Name: Oluwaseyi Awoga

Program: Udacity Deep Reinforcement Learning Nanodegree Program

Topic: Using Deep Q-Networks to Train an Agent to Navigate the Unity-ML Banana Environment.

Introduction & Literature Review:

Deep Q-learning is the combination of the Q-learning process with a function approximation technique such as a neural network. According to Zai & Brown (2020) [6], the main idea behind Q-learning is to use an algorithm to predict a state-action pair, and then to compare the results generated from this prediction to the observed accumulated rewards at some later time. The parameters of the algorithms are then updated so that it makes better predictions next time. While this technique has some advantages that make it very useful for solving reinforcement learning problems, it also falls short for solving complex problems with large state-space.

Palmas et al (2020) [1], noted that the following disadvantages make the use of the tabular Q-learning paradigm less than ideal for solving complex problems with large observation space.

- **Performance issues:** When the state spaces are very large, the tabular iterative lookup operations are much slower and more costly.
- **Storage issue**: Along with performance issues, storage is also costly when it comes to storing the tabular data for large combinations of state and action spaces.
- The tabular method will only work well only when an agent comes across seen discrete states that are present in the Q-table. For the unseen states that are not present in the Q-table, the agent's performance may not be the optimal performance.
- For continuous state spaces for the previously mentioned issues, the tabular Q-learning method won't be able to approximate the Q-values in an efficient or proper manner.

In fact, Google DeepMind (2015) [3] supported the above conclusion in its seminal paper entitled "Human-level control through deep reinforcement learning". In this paper, Minh et al (2015) [3] asserted that *"to use reinforcement learning successfully in situations approaching real-world complexity, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experiences to new situations".* To achieve this objective they stated further, *"we developed a novel agent, a deep Q-network (DQN), which is able to combine reinforcement learning with a class of artificial neural network known as deep neural networks".*

While the use of Q-learning for solving reinforcement learning problems has enjoyed some remarkable successes, in the past it was not until the introduction of DQN that they were able to be used to solve large-scale problems. Prior to that, reinforcement learning was limited to *"applications and domains in which useful features could be handcrafted, or to domains with fully observed, low-dimensional state spaces"*, Minh et al (2015) [3] argued further.

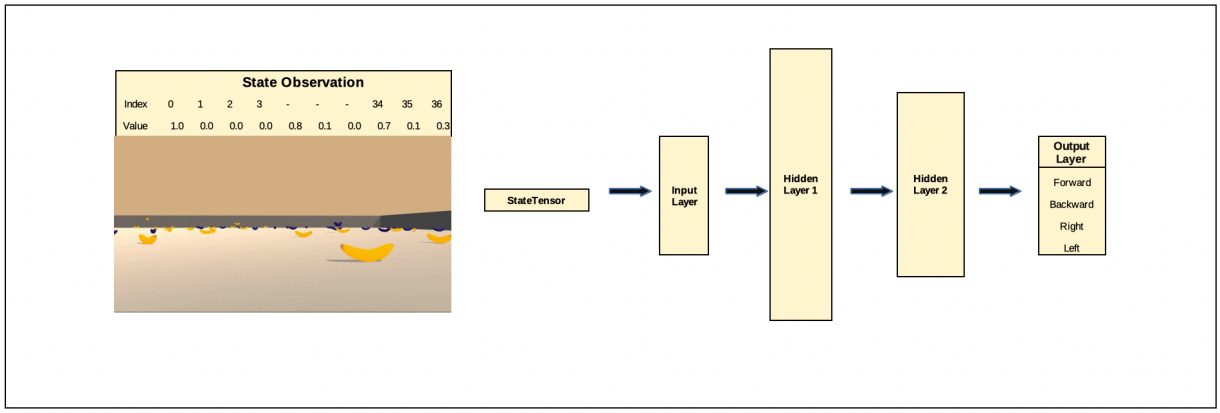
Intuition Behind Deep Q-Networks

Generally speaking, the aim of a reinforcement learning agent is to act in a manner that maximizes the expected return future reward from navigating an environment.

$$Q^*(s,a) = max \in [r_t + \gamma r_{t+1} + \gamma 2\, r_{t+2} + \ldots |s_t = s,\, a_t = a,\, \pi],$$

which is the maximum sum of rewards r_t discounted by γat each time-step t, achievable by a behavior policy $\pi = P(a|s)$,after making an observation $(s)$and taking an action (a) according to Google DeepMind (2015) [3].

