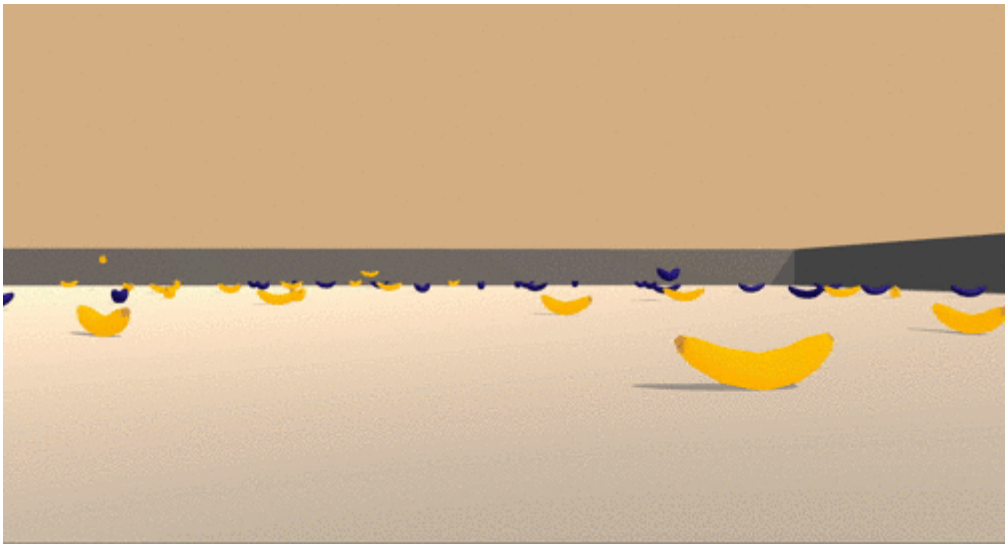
Reinforcement Deep Q Learning for playing a game in Unity

Ravish Chawla

Sep 27 · 6 min read ★

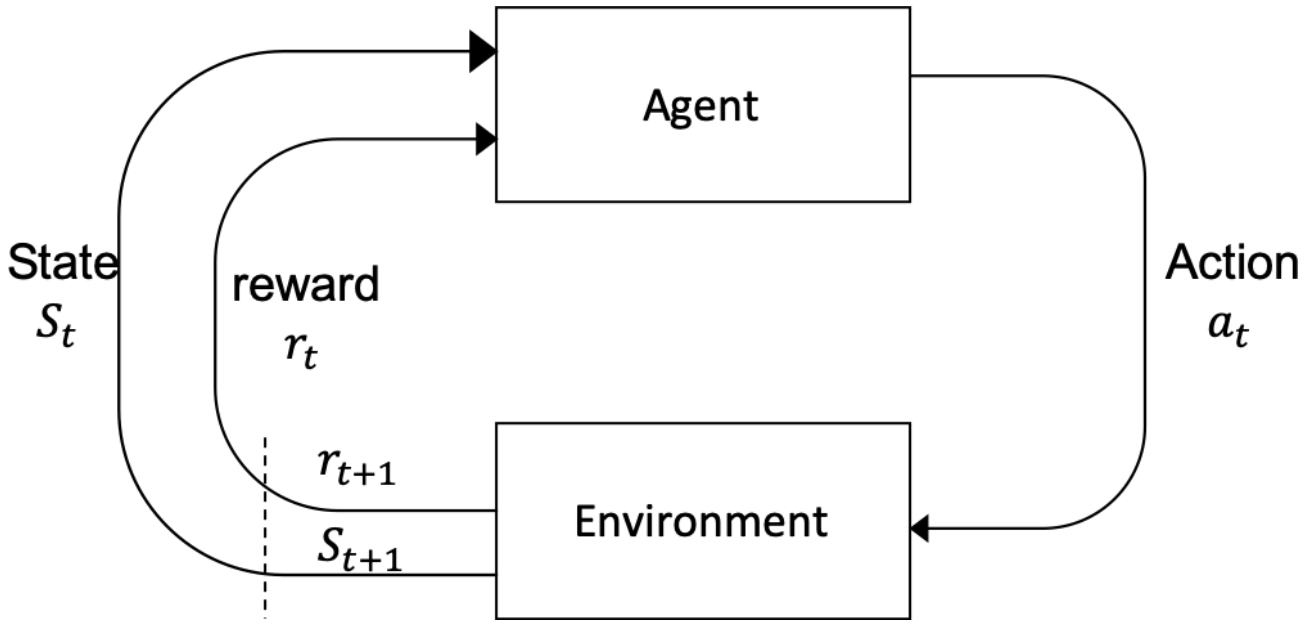
In this environment, there is a continuous space space, represented in 37 dimensions, and 4 actions (left, up, right, down). The goal of the environment is to collect *yellow* bananas and avoid *blue* bananas, which return a reward of +1 or -1 respectively.



