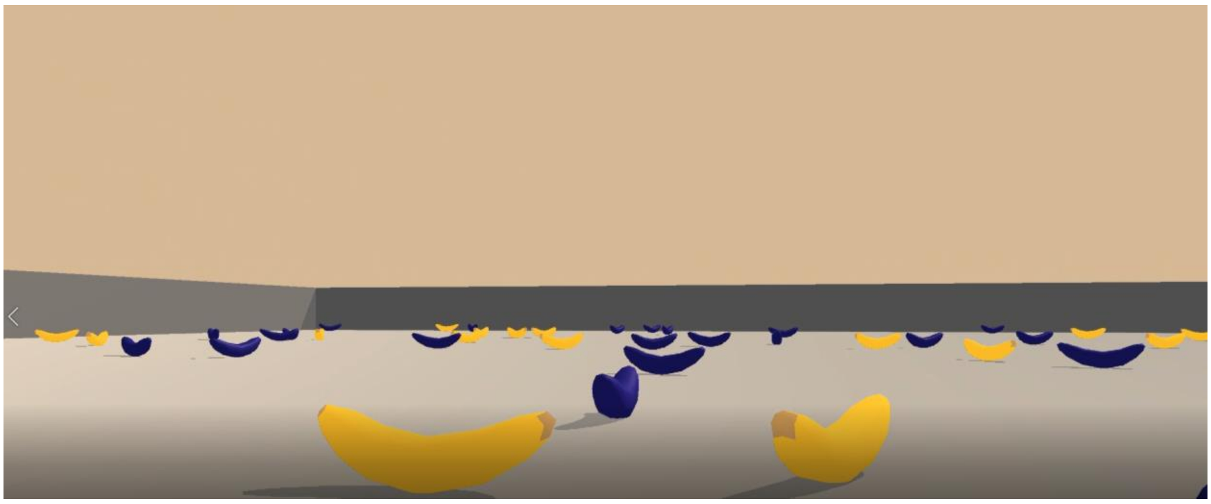
Deep_Reinforcement_learning_Banana_environment

The project focusses on training an agent to collect bananas of particular color in a grid based environment developed in Unity. The agent is trained using deep reinforcement learning algorithms Deep Q learning. The agent gets data about its current state from the environment through the sensor readings which is of 37 dimensional. There is only one agent in this environment and it has 4 actions space which is of Up, Down, Left and Right movements. The agent get a reward of +1 collecting an yellow banana and a reward of -1 for collecting blue banana. The goal of the agent is to collect as many yellow bananas as possible. The task is episodic and comes to end after agent fails. The agent needs to get and average score of +13 to solve the environment.



Model

We use the Deep Reinforcement learning approaches to solve the environment. The Q learning algorithm combined with the Neural Network produces the Deep Q Network. This network is trained by using Mean Squarred loss function to solve the environment.

