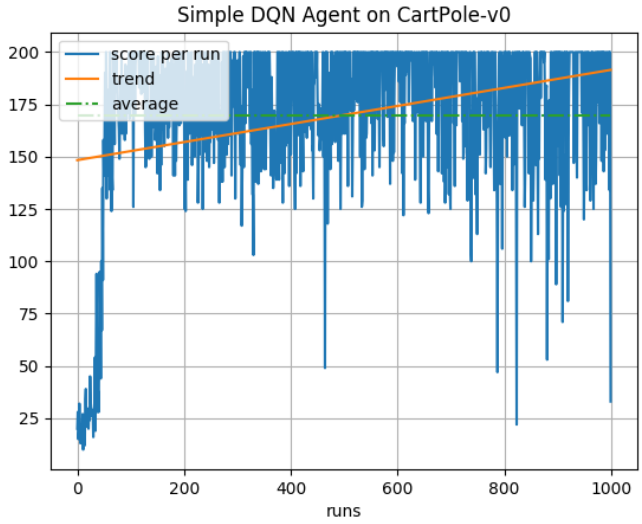

Figure 1: Statistics for Simple DQN Agent on CartPole-v0.

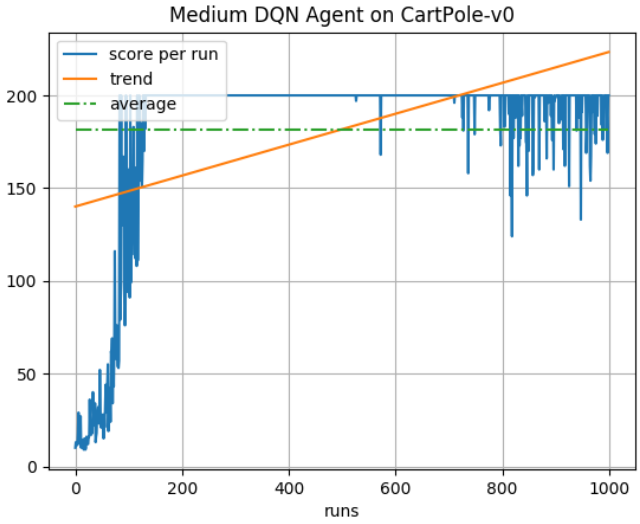

Figure 2: Statistics for Medium DQN Agent on CartPole-v0.

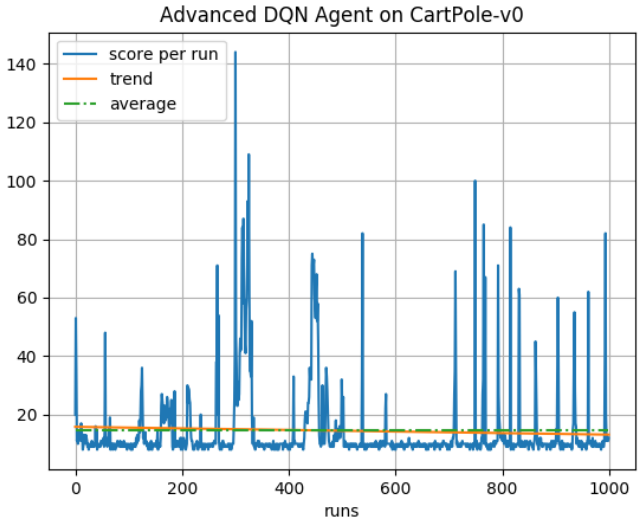

Figure 3: Statistics for Advanced DQN Agent on CartPole-v0.

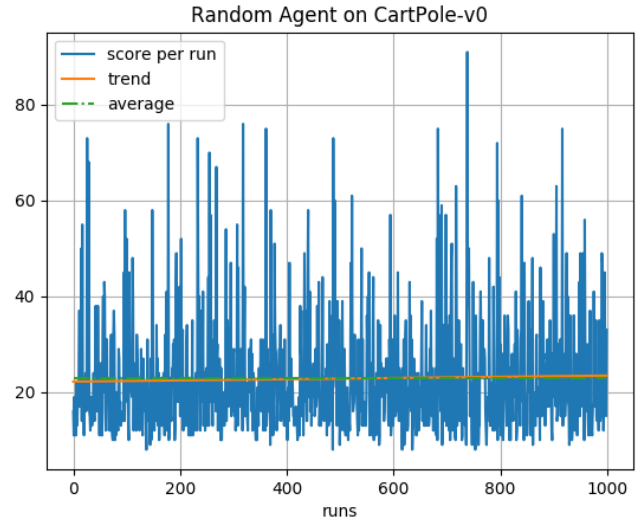

Figure 4: Statistics for Random Agent on CartPole-v0.