Unity Environment

Algorithm Overview

Deep Q Learning is an extension of Q Learning, a value based method for Reinforcement Learning. In Reinforcement learning, we look at states, actions, and rewards, and the goal is to teach the model to predict actions which provide the best long-term reward from each state.



Q Learning

In Q Learning, the agent interacts with the environment iteratively, by taking an action. The environment responds by informing the agent of the reward from that action, and advancing to the next state. This happens continuously until the environment is “solved”. Reinforcement learning tries to learn the best sequence of actions to take. This is done by trying different combinations of actions, first randomly, than using a policy based on what the model has learned from rewards up to this point. This happens until the environment reaches its terminal state.

We refer to it as Q-Learning because of the actual algorithm. Let’s look at the pseudocode:

Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
  Sates $\mathcal{X} = \{1, \ldots, n_x\}$
  Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
  Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
  Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
  Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
  Discounting factor $\gamma \in [0, 1]$
  procedure QLEARNING($\mathcal{X}$, A, R, T, α, γ)
    Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
    while Q is not converged **do**
      Start in state $s \in \mathcal{X}$
      while s is not terminal **do**
        Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow \arg\max_a Q(x, a)$)
        $a \leftarrow \pi(s)$
        $r \leftarrow R(s, a)$ ▷ Receive the reward
        $s' \leftarrow T(s, a)$ ▷ Receive the new state
        $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
        $s \leftarrow s'$
    return Q

Q Learning Pseudo-code

At each iteration of Q Learning, we look at 5 things:

- *a* — action taken by the agent onto the environment at timestep *t*
- *s* — state held in the environment at timestep *t*
- *r* — immediate reward returned from the environment after taking the action *a*
- π — policy used by the agent to decide the next action

- *Q* — the long term return for the agent, when it takes action *a* at state *s*. Unlike *r*, which is the short term reward, Q refers to the combined reward of all future states and actions, starting from its current position.

The goal of Q Learning is to learn these Q Mappings for each state/action pair. Over multiple iterations, the policy used to decide the next action is improved, taking in account which action returns the highest future reward at that state.

Q Learning is the base algorithm, and there are improvements and modifications to it. One of these is ***Deep Q Learning***. In Deep Q Networks, instead of keeping track of the returns in a mapping table, a Neural Network is used to learn the Q-value function instead. These Networks perform better on high-dimensional state spaces than standard on/off policy based methods do.

- **Require:**
  - Replay memory D with capacity N
  - Action value function Q with random weights w
  - Target action-value function $\hat{Q}$ with weights $w^- \leftarrow w$
- **Procedure** QNetwork (*state s*):
  - **for** *episode e* $\leftarrow 1$ *to* M **do:**
    - initialize state = $S \leftarrow s_1$
    - **for** *time step t* $\leftarrow 1$ *to* T **do:**

Sample
- Choose action A from state using policy $\pi \leftarrow$ ε-greedy$(\hat{Q}(S, A, w))$
- Take action A, observe reward R and next state $S' \leftarrow s_{t+1}$
- Store experience tuple (S, A, R, S') in replay memory D
- $S \leftarrow S'$

