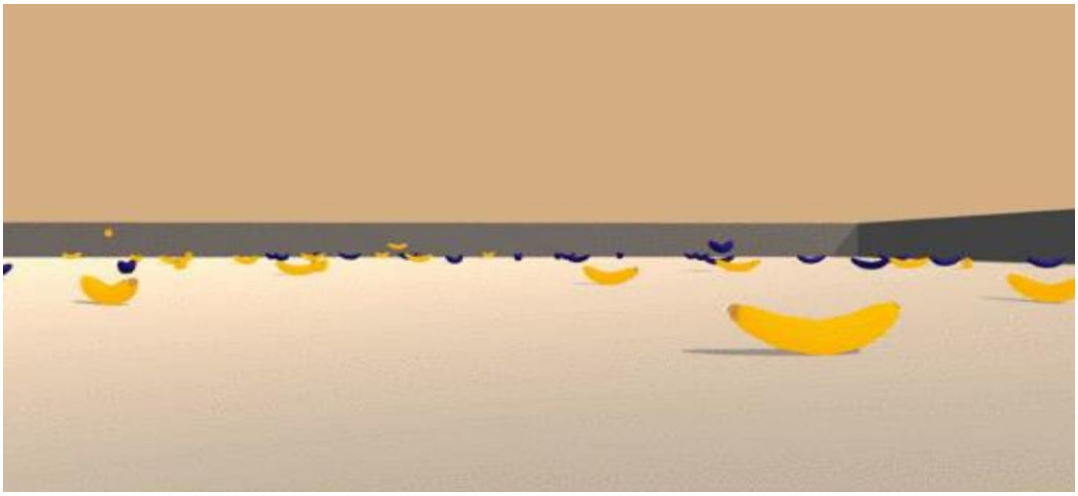
Project Description:

For this project, you will train an agent to navigate (and collect bananas!) in a large, square world.



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 your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

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. Four discrete actions are available, corresponding to:

0 - move forward.
1 - move backward.
2 - turn left.
3 - turn right.
The task is episodic, and in order to solve the environment, your agent must get an average score of +13 over 100 consecutive episodes.

Theory:

One of the early breakthroughs in reinforcement learning was the development of an off-

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

policy TD control algorithm known as Q-learning (Watkins, 1989), defined by

R_{t+1} → is the reward for next state.
Gamma → Is the discount factor
Alpha → Is the learning rate

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated.

In Q-Learning, we *update a guess with a guess*, and this can potentially lead to harmful correlations. To avoid this, we can update the parameters w in the network q^ to better approximate the action value corresponding to state S and action A with the following update rule:

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

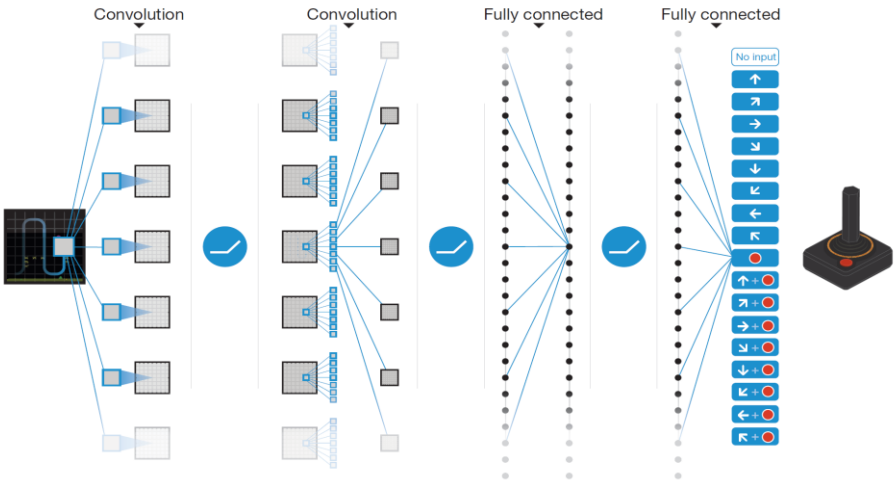
where w− are the weights of a separate target network that are not changed during the learning step, and (S, A, R, S′) is an experience tuple. Goal is to reduce TD Error.

In Deep Q Network, at the heart of the agent is a deep neural network that acts as a function approximator. You pass in images from your favorite video game one screen at a time, and it produces a vector of action values, with the max value indicating the action to take.

As a reinforcement signal, it is fed back the change in game score at each time step.
In the beginning when the neural network is initialized with random values, the actions taken are all over the place. It's really bad as you would expect, but overtime it begins to associate

situations and sequences in the game with appropriate actions and learns to actually play the game well.

