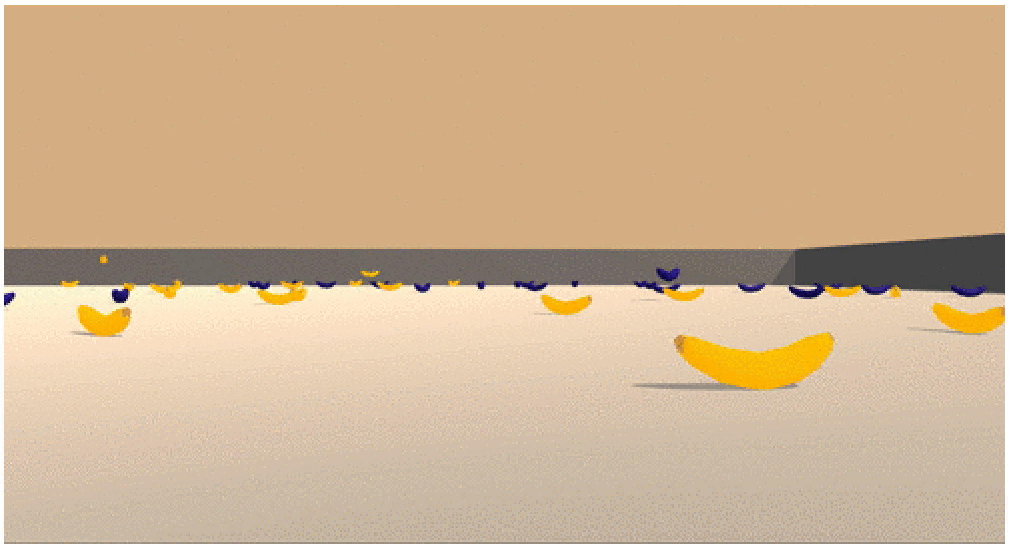
DQN for Navigation Task

In this project we have trained a deep reinforcement learning agent to solve a robotic navigation task. In details, the task environment built in 3D space and sparsely covered with bananas in yellow and blue colors. There are 4 action available for the navigation robot in each state. The measured stated contains robot velocity and ray-based observation of the objects in the forward direction. The task designed episodically and each episode last maximum of 1000 cycles. Our solution uses the DQN (deep Q-Network) reinforcement learning algorithm to solve this task. The optimal navigation policy should maximize its expected discounted reward $Q^*(s,a) = \max_\pi \mathbb{E}[\sum_i \gamma^i r_i]$ by collecting as may yellow bananas as possible and avoid blue bananas. In addition to the vanilla DQN algorithm we add several improvement such as prioritized experience replay buffer , double DQN, and dueling DQN. In the following I will describe each step of our method in details and present the results. In this report the reference to the ideas and publications provide via hyper-link.



Learning Algorithm

- **Reinforcement Learning**: is a type of machine learning method which tries to learn an appropriate closed-loop controller by simply interacting with the process and incrementally improving the control behavior. The goal of reinforcement learning algorithms is to maximize a numerical reward signal by discovering which control commands i.e. actions yield the most reward. Using reinforcement learning algorithms, a controller can be learned with only a small amount of prior knowledge of the process. Reinforcement learning aims at learning control policies for a system in situations where the training information is basically provided in terms of judging success or failure of the observed system behavior.
- **Markov Decision Process**: The type of control problems we are trying to learn in this work are discrete time control problems and can be formulated as a Markov decision process(MDP). An MDP has four components: a set S of states, a set A of actions, a stochastic transition probability function $p(s,a,s')$ describing system behavior, and an immediate reward or cost function $c : S \times A \to R$. The state of the system at time t,

characterizes the current situation of the agent in the world, denoted by $s(t)$. The chosen action by agent at time step t is denoted by $a(t)$. The immediate reward or cost is the consequence of the taken action and function of state and action. Since the rewards for the taken action can be formulated as cost, the goal of the control agent would be to find an optimal policy $\pi* : S \rightarrow A$ that minimizes the cumulated cost for all states. Basically, in reinforcement learning we
try to choose actions over time to minimize/maximize the expected value of the total cost/reward.

