Research Report

Udacity DRLND, Project: Navigation

Andrzej Wodecki

December 25th, 2018

Introduction

This report is a summary of results of the ***Navigation*** **project of Udacity Deep Reinforcement Learning Nanodegree.**

The goal of the project is to **train the Agent to navigate and pick-up bananas** in the Banana environment provided by Unity Environments. After training it should go for yellow bananas, and avoid blue ones. This **episodic** environment is considered **solved** when agents gets an average score of +13 over 100 consecutive episodes.

Implementation strategy, the learning algorithm and a brief discussion of results are described in more details in the next sections of this report.

Implementation

The main steps of the procedure are the following:

1. Initiate the environment
2. Initiate and train the agent
3. Watch the trained agent in action.

The most important part of the process is **agent training**. Following Udacity suggestions I adopted **Vanilla Deep Q-Network (DQN)** implementation provided during classes to 1. Unity Banana Environment and 2. hyperparameter search of learning parameters.

After successful compilation of the Vanilla DQN model, I started with first trials of different sets of learning parameters. After 5-10 experiments, which resulted with the average (over 100 consecutive episodes) score at the level of 5-6 after 2000 training episodes, I decided to run a bit more rigorous grid search over the learning hyparameter space, what required some new, minor changes in the code (see *grid_search* subfolder and *README.dm* file)

First, I made hundreds of runs searching for optimal combination of **learning rate, gamma and tau** parameters. Despite some interested lessons learnt it didn't help me to solve the environment. Thus, I studied in more details different combinations of next two learning parameters: **buffor size** and **UPDATE_EVERY**, with a bit better results, but still with no success. Finally, I've solved the environment **in 1874 episodes** by simply increasing the **max number of time steps during the episode** and **epsilon minimum value with it's decay rate**.

You can watch my smart agent in action here: https://www.youtube.com/watch?v=OS0lHeO5Mn4.

Learning algorithm, hyperparameters and model architecture

Learning algorithm

The goal of the project is to **train the Agent to navigate and pick-up bananas** in the Banana environment provided by Unity Environments.

Agents receives +1 reward for collecting a yellow banana, and -1 for collecting a blue one. After training it should go for yellow bananas, and avoid blue ones.

The state space has 37 dimensions like the agent's velocity, along with ray-based perception of objects around agent's forward direction.

The action space consists of 4 discrete actions:

1. move forward: 0
2. move backward: 1
3. turn left: 3
4. turn right: 4

This is **episodic** environment. It is considered **solved** when agents gets an average score of +13 over 100 consecutive episodes.

For Deep Q-Network algorithm I adopted the Udacity's DQN implementation available here: https://github.com/udacity/deep-reinforcement-learning/tree/master/dqn/solution, with theoretical background described eg. in Mnih *at al*, *Human-level control through deep reinforcement learning* Nature **518**, (2015): 529-533. https://www.nature.com/articles/nature14236.

The most important part of the training is implemented in the `dqn` function. For each episode:

1. Environment is reset (with `train_mode` parameter set to `True`), it's `state` is presented to the agent and `score` parameter set to `0`
2. Next, every time step:
    1. agent chooses the action

    2. this action is "presented" to the environment
    3. environment returns `next_state, reward` and `done` information.
    4. agents absorbs this parameters to update it's Q-network (`step` method)
    5. `state` and `score` parameters are updated, and appropriate average scores calculated.
3. Overall `dqn` function returns the final `score` list.

A precise description of the Deep Q-Network algorithm is presented eg. in the paper by Mnih *et al* cited above. In short, it consists of the following steps:

1. sample a batch of experiences (`BATCH_SIZE` parameter) from the **replay memory**
2. for each experience, select the next action for the next state basing on the **online** Q-network
3. compute **target** Q-values for the current state-action pair as the sum of the rewar and the Q-value obtained from the **target** Q-network
4. identify the **expected** Q-values from the online Q-network
5. use the difference between target and expected Q-values to calculate the **temporal difference errors** and compute **the mean squared loss error** (`mse_loss` method).
6. update Q-networks (online and target) through backpropagation.