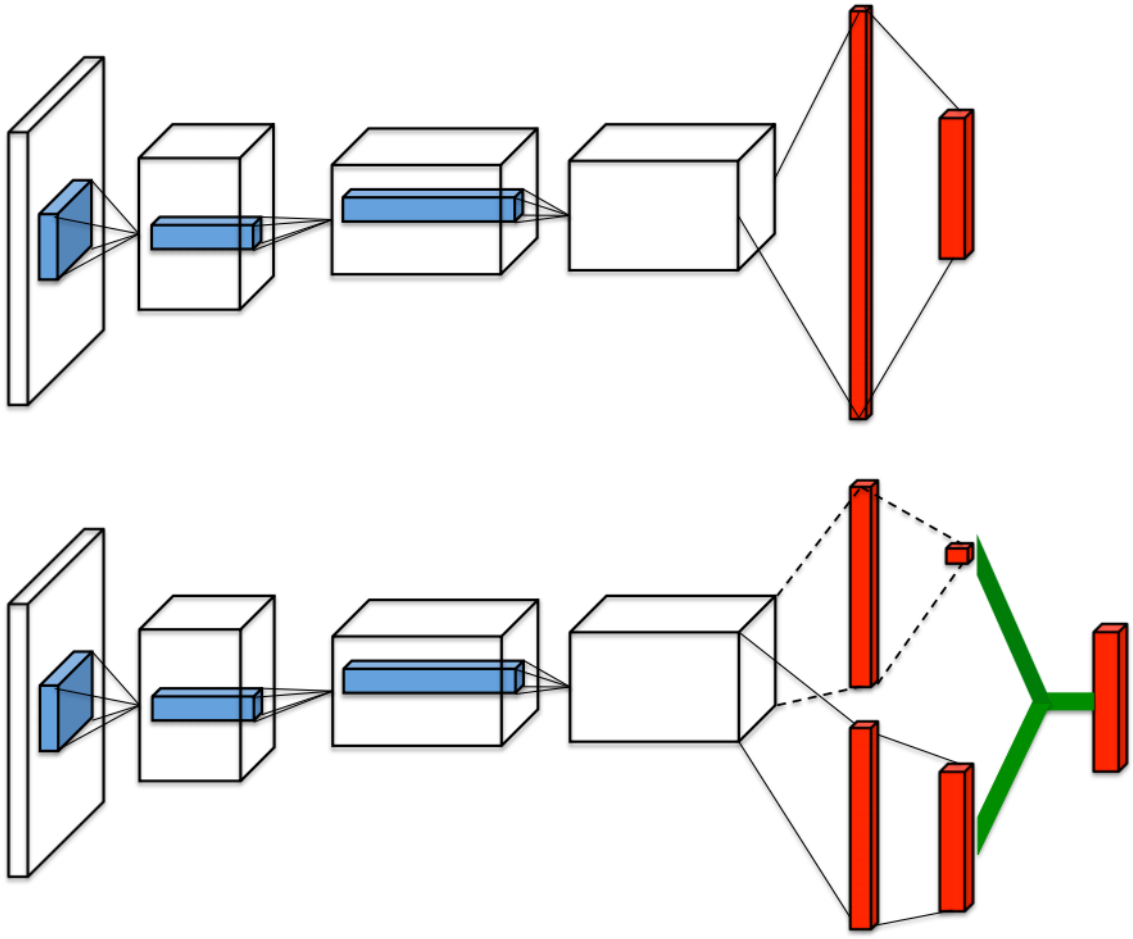
Learn
- Obtain random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from D
- Set target $y_j \leftarrow r_j + \gamma \, max_a \hat{Q}(s_{j+1}, a, w^-)$
- Update target $\Delta w = \alpha(y_j - \hat{Q}(s_j, a_j, w))\nabla_w(\hat{Q}(s_j, a_j, w))$
- Every C steps, reset: $w^- \leftarrow w$

Deep Q Learning

In Deep Q Learning, we do a 2-step training at each iteration. First, we use the epsilon-greedy policy to decide and take an action based on the current state. The second step

involves a “Replay buffer”.

