Navigation in Unity Banana Collector Environment using Deep Q-Learning

James Gallagher

This report was produced as part of the submission for the first assignment of the Udacity Deep Reinforcement Learning nanodegree. The report describes the methods used and results obtained in training an agent to solve a provided Unity environment.

Environment

We aim to solve a Unity environment (*6*) by teaching an agent to collect as many yellow bananas as possible while avoiding blue bananas. The agent must navigate a square grid containing yellow and blue bananas.

The observation space consists of ray-based perception (35 vectors) around the agent's forward direction as well as 2 vectors for the agent's velocity. Fig 1. is a schematic representation of the agent's perception of the environment. The agent is aware of yellow and blue bananas and the boundary of the environment when they intersect with one or more of several finite rays projecting from the front side of the agent. The state includes which object(s) the rays intercept and the distance to the closest object intersecting each ray.

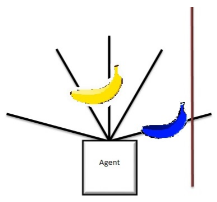

Figure 1: Ray Based Perception

Four discrete actions are available, corresponding to:

- 0 - move forward.

- 1 - move backward.
- 2 - turn left.
- 3 - turn right.

The task is episodic, consisting of 300 time steps per episode. The environment is considered solved when an average score of +13 is obtained over 100 consecutive episodes.

Methods

The solution to the problem makes use of Deep Q-Learning as introduced in (*1*) . The idea behind Deep Q-Learning is to adapt the classical temporal difference learning framework by approximating the state value function with a neural network. Under the temporal difference framework, under a policy π, each state and action pair, (s, a), is assigned a value $Q^{\pi}(s, a)$ - the expected discounted future return when taking action a in states s and then following policy π. It can be shown that there exists an optimal policy, π^* such that the expected future return from following policy π^* is greater than or equal to the expected return from any other policy all states. Furthermore, with a discrete state and action space TD-control algorithms such as SARSA max have been developed that are guaranteed to converge to the optimal policy. The details of the algorithm can be found in (*?*).

The problem occurs when we move to continuous state spaces, or state spaces large enough that it is better to approximate them with a continuous model. The SARSA max requires an update to the approximation of Q at every iteration.

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha\Big(r_{t+1} + \gamma \max_a [Q(s_{t+1}, a)] - Q(s_t, a_t)\Big)$$

Where γ is the discount factor and α is a hyper-parameter controlling the learning rate. Once we move to a continuous state space it is not possible to carry out this update for every value of S. The Deep Q-Network model uses a neural network $\tilde{Q}$ to approximate the $Q(S, A)$ function of temporal difference learning. This allows us to instead apply the update to the finite dimensional parameter space of the neural network, with the appropriate update step given by.

$$\Delta w = \alpha[r_{t+1} + \gamma \max_a \tilde{Q}(s_{t+1}, a; w) - \tilde{Q}(s_t, a_t; w)].\nabla\tilde{Q}(s_t, a_t; w)$$

Where w is the parameterisation of $\tilde{Q}$ and $\nabla\tilde{Q}$ is the gradient of $\tilde{Q}$ w.r.t w at (s_t, a_t). The key technical insight of the paper (*1*) was to address the problem of instability in the update of $\tilde{Q}$. This is done by storing experiences in a buffer that is randomly sampled from and by maintaining a separate target network. The respective purposes being to reduce the correlation between consecutive experience tuples and the correlation between the current estimation of $\tilde{Q}$ and the target value of $\tilde{Q}$.

With the experience buffer in place, the Q-learning updates can be made using batches of experiences drawn from the buffer. The update step is then precisely the update to $\tilde(Q)$ applied under gradient descent using the loss function

$$L(w) = \mathbb{E}_{e\in U}\Big[r_U + \gamma \max_a[\tilde{Q}(s', a'; w^-)] - \tilde{Q}(s, a; w)\Big]$$

where $U(D)$ is a uniform sampling of experiences $e = (s, a, r, s', a')$ from the replay buffer D. Note that we use w^- to refer to the parameters of our separate target network,$\tilde{Q}^-$, which is held fixed until updated every C steps to be equal to the local network $\tilde{Q}$.