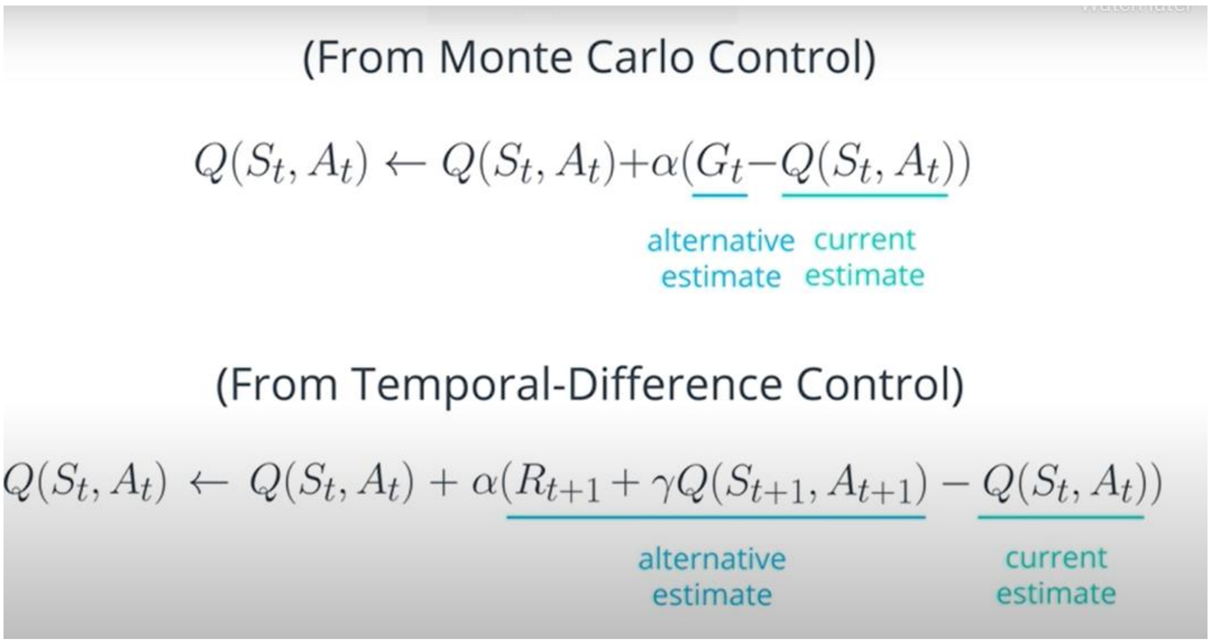
Q learning

The Q learning algorithm uses a Q table which contains both states and actions and the value corresponding to each state action pair.Each time the agent chooses an action which has maximum value among the actions of a particular state. In Monte Carlo based control approaches,the agent updates its Q value only when the episode is completed. Sometimes the episode may take very large time and instead of waiting to update the Q table, we use temporal difference approach to update the Q table after each action.

The Q table for TD control Q learning (SarsaMax) is updated using the below equation

Q(S1,A1) = Q(S1,A1) + alpha * (Reward2 + gamma * max(Q(S2,a) for all actions) - Q(S1,A1))

Here alhpa determines the learning rate of updating the Q table. For updating the Q value of the state S1 and action A1, we use the Q(S1,A1),reward obtained after performing the action A1 (Reward2) and the Q value of next state and maximum probable action in that state. Here we try to reduce the difference between the current estimate Q(S1,A1) and the alternative estimate (Reward2 + gamma * max(Q(S2,a) for all actions)).



(From Monte Carlo Control)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\underbrace{G_t}_{\text{alternative estimate}} - \underbrace{Q(S_t, A_t)}_{\text{current estimate}})$$

(From Temporal-Difference Control)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}_{\text{alternative estimate}} - \underbrace{Q(S_t, A_t)}_{\text{current estimate}})$$

Here is Gt is the expected return after an episode in MC control method while in TD method we use the expected return after each action.

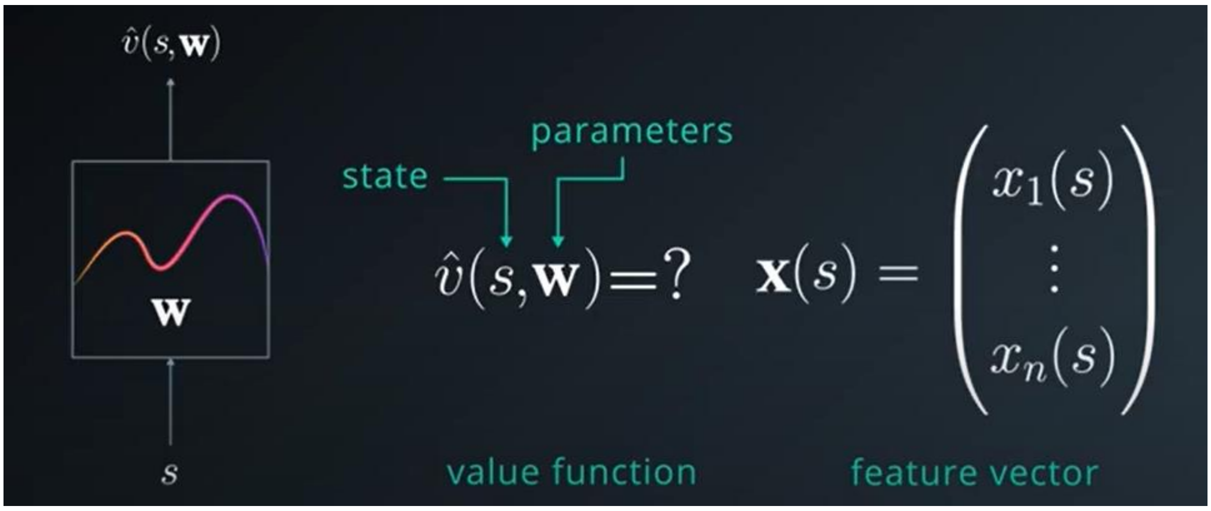
Discretization

In order to deal with a environment which is continuous we discretize into a grid to apply Q learning. But the problem is that the state space is very large and so discretization won't be of much help. So we use function approximation here. Here the state space is repesented as a 37 dimensional feature vector and contains agent's velocity along with ray based perception of the objects. Here we only use the 37 dimensional values and we ignore the images.

