Report

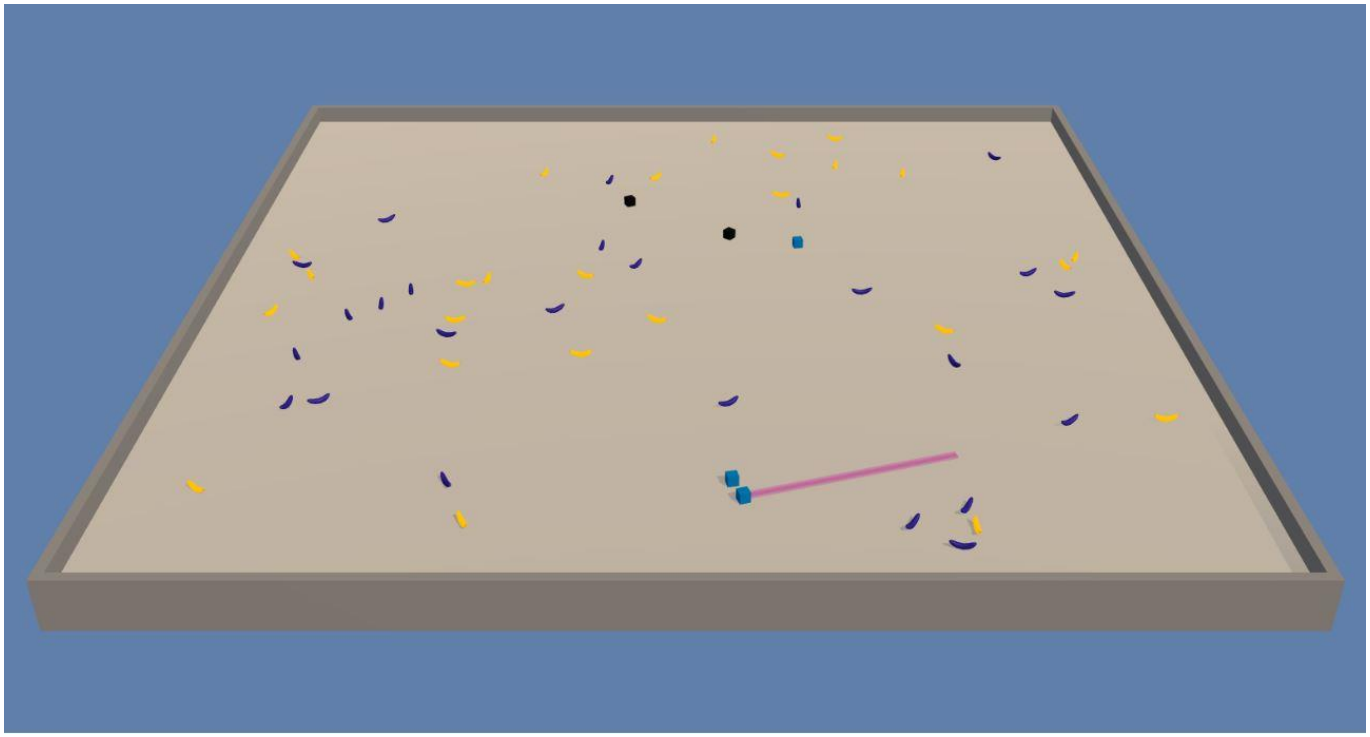
September 13, 2019

1. Deep Q-Network - Navigation project

1.1 Introduction

Reinforcement learning has long be thought to be an important tool in achieving human level Artificial Intelligence (AI). While we are still far away from anything remotely like human level AI, the advent of deep learning has significantly improved the performance of traditional reinforcement learning algorithms. In this project, we will look at an implementation for the banana collection project in the Udacity Deep Reinforcement Learning Nanodegree program.

Using a simplified version of the Unity Banana environment, the objective of the project is to design an agent to navigate (and collect bananas!) in a large, square world. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas. The agent's observation space is 37 dimensional and the agent's action space is 4 dimensional (forward, backward, turn left, and turn right). The task is episodic, and in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes. 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. You can learn more about the Banana Collector at official Unity Technologies Environments



2. Important notice

The content of this report is the same as the comments of the Navigation notebook at the specific code cells!!

3. The Unity Environment First we import the Unity environment and we start the specific Banana environment:

- env = UnityEnvironment(file_name="/data/Banana_Linux_NoVis/Banana.x86_64")

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.

- brain_name = env.brain_names[0]

- brain = env.brains[brain_name]

4. First we define the Neural Network This is the most important factor in DQN. We used 4 Linear layers as part of this network and we changed the nodes a lot to find the best parameters. We examined a lot the middle layer and saw that when we use larger values than 64 etc 128 or 156 then we solve the environment in fewer episodes. With 128 nodes task is solved in 383 episodes and with 156 is solved in 419. The Deep Neural Network is used so the agent can approximate the action selection for any state. The input size of the NN is 37 like the state size and the output is 4 like the action size parameter. We use the argmax function so we pick the action with the higher probability.