Each seen state/action/reward/next-action pair is stored in a dictionary called the Replay buffer. A small batch of these is sampled, and use them to update the local Q Network. The advantage of the Experience Replay buffer is that it prevents the Neural Network from relying on sequential experiences. Experiences in an environment can be highly correlated, and training the model with a random sample breaks the temporal correlation by using independently distributed data. This prevents the model from oscillating between states, and it easier for the model to converge.



Q Network and Dueling Q Network

The Q Value Function used is learnt using a Neural Network in Deep Q Learning. For this project, there were two models implemented, a regular Deep Network (top) and a Dueling Deep Network (bottom). The Q Value function is represented as a sum of two different functions, the *Value function* V(s) and *Advantage function* A(s, a). The Value function represents the action for the given state, while the Advantage function is used

to represent how the one action at that state compares to all other actions at that state. The Q function is the sum of V(s) and A(s, a).

Instead of having both functions being learnt by a single network, in a Dueling Network, the output from the final layer is split into two branches, each used to train one of the two functions.

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|A|}\sum_a A(s,a))$$

Q Value Function using Value and Advantage

And the final Q value is calculated as the above function.

Implementation

We will now go over the code implementation of the agent and network. Let's first look at how that above network is implemented using PyTorch

```
class DuelingQNetwork(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_size=64, fc2_size=64):
        super(DuelingQNetwork, self).__init__();

        self.state_size = state_size;
        self.action_size = action_size;

        self.seed = torch.manual_seed(seed);
        self.fc1 = nn.Linear(state_size, fc1_size);
        self.fc2 = nn.Linear(fc1_size, fc2_size);

        self.value_fc = nn.Linear(fc2_size, 1);
        self.advantage_fc = nn.Linear(fc2_size, action_size);

    def forward(self, state):
        x = F.relu(self.fc1(state));
        x = F.relu(self.fc2(x));

        v = self.value_fc(x);
        a = self.advantage_fc(x);

        q = v + a - a.mean(1).unsqueeze(1).expand(x.size(0), self.action_size) / self.action_siz
```

```
        q = v + a - a.mean(1).unsqueeze(1).expand(x.size(0), self.action_size) / self.action_siz
        return q;
```

duel-network-1.py hosted with ❤ by GitHub view raw

Dueling Q Network

Output from the last layer is calculated into two functions, the Value and Advantage, and its final Q Value is calculated using the above formula before being returned.

The next important part of that we'll look at is how Training is done.

```
state, dqn_agent = env.reset(train_mode=True)[brain_name].vector_observations[0], Agent(state_si

scores, discount = [], EPS;

for ite in range(1, num_iterations+1):
    score, env_info = 0, env.reset(train_mode=True)[brain_name];
    state = env_info.vector_observations[0];

    for t_step in range(max_timesteps):
        action = dqn_agent.act(state, discount);
        env_info = env.step(action)[brain_name];
        next_state = env_info.vector_observations[0];

        reward, done = env_info.rewards[0], env_info.local_done[0];

        dqn_agent.step(state, action, reward, next_state, done);
        score, state = score + reward, next_state;

        if done:
            break;

    scores.append(score);
    discount = max(EPS_LIMIT, EPS_DECAY * discount);
```

duel-network-3.py hosted with ❤ by GitHub view raw

This code follows the pseducode that we looked at previously for Deep Q Networks. The Agent for the Banana game implements many of the important steps for training, including learning from Experience replay. For the implementation of the Agent, as well as how the trained model is used for actual play, please refer to the GitHub repository.

There are several important hyperparameters used in this implementation, that affected the training.

Search this file...