Consider Atari games are displayed at a resolution of 210 by 160 pixels, with 128 possible colors for each pixel. This is still technically a discrete state space but very large to process as is. To reduce this complexity, the Deep Mind team decided to perform some minimal processing, convert the frames to gray scale, and scale them down to a square 84 by 84 pixel block.
Square images allowed them to use more optimized neural network operations on GPUs. In order to give the agent access to a sequence of frames, they stacked four such frames together resulting in a final state space size of 84 by 84 by 4.



On the output side, unlike a traditional reinforcement learning setup where only one Q value is produced at a time, the Deep Q network is designed to produce a Q value for every possible action in a single forward pass. Use this vector to take an action, either stochastically, or by choosing the one with the maximum value.

These innovative input and output transformations support a powerful yet simple neural network architecture under the hood. The screen images are first processed by convolutional layers. This allows the system to exploit spatial relationships, and can exploit spatial rule space.

Also, since four frames are stacked and provided as input, these convolutional layers also extract some temporal properties across those frames. The original DQN agent used three such convolutional layers with RLU activation, regularized linear units. They were followed by

one fully-connected hidden layer with RLU activation, and one fully-connected linear output layer that produced the vector of action values.

This same architecture was used for all the Atari games they tested on, but each game was learned from scratch with a freshly initialized network. Training such a network requires a lot of data, but even then, it is not guaranteed to converge on the optimal value function. In fact, there are situations where the network weights can oscillate or diverge, due to the high correlation between actions and states. This can result in a very unstable and ineffective policy.

In order to overcome these challenges, the researchers came up with several techniques that

slightly modified the base Q learning algorithm. Two of these techniques are experience **replay, and fixed Q targets.**

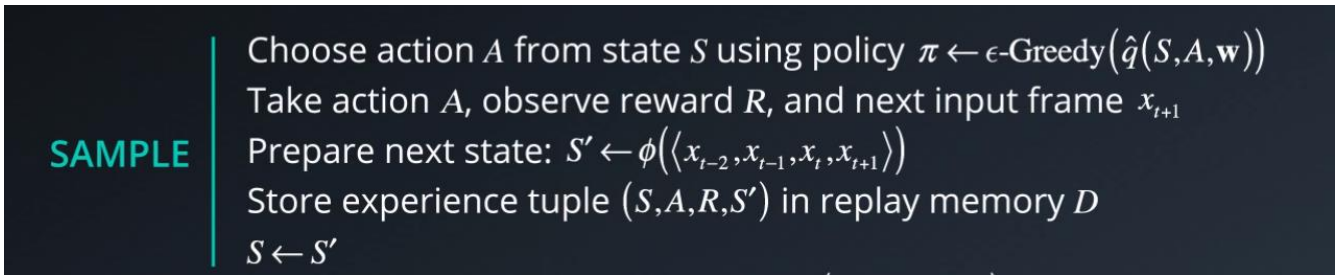
By keeping track of a **replay buffer** and using **experience replay** to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically.

The **replay buffer** contains a collection of experience tuples (S, A, R, S′). The tuples are gradually added to the buffer as we are interacting with the environment.

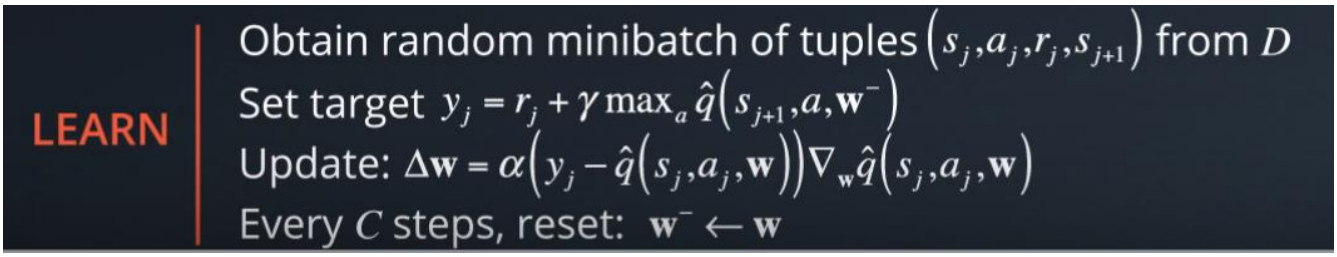
The act of sampling a small batch of tuples from the replay buffer in order to learn is known as **experience replay**. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Algorithm:

There are two main processes that are interleaved in this algorithm. One, is where we sample the environment by performing actions and store away the observed experienced tuples in a replay memory.



The other is where we select the small batch of tuples from this memory, randomly, and learn from that batch using a gradient descent update step.



These two processes are not directly dependent on each other. So, you could perform multiple sampling steps then one learning step, or even multiple learning steps with different random batches.