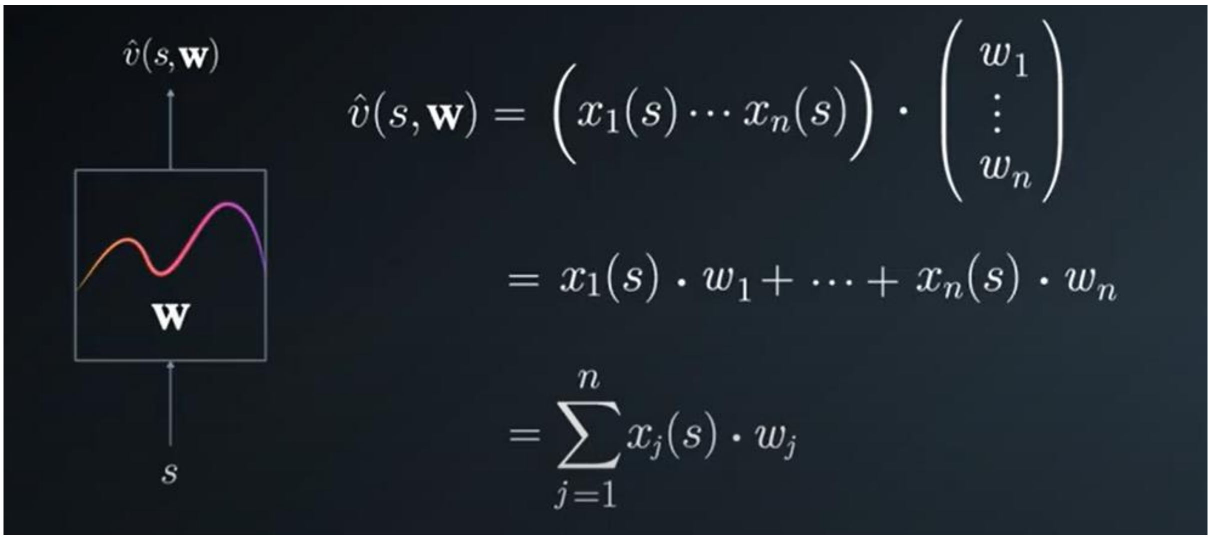
Function approximation

When the state space is very large and complicated, the number of states after discretization becomes very large and so we loose the advantages of discretization. For positions is the nearby states, the values are very much similar and smoothly changing. Discretization doesn't exploit this characteristic failing to generalize well across the state space. The true state value function V_pi or action value function Q_pi are typically smooth and continuous. Capturing this completely is practically infeasible except for some simple problems. So our best hope is function approximation.

So we introduce a parameter vector w to shape the function. These w are learned by optimization algorithms to get the desired approximation. The appoximation can map a state to value or a state action pair to corresponding Q value. In our case we use the function approximation to map the state action pair to corresponding Q value,since we need our agent to take actions in an unknown environment. We first use a Linear function approximation by converting the state space to a set of feature vectors and we tune the weights to get the desired approximation. The feature vector is dot product with the weights to get the desired Q value function.



$\hat{v}(s,\mathbf{w})$
$\mathbf{w}$
s
parameters
state
$\hat{v}(s,\mathbf{w})=?$
$\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$
value function
feature vector