Our implementation also makes use of two innovations developed since the original publication (*1*). These are Double Q-Networks, Dueling Q-Networks and soft-update.

Double Q-Networks, introduced in (*3*) address the problem that by choosing the next action with the maximum state action value there is a tendency to overestimate action values as the estimates are very susceptible to noise in the estimates of state action values at the next state. Ideally we would divorce the choice of action and the evaluation of the action. The Double Q-Network method achieves this by maintaining two Q-Networks with separate estimations of the state-action values. In practice, our target Q-Network, which is held fixed for C iterations, is sufficiently uncorrelated from our local network that we can use our local network to choose the next action and our target network to evaluate it.

Dueling Q-Networks, introduced in (*4*) make use of a novel neural network architecture based around the decomposition of the state-action value into the state value and the advantage function.

$$Q^{\pi}(s,a) = V^{\pi}(s) + A^{\pi}(s,a)$$

Where$A(s,a)$ is the advantage function, giving the expected difference in the total rewards from taking action a over following the policy π. This observation inspires a network architecture that starts with fully connected layers but then splits into separate streams $V(s,w,\beta)$ and $A(s,w,\alpha)$ for computing the state and advantage values. Here, α and β represent the parameter estimates for the separate streams and w the parameter estimates for the shared portion of the network.

The two streams are recombined using the function.

$$\tilde{Q}(s,a;w,\alpha,\beta) = \tilde{V}(s;w,\beta) + \tilde{A}(s,a,w,\alpha) - mean[\tilde{A}(s,a,w,\alpha)]$$

Where the addition of the mean of A is used to address the unidentifiability of the decomposition and stabilize the outputs. The advantage of this architecture is that for many environments the actions chosen in many states have little importance and only a sparse set of states have actions that matter. The decomposition allows the agent to learn which states have value independent of the actions taken in that state and which actions have the largest differential impact from other actions available in that state.

Implementation and Results

Network architecture and Hyper-parameters

The dueling network has the following architecture;

- Layer 1 - A fully connected with 64 nodes
- Relu activation function
- Layer 2 - A fully connected layer with 64 nodes

- Relu activation function
- V Layer 1 - The first fully connected layer of stream V branching off Layer 2 with 24 nodes
- Relu Activation following V Layer 1
- V Layer 2 - The second fully connected layer of stream V with 1 node
- A layer 1 - The first fully connected layer of stream A branching off Layer 2 with 24 nodes
- Relu Activation following A Layer 1
- A Layer 2 - The second fully connected layer of stream A with 4(number of actions nodes)
- Combination function - Q = V + A - mean(A)

Variable Name	**Value**	**Description**
epsilon_start	0.1	The initial value of ϵ for the ϵ-greedy policy.
epsilon_decay	0.99	Each episode ϵ is decayed by multiplication with this constant until the minimum.
epsilon_min	0.01	The minimum value of ϵ (no decay below this value).
buffer_size	10^5	The size of the replay buffer
batch_size	64	The number of observations in each minibatch
gamma	0.99	The discount factor, γ
tau	10^-3	The soft update rate, τ
lr	5×10^-4	The base learning rate of Adam optimization on the Q-Network
update_every	4	The number of steps, C , between updates of the target Q-network
n_episodes	700	The number of episode per training run

Table 1: Hyper-parameters for Q-Learning

Code

The implementation makes heavy use on the provided solution to the Lunar Lander-v2 environment. This code is available in the github repository for the course (*5*). The structure of the code is as follows:

- The network architecture is defined in the DuelingQNetwork class in model_dueling.py - this extends PyTorch's nn.module class, thus allowing the model to be trained using PyTorch auto gradient tools.

- The agent class is used to represent the agent to be trained. The local and target networks are maintained here. The method act returns the action for a given state and policy, the method step updates the experience replay buffer and the method learn updates the target and local networks.
- The python notebook Navigation_Solution.ipynb sets ups the environment and controls the online training in that environment.

Where the code differs from the provided solution is in the implementation of a dueling networks architecture and in the use of the double-update of the Q-Network. The Double Q-Network feature is implemented using the following python code;

```
if(double_update):
next_actions = self.qnetwork_local(next_states).detach().max(1)[1].unsqueeze(1)
Q_targets_next = self.qnetwork_target(next_states).gather(1,next_actions)
```

For each experience in the replay buffer, we select the actions based on the maximum state-action values of the local network - as opposed to the target network. We then use the values assigned to those actions from the target network as normal.