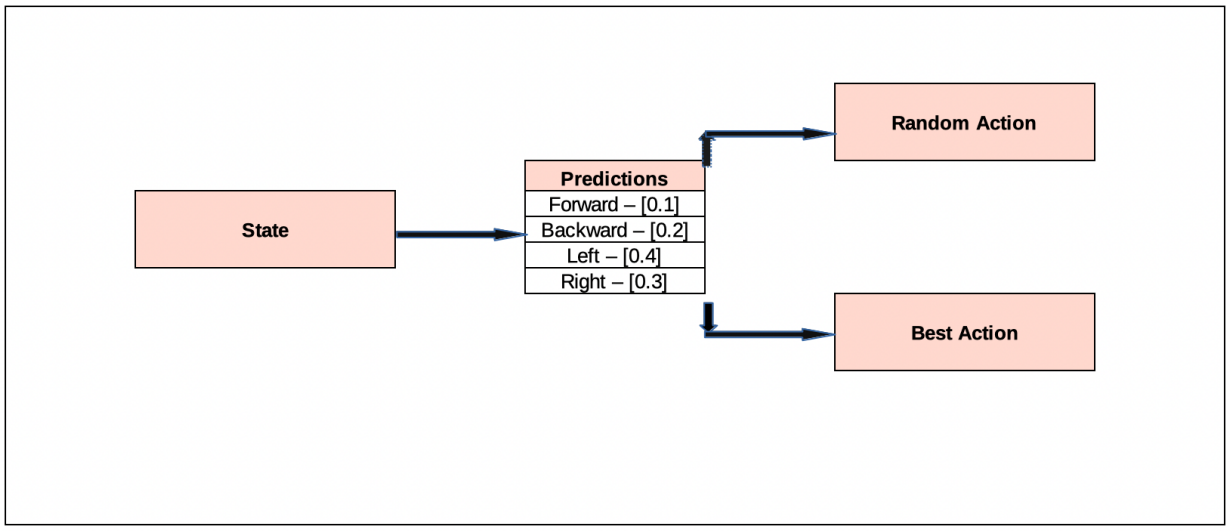
To achieve this in a DQN paradigm, Cai, Bileschi, Nielsen & Chollet (2020) [4], the deep neural network is pressurized to match the Bellman Equation. Palmas et al (2020) [1] stated that the Q function accepts a state and returns a vector of state-action values, one for each possible action given the input state.



For instance, in the above table, the DQN takes as input the state-observation space of an environment and returns a vector or list of state-action values or Q-values. The vector contains the expected value of each possible action in that particular environment, namely forward, backward, left, and right in this case.

Palmas et al (2020) [1] stated further to avoid the exploration-exploitation dilemma, the DQN

algorithm uses the epsilon-greedy action selection method, in which the epsilon parameter is set to some decimal value e.g. 0.1, and with a probability, we randomly select an action without considering the Q-values returned by the DQN and with a probability of 1- epsilon, we select the action associated with the maximum or highest Q-value. Alternatively, one could start with an epsilon of 1 and incrementally decrease this to a very low value. This approach will ensure that the agent explores a lot in the beginning and exploits more after the initial learning phase.

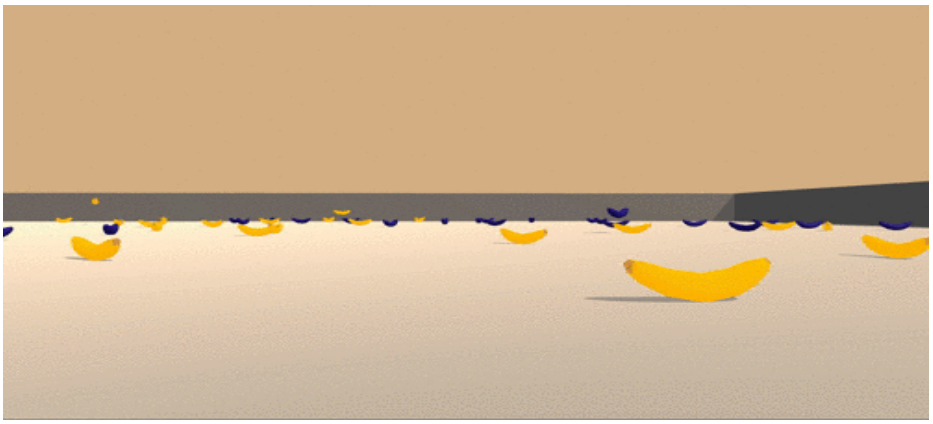


The difference between the Q-value predicted by the DQN and a target Q-value created by another network after applying the Bellman equation is used to determine the loss. This loss is then minimized using backpropagation and gradient descent. Essentially, reducing the problem to a supervised learning problem of the linear type. These concepts will be explained in detail later but first, let us explore our environment.