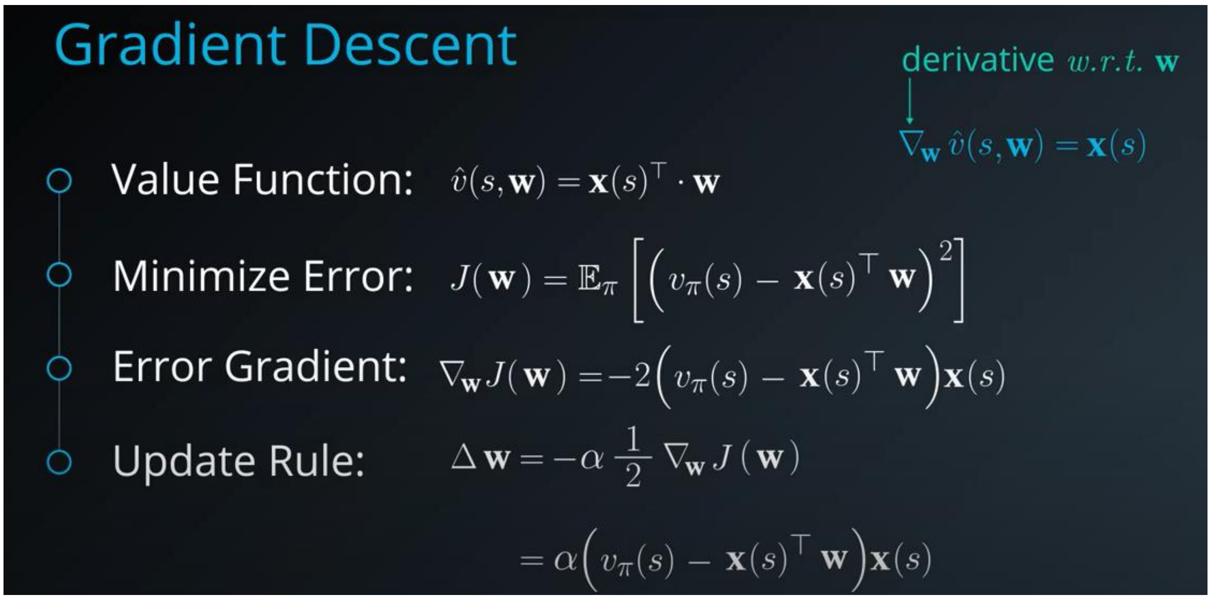
$$\hat{v}(s,\mathbf{w}) = \big(x_1(s) \cdots x_n(s)\big) \cdot \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$$

$$= x_1(s) \cdot w_1 + \cdots + x_n(s) \cdot w_n$$

$$= \sum_{j=1}^{n} x_j(s) \cdot w_j$$

Gradient descent

We need to minimize the difference between true value function V_pi and the approximation value function which we obtained through the dot product. To minimze the mean square error function we differentiate with respect to the weights the loss function and equate it to 0. But for complex problems solving the differential equations can become very painful and so we use gradient descent optimization algorithms.



Gradient Descent

derivative $w.r.t.\ \mathbf{w}$

$\nabla_{\mathbf{w}} \hat{v}(s,\mathbf{w}) = \mathbf{x}(s)$

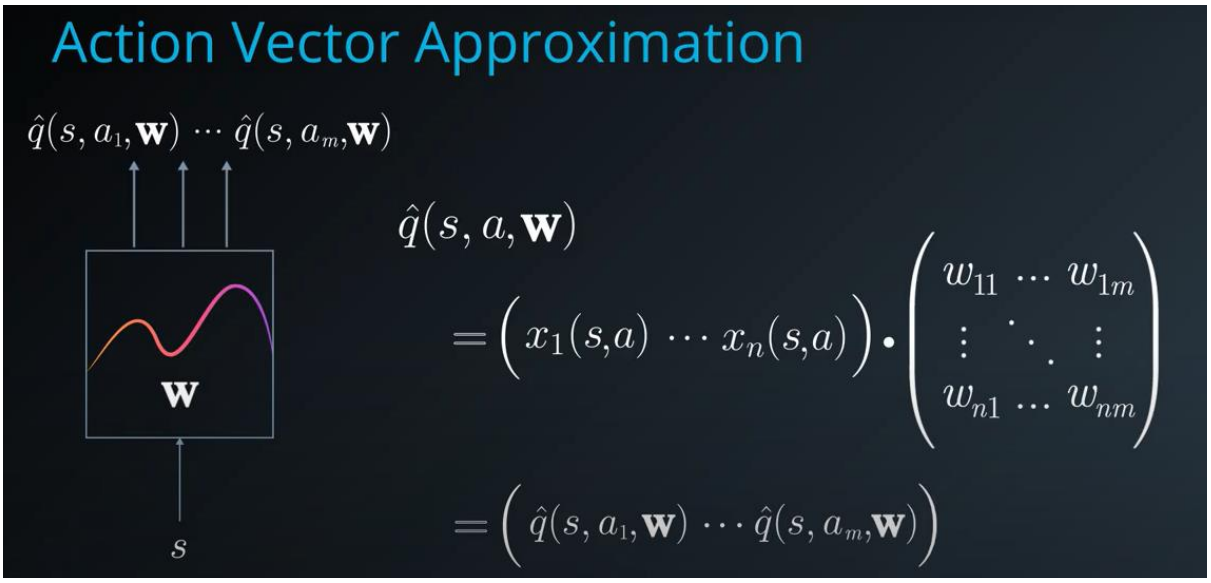
- Value Function: $\hat{v}(s,\mathbf{w}) = \mathbf{x}(s)^\top \cdot \mathbf{w}$
- Minimize Error: $J(\mathbf{w}) = \mathbb{E}_\pi\left[\left(v_\pi(s) - \mathbf{x}(s)^\top \mathbf{w}\right)^2\right]$
- Error Gradient: $\nabla_{\mathbf{w}} J(\mathbf{w}) = -2\left(v_\pi(s) - \mathbf{x}(s)^\top \mathbf{w}\right)\mathbf{x}(s)$
- Update Rule: $\Delta \mathbf{w} = -\alpha \frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w})$

