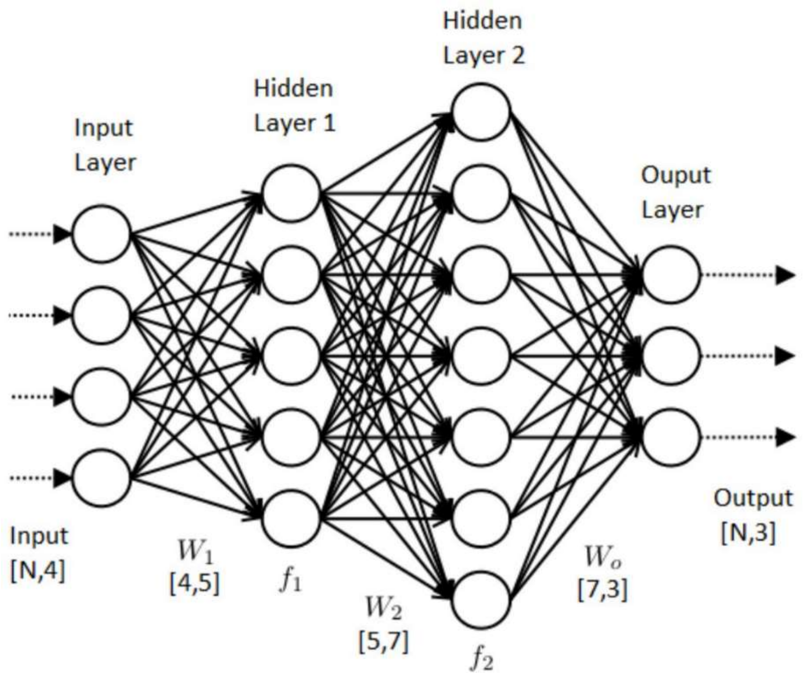
Solving Navigation using Deep Reinforcement Learning (Value-Based Methods)

By: Adrian Chow

Deep reinforcement learning is a subfield of machine learning that combines reinforcement learning and deep learning. Rather than using traditional RL methodologies, such as a Q-Table, artificial neural networks allow us to expand an agent's ability to solve a complex environment. While this post focuses on tackling the navigation issue, there is a lot to learn about how a DQN (Deep Q-Network) can use non-linear approximators to estimate the optimal action-value function.

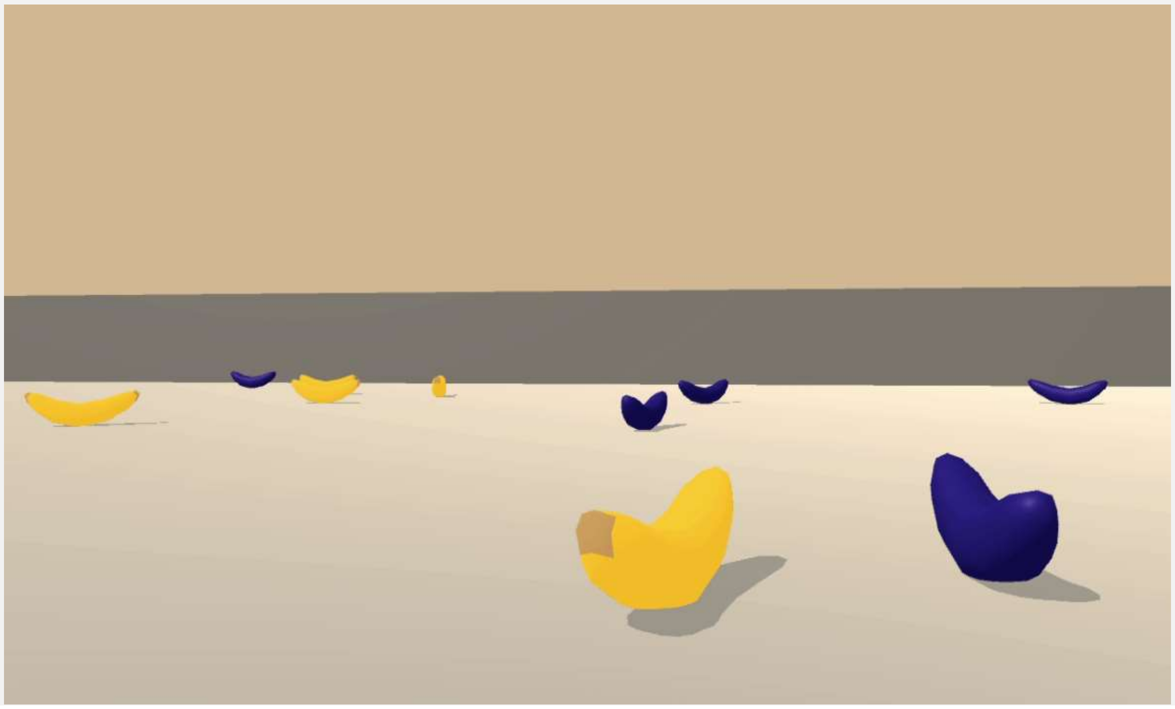


Artificial Neural Network (Source: VIASAT)

The Environment

To solve the task of navigation, I have used a task similar to the Food Collector environment on the Unity ML-Agents GitHub page. Instead, the project is provided by Udacity as a part of their Deep Reinforcement Learning Nanodegree.