The Unity-ML Banana Environment

For this project, we will be training an agent to navigate the Unity_ML Banana environment and collect bananas in a large square world. The agent earns a reward of +1 for collecting a yellow banana. Conversely, the agent incurs a cost of -1 for collecting blue bananas. The objective of the agent is to maximize its total rewards or put another way to collect as many yellow bananas as possible.

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. Four (4) discrete actions are available, corresponding to (a) forward move (b) backward move (c) right move, and (d) left move.



The abstract reinforcement learning canonical representation of the environment is shown below:

Abstract RL Concept	Realization in the Unity-ML Banana Environment
Environment	A large square world with yellow and blue bananas.
Agent	Discrete Choice: The agent can move forward, backward, right and left
Reward	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 while avoiding blue bananas.
Observation	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. Given this information, the agent has to learn how to best select actions.

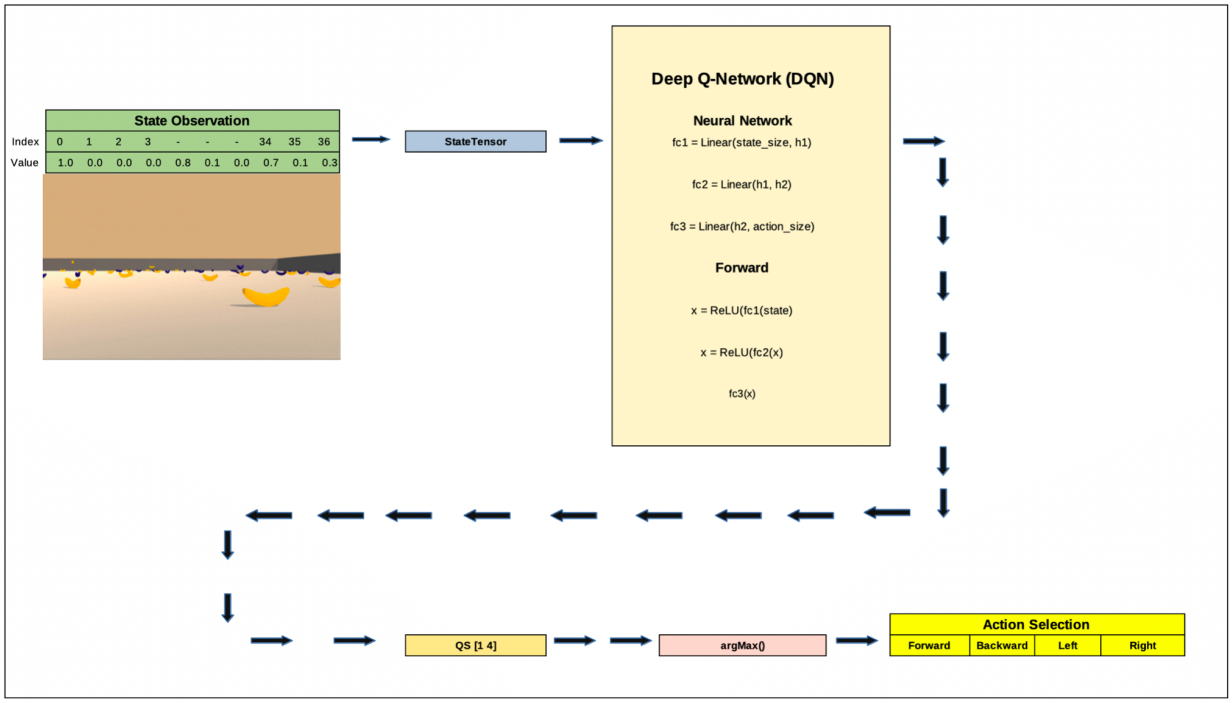
The Deep Q-learning Process

Although we have already touched on some of the deep Q-learning process above, let us go over the process again albeit slightly differently to further explain out the process works. This is because seeing the process presented in different ways may further our understanding of the topic. According to Palmas et al 2020 [2] in a book entitled "The Reinforcement Learning Workshop" [Pages 514-515], the following are the break downs of the deep Q-learning process:

- **Inputs to the DQN:** The neural network accepts the states of the environment as input. A state can be a simple coordinate of a grid for example. In the case of the Unity-ML Banana environment, this consists of an an array of length 37. Palmas et al (2020) [1], noted that in more complex games such as Atari, the input can be a few consecutive snapshots of the screen in the form of an image as state representation. The number of

nodes in the input layer will be the same as the number of states present in the environment.