$$= \alpha\left(v_\pi(s) - \mathbf{x}(s)^\top \mathbf{w}\right)\mathbf{x}(s)$$

We now use this update rule to apply gradient descent by plugging in random values of w initially and then optimizing the weights so that the we reach the minimum point in the loss curve. The parameter alpha denotes the learning rate.

Action Vector Approximation

Previously we computed a single action for each state.So we need to reiterate to get the action value function for all possible actions for a particular state. We can now optimize this by computing the action vector which computes all the action values for a state at once.


Action Vector Approximation

$$\hat{q}(s, a_1, \mathbf{w}) \cdots \hat{q}(s, a_m, \mathbf{w})$$

$$\hat{q}(s, a, \mathbf{w}) = \begin{pmatrix} x_1(s,a) & \cdots & x_n(s,a) \end{pmatrix} \cdot \begin{pmatrix} w_{11} & \cdots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nm} \end{pmatrix} = \begin{pmatrix} \hat{q}(s, a_1, \mathbf{w}) & \cdots & \hat{q}(s, a_m, \mathbf{w}) \end{pmatrix}$$

Here we extended our weight vector to a matrix where each column emulates a separate linear function but common features brings the correlation between columns.This parallel processing enables faster computation.

Non Linearity

The main problme with the above function approximation is that it can only handle linear relations. Most of the natural processes are non linear. So apply a non linearity to our previous dot product computation so as to capture the non linear relations. We can use sigmoid or relu or other activation functions for this purpose.

Experience Replay training technique

Previously for each state we perform action, learn from the rewards and then discard the values moving onto next state.But some states may occur very rarely and the actions performed in those states may ber very much crucial. So discarding them is not a good option. So we store the state action experiences in a replay buffer and then learn from this buffer. This Naive Q learning approach runs in the risk of getting stuck by effects of correlations between the sequence of experienced tuples in this buffer. The agent may be biased to certain set of action for a certian set of states and stops exploring the remaining states.

To solve this we can use experience replay. Instead of learning each step, we first allow the agent to explore the environment and get the experiences in a replay buffer. The agent may randomly choose an action for a random state and can explore the environment very well. After sometimes when we get a batch of experiences we now train our agent by randomly choosing the samples from the replay buffer to avoid any correlation.