In this environment, an Agent navigates a large, square world collecting bananas. Each episode of this task is limited to 300 steps. 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, avoiding the blue ones.



RL Problem Specifications:

- The Goal of the Agent: Collect yellow bananas, avoid blue bananas
- Rewards : `+1` collecting yellow bananas, `-1` collecting blue bananas
- Action Space — Discrete, 4 Actions [Forward, Back, Left, Right]
- State Space — Continuous, 37-Dimension
- Solving Condition: Average score of +13 over 100 consecutive episodes.

The state-space has 37 dimensions and contains the agent's velocity, along with the ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions.

Learning Algorithm

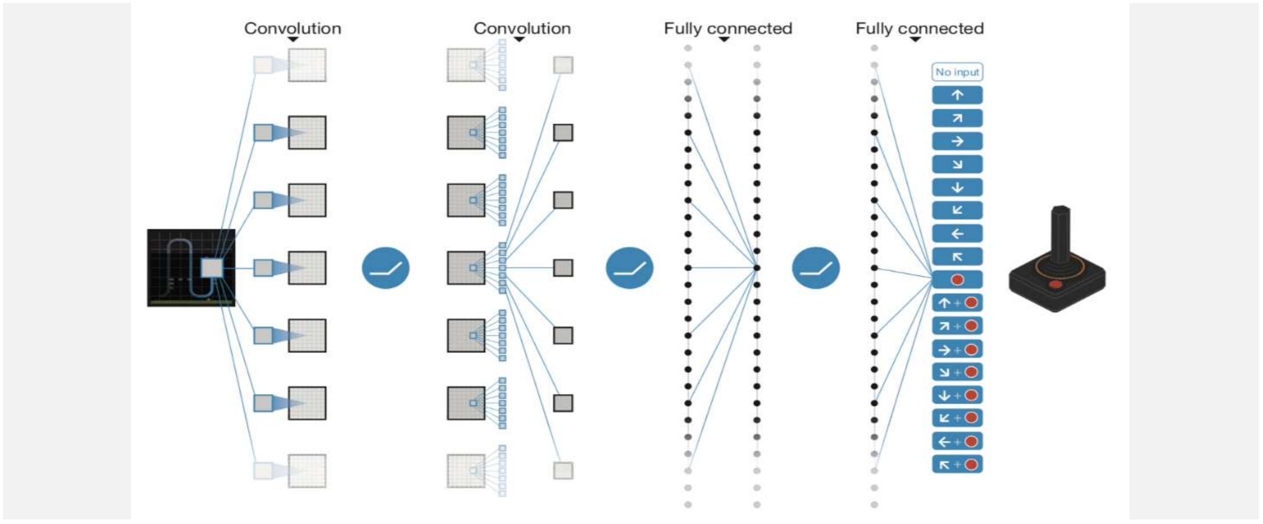


Illustration of DQN Architecture (Source)

The learning algorithm was developed by researchers working towards achieving human-level control to play a challenging domain of classic Atari 2600 games! To learn more about this, you read this research paper which outlines the algorithm in full detail.

Initialization:

- Initialize Replay Buffer Data Structure. A memory bank/buffer with a capacity of N=10000 memories should suffice. This data structure is essential a deque that stores nodes (named tuples) with the following data: [“state”, “action”, “reward”, “next_state”, “done”]
- Initialize Local Model: A feed-forward NN that will estimate the action-value function.
- Initialize Target Model: A feed-forward NN that will estimate the target action-value function.