- **Outputs from the DQN:** The output from the DQN would be the Q values for each action possible given the environment. For example, for any given environment, if there are four possible actions, then the output would have four Q values for each action. To choose the optimal action, we will select the action with the maximum Q value using the argmax function. In the case of the Unity-ML Banana environment, the possible actions are the backward, forward, left and right moves.



The chart above depicts the input into and output from our environment. The input consists of an array of of length 37 which is first converted to a tensor using Pytorch functionalities. The tensor is then passed into a DQN. The output is an array of length four and the content of that array are the Q values. Finally, we use the argmax function to retrieve the action with the maximum Q value.

- **The Loss Function and Learning Process:** The DQN will accept states from the environment and, for each given input or state, the network will output an estimated Q value for each action. The objective of this is that it approximates the optimal Q value, which will satisfy the right-hand side of the Bellman equation, as shown in the expression below:

$$Q(s,a) = r + \gamma maxQ(s',a')$$

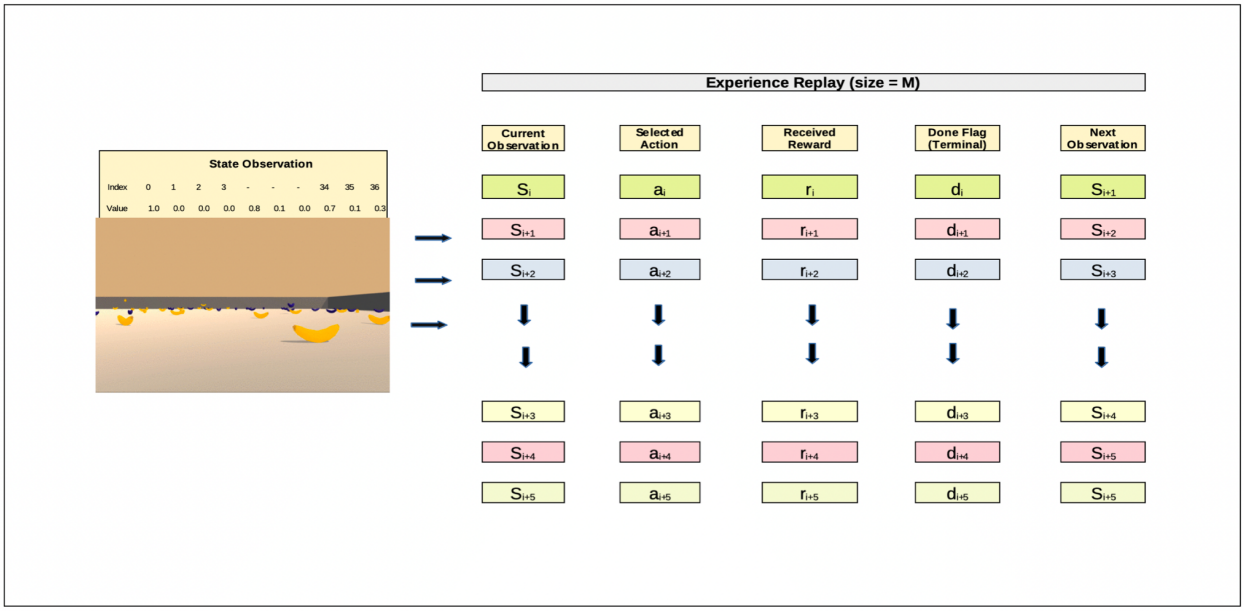
However, according to Google DeepMind (2015) [3], *"reinforcement learning is known*

to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function. It stated further that, this instability has several causes: the correlations present in the sequence of observsations, the fact that small updates to Q may significantly change the policy and therefore change the data distributions, and the correlations between the action-values (Q) and the target values $r + \gamma \, max \, Q(s', a')$".