- **Q-Learning**: In many real-world problems the state transition probabilities and the reward functions are not given explicitly. But, only a set of states S and a set of actions A are known and we have to learn the dynamic system behaviors by interacting with it. Methods of temporal differences such as Q-Learning were invented to perform learning and optimization in exactly these circumstances. The basic idea in Q-learning is to iteratively learn the value function, Q-function, that maps state-action pairs to expected optimal path costs. The goal of a Q-learning is to find optimal policy which returns highest expected reward give the action-value function. In order to find the optimal action in each state we use Bellman optimality principal. By definition that means at each time step the optimal value of action in a particular state is equal sum its action-value and discounted reward collected after that time stamp onward. The update equation for the action-value function is : $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a{}' Q(S_{t+1}, a') - Q(S_t, A_t)]$.
- **Monte Carlo Method**: are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, here we define Monte Carlo methods only for episodic tasks. The underling idea of all Monte Carlo methods is to estimate value of the given state using experience. As more returns are observed, the average should converge to the expected value. In particular, suppose we wish to estimate $v_\pi(s)$ the value of a state s under policy π, given a set of episodes obtained by following π and passing through s.   Recall that the value of a state is the expected return—expected cumulative future discounted reward—starting from that state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods. The following are the advantages of Monte Carlo methods :
    - they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics
    - they can be used with simulation or sample models. the ability to learn from simulated experience
    - One can generate many sample episodes starting from the states of interest, averaging returns from only these states, ignoring all others. it is easy and efficient to focus Monte Carlo methods on a small subset of the states. A region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set.
    - they may be less harmed by violations of the Markov property. This is because they do not update their value estimates on the basis of the value estimates of successor states. In other words, it is because they do not bootstrap.

- **Batch Reinforcement Learning**: At each time point t it observes the environment state s_t , takes an action a_t , and receives feedbacks from the environment including next state s_{t+1} and the instantaneous reward r_t. The sole information that we assume available to learn the problem is the one obtained from the observation of a certain number of one-step system transitions (from t to $t+1$). The agent interacts with the control system in the environment

and gathers state transitions in a set of four-tuples (s_t, a_t, r_t, s_{t+1}). Except for very special conditions, it is not possible to exactly determine an optimal control policy from a finites et of transition samples. Batch reinforcement learning aims at computing an approximation of such optimal policy π^*, from a set of four-tuples: $D = [(s_t^l, a_t^l, r_t^l, s_{t+1}^l), l = 1, \ldots, \#D]$. This set could be generated by gathering samples corresponding to one single trajectory (or episode) as well as by considering several independently generated trajectories or multi-step episodes. Training algorithms with growing batch have two major benefits. First, from the interaction perspective, it is very similar to the 'pure' online approach. Second, from the learning point of view, it is similar to an
off-line approach that all the trajectory samples are used for training the algorithm. The main idea in growing batch is to alternate between phases of exploration, where a set of training examples is grown by interacting with the system, and phases of learning, where the whole batch of observations is used. The distribution of the state transitions in the provided batch must resemble the 'true' transition probabilities of the system in order to allow the derivation of good policies. In practice, exploration cultivates the quality of learned policies by providing more variety in the distribution of the trajectory samples.

- **Off-Policy Q-Learning**: All learning control methods face a dilemma: They seek to learn action values conditional on subsequent optimal behavior, but they need to behave non-optimally in order to explore all actions (to find the optimal actions). A more straightforward approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. The policy being learned about is called the target policy, and the policy used to generate behavior is called the behavior policy . In this case we say that learning is from data "off" the target policy, and the overall process is termed off-policy learning. An advantage of this separation is that the target policy may be deterministic (e.g., greedy), while the behavior policy can continue to sample all possible actions. The assumption of coverage requires that $\pi(a|s) > 0$ implies $b(a|s) > 0$, meaning every action taken under target-policy(π) should at least occasionally be taken by behavior-policy(b), which enables the use of episodes generated from b to estimate values for π. To explore all possibilities, we require that the behavior policy be soft (i.e.,that it select all actions in all states with nonzero probability).

- **DQN**: Deep Reinforcement Learning refers to usage of neural network function approximators in order to measure the $Q(s_t, a_t)$. In its essence DQN uses off-policy Q-learning in combination with batch reinforcement learning. The non-linearity of the neural networks could cause/add to the instability of the reinforcement learning agent's behavior. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma\, max_a' Q(s', a')$. The DQN has two Q-function, one for generating samples and interacting with the environment, on-line Q-network, and one for calculating the expected rewards target Q-network. The two key ideas of DQN algorithm for solve instability and correlation in the sampled training batch are:
    - First, used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution.
    - Second, used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target.
    - DQN Loss function is defined as following, $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$, $L_i(\theta) = \mathbb{E}_{(a,s,r,s')} \sim U(D) \left[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2 \right]$, Meaning that the expected target values is calculated via target Q-network which its parameters θ^- are update after certain number time stamps by on-line Q-network parameters θ.

