Deep Q-Networks – Navigation Project

Introduction

About Reinforcement Learning

RL is the science behind self-taught software agents who explore a previously unknown world and eventually excel in that world. It is incredible how you can introduce an agent to a hostile and utterly unknown environment, where the agent only knows what available actions it can perform, but nothing more, and through trial and error discover policies that improve with each consecutive move, reaching super-human performance.

About this project

In this document we will present an early approach to DQNs to play a simple game using a UnityEnvironment that simulates a world with yellow and blue bananas. Our agent's task is to understand the dynamics of this world, by collecting yellow bananas that give +1 Reward as fast as possible, while avoiding blue bananas that give -1 Reward.

Without a DQN, we can watch our agent using the equiprobable policy. During that phase we will see that the agent randomly selects actions, making it truly useless.

Important Note: *The content on this report totally reflects the code found at the* ***`Navigation.ipynb`*** *file which also contains as comments all that is contained in this very report.*

So if you want to see implementation details on anything we refer here, please check the ***`Navigation.ipynb`*** *file instead.*

The Unity environment

By running our notebook (*`Navigation.ipynb`*) we first import the `UnityEnvironment` and then initialize it by the downloaded environment for the bananas (check the README.md to see how to obtain it).

```
env = UnityEnvironment(file_name="Banana_Windows_x86_64/Banana.exe")
```

Now we have a link between our notebook and the actual Unity Environment. We can now start interacting with the environment and get feedback by our interactions.

Brains

Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. Here we check for the first brain available, and set it as the default brain we will be controlling from Python.

Taking Actions at the Environment

The `play_game` function takes a `use_DQN` boolean parameter that by default is set to `False`.

Here we make use of the Python API to control the agent and receive feedback from the environment. Once this function is called, you will watch the agent's performance.

If `use_DQN` is set to `False`, we will select random actions at each time step, otherwise we will use the Deep-QNetwork to pick the best action based on the DQN's approximation over the environment's state.

1. Defining the Neural Network Architecture

Well, in the heart of a Deep Q-Learning system beats the Deep Neural network model that will be used to approximate action selection for any given state.

Our neural network will be trained using regression approach, however we will use it as a classifier among the available action set based on the state input. This means that the input size will be as large as our state size (37) and the output size will be as large our action set size (4).

How it turns that a regression model will be used as a classifier? Well, we have multiple outputs. Each output measures over the same result, the potential total reward. Thus, since we have multiple outputs, all of which measure the same result, the output with the highest score beats the others.

Well, this sounds like classification, where the highest label prevails, and indeed it is. The action to select can be considered as our potential label. As a good refinement we can even use softmax at the final layer if we like, as our outcome won't change, since we will still be considering as the topmost action the one with the highest overall score (no matter if that score is value or probability, we still care about the highest).

We will be defining two identical neural networks (local and target). These neural networks should be clones. They should have the same architecture and same initial weights (θ). We define a seed so that both the online and the target networks will be initialized equally with a random weight set (θ).

It appears that three fully connected layers of increasing size contributed to faster training, by consuming less episodes to efficiently approximate the best actions for any state input.

2 The Agent

The system will utilize a Deep Q-Network that take advantage of the "Experience Replay" and the "Fixed Q-Targets" optimizations.

2.1. - The Params

After several configurations test, it appears that the following configurations contribute to a faster training, using less episodes to come up with a policy that meets the needs of this specific Reinforcement Learning task.