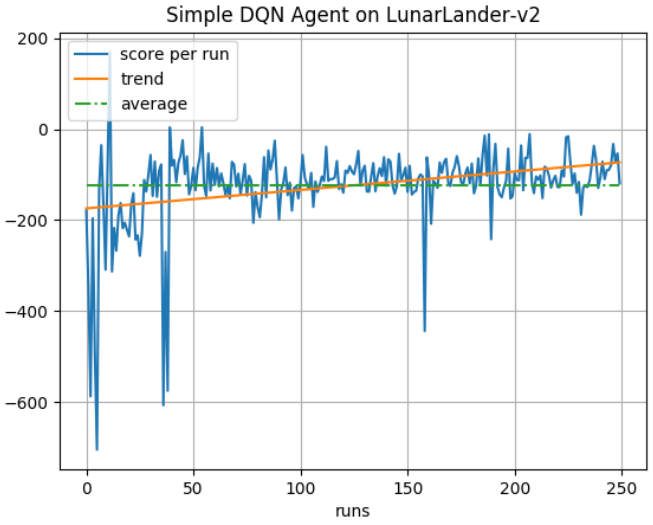

Figure 5: Statistics for Simple DQN Agent on LunarLander-v2.

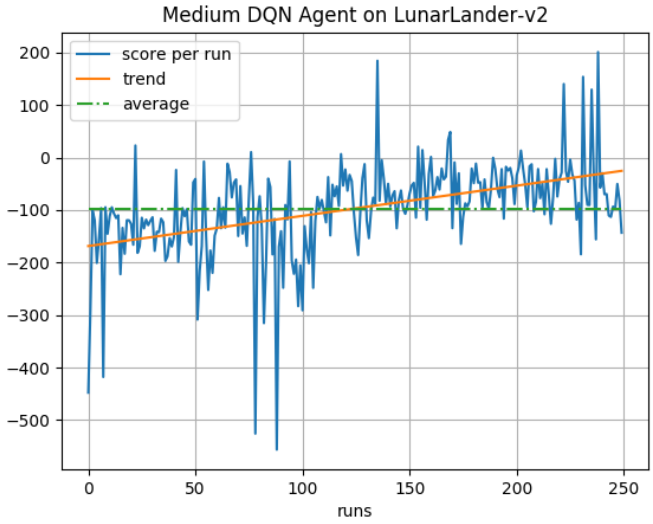

Figure 6: Statistics for Medium DQN Agent on LunarLander-v2.

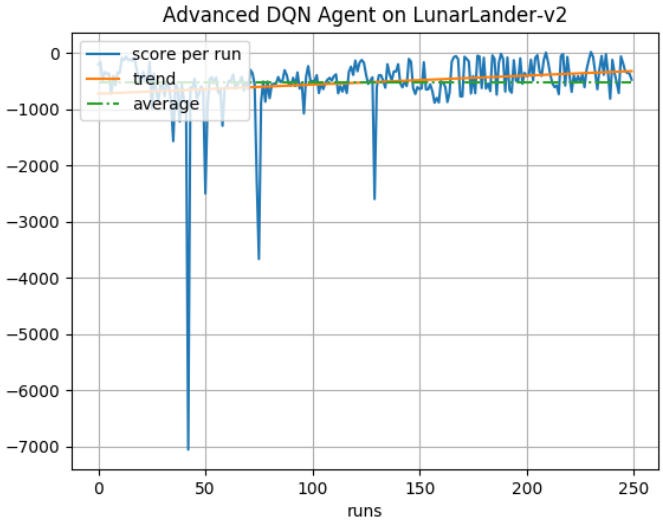

Figure 7: Statistics for Advanced DQN Agent on LunarLander-v2.

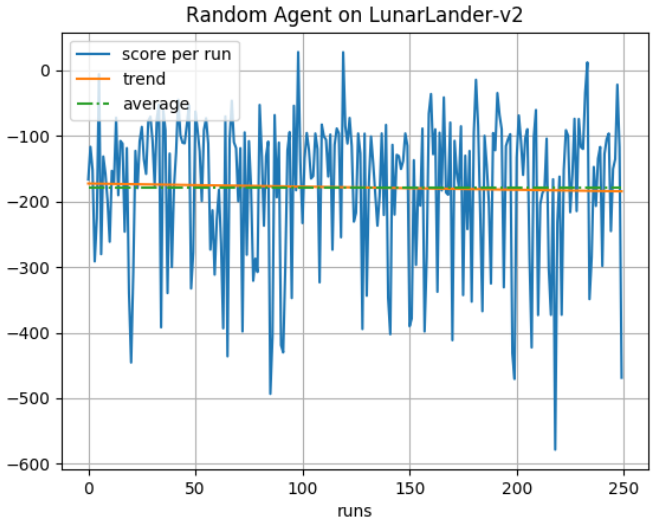

Figure 8: Statistics for Random Agent on LunarLander-v2.