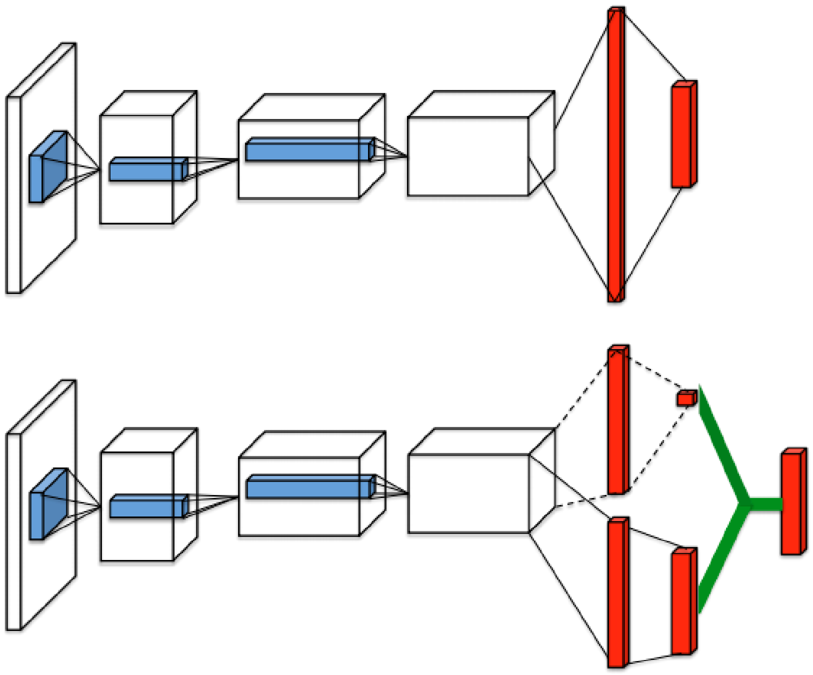
$Deep\ Q - Learning\ Algorithm\ with\ Experience\ Replay$

```
$Initialize\ replay\ memory\ D\ to\ capacity\ N$
$Initialize\ action - value\ function\ Q\ with\ random\ weights\ \theta$
$Initialize\ target\ action - value\ function\ \hat{Q}\ with\ weights\ \theta^- = \theta$
$For\ episode = 1, M\ do$
    $Initialize\ sequence\ s_1 = \{x_1\}$
    $For\ t\ = 1, T\ do:$
        $With\ probability\ \epsilon\ select\ a\ random\ action\ a_t\ otherwise\ select\ a_t = arg\max_a Q(s_t, a; \theta)$
        $Execute\ action\ at\ in\ emulator\ and\ observe\ reward\ r_t\ and\ successor\ state\ s_{t+1}$
        $Store\ transition\ (s_t, a_t, r_t, s_{t+1})\ in\ D$
        $Sample\ random\ minibatch\ of\ transitions\ (s_t, a_t, r_t, s_{t+1})\ from\ D$
        $if\ episode\ terminates\ at\ step\ j+1\ then\ y_i = r_j$
        $else\ y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \theta^-)$
        $Perform\ a\ gradient\ descent\ step\ on\ ((y_j - Q(s_j, a_j; \theta))^2$
        $with\ respect\ to\ the\ network\ parameters\ \theta$
        $Every\ C\ steps\ reset\ \hat{Q} = Q$
    $EndFor$
$EndFor$
```

- **Improvements To DQN**: there multiple way which we could improve vanilla DQN algorithm. In this work we applied two major class of improvements, one is improving deep learning model, other is improving Q-Learning algorithm.
  - **Deep Leaning Model Improvements**
    - **batch normalization**: it is a method of adaptive re-parametrization, motivated by the difficulty of training very deep models. One of the key motivations for the development of BatchNorm was the reduction of so-called *internal covariate shift* (ICS). It is a mechanism that aims to stabilize the distribution (over a mini-batch) of inputs to a given network layer during training. It provides an elegant way of re-parameterizing almost any deep network. The re-parametrization significantly reduces the problem of coordinating updates across many layers. To do that, we apply normalization using μ and σ on activation of each hidden layer H before applying nonlinear function: $H' = \frac{H-\mu}{\sigma}$ where $\mu = \frac{1}{m}\sum_i H_i$, and $\sigma = \sqrt{\epsilon + \frac{1}{m}\sum_i (H_i - \mu)^2}$, The major innovation of the batch normalization is that, it prevents the gradient from increasing the standard deviation or mean of h_i; the normalization operations remove the effect of such an action and zero out its component in the gradient.
    - **Gradient Clipping** One difficulty that arises with optimization of deep neural networks is that large parameter gradients can lead an SGD optimizer to update the parameters strongly into a region where the loss function is much greater, effectively undoing much of the work that was needed to get to the current solution. On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off of the cliff structure altogether