Hyperparameter	value
Number of Episodes	2000
Number of Timesteps	1000
Print Checkpoint step every	4
Training Batch Size	64
Discount Rate / Gamma	0.99
Learning Rate / alpha	5e-4
TAU	1e-3
Epsilon	0.1
Epsilon-Min	0.01
Epsilon-Decay	0.995

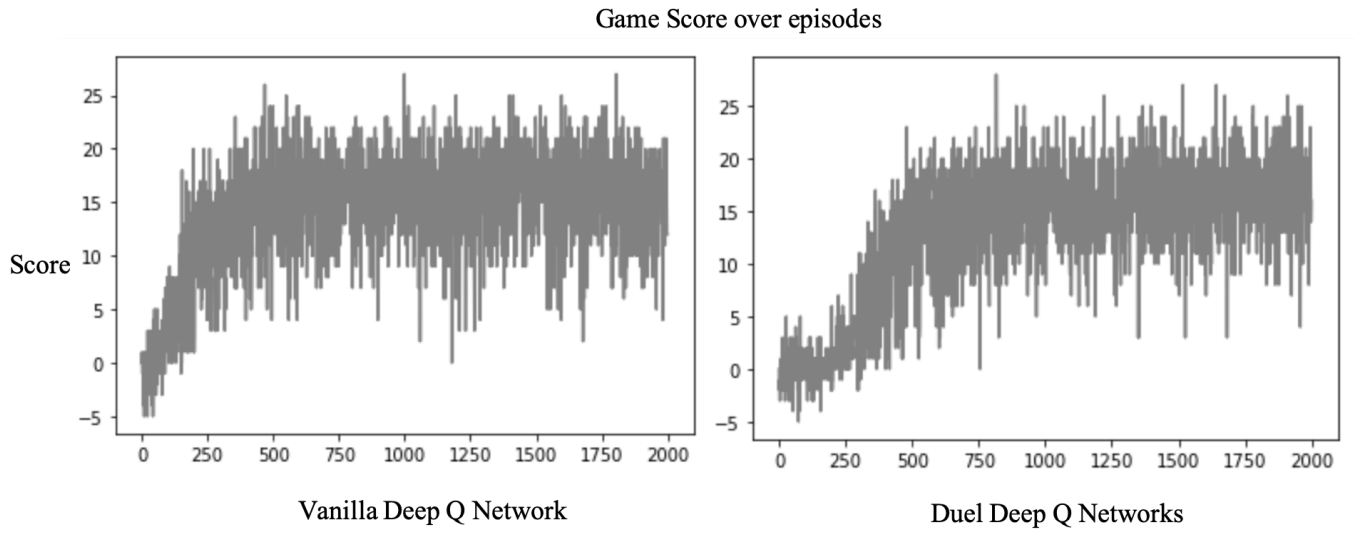
duel-network-4.csv hosted with ❤ by GitHub view raw

Hyperparameters

Results

Playing the Banana game with a trained Deep Q Network Model

Using the final trained Duel Networks model, the above episode yielded 17 total points. Different episodes yielded different scores, in some cases even better.



Results on Vanilla and Duel Networks

The graph on the right shows the score results obtained during training of the models. The two approaches had very different training behavior. Surprisingly, the Vanilla method had a better start for game score, with a high quicker rise in score over the first 250 episodes. However, after 500 episodes, both models converge to a similar pattern.

Dueling Networks were expected to improve upon the vanilla method, but it can be attributed to the fact that the model wasn't adjusted with any hyperparameter tuning. However, regardless with the base hyperparameter values the agent is able to solve the environment fairly quickly and well.

Next Work

There are several ideas for improvement on this project, which can be explored next. The work that would be important to focus on next are:

- Tuning the hyperparameter for the Dueling Networks to get an improvement over the vanilla Deep Q Networks.
- Try other variations of Deep Q Networks, including Double Q Networks, and the Rainbow algorithm

- Add Prioritized Replay for the Deep Q Algorithm, which changes the way samples are selected from the Replay Buffer. This method has been shown to show significant improvement over random sampling.

GitHub

All code for this project is available on GitHub at https://github.com/ravishchawla/Reinforcement-Learning-Navigation

Licensing, Authors, Acknowledgements

Credit to Udacity for providing the data and environment. You can find the Licensing for the data and other descriptive information from Udacity. This code is free to use.