Return to Classroom

Navigation

REVIEW | CODE REVIEW 3 | HISTORY

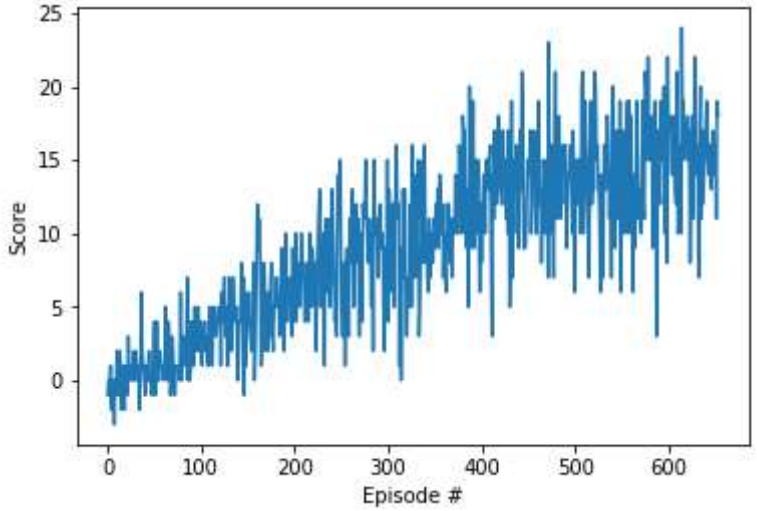
Meets Specifications

Congratulations

Great submission!

You have implemented and trained the agent successfully.

```
Episode 100     Average Score: 1.15
Episode 200     Average Score: 4.49
Episode 300     Average Score: 8.09
Episode 400     Average Score: 10.28
Episode 500     Average Score: 13.35
Episode 600     Average Score: 14.18
Episode 654     Average Score: 15.05
Environment solved in 554 episodes!     Average Score: 15.05
```



All functions were implemented correctly and the Deep Q-learning algorithm seems to work quite well.

Suggestions to make your project even better!

- Include a GIF and/or link to a YouTube video of your trained agent!
- Write a blog post explaining the project and your implementation!

- For an extra challenge, try to train an agent from raw pixels! In case you get stuck, take a look at this github repository and this other one

Training Code

The repository (or zip file) includes functional, well-documented, and organized code for training the agent.

You implemented a Deep Q-learning algorithm, a very effective reinforcement learning algorithm.

Your code was functional, well-documented, and organized for training the agent.

Suggested reading:

- Google Python Style Guide
- PEP 8 — the Style Guide for Python Code
- Python Best Practices
- Clean Code Summary

The code is written in PyTorch and Python 3.

The code was written in PyTorch and Python 3.

Suggested reading about the discussion of what machine learning framework should be used.

- TensorFlow vs. Pytorch.
- Tensorflow or PyTorch : The force is strong with which one?
- What is the best programming language for Machine Learning?
- Deep Learning Frameworks Comparison – Tensorflow, PyTorch, Keras, MXNet, The Microsoft Cognitive Toolkit, Caffe, Deeplearning4j, Chainer.

The submission includes the saved model weights of the successful agent.

You included the saved model weights of the successful agent.

Suggested reading:

- Saving and Loading Models
- Best way to save a trained model in PyTorch?

README

The GitHub (or zip file) submission includes a `README.md` file in the root of the repository.

Well done! A detailed README file has been provided and is

Well done! A detailed README file has been provided and is present in the repository.

The README.md file is important for many reasons

- It is an industry standard practice and helps to make the repository look professional.
- It make easy for the general audience to reuse the code.
- A README file shows:
  - Why the project is useful,
  - What to do with the project, and
  - How to use it.

That's what a README is for.

I recommend you to check

- This awesome template to make good README.md.
- This article: Make a README - Because no one can read your mind (yet).

The README describes the the project environment details (i.e., the state and action spaces, and when the environment is considered solved).

Good work with environment details.

The README described all the project environment details.

Suggested description of the project environment details

- **Environment**: How is it like?

- **Agent** and its **actions**: When is it considered resolved?; What are the possible actions the agent can take?
- **State space**: Is it continuous or discrete?
- **Reward Function**: How is the agent rewarded?

- **Task**: What is its task?; Is the task episodic or not?

The README has instructions for installing dependencies or downloading needed files.

The README described all instructions for installing dependencies or downloading needed files in Getting Started item.