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 5        # how often to update the network
```

2.2. - Defining the Replay Buffer

To take advantage of the Experience Replay optimization we need to create a buffer that will hold the SARS experience tuples in a memory double-ended queue.

2.2.1. - `__init__` constructor

In the initialization section we will create

- a double-ended queue with the buffer_size provided to the constructor that will hold the experience tuples.
- a named tuple to index the stored experiences by their titles

2.2.2. - `add` method

The add method appends the SARS tuple (given in the parameters) to the memory

2.2.3. - `sample` method

Returns a set of random tuples from the memory buffer based on the batch_size (the number of random samples to return).

2.2.4. - `__len__` method

When using len(myAgent) it will return the size of the memory tuples buffer.

2.3. - Defining the Agent class

The agent will train the Deep Q-Networks to explore/exploit which will be used to define the policy that solve the RL problem.

It will consist of the following methods with their described functionality:

2.3.1 - `__init__` constructor method

The constructor will take three parameters, these are `state_size`, `action_size` and `seed`

1) in the constructor we will initialize `two QNetwork instances` with the same random seed as to be initialized with the same weights. The first QNetwork instance will be the online network, while the target network will be the to compare over the changes of the online network and synchronize with the online network based on the `UPDATE_EVERY` frequency from the parameters above.

2) We will define the `Optimizer`, in our case we will use Adam, with the specified learning rate `LR` as defined from the parameters above.

3) Finally, the agent will instantiate a ReplayBuffer to store its experiences.

2.3.2. - `step` method

This method will add experiences to the buffer. If our current step matches the amount of steps based on which we wish to perform learning (UPDATE_EVERY), then we will learn based on random SARS tuple samples as long as the memory contains at least as many items as our batch size (128)

2.3.3. - `act` method

The act method will fetch the existing state, then in evaluation mode we make a forward pass the state to the neural network and get the output with the scores for the available actions. Now according to the epsilon-greedy-policy probabilities we will either select the highest scored action, or select randomly among any of the available actions.

2.3.4. - `learn` method

Q-Learning (SarsaMax) dictates that the next action we will take into account is the one that always maximizes the reward at the next state based on our policy. Now applying Q-Learning in a Deep-QN...

So the `Q_targets_next` will obtain the the highest action for next state, populated by our batch size.

We use this to calculate our `Q_targets` based on our existing `rewards`, so we have a complete SARSA now.

Ok, so far we have obtained our SARSA and its now time to see how this change compares to our existing network weights configuration so we make any alterations necessary!

Now with a forward call to our DQN we can gather the state values, which we call `Q_expected`.

Great... now its time to measure the error on our Neural Network. To do so, we will use a regression error function, the MSE (Mean of Squared Errors), which is used to quantify the distance between Q_expected and and Q_targets. This distance is our error.

Cool! Now all that is left is utilize this error to actually train the neural network, expecting to get a new smaller-error.

- So we zero our gradients
- quantify the loss to all weights using backpropagation by using backward() call
- and finally apply the gradient change to existing weights using the optimizer's step() to come up with new weights

Ok! Great! We have finished our learning step now!

The only thing is that our target network should sync to our local (online) network until the next learning step (whenever it will reoccur based on the UPDATE_EVERY variable).

So we will `soft_update` our target network to match our local network.

2.3.5. - `soft_update` method

Ok, finally we reach the point where the target network will sync to our online local network! That means that learning has concluded, and we sync the networks to prepare for our next learning.

The soft update takes a constant τ (tau), which defines by how much the target network will match the weight configuration of the local network. Passing value 1 will result in clones (fully sync), while anything lower than that will reflect slight variations between the old weights to the new ones. Setting a tau of 0 will never update the target network and it will always keep its original initial configuration.

2.4 Initializing the agent object

Now we have all the tools to initialize the agent which will explore/exploit the banana world!

We initialize the agent setting the

- `state_size  = brain.vector_observation_space_size`, which is 37
- `action_size = brain.vector_action_space_size`, which is 4

