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 Dueling Deep Q-Network. The code was based upon the Luna example from the Udacity Deep Reinforcement Learning GitHub repo (https://github.com/udacity/deep-reinforcement-learning/tree/master/dqn). This was modified and updated to work with the Unity-ML environment and extended with new model architecture. Some of the code was separated out to modules, for example ReplayBuffer class in memory.py.

The Jupyter notebook results.ipynb contains the implementation for training the agent in the environment.

Agent.py contains the Deep Q-learning agent which interacts with the environment to optimize the reward.

Memory.py contains the Replay Buffer which is used by the Agent to record and sample a (state, action, reward, next_state) tuples for training the model.

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 (https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf) 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.

Three implementations were built and tested using different Deep Q-Network architectures. A Vanilla DQN, A Double DQN (https://arxiv.org/abs/1509.06461) and a Dueling DQN (https://arxiv.org/abs/1511.06581) . The final implementation used was a Dueling DQN as it performed the best in trials. All three variations consist of Q-Learning Agent and two Neural networks for approximating the Q-values function.

Once the Vanilla DQN was up and running I decided to implement a Double DQN. It uses the target network to generate the Q-values for the optimal actions provided by the online network. Those Q-values are then used to update the Q-Targets as input for the minimizing loss for the online network.

I built upon this by implementing a Dueling DQN. This was done by modifying the neural network architecture by splitting the last fully connected layer into an Advantage and Value layer and then combining the results of the two to get the Q-values. In this case a Multilayer perceptron was used but common features from a convolutional neural network can be passed to the fully connected networks.

Replay Buffer

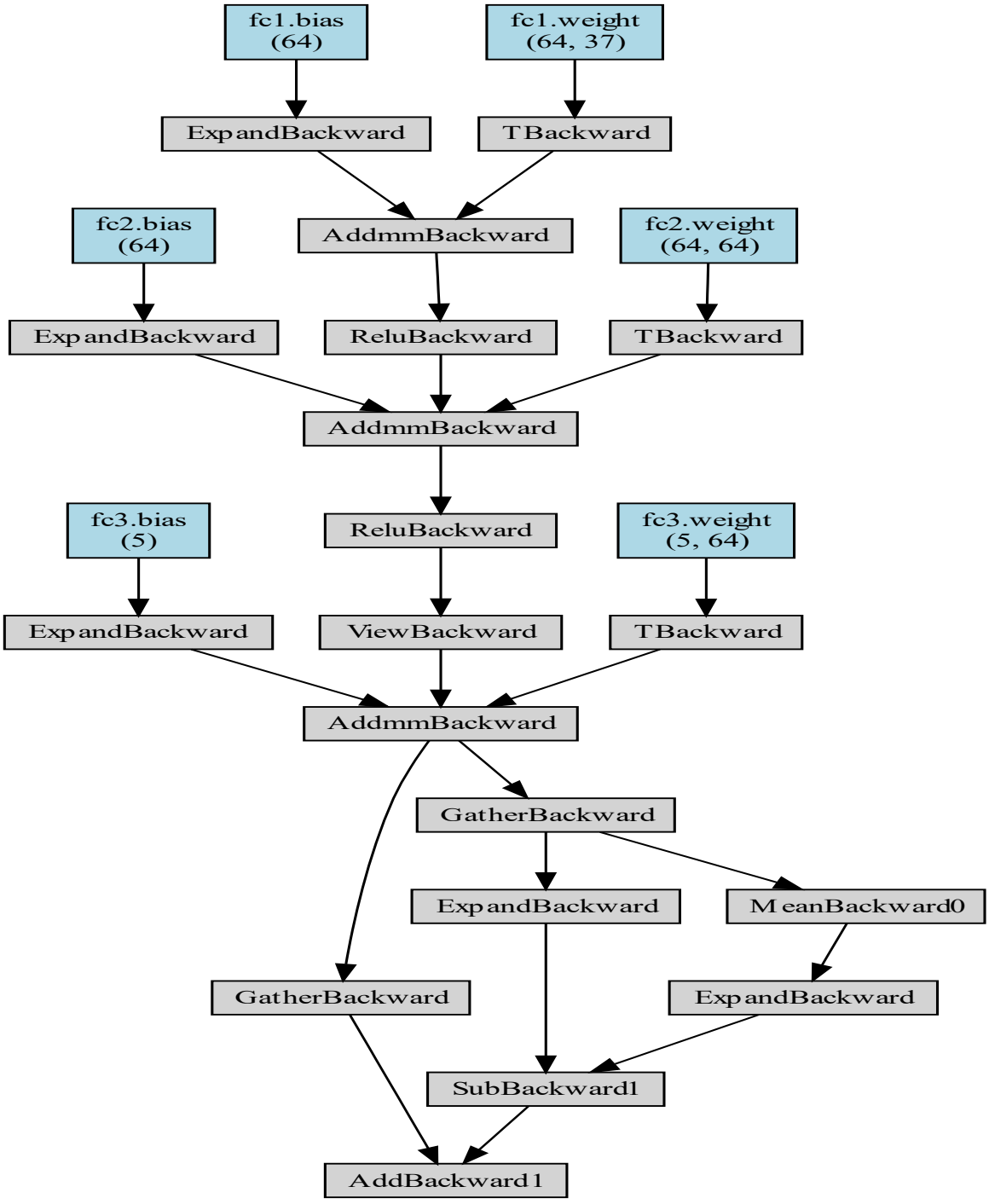
The agent uses a separate class imported from memory.py. It stores experiences in a replay buffer so that learning is separated and can take place on both the current experience and past experiences. This allows us to use Supervised Deep learning techniques. This is done by randomly sampling the stored tuples in the replay buffer for training the model.