Gradient Clipping clips the size of the gradients to ensure optimization performs more reasonably near sharp areas of the loss surface. It can be performed in a number of ways. The basic idea is to recall that the gradient does not specify the optimal step size, but only the optimal direction within an infinitesimal region. When the traditional gradient descent algorithm proposes to make a very large step, the gradient clipping heuristic intervenes to reduce the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent. One option is to simply clip the parameter gradient element-wise before a parameter update. Another option is to clip the norm $||g||$ of the gradient g before a parameter update: $if\ ||g|| > v\ then\ g \leftarrow \frac{g^v}{||g||}$ where v is a norm threshold.

- **Dueling Q-Network** : The dueling architecture consists of two streams that represent the value and advantage functions, while sharing a common feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function Q. The dueling network automatically produces separate estimates of the state value function and advantage function, without any extra supervision. Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way.



The Advantage Function is relating the value and Q function: $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. Intuitively, the value function V measures the how good it is to be in a particular state s. The Q function, however, measures the the value of choosing a particular action when in this state. The advantage function subtracts the value of the state from the Q function to obtain a relative measure of the importance of each action.

- **Q-Learning Algorithm Improvements**:
  - **Soft Update**: Instead of updating parameters of target Q-network θ^- using of on-line Q-network θ via a direct copy, we use the factor τ to change the target parameters gradually and avoid instability in the expected values: $\theta^- = \tau * \theta + (1-\tau) * \theta^-$

- **Double DQN**: Q-learning tends to learn overestimated to underestimated action values. This preference of unrealistically high action values caused by including a maximization step over estimated action values. Mainly overestimation of action values are attributed to Inflexible function approximation and measurement noise. The max operator in standard Q-learning and DQN, uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overestimated value estimates. To prevent this, we can decouple the selection from the evaluation , $y = r + \gamma Q(s', \max_{a'} Q(s', a'; \theta); \theta^-)$, $L_i(\theta) = \mathbb{E}_{(a,s,r,s')} \sim U(D) \left[(r + \gamma Q(s', \max_{a'} Q(s', a'; \theta_i); \theta_i^-) - Q(s, a; \theta_i))^2\right]$.
- **Prioritized Experience Replay**: On-line reinforcement learning (RL) On Policy agents incrementally update their parameters (of the policy, value function or model) while they observe a stream of experience. In their simplest form, they discard incoming data immediately, after a single update. Two issues with this are (a) strongly correlated updates that break the i.i.d. assumption of many popular stochastic gradient-based algorithms, and (b) the rapid forgetting of possibly rare experiences that would be useful later on. with experience stored in a replay memory, it becomes possible to break the temporal correlations by mixing more and less recent experience for the updates, and rare experience will be used for more than just a single update.

  In general, experience replay can reduce the amount of experience required to learn, and replace it with more computation and more memory – which are often cheaper resources than the RL agent's interactions with its environment. The key idea is that an RL agent can learn more effectively from some transitions than from others. In particular, we propose to more frequently replay transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error. This priortization can lead to a loss of diversity, which we alleviate with stochastic priortization, and introduce bias, which we correct with importance sampling. The central component of prioritized replay is the criterion by which the importance of each transition is measured $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}; \quad w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta$. The estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation. Prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to (even if the policy and state distribution are fixed). We can correct this bias by using importance-sampling (IS) weights.
- **Combined Experience Replay**: Experience replay introduces a new hyperparameter, the memory buffer size, which needs carefully tuning. However unfortunately the importance of this new hyper-parameter has been underestimated in the community for a long time. With simple trick of adding latest transition to the sampled batch transition we could improve the expected reward using smaller buffer size. It is important to note that experience replay itself is not a complete learning algorithm, it has to be combined with other algorithms to form a complete learning system. This technique is not applied to prioritized replay buffer, but only regular replay buffer with uniform sampling.