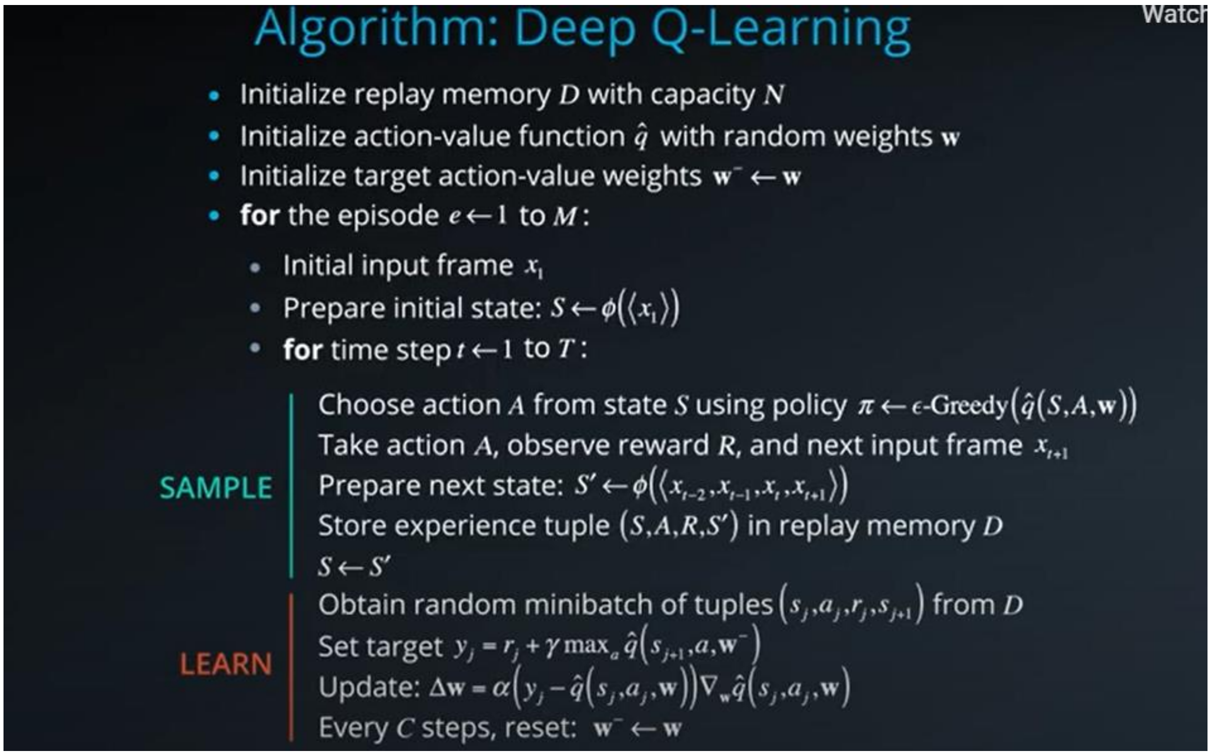
Fixed Q targets

The other problem in the Q learning is that from the update equation we have a correlation between the target and the parameters which the agent is learning. We have the weights which approximate the action value function Q and we use this action value function to update the weights of Q. Thus we a chasing a moving target which is not efficient learning approach. So to solve this problem we use a Fixed Q target function and we use this to find the MSE between the local and target network. After few steps of training we update the target network using the local network. We perform soft update of the target network perventing any drastic changes from the local network.

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

where W minus is the weights of the fixed target network.

Deep Q learning algorithm



Algorithm: Deep Q-Learning

```
• Initialize replay memory $D$ with capacity $N$
• Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
• Initialize target action-value weights $\mathbf{w}^- \leftarrow \mathbf{w}$
• for the episode $e \leftarrow 1$ to $M$:
    • Initial input frame $x_1$
    • Prepare initial state: $S \leftarrow \phi(\langle x_1 \rangle)$
    • for time step $t \leftarrow 1$ to $T$:
SAMPLE      Choose action $A$ from state $S$ using policy $\pi \leftarrow \epsilon\text{-Greedy}(\hat{q}(S,A,\mathbf{w}))$
            Take action $A$, observe reward $R$, and next input frame $x_{t+1}$
            Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
            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 $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, \mathbf{w}^-)$
            Update: $\Delta\mathbf{w} = \alpha(y_j - \hat{q}(s_j, a_j, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s_j, a_j, \mathbf{w})$
            Every $C$ steps, reset: $\mathbf{w}^- \leftarrow \mathbf{w}$
```

Implementation

The model uses 2 Linear layers with a Relu activation function. The input state feature vector is passed to the network and the final third linear layer provides the Q value for the 4 possible actions. We use a batch size of 64 and update the target Q network every 4 iterations. The model is trained for 2000 episodes with an average score of 14.14.