Model Architecture

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

- The model has 3 fully connected layers with 64 nodes each.
- It takes an input of 37 (state vector) and output of 4 (action space)
- Third layer is split into an Advantage A(s,a) and Value V(s)
- The advantage layer outputs 4 (number of actions)
- the Value layer outputs 1
- The final output is calculated by Q(s,a)=V(s)+(A(s,a)-avg_a(A(s,a)))
- Rulu activation is used between the first two layers
- Adam optimizer was used for optimization

The following diagram displays the Dueling DQN architecture



Agent

The agent class in agent.py is a Deep Q-learning agent which interreacts with 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.

DQN

The DQN function in results.ipynb brings all these components together to train the network. The agent interacts with the environment by viewing the state and performing an action using epsilon-greedy. The rewards and next state are then sent back to the agent. The agent records a tuple (s,a,r,s') into replay memory. It then samples these to updates the Q-values to optimize rewards. This process is done until the desired average score is achieved, in this case 17, it discontinues training and finishes the loop.

Hyperparameters

The hyperparameter used for training are the following:

BUFFER_SIZE = 100000 (replay buffer size)

BATCH_SIZE = 64 (minibatch size)

GAMMA = 0.99 (discount factor)

TAU = 0.01 (soft update of target parameters)

LR = 0.0005 (learning rate)

UPDATE_EVERY = 4 (how often to update the network)

eps_start=1.0 (Initial epsilon for Epsilon greedy Action)