To implement Dueling Q-Network we simply have to update the architecture of our Q-Network. With PyTorch this is easy to do, we simply have to change the forward propagation in the forward method.

```
def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        v = F.relu(self.vfc1(x))
        v = self.vfc2(v)
        a = F.relu(self.afc1(x))
        a = self.afc2(a)
        q = v + a - torch.mean(a)
        return q
```

Results

A single online training run was performed for a total of 700 episodes. The environment is considered solved after the agent obtains an average score of +13 or greater over 100 consecutive episodes. This was achieved by our agent after **381** episodes. Fig. 2 shows the scores and moving average score over the 700 episodes of training.

Performance appears to plateau after around 500 episodes at about +15.

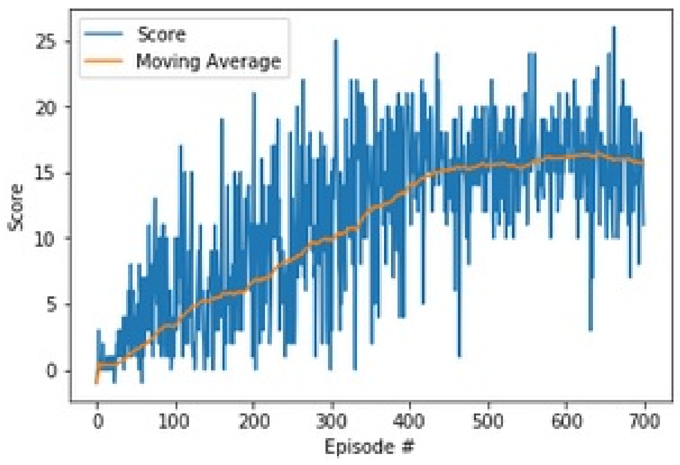

Figure 2: Evolution of scores during online learning.

The results demonstrates that Q-Learning enables the agent to learn to perform to a reasonable standard in the environment even when hyper-parameters were chosen in an arbitrary fashion. It is expected that performance would improve significantly if hyper parameter tuning was performed.

Ideas for Future Work

- A comparision of the performance with and without double and dueling features and an analysis of why the environment was amenable or otherwise to using these features. Hyper-parameter tuning of each model separately would be required in order to provide a valid comparision.

- The default state representation from the unity environment uses 37 dimensions. A lower dimensional representation would be possible by multiplying the distance variable by the binary variables for encountering each object and setting the variables to -1 when the object is not encountered (the latter introducing a discontinuity - the effect of which would need to be investigated). It would then be interesting to look at how different state representations may perform better or worse with different architectures and why. It may also be possible to incorporate transformations of the state space into the network architecture to allow them to be trained.

- Prioritised experience replay (*2*) could be implemented and incorporated into the learning algorithm.
- It would be interesting to look at the effect of missing and incorrect state information on the agent's performance. I suspect that an agent would be able to deal reasonably well with missing information but would perform poorly if there was a significant amount of incorrect information during the online training.
- The DQN paper (*1*) used images at 4 consecutive time-points in the definition of a state. The effect of grouping consecutive states together could be investigated.

References and Notes

1. Mnih et al, *Human-level control through deep reinforcement learning*, , Nature, 1426 p.529.

2. Tom Schaul, John Quan, Ioannis Antonoglou, David Silver, *Prioritized Experience Replay*, , arXiv:1511.05952

3. Hado van Hasselt, Arthur Guez, David Silver, *Deep Reinforcement Learning with Double Q-learning*, , arXiv:1509.06461

4. Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas *Dueling Network Architectures for Deep Reinforcement Learning*, , arXiv:1511.06581

5. Alexis Cook, Deep Reinforcement Learning Repository, Github *https://github.com/udacity/deep-reinforcement-learning*,

6. Unity ML Agents, website *https://unity3d.com/machine-learning*

7. Udacity Deep Reinforcment Learning Course Notes *https://github.com/udacity/deep-reinforcement-learning/blob/master/cheatsheet/cheatsheet.pdf*