The implementation of the simple feed-forward model can be seen below. In addition, you can see the full implementation of the replay buffer here.

```python
import torch
import torch.nn as nn

class QNetwork(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):

        """


        Actor (Policy) Model

        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """

        # Inherit nn.Module as subclass
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.action_size = action_size
        self.state_size = state_size

         # Define layer parameters
        self.fully_connected_1 = nn.Linear(state_size, fc1_units)
        self.fully_connected_2 = nn.Linear(fc1_units, fc2_units)
        self.fully_connected_3 = nn.Linear(fc2_units, action_size)

    def foward(self, state):
        """
        Foward Pass that maps state -> action values.

        """
        X = nn.functional.relu(self.fully_connected_1(state))
        X = nn.functional.relu(self.fully_connected_2(X))
        return self.fully_connected_3(X)
```

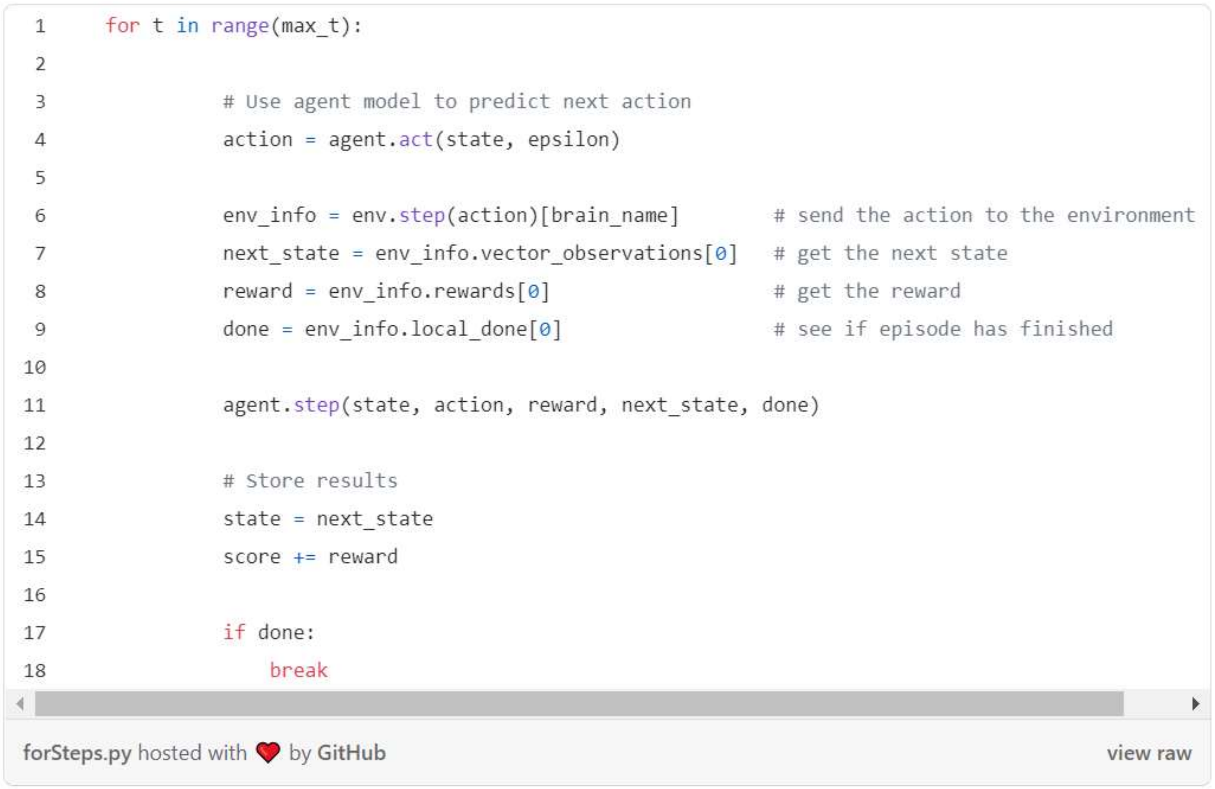
For episode e ⟵1 to M:

- Initialize the input frame x, a.k.a the initial state. Most OpenAI Gym environments will provide a simple API to grab a restated state.

```
for i_episode in range(1, n_episodes + 1):
     env_info = env.reset(train_mode=True)[brain_name]
     state = env_info.vector_observations[0]
```

For step t ⟵ 1 to T:

- We break this down into two parts: **SAMPLE** and **LEARN**



```
for t in range(max_t):

          # Use agent model to predict next action
          action = agent.act(state, epsilon)

          env_info = env.step(action)[brain_name]       # send the action to the environment
          next_state = env_info.vector_observations[0]  # get the next state
          reward = env_info.rewards[0]                  # get the reward
          done = env_info.local_done[0]                 # see if episode has finished

          agent.step(state, action, reward, next_state, done)

          # Store results
          state = next_state
          score += reward

          if done:
              break
```

- Sample Step:

1. Choose action A from state S using policy π ⟵ ε-Greedy(q(S, A, w)).

2. Take Action A, observe reward R, and next input frame X{t+1}

3. Prepare next state and store experience tuple in Replay Buffer

4. Set S ⟵ S’

```
def step(self, state, action, reward, next_state, done):
        """

        Process a step from time step t to t+1 by updating agent models.

        Params
        ======
            state (continious): current state before action
            action (discrete): action take for given state
            reward (int): reward recieved after performing action
            next_state (int): state achieved at timestep t+1
            done (bool): episode completed on this timestep

        """

        self.memory.add(state, action, reward, next_state, done)

        # Increase counter until we are ready to take an update step
        self.t_step = (self.t_step + 1) % UPDATE_EVERY

        if self.t_step == 0:
            # If enough samples are available in memory, get random subset
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)
```

- Learn Step:

1. Obtain random minibatch of tuples (“state”, “action”, “reward”, “next_state”, “done”) from Replay Buffer
2. Set TD target and perform the update step on the local model

$$\Delta w = \alpha \cdot \overbrace{(\underbrace{R + \gamma \max_a \hat{q}(S', a, w^-)}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{old value}})}^{\text{TD error}} \nabla_w \hat{q}(S, A, w)$$

Update step on weights

3. Perform a soft update step of the target model by factoring in a portion of the local model’s parameters

```python
def learn(self, experiences, gamma):
    """

    Update value parameters using given batch of experience tuples.

    Params
    ======
        experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done) tuples
        gamma (float): discount factor

    """

    # Note: These tensors make up a batch of 64 experiences

    states, actions, rewards, next_states, dones = experiences

    # Get max predicted Q values (for next states) from target model
    q_prime = self.qnetwork_target.foward(next_states)
    # Choose the reward from action that gives max return
    q_prime = q_prime.detach().max(1)[0].unsqueeze(1)

    not_done_bool = (1 - dones) # If done, no need to include next return
    td_target = rewards + (gamma * q_prime) * not_done_bool

    # This is the model we will update
    q_expected = self.qnetwork_local.foward(states)
    # Gathers the expected values for each action
    q_expected = q_expected.gather(1, actions)

    # Compute the loss, minimize the loss
    loss = F.mse_loss(td_target, q_expected)
    self.optimizer.zero_grad() # reset gradient
    loss.backward() # Calculate the gradient
    self.optimizer.step()  # Update weights

    # Update the target model parameters (Soft Update)
    # Soft Update: Factor in local parameter changes by a factor of TAU
    # Rather than update for every C steps, this helps inch closer to local parameters

    for target_param, local_param in zip(self.qnetwork_target.parameters(),
                                         self.qnetwork_local.parameters()):

        upd_wghts = ((1.0 - self.tau) * target_param.data) + (self.tau * local_param.data)
        target_param.data.copy_(upd_wghts)
```

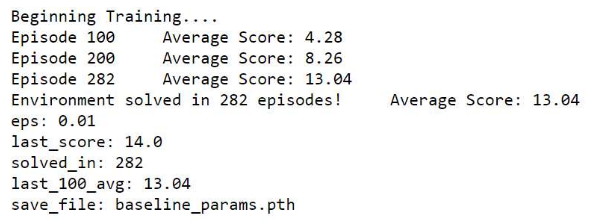
Yeah, so that's it! The model is able to learn over time and develop the ability to collect yellow bananas while avoiding the blue bananas. The full repository can be viewed here.

The application of this project is targeted for, while not limited to, robotic navigation. Some ideas that one could build off of this project are Garbage Collector Robots and Agricultural Robots (Muti-purpose)!