To address the challenges associated with correlations between steps and the convergence issues, DeepMind used a concept known as experience replay. It stated that *"we parametize an approximate value function* $Q(s, a; \theta i)$ *using the deep convolutional neural networks, in which* θi *are the parameters (that is weights) of the Q-network at iteration* i*. To experience replay we store the agent's experiences* $e_t = (s_t, a_t, r_t, s_{t+1})$ *at each time-step t in a data set* $D_t = \{e_1 \ldots\ldots e_t\}$*. During learning, we apply Q-learning updates, on samples (or minibatches) of experience* $(s, a, r, s') - U(D)$*, drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration* i *uses the following function".*

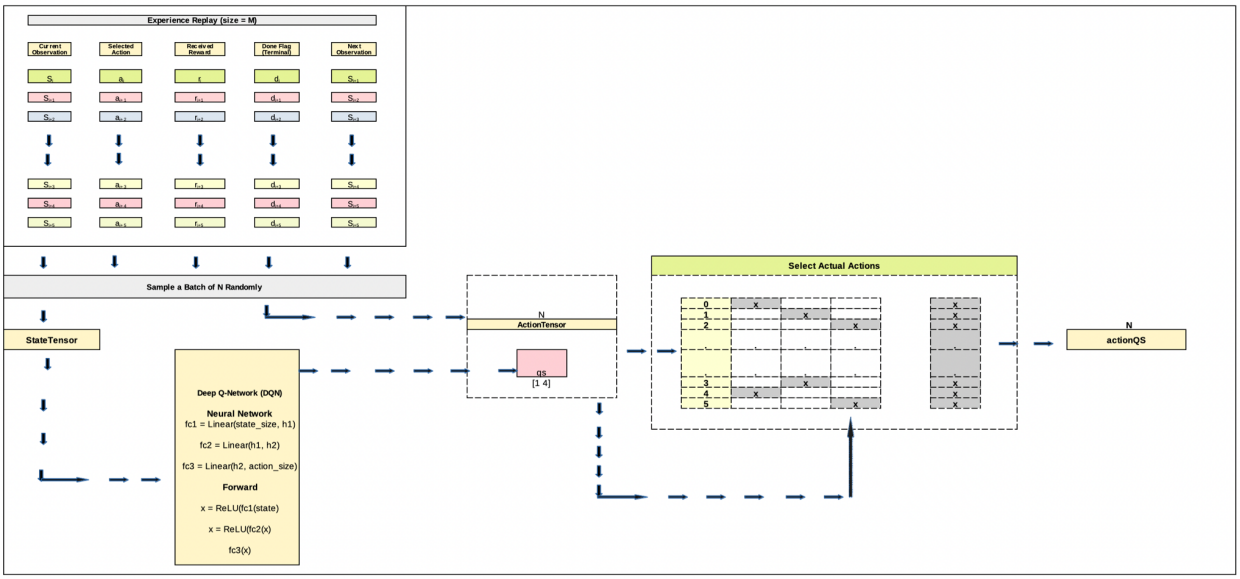
$$L_i Q_i = E_{(s,a,r,s')} - U(D) = [(r + \gamma maxQ(s', a': \theta_i') - Q(s,a:\theta_i))^2]$$

in which $y$is the discount factor determining the agent's horizon, θ_i are the parameters of the Q-network at iteration i and θ'_i are the network parameters used to compute the target at iteration.