5. We set some hyperparameters These parameters are very important for the DRL algorithm and slight changes make big impact at training. It appears that specific values contributed to faster training.

```
import torch.optim as optim

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
```

6. Also we define the agent class. The DQN network will use the "Experience Replay" and the "Fixed Q-Targets" optimizations.

In the constructor we define state size, action size and seed. We initialize 2 identical Neural Networks with the same seed so they get the same weights at the beggining. We use Adam optimizer with very small Learning rate(0.0005). Also we initialize ReplayBuffer which is responsible to store the "experiences" of the agent etc keeps history of different states, actions, rewards, next states and done parameters.

At the step function we save experiences in replay memory and if there are enough experiences in replay buffer then it gives the order to sample some experiences and update the target network with a gamma discount factor to make rewards that are in future less relative than that of current rewards.

At the act function we make a forward pass at the local neural network and we select an action. Now according to the epsilon-greedy-policy we either select the highest scored action, or select randomly among any of the available actions.

At learn function we take use of the SarsaMax algorithm. By this algorithm we tell the agent to pick an action that maximizes reward in the next state based to the specific policy. Then we compute the Q_targets based on reward and gamma factor. We also get Q_expected from local network and after that we compute the loss with Pytorch MSE loss function. We continue the algorithm with making step with the optimizer to correct the weights and at the end we use the soft_update function.

Q-Learning algorithm

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.

Applying Q-Learning in a DeepQN

The Q_targets_next will obtain the action with the highest probability for next state, with respect to our batch size. We use this to calculate our Q_targets based on our existing rewards. We have obtained SARSA and its now time to see how this change compares to our existing network weights configuration so we make some changes. Now with a forward call to our DQN we can gather the state values, which we call Q_expected. We 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 mesuarement between Q_expected and and Q_targets. This distance is our error. Next we 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 get new weights.

Deep Q-learning with experience replay

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

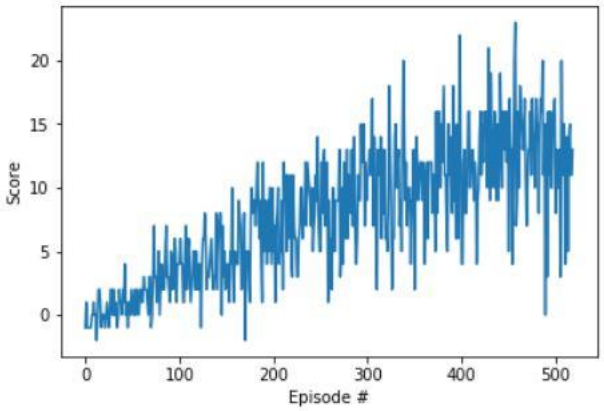
At last soft_update function we have a constant TAU. This function updates the target network's weights to match the weights of the local network. Parameter of 1 will result in full synchronization and parameter of 0 will result to NO synchronization

7. Last we have the ReplayBuffer class This class is responsible to store the experiences of the agent which include different states, actions, rewards, next states and done parameter. The add method appends the SARS tuple to the memory. The sample method gets random sample of data for the agent with a specific buffer size. The samples are random so that our agent can avoid memorizing sequences and take advantage of experiences that are rare.

8. Below we can see a plot of the episodes

and the average score every 100 ones.

```
Episode 100     Average Score: 1.36
Episode 200     Average Score: 4.98
Episode 300     Average Score: 8.13
Episode 400     Average Score: 10.66
Episode 500     Average Score: 12.78
Episode 519     Average Score: 13.02
Environment solved in 419 episodes!     Average Score: 13.02
```



9. Improvements We can make some improvements to solve the environment to fewer episodes

• Different Batch_Size may result to faster training
• GAMMA factor can be set closer to 1 so that we can make immediate results more effective for our agent when bananas are close enough one to another
• TAU parameter will tweak target network's weight to match those of the local network. Closer to 1 will result in full sync and closer to 0 at NO sync.
• UPDATE_EVERY is a factor that we can change and result to improvements. Here we selected 4 and if we set it to larger numbers will result to improvement of generalization.

We can also use Reinforcement Learning refinements.

• Double DQN - Deal with overestimation of action values.
• Prioritized Experience Replay - To make important experiences more relevant.
• Dueling DQN - Improve performance by dividing a neural network in two streams. One for state values and other for advantage values.

• Rainbow DQN which incorporates all six of below algorithms

1) Double DQN (DDQN)
2) Prioritized experience replay
3) Dueling DQN
4) Learning from multi-step bootstrap targets
5) Distributional DQN
6) Noisy DQN