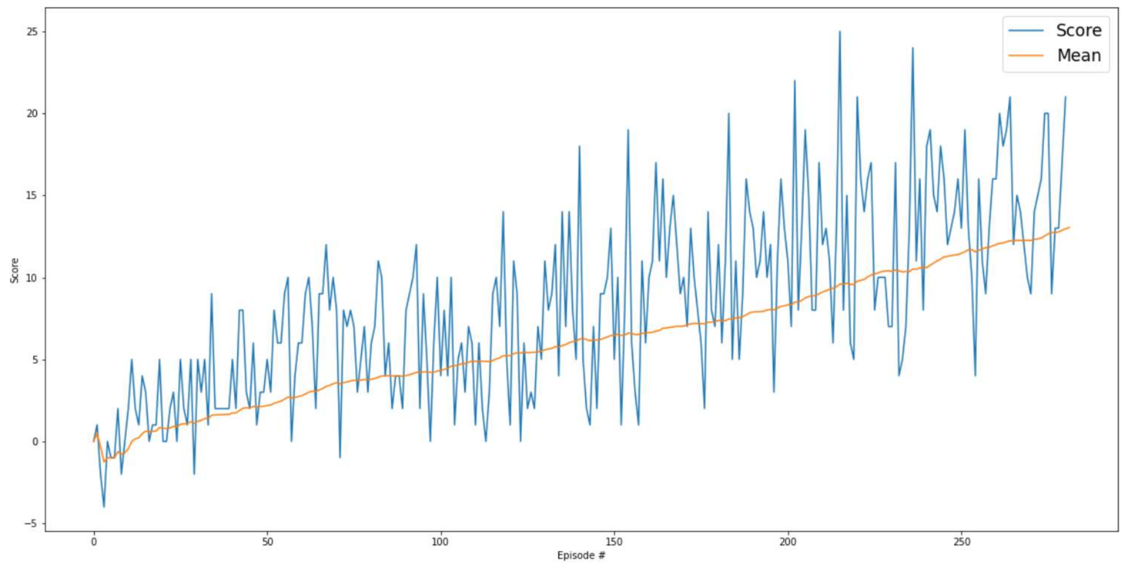
Plot of Results

Training Parameters:

- Max Episodes: 2000
- Max Time Steps: 1000
- Buffer Size: 10000
- Batch Size: 64
- Gamma: 0.99
- Tau: 1e-3
- Learning Rate = 5e-4
- Epsilon (Start, End, Decay Rate): [0.10, 0.01, 0.987]



```
Beginning Training....
Episode 100     Average Score: 4.28
Episode 200     Average Score: 8.26
Episode 282     Average Score: 13.04
Environment solved in 282 episodes!     Average Score: 13.04
eps: 0.01
last_score: 14.0
solved_in: 282
last_100_avg: 13.04
save_file: baseline_params.pth
```

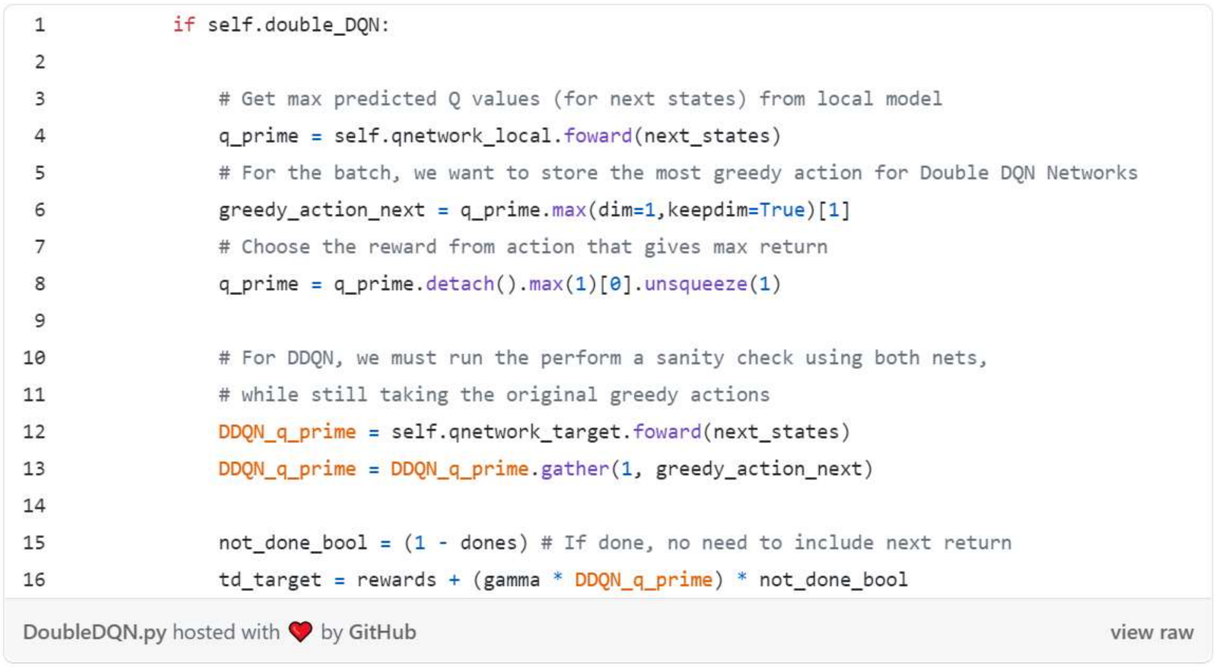


Going Further…

Several improvements to the original Deep Q-Learning algorithm have been suggested.