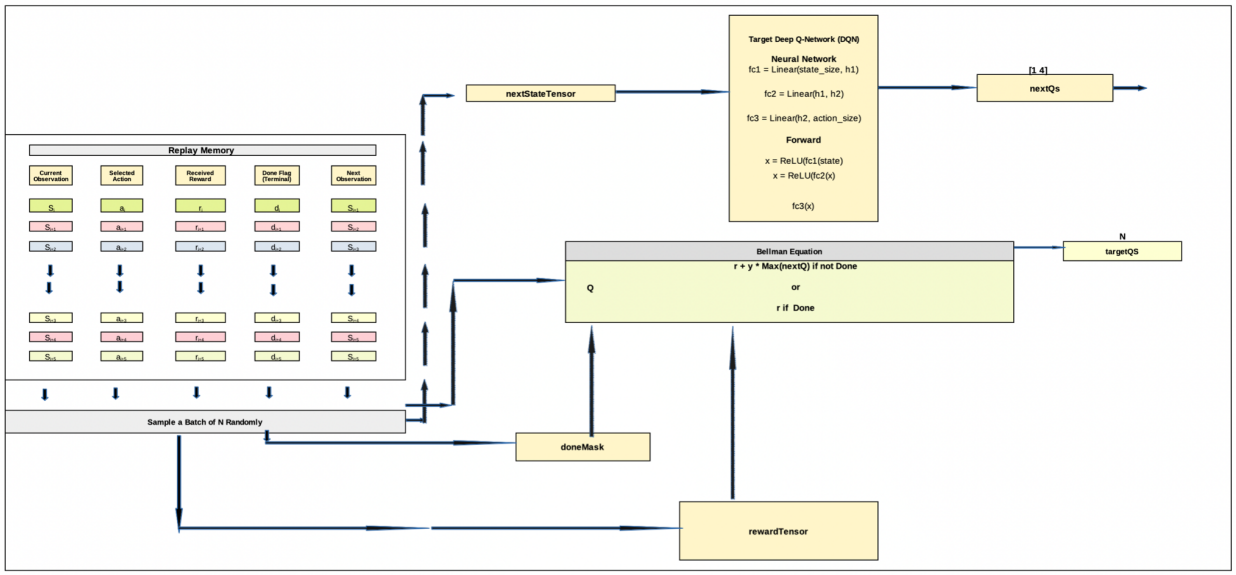


During learning, Q-learning updates is applied on samples (or minibatches) of experience $(s, a, r, s') \sim U(D)$, drawn uniformly at random from the pool of stored samples.

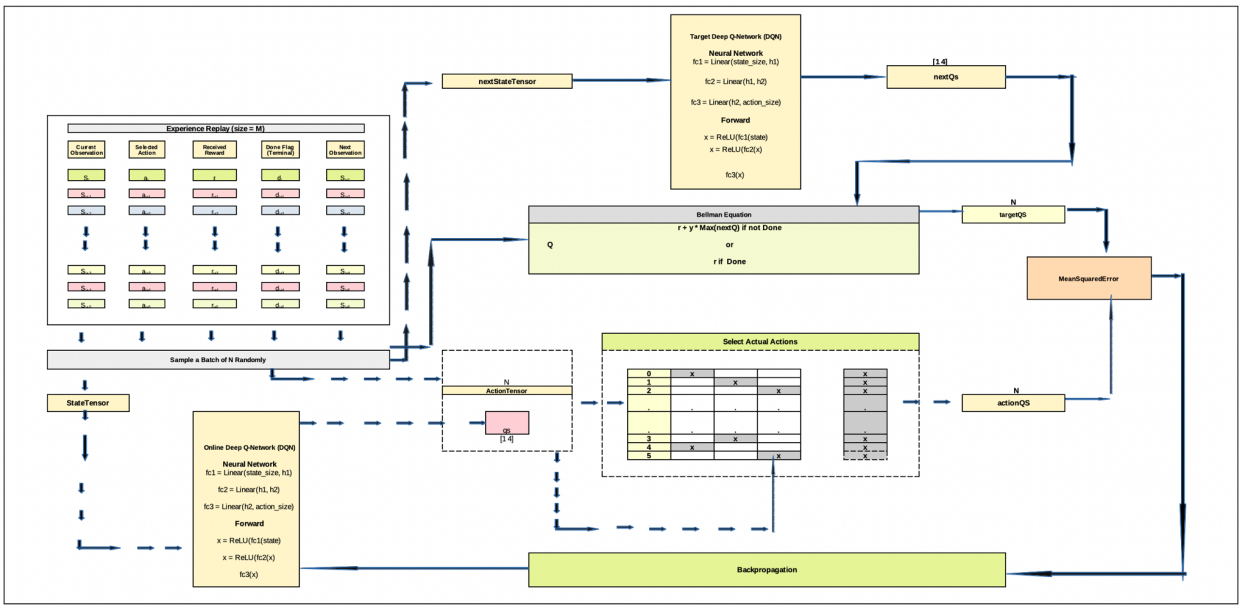
Palmas et al 2020 [2], noted that to calculate the loss, we need target Q values and the Q values coming from the network. The Q values which represent our dependent variables are coming from the online network which is separate and distinct from the target network.



Google DeepMind [2015], stated that target network parameters θ'_i are only updated with the Q-network parameters θ_i, every C steps and are held fixed between individual updates. The target Q values are then calculated using the Bellman equation.



The loss from the DQN is calculated by comparing the output Q values from the DQN to the target Q values. Once the loss is calculated the weights of the DQN are updated via backpropagation to minimize the loss. This process is depicted in the chart below:



Implementation of the Deep Q-learning Process in PyTorch

Having established sufficient background about the workings of DQN, it is now pertinent to implement it in Python.