Hyperparameters

Below please find the most important laerning hyperparameters, where **value** presents the of the algorithm which solved the environment (parameters of the Deep Q-network learning are named with capital letters).

Hyperparameter	Value	Description
n_episodes	2000	max number of the episodes in the training
max_t	1900	max number of time steps in the training episode
eps_start	1.0	start value of the epsilon parameter (epsilon-greedy strategy)
eps_end	0.01	end value of the epsilon parameter (epsilon-greedy strategy)
eps_decay	0.999	decrease of the epsilon parameter (epsilon-greedy strategy
BUFFER_SIZE	10000	replay memory size
BATCH_SIZE	64	minibatch replay buffer size
GAMMA	0.99	discount factor (for future rewards)
TAU	0.002575	soft update of the target Q-network parameter
LR	0.00009	learning rate
UPDATE_EVERY	8	the frequency of updating the online Q-network

Model architecture

The Q-network implemented as a part of the agent used (see `model.py`):

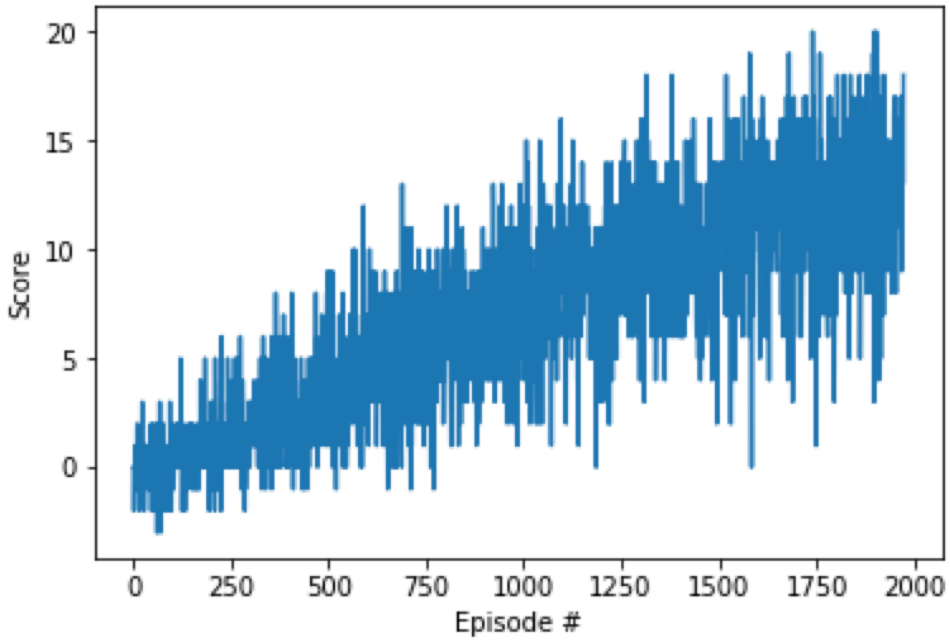
1. Input layer with 37 input nodes (which is a state space size)
2. 2 hidden layers, 64 nodes each and `RELU` activation function
3. Output node with 4 nodes (action space size).

Results

My agent, with hypee parameters set as in the table above, was able to achieve the avarage reward >13 over 100 consecutive episodes in 1874 training episodes.

```
Episode 100     Average Score: -0.08
Episode 200     Average Score: 0.572
Episode 300     Average Score: 1.38
Episode 400     Average Score: 2.35
Episode 500     Average Score: 2.79
Episode 600     Average Score: 3.98
Episode 700     Average Score: 4.81
Episode 800     Average Score: 5.54
Episode 900     Average Score: 6.49
Episode 1000    Average Score: 7.42
Episode 1100    Average Score: 7.79
Episode 1200    Average Score: 8.81
Episode 1300    Average Score: 8.94
Episode 1400    Average Score: 9.92
Episode 1500    Average Score: 9.53
Episode 1600    Average Score: 10.78
Episode 1700    Average Score: 11.70
Episode 1800    Average Score: 11.85
Episode 1900    Average Score: 12.37
Episode 1974    Average Score: 13.02
Environment solved in 1874 episodes!    Average Score: 13.02

<Figure size 432x288 with 1 Axes>
```



But the path to this success was very painful...