1. **Double DQN**

Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this. The main idea is to gain a second opinion on the target action-value (TD Target) by running the state through the local model first, then the target model. See the changes below:



```
        if self.double_DQN:

            # Get max predicted Q values (for next states) from local model
            q_prime = self.qnetwork_local.foward(next_states)
            # For the batch, we want to store the most greedy action for Double DQN Networks
            greedy_action_next = q_prime.max(dim=1,keepdim=True)[1]
            # Choose the reward from action that gives max return
            q_prime = q_prime.detach().max(1)[0].unsqueeze(1)

            # For DDQN, we must run the perform a sanity check using both nets,
            # while still taking the original greedy actions
            DDQN_q_prime = self.qnetwork_target.foward(next_states)
            DDQN_q_prime = DDQN_q_prime.gather(1, greedy_action_next)

            not_done_bool = (1 - dones) # If done, no need to include next return
            td_target = rewards + (gamma * DDQN_q_prime) * not_done_bool
```

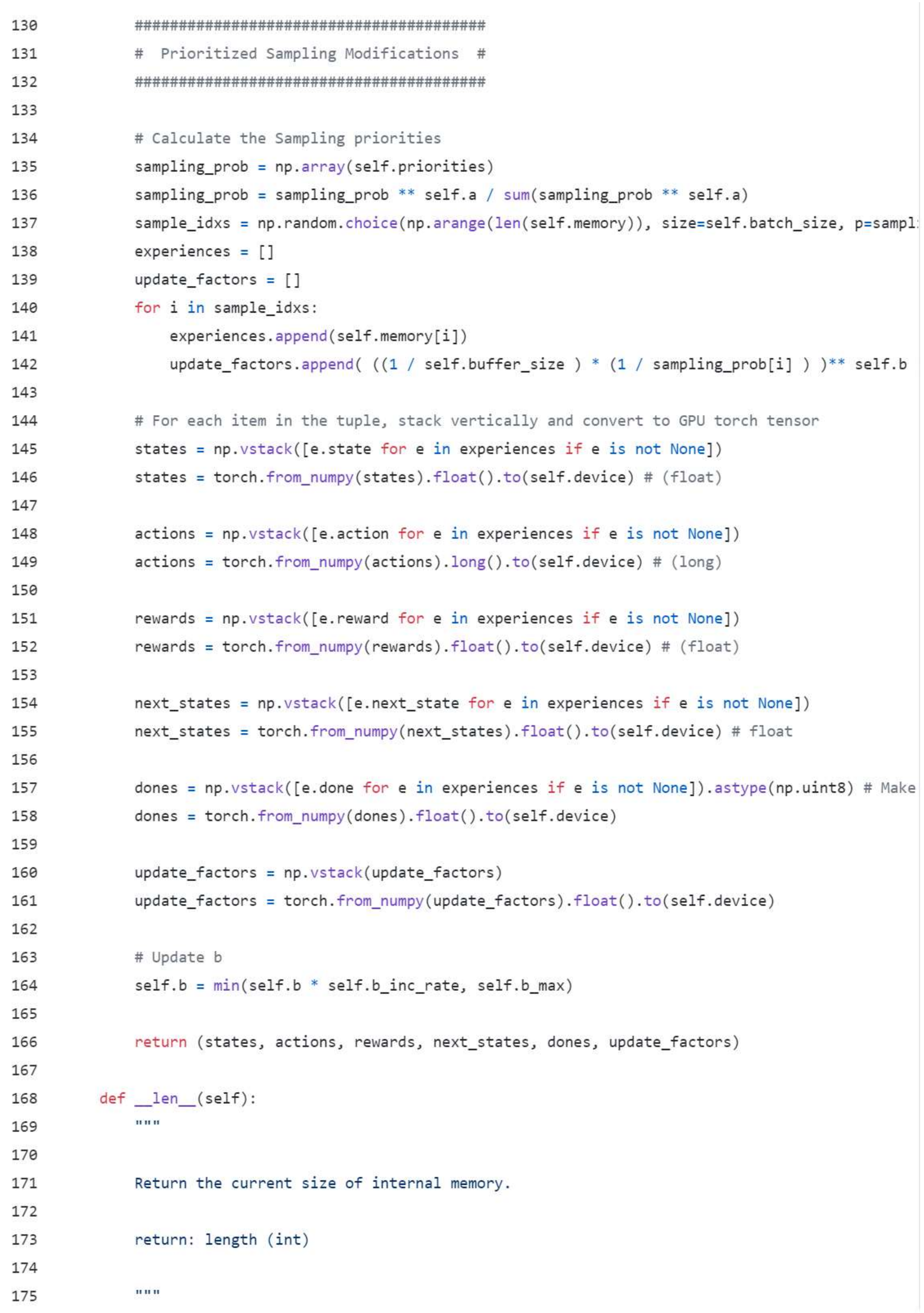
DoubleDQN.py hosted with ❤ by GitHub view raw