In the beginning you need to initialize an empty replay memory. Note that memory is finite, so you may want to use something like a circular Q that retains the N most recent experience tuples. Then, you also need to initialize the parameters or weights of your neural network. There are certain best practices that you can use, for instance, sample the weights randomly from a normal distribution with variance equal to two by the number of inputs to each neuron. These initialization methods are typically available in modern deep learning libraries like Keras and TensorFlow.

To use the fixed Q targets technique, you need a second set of parameters w^- which you can initialize to w. So, for each episode and each time step t within that episode you observe a raw screen image or input frame x_t .

Also, in order to capture temporal relationships, you can stack a few input frames to build each state vector. Let's denote this pre-processing and stacking operation by the function phi, which takes a sequence of frames and produces some combined representation. Note that if we want to stack say four frames will have to do something special for the first, three time steps. For instance, we can treat those missing frames as blank, or just used copies of the first frame, or we can just skip storing the experience tuples till we get a complete sequence.

In practice, you won't be able to run the learning step immediately. You will need to wait till you have sufficient number of tuples in memory. We do not clear out the memory after each episode, this enables us to recall and build batches of experiences from across episodes.

Implemented Architecture For Q-network:

Model.py contains the neural network architecture Dqn_agent.py uses pytorch to construct two neural networks for local and target Q functions both have the same structure. Input layer is composed of 37 neurons, two hidden layers has 64 neurons and final fully connected layer with output size equivalent to action size of 4.

```
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
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        "*** YOUR CODE HERE ***"
        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)
```

Training:

Method **def** dqn(agent,n_episodes=1800, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995,train=**True**) of Navigation.npyb is used for training the network.

1. Initial arguments:

```
n_episodes (int)=1800      (maximum number of training episodes)
max_t (int)=1000           (maximum number of timesteps per episode)
eps_start float)= 1        (starting value of epsilon, for epsilon-greedy action selection)
eps_end (float)=0.01       (minimum value of epsilon)
eps_decay (float)=0.995    (multiplicative factor (per episode) for decreasing epsilon)
```

2. We now initialize score and store last 100 score set epsilion value to the start value

```
    scores = []                                # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                            # initialize epsilon
```

3. For each episode reset the environment and get the current state

```
env_info = env.reset(train_mode=True)[brain_name]      # reset the environment
state = env_info.vector_observations[0]                # get the current state
```

4. For each time step within each episode get action value , send action to the environment, get next state, reward and check if the episode has finished . If episode has finished then break.

```
action = agent.act(state, eps)
env_info = env.step(int(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
```

5. Step the agent forward one step and store SARS' tuple for learning

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

6.Update total score for episode and for rolling 100 window

```
scores_window.append(score)      # save most recent score
```

```
scores.append(score)          # save most recent score
```

7. Next reduce the epsilion

```
eps = max(eps_end, eps_decay*eps) # decrease epsilon
```

Hyper-parameter:

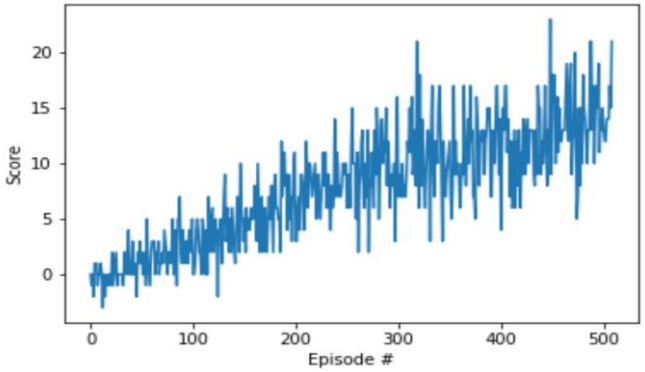
Hyper-parameters are configured in dqn_agent.py

```
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
```

Results:

Based on the architecture, hyper-parameters and training network algorithm reached a Average score of 13 at 409th episode.

```
Episode 100    Average Score: 1.16
Episode 200    Average Score: 4.77
Episode 300    Average Score: 8.63
Episode 400    Average Score: 10.84
Episode 500    Average Score: 12.81
Episode 509    Average Score: 13.00
Environment solved in 409 episodes!    Average Score: 13.00
```



Future Ideas:

As a future improvements we can implement several optimization techniques such as :

Reward clipping, error clipping, storing past actions as part of the state vector, dealing with terminal states, digging epsilon over time. Optimizing hyper-parameter including the decay rate and starting epsilon.

Also, it will be good to implement Double DQN, Prioritized Experience Replay and Dueling DQN as:

Double DQN which can eliminate overestimation by Deep Q Learning.

Prioritized Experience Replay which 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. Whereas Deep Q Learning experience transitions *uniformly* from a replay memory

With Deep Q Learning 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 DQN** architecture, we can assess the value of each state, without having to learn the effect of each action.