Grid search #1: LR (learning rate), GAMMA and TAU

As mentioned in the introduction, I made hundreds of runs searching for optimal combination of **learning rate, gamma and tau** parameters. **To make computations faster, I ended each training session after 1000 episodes,** just to get the comparison of the hyperparameter sets efficiency.

The raw data of results can be found in `grid_search/1. lrate gamma tau.txt`, and are summarized the Fig. 1 below:

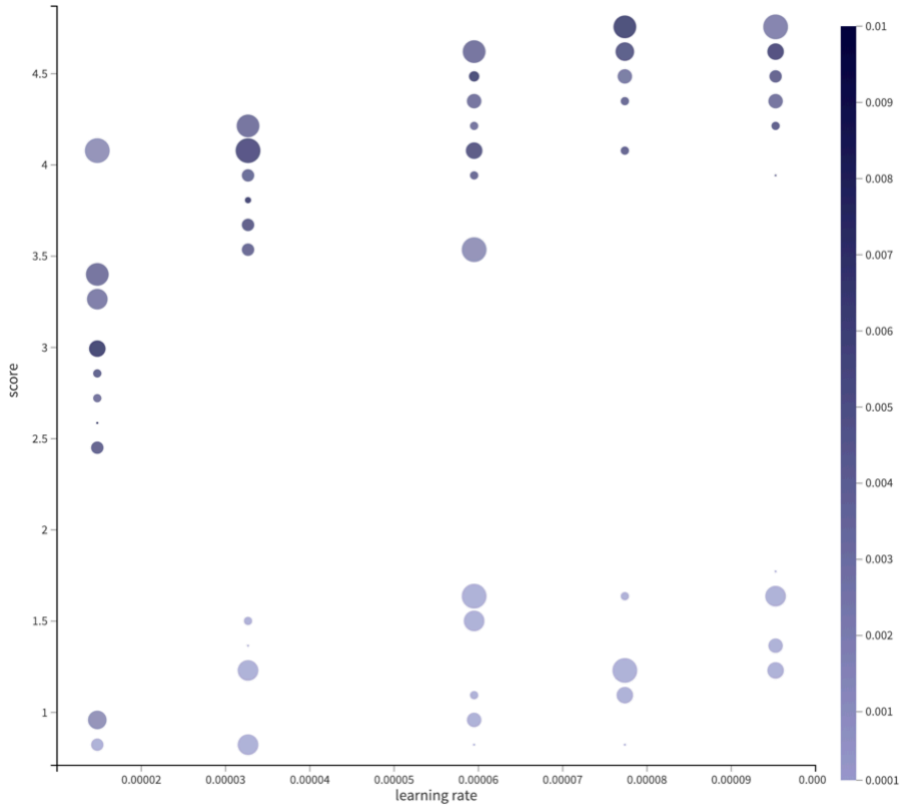


Fig. 1. Average score rate after 1000 episodes as a function of "LEARNING RATE" (X axis), "GAMMA" (dot size) and "TAU" (dot color) parameters. Source: own elaboration.

The parameters were scanned in the following ranges:

1. **LR:** from 1e-5 to 1e-4, 5 points
2. **GAMMA:** from 0.90 to 0.99, 5 points
3. **TAU:** 1e-4 to 1e-2, 5 points

As can be seen on this synthetic plot, the best results were achieved with:

1. the **learning rate** at the level of 8e-5, with slightly visible decrease for it's larger values (what was partially confirmed in my next experiments)
2. **GAMMA** at the level of 0.99
3. **TAU** close to 0.01.

Experiment with the best performing set of these parameters resulted with the average reward at the level of 7 after 2000 episodes, still very far from the target.

Grid search #2: LR (learning rate), GAMMA and TAU

Next, I studied in more details different combinations of next two learning parameters: **buffor size** and **UPDATE_EVERY**. Results were a bit better results, but still with no success (see Fig 2. below).