Double DQN Changes

2. Prioritized Experience Replay

Deep Q-Learning samples experience transitions *uniformly* from a replay memory. Prioritized experience replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability. By assigning priority we can steer the agent towards understanding the more important experiences.

```python
import numpy as np
import torch
import random
from collections import namedtuple, deque

version = 1.0
EXP_FIELD_NAME = ["state", "action", "reward", "next_state", "done", "prob"]

class PrioritizedReplayBuffer:
    """
    ReplayBuffer (Data Structure): Fixed-size buffer to store experience tuples.
    Experience (Data): NamedTuple with structure of
                       ["state", "action", "reward", "next_state", "done"]
    """

    def __init__(self,
                 action_size,
                 buffer_size,
                 batch_size,
                 seed,
                 device,
                 prioritized_params):
        """
        Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
            device (string): compute resource
            prioritized_params (dict): contains the constants for hyperparameters
              α (int): α = 0 corresponding to the uniform case, α = 1 to using priorities
              b (int): b is the bias used to scale the change in weights for importance-sampling

        """

        self.action_size = action_size
        self.buffer_size = buffer_size
        self.batch_size = batch_size
        self.seed = seed
        self.device = device
        self.a = prioritized_params['a']
        self.b = prioritized_params['b']
        self.b_inc_rate = prioritized_params['b_inc_rate']
        self.e = prioritized_params['e']
        self.b_max = 1.0
        self.max_prob = 1.0

        # Memories will be stored on a double-ended queue (deque)
        self.memory = deque(maxlen=self.buffer_size)
        # Now we have to keep track of priorities as well
        self.priorities = deque(maxlen=self.buffer_size)

        # Experiences will be a named tupile structure
        self.experience = namedtuple("Experience", field_names=EXP_FIELD_NAME)



    def get_td_errors(self, state, action, reward, next_state, done, models):

        """ Calculate td error."""
        models['local'].eval() # Evaluation Mode
        with torch.no_grad(): # No Gradient Descent

            # Make 64 copies to fit into model
            states = []
            rewards = []
            next_states = []
            dones = []

            for i in range(self.batch_size):
                states.append(state)
                rewards.append(reward)
                next_states.append(next_state)
                dones.append(done)

            states = torch.from_numpy(np.vstack(states)).float().to(self.device)
            q_expected = models['local'].foward(states)

            next_states = torch.from_numpy(np.vstack(next_states)).float().to(self.device)
            q_prime = models['target'].foward(next_states)
            rewards = torch.from_numpy(np.vstack(rewards)).float().to(self.device)
            dones = torch.from_numpy(np.vstack(dones)).float().to(self.device)
            not_done_bool = (1 - dones) # If done, no need to include next return
            td_target =  rewards + (models['GAMMA'] * q_prime) * not_done_bool

            td_error = td_target - q_prime

        models['local'].train()

        return abs(td_error.detach()[0][action])


    def add(self, state, action, reward, next_state, done, models):
        """

        Append an experience to the memory.

        Params
        ======
            state (continious): current state before action
            action (discrete): action take for given state
            reward (int): reward recieved after performing action
            next_state (int): state achieved at timestep t+1
            done (bool): episode completed on this timestep

        """
        # Create an experience

        #Calculate the td_error
        td_error = self.get_td_errors(state, action, reward, next_state, done, models)
        td_error += self.e
        exp = self.experience(state, action, reward, next_state, done, td_error)
        self.memory.append(exp)
        self.priorities.append(td_error)


    def sample(self):
        """

        Randomly sample an experience from the memory.

        return: experience (tuple)

        """
```