The README must describe all instructions for

- Installing dependencies
- Downloading needed files in Getting Started instructions item.

The README describes how to run the code in the repository, to train the agent. For additional resources on creating READMEs or using Markdown, see here and here.

Nice job!

The README contains the instructions to run the code in the repository.

I recommend you to check the Mastering Markdown, there are great tips about how to use Markdown

Report

The submission includes a file in the root of the GitHub repository or zip file (one of `Report.md`, `Report.ipynb`, or `Report.pdf`) that provides a description of the implementation.

All required files were included

You included the report of the project with the description of the implementation. Well done!

Report content guide:

- Description of the learning algorithm
- The hyperparameters used.
- The model architecture.
- A plot showing the increase in average reward as the agent learns.
- The weakness found in the algorithm and how to improve it.

The report clearly describes the learning algorithm, along with the chosen hyperparameters. It also describes the model architectures for any neural networks.

Nice job!

The report described the learning algorithm, the chosen hyperparameters and the model architectures.

Learning algorithm

The main idea behind the algorithm used in this project is using the nonlinear function performed by Deep Neural Network with 2 hidden layers of 64 units with ReLU (Rectified Linear Units) function as activation to estimate the Q (state-action value) function in Reinforcement Learning.

There are 2 novel points in this algorithm to take care about:

- *Experience Replay*
    - At first, the algorithm starts with initializing the memory D to store N transitions, which is randomized and updated every step.

      The newly randomized action in the transition (state+action ⇒ reward+new_state) is selected with the continuingly updated policy (this can be ϵ-greedy or greedy policy).

      The memory D is then used as the labeled data in training the Deep Neural Network to estimate the function, but this is only within a step, after that step, the memory would be updated.
- *Target Network*
    - The main idea behind this target network is to prevent the Deep Neural Network (which is the core of deep Q-learning) to diverge during the training session.

      This target network is used to estimate the future reward (from step ***j+1***). Plus with the reward ***r*** at step ***j***, it give the estimate Q-value for the action chosen by the policy ***(y_j)***. This Q-value is then passed into the loss function of the Deep Neural Network, the effort to minimize the mean-squared error loss would then optimize the performance of the deep network in estimating Q-values.

      This network is updated every step. In other word, if we consider each time the parameters in Deep Neural Network update it produce a new network and then we store all those networks into a list, the current network is at index ***i***, then the index of current target network is ***i-1***.

Hyperparameters:

Hyperparameter	Value
Batch size	64
Buffer size	1e5
Discount factor	0.99
Soft update of target parameters	0.001
Learning rate	0.0005
Update interval (after every steps)	4
maximum number of episodes	2000
maximum number of timesteps per episode	1000
starting value of epsilon	1.0
minimum value of epsilon	0.01
epsilon decay	0.995

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

Model architecture:

Layer	number of units	Activation function
Input layer	37	
1^{st} hidden layer	64	ReLU