1. Import required libraries and packages:

```
from unityagents import UnityEnvironment
import numpy as np
import pprint as pp
import gym
import math
import random
import torch
import torch.nn as nn
from collections import namedtuple, deque
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Load the Unity-ML Banana environment. The code below assumes that the Unity ML Banana app has been downloaded (see the requirements.txt) and it is in the same directory from which this programmed is be called:

```
env = UnityEnvironment(file_name="Banana.app")
```

3. Get the environment information from the Unity-ML environment:

```
#Get the Default Brain
brain_name = env.brain_names[0]
print(brain_name)
brain = env.brains[brain_name]
print(brain)

env_info = env.reset(train_mode=True)[brain_name]
print(env_info)

#Output:

BananaBrain
Unity brain name: BananaBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 37
        Number of stacked Vector Observation: 1
        Vector Action space type: discrete
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,

<unityagents.brain.BrainInfo object at 0x7fec01cc3048>
```

4. Retrieve information about the observation state-space and number of actions from the environment:

```
env_info.vector_observations.shape
state = env_info.vector_observations.shape
print(state)

# Fetching the number of states and actions
number_of_states = brain.vector_observation_space_size
number_of_actions = brain.vector_action_space_size
number_of_agents = len(env_info.agents)

# checking the total number of states and action
print('Total number of States : {}'.format(number_of_states))
print('Total number of Actions : {}'.format(number_of_actions))
print('Total number of Agents : {}'.format(number_of_agents))

#Examine the State Space
state = env_info.vector_observations[0]
print("State Looks Like: ")
pp.pprint(state)

#Output:

Total number of States : 37
Total number of Actions : 4
Total number of Agents : 1
State Looks Like:
array([1.        , 0.        , 0.        , 0.        , 0.84408134,
       0.        , 0.        , 1.        , 0.        , 0.0748472 ,
       0.        , 1.        , 0.        , 0.        , 0.25755   ,
       1.        , 0.        , 0.        , 0.        , 0.74177343,
       0.        , 1.        , 0.        , 0.        , 0.25854847,
       0.        , 0.        , 1.        , 0.        , 0.09355672,
       0.        , 1.        , 0.        , 0.        , 0.31969345,
       0.        , 0.        ])
```

5. Select the available device from the environment. If a GPU is available the program uses a GPU, otherwise it defaults to CPU:

```
#Selecting the available device (cpu/gpu)
use_cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if use_cuda else "cpu")
print(device)

#Output
cpu
```

6. Initialize the weights to get an initial approximation of Q(s,a). The is the online DQN and it accepts the number of state size, action size and a random seed as input. This class outputs Q values for the number of actions present in the environment and has two hidden layers of size 64:

```
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
        """Initialize parameters and build 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
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

7. Create a DQN agent class and implement the constructor's __init__ function. This function will create an instance of the DQN class within which the hidden layer size is passed. It will also define the MSE as a loss criterion. Next define the Adam as the optimizer with model parameters and a predefined learning rate. Other methods within this class includes the step, act, learn and replaybuffer functions and they implement the architectures we discussed above.
8.

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_state, done)

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    def act(self, state, eps=0.):
        """Returns actions for given state as per current policy.

        Params
        ======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
```

```
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next =
self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        θ_target = τ*θ_local + (1 - τ)*θ_target
        Params
        ======
            local_model (PyTorch model): weights will be copied from
            target_model (PyTorch model): weights will be copied to
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(),
```

```
local_model.parameters()):
            target_param.data.copy_(tau*local_param.data +
(1.0-tau)*target_param.data)


class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """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
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action",
"reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is
not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is
not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is
not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences
if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not
None]).astype(np.uint8)).float().to(device)