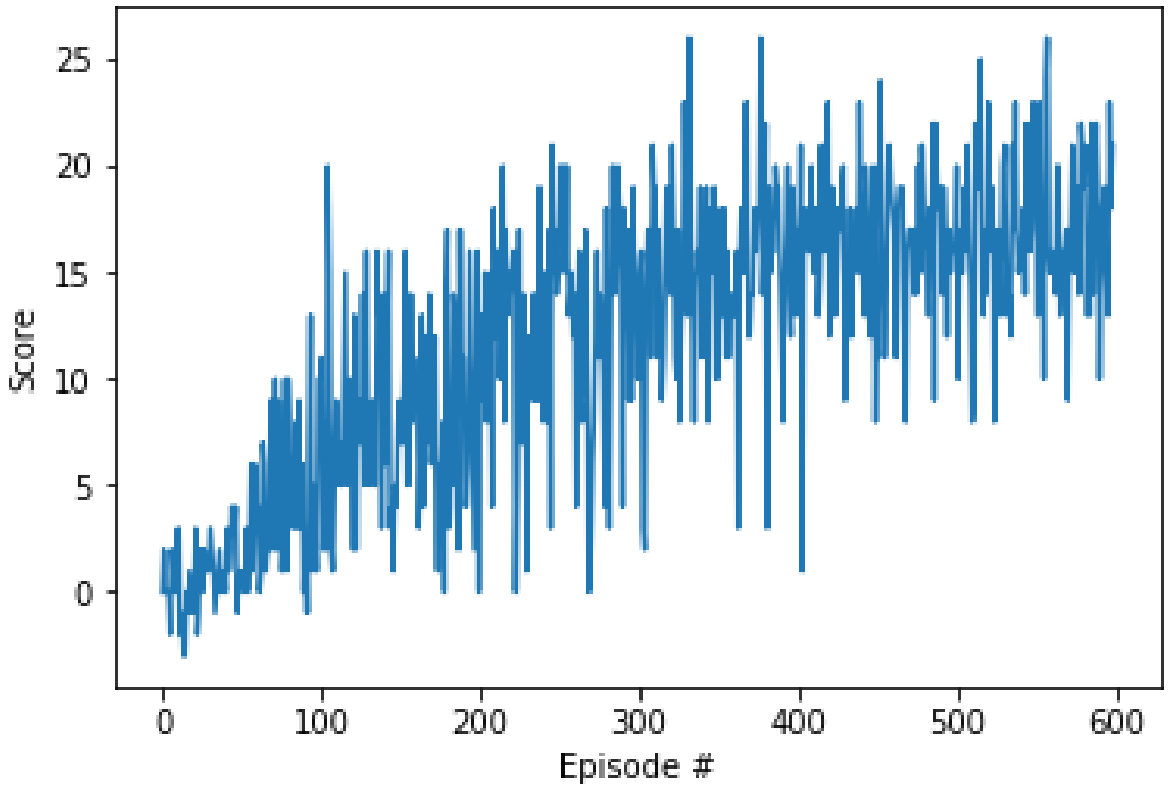
eps_end=0.01 (Minimum Epsilon)

eps_decay=0.95 (Epsilon decay rate)

Results

The results from the Dueling Deep Q-Network were impressive. It easily achieved an average score of 13 and then went further to a high of 17.

```
Episode 100    Average Score: 2.43
Episode 200    Average Score: 8.10
Episode 300    Average Score: 12.44
Episode 400    Average Score: 14.98
Episode 500    Average Score: 16.27
Episode 598    Average Score: 17.04
Environment solved in 498episodes!  Average Score: 17.04
```



Improvements

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

- Modifying the Hyperparameters. Changing the hyperparameter could potentially speed up training or increase the final score.
- Prioritized action replay could be used to improve learning. It adds to experience replay by prioritizing important experiences and replaying them more frequently. This has shown to improve learning efficiency. (https://arxiv.org/abs/1551.05952)
- Modifying the model architecture by changing the number of layers or neurons
- Try to implement features from the Rainbow DQN (https://arxiv.org/abs/1710.02298)

Next, I will implement a variation to this project by using an input of raw pixels for the state. This should make it more difficult to train and will need the help of a Convolutional neural network.

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.

The ages old reinforcement learning issue of exploration vs exploitation can not be underestimated. I found that more exploration gave steadier and higher gains but can also dramatically increase training times. Increasing the epsilon decay rate allowed the model to get a score of +13 in under 100 but this varied massively from training to training. It also would drop of quickly and achieve lower total training results.

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 a lot. I recommend playing with those parameters while you run this project.