```
agent = Agent(state_size  = brain.vector_observation_space_size,
              action_size = brain.vector_action_space_size,
              seed = 0)
```

3. Interacting with the Environment to learn

At this stage, we will put everything together so we can interact with the environment for several episodes, until we come up with a good policy estimation via our DQN network, and eventually solve the RL task.

We will define a function `DQN` which will put everything together and drive the learning through interaction.

Our DQN method will run for a maximum of 2000 episodes, after which if we have not reached our desired per 100 episodes score, the method will terminate.

In a nutshell, at every episode we will

- Reset the environment
- For each state, loop by getting the current state
    - Follow the epsilon-greedy-policy so the agent acts on that state according policy and epsilon
    - Get observation and reward
    - Use the agent's `step` function as described above to populate the experience buffer and possibly trigger the `learn` function

We keep doing this until our average score over 100 episodes is equal or higher than 13 (which relates to gathering 13 yellow bananas on time).

After every 100 episodes we save a `checkpoint` if the average score has increased, but we also overwrite this checkpoint once we reach our goal. Now one might ask, what is the use to save intermediate scores. Well, sometimes you might interrupt this method earlier, and just want to check how your model performs with this "early" training (so it is there just for curiosity reasons).

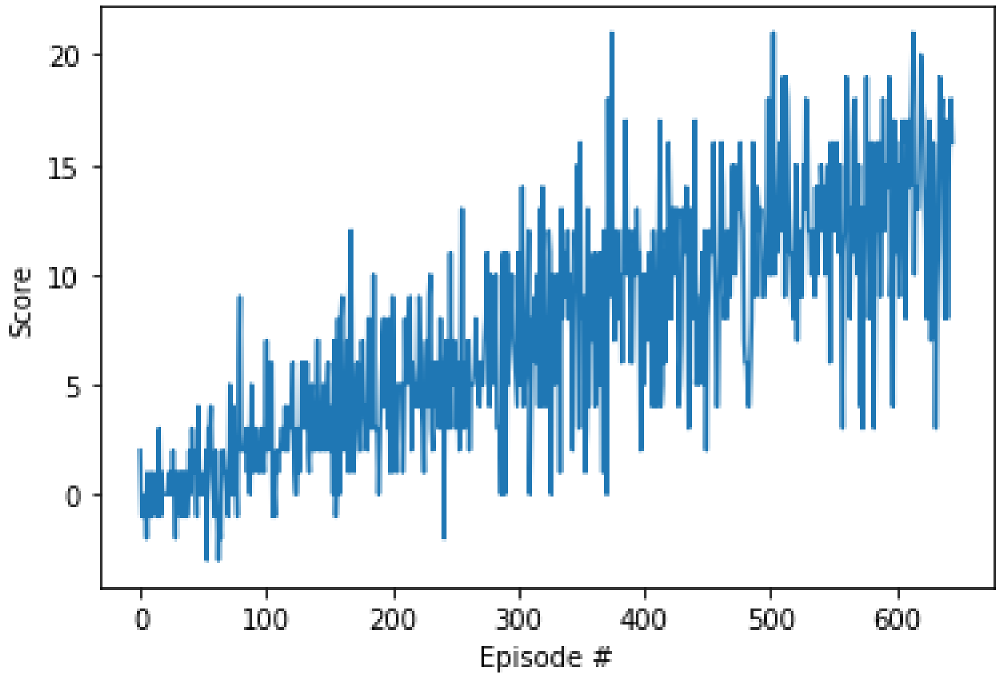
4. Performance

Given our existing approach, we reached the desired 13+ cumulative result (over 100 episodes) at episode 645.

More specifically, here comes the outcome of the DQN function which took 645 episodes to conclude:

```
Episode 100    Average Score: 0.91
Episode 200    Average Score: 3.62
Episode 300    Average Score: 5.53
Episode 400    Average Score: 8.55
Episode 500    Average Score: 9.93
Episode 600    Average Score: 12.47
Episode 645    Average Score: 13.03
Environment solved in 645 episodes!  Average Score: 13.03
```

And here is the visualization of the cumulative reward (over 100 episodes) all the way from episode one to 645.



6. Further Refinements

There are several refinements to the existing ones that could add up to the performance of our smart DQN agent.

Based on our existing setup, it is much likely that playing further with the hyperparameters we might came up with an agent that could learn faster.

However, the agent could greatly benefit from additional RL-specific refinements which we could introduce to our existing setup.

More specifically:

- Prioritized learning could be used so rather than randomly sampling SARS tuples from our experience buffer, we could revisit more frequently the SARS tuples that better relate to rational actions over specific states.

- A Double-DQN could be used to address the DQN problem of action value overestimation

- The QNetwork pytorch model could be changed to adapt a dueling DQN architecture, which incorporates two streams. One for the state-values function and another for the advantage-values function.