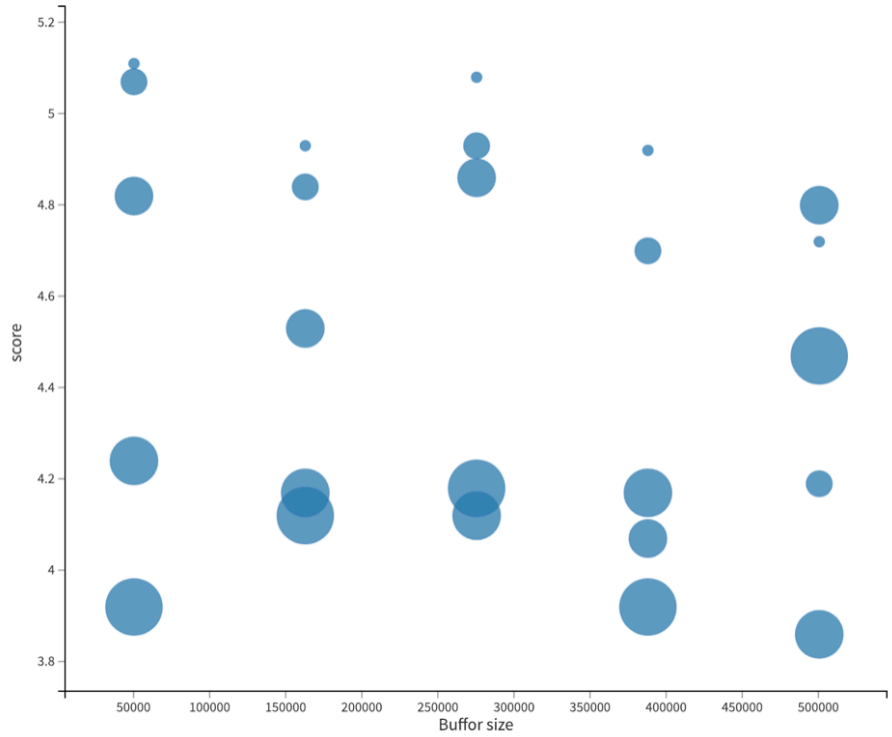


Fig. 2. Average score after 1000 episodes as a function of "BUFFOR SIZE" (X axis) and "UPDATE EVERY" (dot size) parameters. Source: own elaboration.

The parameters were scanned in the following ranges:

1. **BUFFOR_SIZE:** from 5e-4 to 5e-5, 5 points
2. **UPDATE_EVERY:** from 2 to 10, 5 points

As can be seen on this synthetic plot, the best results were achieved with:

1. the **BUFFOR_SIZE** at the level of 50000 (5e-4), interestingly: the smaller, the better
2. the **UPDATE_EVERY** at the level of 2, also with small sizes as more efficient.

But, similarly to Grid Search #1, experiment with the best performing set of these parameters resulted with the average reward at the level of 8-9 after 2000 episodes, still very far from the target.

The final trick: let the agent learn a bit longer...

Finally, I've solved the environment **in 1874 episodes** by simply increasing the **max number of time steps during the episode** and **epsilon minimum value with it's decay rate**. Compared to my previous analysis this solution seems to be primitive, but worked out very well. I made app. 20 different runs, observing different behaviours specifically with relation to **learning rate**: with relatively high values it resulted with quite dynamic increase of the average reward for first 1000 episodes, but then quite flat and/or unstable behavior in episodes from 1000 to 2000. And opposite: smaller learning rates gave me a bit weaker results in early phases of training, but quite steady growth over first 2000 training episodes.

Ideas for the future improvements

For the future improvements I recommend to focus mostly on the new learning algorithms rather than fine-tune the hyper parameters of Vanilla DQN. Specifically, the most promising may be:

1. (recommended by Udacity):
    1. prioritized replay
    2. double DQN
    3. duelling DQN
2. The set of different DQN "flavors" presented eg. here: https://github.com/higgsfield/RL-Adventure, including the RAINBOW model.

Conclusions

The most important conclusions and my personal take-aways from this project:

1. the "training procedure" hyper parameters (like `max_t` or `epsilon`) proved to be more important for the final solution than "learning algorithm" ones
2. watching (and contemplating) the behaviour of scores for different combinations of hyper-parameters gave me a real feeling of their role
3. I guess much more can be obtained from implementing new learning architectures rather than fine tuning learning hyper parameters.