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

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

9. Instantiate an instance of the the DQN agent as shown below:

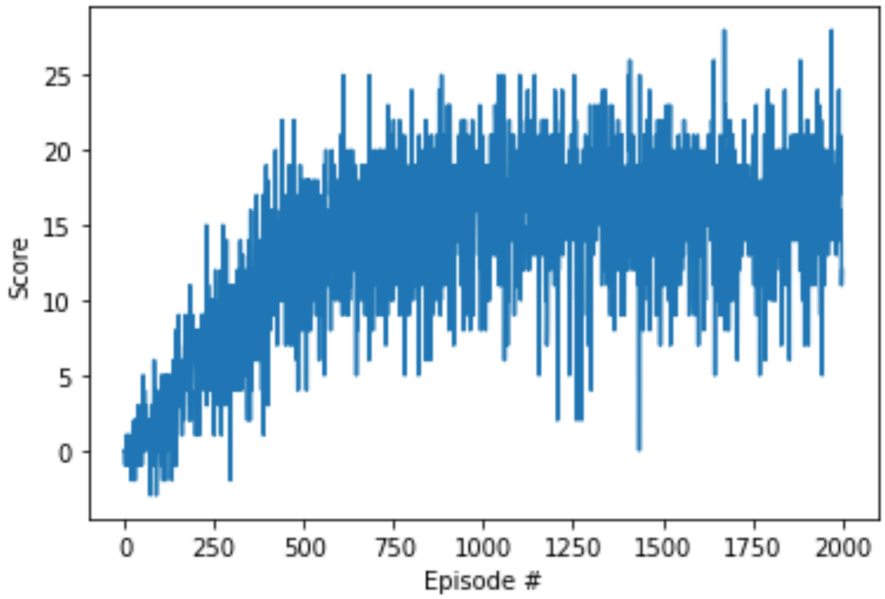
```
agent = Agent(state_size=37, action_size=4, seed=0)
```

10. Train the agent using the code block shown below:

```
def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action
selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing
epsilon
    """
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon
    for i_episode in range(1, n_episodes+1):
        state = env.reset(train_mode=True)[brain_name]
        state = state.vector_observations[0]
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
            env_info = env.step(action)[brain_name]
            next_state = env_info.vector_observations[0]
            reward = env_info.rewards[0]
            done = env_info.local_done[0]
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        scores_window.append(score)       # save most recent score
        scores.append(score)              # save most recent score
        eps = max(eps_end, eps_decay*eps) # decrease epsilon
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode,
np.mean(scores_window)), end="")
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode,
```

```
np.mean(scores_window)))
        if np.mean(scores_window)>=200.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score:
{:.2f}'.format(i_episode-100, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores

scores = dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()

#Output:

Episode 100  Average Score: 0.75
Episode 200  Average Score: 3.98
Episode 300  Average Score: 6.75
Episode 400  Average Score: 9.45
Episode 500  Average Score: 12.99
Episode 600  Average Score: 13.88
Episode 700  Average Score: 14.86
Episode 800  Average Score: 14.49
Episode 900  Average Score: 15.71
Episode 1000 Average Score: 15.91
Episode 1100 Average Score: 16.12
Episode 1200 Average Score: 16.96
Episode 1300 Average Score: 16.08
Episode 1400 Average Score: 16.20
Episode 1500 Average Score: 16.39
Episode 1600 Average Score: 15.44
Episode 1700 Average Score: 16.61
Episode 1800 Average Score: 15.72
Episode 1900 Average Score: 15.97
Episode 2000 Average Score: 16.53
```

11. Preview the chart generated by the codeblock above:



12. Close the Unity-ML Banana environment:

```
env.close()
```

Conclusion

In the paper, we examined DQN and discussed how it could be implemented to solved reinforcement learning problems that approaches human level complexity. We also discussed an implementation of the technique in Python. Our agent was trained using a real life implementation of the technique. The agent scored an average score of 16.53 over one hundred (100) episodes which is considered better than the benchmark score of 13.

We used the concept of experience replay and target networks to reduce the problems of correlation between steps and convergence issues. Yet the DQN technique may still suffer from other problems such overestimation due to overfitting. Therefore, other variants of the DQN technique, for example, the Double Deep Q-Network (DDQN) method could be explored to further improve the performance of our agent.

References

1. Alessandro Palmas, Emanuele Ghelfi et al (2020). "*The Reinforcement Learning Workshop"*. Packt Publication, United Kingdom.
2. Miguel Morales (2020). *"Grokking Deep Reinforcement Learning"*. Manning Publications, Shelter Island, New York, United States.

3. Minh et al (2015). *"Human-level control through deep reinforcement learning"*. Google DeepMind, London, United Kingdom.
4. Shanqing Cai, Stanley Bileschi, Eric Nielsen & Francois Chollet (2020). *"Deep Learning with Javascript"*. Manning Publications, Shelter Island, New York, United States.
5. Udacity (2020). *"Deep Reinforcement Learning Nanodegree Program - Class Resources - Videos & Lectures"*. Mountain View, United States of America.
6. Zai & Brown (2020). *"Deep Reinforcement Learning in Action"*. Manning Publications, Shelter Island, New York, United States.