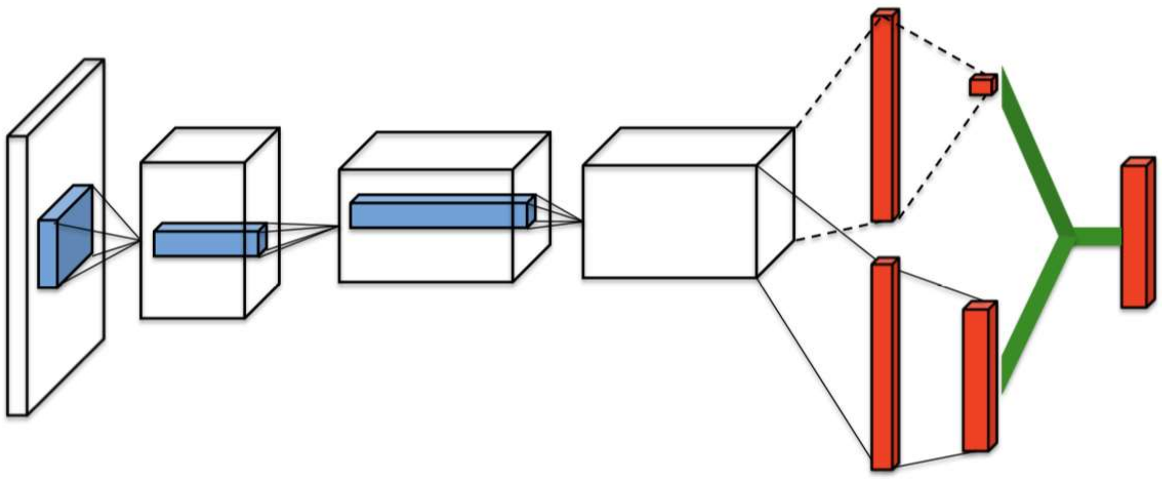


```
        ########################################
        #  Prioritized Sampling Modifications  #
        ########################################

        # Calculate the Sampling priorities
        sampling_prob = np.array(self.priorities)
        sampling_prob = sampling_prob ** self.a / sum(sampling_prob ** self.a)
        sample_idxs = np.random.choice(np.arange(len(self.memory)), size=self.batch_size, p=sampl
        experiences = []
        update_factors = []
        for i in sample_idxs:
            experiences.append(self.memory[i])
            update_factors.append( ((1 / self.buffer_size ) * (1 / sampling_prob[i] ) )** self.b

        # For each item in the tuple, stack vertically and convert to GPU torch tensor
        states = np.vstack([e.state for e in experiences if e is not None])
        states = torch.from_numpy(states).float().to(self.device) # (float)

        actions = np.vstack([e.action for e in experiences if e is not None])
        actions = torch.from_numpy(actions).long().to(self.device) # (long)

        rewards = np.vstack([e.reward for e in experiences if e is not None])
        rewards = torch.from_numpy(rewards).float().to(self.device) # (float)

        next_states = np.vstack([e.next_state for e in experiences if e is not None])
        next_states = torch.from_numpy(next_states).float().to(self.device) # float

        dones = np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8) # Make
        dones = torch.from_numpy(dones).float().to(self.device)

        update_factors = np.vstack(update_factors)
        update_factors = torch.from_numpy(update_factors).float().to(self.device)

        # Update b
        self.b = min(self.b * self.b_inc_rate, self.b_max)

        return (states, actions, rewards, next_states, dones, update_factors)

    def __len__(self):
        """

        Return the current size of internal memory.

        return: length (int)

        """
```

PER Replay Buffer Implementation

As you can see the implementation is more involved than the previous replay buffer. It is important to note that prioritized experience replay adds additional time and space complexity to the algorithm.

3. Dueling DQN



Dueling DQN

Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action values *for each action*. However, by replacing the traditional Deep Q-Network (DQN) architecture with a dueling architecture, we can assess the value of each state, without having to learn the effect of each action. By using two streams (value and advantage), we can gain a better estimate of the action-value.

Here I have implemented my own version of the dueling network with the help of Chris Yoon's post, feel free to take a look.

```python
class DuelingQNetwork(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64, fc3_units=32):


        """

        Actor (Policy) Model

        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
            fc2_units (int): Number of nodes in thirs hidden layer
        """

        # Inherit nn.Module as subclass
        super(DuelingQNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)

        self.feauture_layer = nn.Sequential(
            nn.Linear(state_size, fc1_units),
            nn.ReLU(),
            nn.Linear(fc1_units, fc2_units),
            nn.ReLU()
        )

        self.value_stream = nn.Sequential(
            nn.Linear(fc2_units, fc3_units),
            nn.ReLU(),
            nn.Linear(fc3_units, action_size),
        )

        self.advantage_stream = nn.Sequential(
            nn.Linear(fc2_units, fc3_units),
            nn.ReLU(),
            nn.Linear(fc3_units, action_size),
        )



    def foward(self, state):
        """
        Foward Pass that maps state -> action values.

        """
        # Foward Pass
        features = self.feauture_layer(state)

        # Value Stream - State Values
        values = self.value_stream(features)
        # Advantage Stream - Advantage Values
        advantages  = self.advantage_stream(features)

        # Q Estimate = V(state) + (A(state) - mean(A(state))
        qvals = values + (advantages - advantages.mean())

```

Ideas for Future Work

The next step for this project would be make the priority replay buffer more efficient. When training the algorithm, it seemed that the training time took too long. In reality, we want to have efficient algorithms in order to perform real-time computations. Maybe we could write the implementation in C++ or use a more efficient method of estimating priority.