2^{nd} hidden layer	64	ReLU
Output layer	4	

```python
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

A plot of rewards per episode is included to illustrate that the agent is able to receive an average reward (over 100 episodes) of at least +13. The submission reports the number of episodes needed to solve the environment.

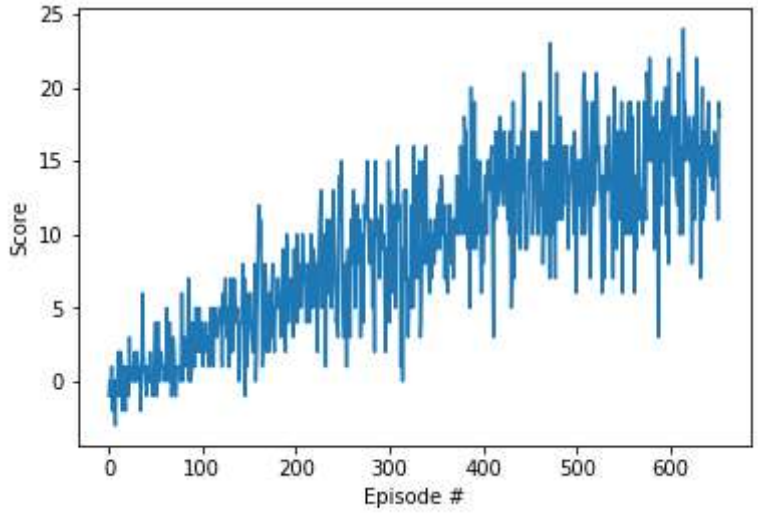
Nice work!

The agent seems to perform very well! The agent was able to achieve an average score of +13 over last 100 episodes!

The report informed the number of episodes needed to solve the environment

__Note:__ The agent only stop learning when its average score over 100 recently continuous episodes reaches 15.0.
This is also a challenge for my designed agent.

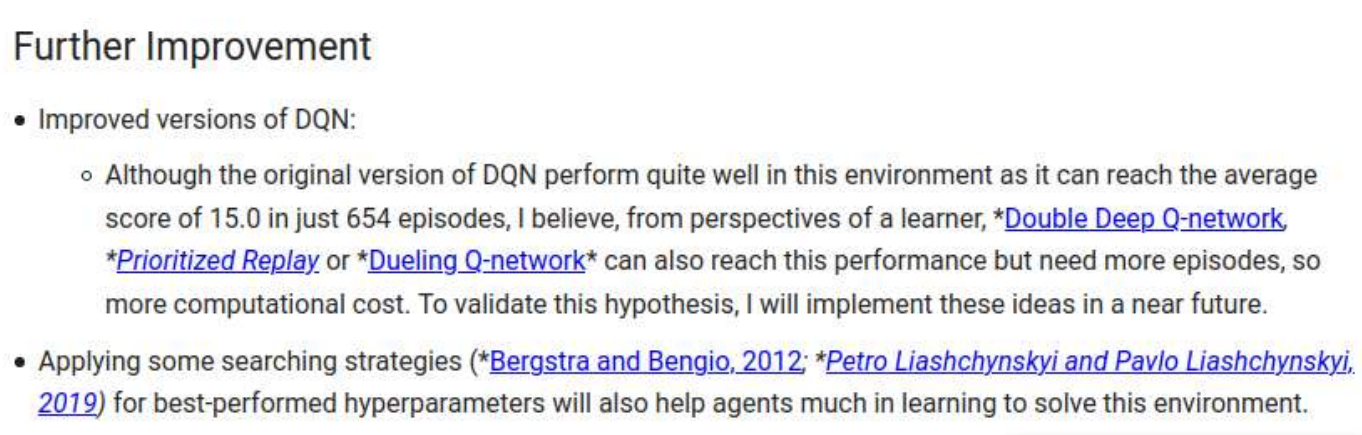
The report also included a plot of rewards per episode.



The submission has concrete future ideas for improving the agent's performance.

The ideas you suggested to improve agent performance were

The ideas you suggested to improve agent performance were very interesting



Further Improvement

- Improved versions of DQN:
  - Although the original version of DQN perform quite well in this environment as it can reach the average score of 15.0 in just 654 episodes, I believe, from perspectives of a learner, *Double Deep Q-network, *Prioritized Replay or *Dueling Q-network* can also reach this performance but need more episodes, so more computational cost. To validate this hypothesis, I will implement these ideas in a near future.
- Applying some searching strategies (*Bergstra and Bengio, 2012; *Petro Liashchynskyi and Pavlo Liashchynskyi, 2019) for best-performed hyperparameters will also help agents much in learning to solve this environment.

More about Prioritized Experience Replay (PER):

- Prioritized Experience Replay
- Implementation of Prioritized Experience Replay
- Let's make a DQN: Double Learning and Prioritized Experience Replay

Prioritized Experience Replay (PER) helps to improve the performance and also helps to significantly reduce the training time, using Sum Tree, a special data structure. Take a look at this implementation

However, in this project using Prioritized Experience Replay (PER) does not make a significant difference.

I agree that you could spend some more time working on hyper-parameters

tuning and optimizing the neural network structure.

- You can almost always obtain better parameter tuning.
- When you do, you can even reduce the number of layers needed.
    - Simpler architectures need less testing time.
- Therefore, they are faster, which is better for end-users.

Take a look at the following resources to get familiar with the Rainbow algorithm:

- Rainbow: Combining Improvements in Deep Reinforcement Learning
- Conquering OpenAI Retro Contest 2: Demystifying Rainbow Baseline

Reinforcement learning algorithms are really hard to make work. Take a look at this article: Deep Reinforcement Learning Doesn't Work Yet.

I hope this helps.

Keep it up.

Stay safe.

DOWNLOAD PROJECT

3 CODE REVIEW COMMENTS ›

RETURN TO PATH