Deep Reinforcement Learning - Navigation

Overview

The goal of this project is to train an agent to achieve an average reward of at least +13 over 100 consecutive episodes in the environment.

The environment is a square world filled with yellow and blue bananas.

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Four discrete actions are available, move forward, move backward, turn left and turn right

Implementation

The project was solved using deep reinforcement learning, more specifically a Deep Q-Network. The code was based upon the Luna example from the Udacity Deep Reinforcement Learning GitHub repo (h ttps://github.com/udacity/deep-reinforcement-l earning/tree/master/dqn). This was modified and updated to work with the Unity-ML environment and extended with new model architecture.

- The Jupyter notebook **Navigation.ipynb** contains the implementation for training the agent in the environment.
- **dqn_agent.py** contains the Deep Q-learning agent which interacts with the environment to optimize the reward.
- **model.py** contains the Neural Network which takes in the input state and outputs the desired Q-values

Learning algorithm

The Deep Q-Learning algorithm was chosen to solve the environment. It consists of two main processes. Firstly, it samples the environment by performing actions and storing the experience tuple in memory (replay buffer). Secondly, it samples a small batch of tuples randomly to learn from using a gradient decent update step. Both are not dependent on the other and can run at different times or frequency.

On a higher level, Deep Q learning works as such:

1.Gather and store samples in a replay buffer with current policy

2.Random sample batches of experiences from the replay buffer (known as Experience Replay)

3.Use the sampled experiences to update the Q network

4.Repeat 1-3

I built this implementation using Simple Vanila DQN. It solved the Environment in 587 episodes. For further improvements in convergence time we can use Double DQN or Dueling DDQN.

Replay Buffer

One of the breaks through with the DQN architecture was the introduction of Experience replay. It stores experiences in a replay buffer so that learning is separated and can take place on both the current experience and past experiences. This allows us to use Supervised Deep Learning techniques. This is done by randomly sampling the stored tuples in the replay buffer for training the model. The code is under the ReplayBuffer Class in **agent.py**

Model Architecture

The Neural Network used for the Vanila DQN is Multilayer perceptron. It returns the Q- values for each state action pair.

- The model has 2 fully connected layers with 64 nodes each.
- It takes an input of 37 (state vector) and output of 4 (action space)
- Relu activation is used between the first two layers
- Adam optimizer was used in optimization for better result

Agent

The agent class in agent.py is a Deep Q-Learning agent which interacts with the environment to maximize reward. It has a Local and Target network which is imported from model.py and replay memory imported from memory.py. It consists of four methods and hyperparameters. The methods are described below.

Step: Saves experiences in Replay memory. Runs the learn function every 4 steps which is set in variable UPDATE_EVERY

Act: Returns actions for the given state as per current policy using Epsilon-greedy action selection

Learn: Updates the value parameters using a given batch of experience tuples from Replay Memory. It then gets the Q values from the network for the next state and chosen action. Then it computes the target for the current state and gets the expected Q values from the target model using current actions. Next it computes and minimizes loss. Finally, it runs the soft_update method.

Soft_update: Performs a soft update of model parameters based on the TAU hyperparameter. The local model is updated based on TAU while the target model is updated based on 1.0 – TAU.

Hyperparameters

The hyperparameter used for training are the following:

BUFFER_SIZE = int(3e5) # replay buffer size

BATCH_SIZE = 128   # minibatch size

GAMMA = 0.99   # discount factor

TAU = 1e-3   # for soft update of target

parameters

LR = 5e-3   # learning rate

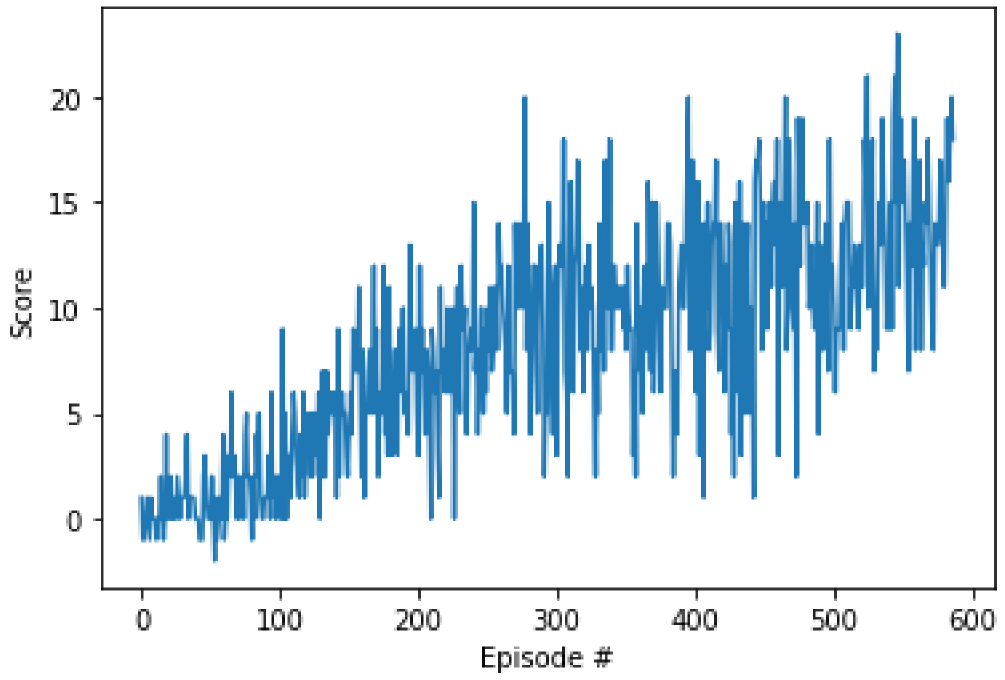
UPDATE_EVERY = 4   # how often to update the

network

Results

The results from the DQN were impressive. It easily achieved an average score of 13 pretty easily in 491 episodes.

```
Episode 100	Average Score: 0.99
Episode 200	Average Score: 5.11
Episode 300	Average Score: 8.40
Episode 400	Average Score: 10.18
Episode 500	Average Score: 11.54
Episode 587	Average Score: 13.06
Environment solved in 587 episodes!	Average Score: 13.06
```



Improvements

While results achieved were impressive, there is always room for improvement. To try and achieve faster training or improved final scores I would implementing the following.

- Modifying the Hyperparameters. Changing the hyperparameter could potentially speed up training or increase the final score.
- For further improvements in convergence time we can use Double DQN or Dueling DDQN. It may result in reaching goal in less no. of episodes.
- Modifying the model architecture by changing the number of layers or neurons
- Try to implement features from the Rainbow DQN (h ttps://arxiv.org/abs/1710.02298)

Conclusion

The project was a fantastic learning experience. Troubleshooting various parts of the agent and playing around with endless variation of hyperparameters and Neural Network architectures was particularly useful. Understanding how small changes can make a significant difference in performance and seeing the algorithms working to solves problems is incredible.

More is not always better. Too many layers or nodes in a neural network can rapidly slow down training and cause it to overfit.

Modifying the TAU and learning rate can change the outcome and speed of training. I recommend playing with those parameters while you run this project.