For Udacity Environment

```
//Download the ML agents for banana environment from the repo for
* [Linux](https://s3-us-west-1.amazonaws.com/udacity-
drlnd/P1/Banana/Banana_Linux.zip)
* [Mac OSX](https://s3-us-west-1.amazonaws.com/udacity-
drlnd/P1/Banana/Banana.app.zip)
* [Windows (32-bits)](https://s3-us-west-1.amazonaws.com/udacity-
drlnd/P1/Banana/Banana_Windows_x86.zip)
* [Windows (64 bits)](https://s3-us-west-1.amazonaws.com/udacity-
drlnd/P1/Banana/Banana_Windows_x86_64.zip)
Source : Udacity

Clone the repo
git clone
https://github.com/RamAIbot/Deep_Reinforcement_learning_Banana_environment.git

Place the simulator inside the folder (for Windows its already added).

To Run Training
jupyter notebook Navigation.ipynb
```

```
To Run Testing
jupyter notebook Testing.ipynb
```

For testing in local environment

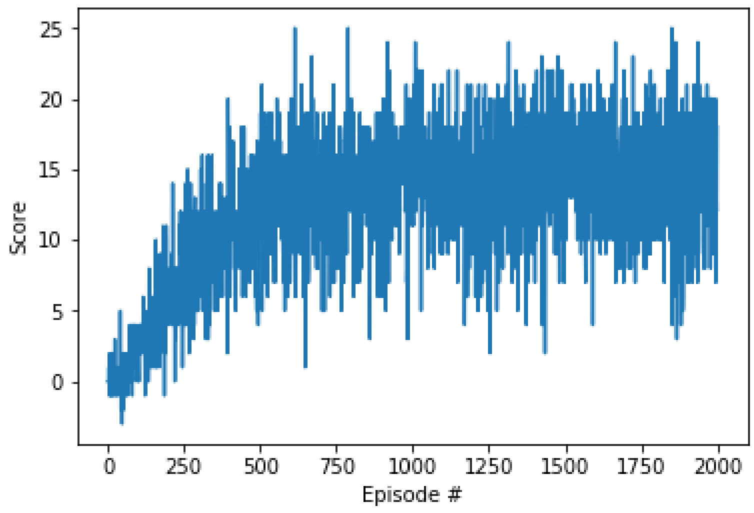
```
Install anaconda python

Create new conda environment
conda create -n ml-agents python=3.6
conda activate ml-agents

Install Cuda toolkit and CuDNN library
Cuda 11.2.2
cuDNN 8.1.0

pip install tensorflow-gpu
conda install pytorch torchvision torchaudio cudatoolkit=11.0 -c pytorch
pip install unityagents
pip install mlagents

move to the path
jupyter notebook Navigation.ipynb
```

Results

Hyper Parameters

- BUFFER_SIZE = int(1e5)
- BATCH_SIZE = 64
- GAMMA = 0.99
- TAU = 1e-3
- LR = 5e-4
- UPDATE_EVERY = 4
- SEED = 0
- MAX TIME = 1000
- EPSILON_START = 1.0
- EPSILON_MINIMUM = 0.01
- EPSILON DECAY = 0.995

Training



Train Results

The agent solved the environment with an average score greater than +13 at episode 900. From there we can see that the network is converged and the score remains stable of above +13.

```
Episode 100	Average Score: 0.92
Episode 200	Average Score: 4.42
Episode 300	Average Score: 7.47
Episode 400	Average Score: 9.54
Episode 500	Average Score: 10.93
Episode 600	Average Score: 12.72
Episode 700	Average Score: 13.56
Episode 800	Average Score: 12.69
Episode 900	Average Score: 13.83
Episode 1000	Average Score: 14.36
Episode 1100	Average Score: 15.68
Episode 1200	Average Score: 14.31
Episode 1300	Average Score: 13.95
Episode 1400	Average Score: 15.03
Episode 1500	Average Score: 15.63
Episode 1600	Average Score: 15.50
Episode 1700	Average Score: 14.71
Episode 1800	Average Score: 14.41
Episode 1900	Average Score: 13.60
Episode 2000	Average Score: 14.14
```

Testing

To test the model,use Testing.ipynb notebook. The simulator can run only once each time the kernel is opened as once the environment is closed the port is lost. So each time we need to restart kernel and run.

```
Video link
https://youtu.be/FGR21U1rhLI
```

Further Improvements

Sometimes the agent fails very badly in some states. This means that some states are not well explored. We can further train the model by tuning hyperparameters to stabilize the result or use prioritized experience replay to give more priority to rare states. Here in this approach we have ignored the images of the camera from the agent. So in future we can also include that to produce more sophisticated agent.