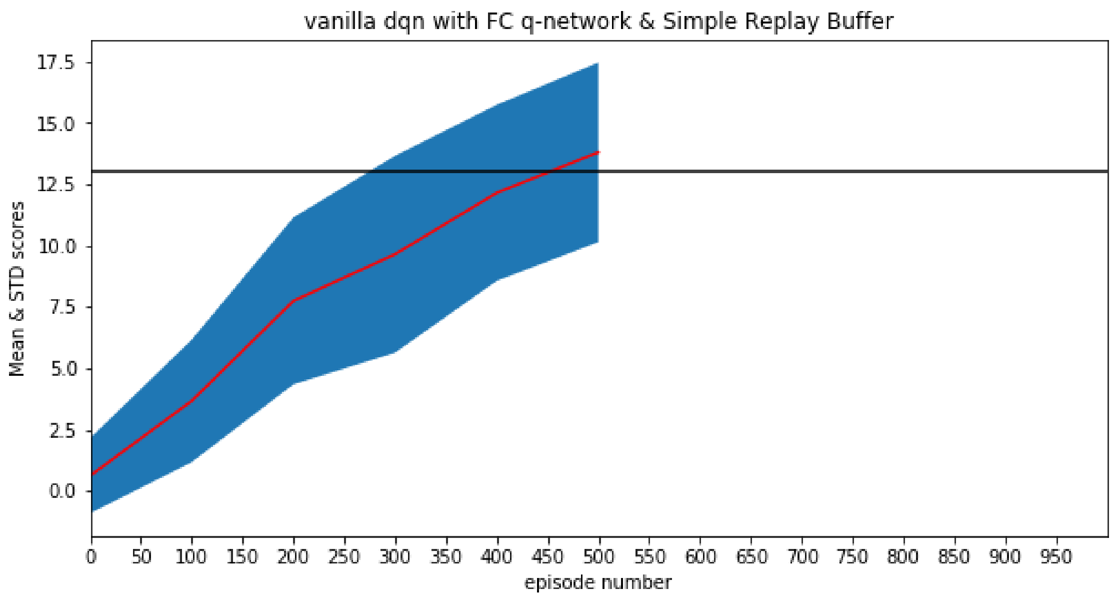
The Experiment Setup :

- The robot get rewarded $+1$ for collecting yellow bananas and -1 for collecting blue bananas
- Each episode last maximum 1000 cycles or terminates earlier due to reaching terminal state
- The training stopped when agent has collected 13 reward points averaged over past 100 episodes.

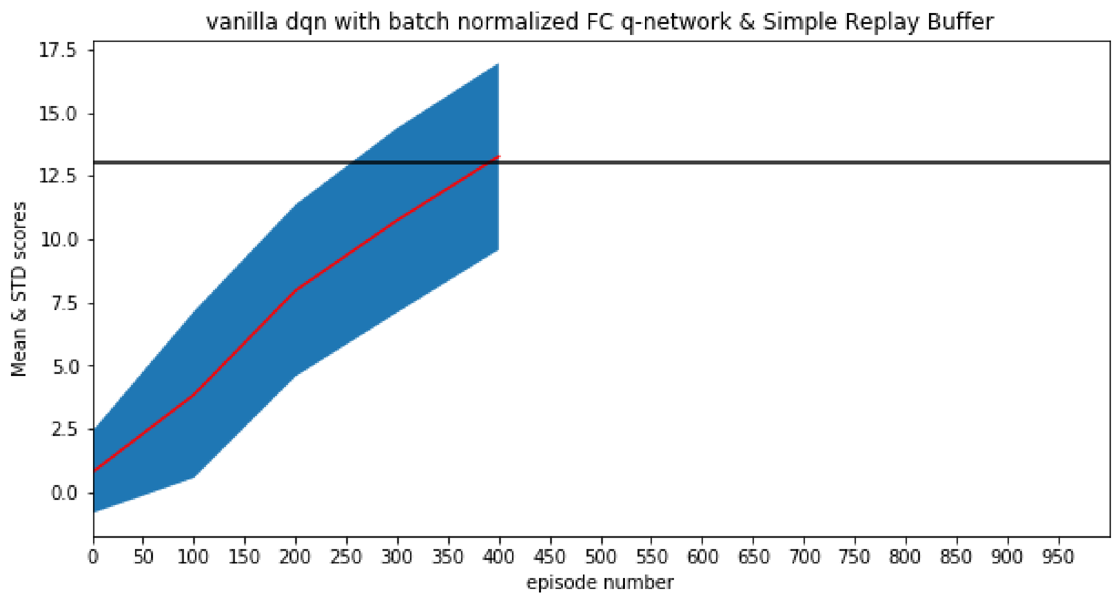
- Epsilon greedy exploration has been applied, starting from 100% greedy exploration and gradually reduced to 5% during 600 episodes.
- After training the trained models and agent construction configurations are stored in the model directory.
- We start by vanilla DQN algorithm and add one improvement, described above, at a time and show the results.
- The hyper parameters are equal to the original DQN paper, the followings are changed(similar for all the variation of the algorithm we have tried):
    - gamma : 0.99
    - learning_rate : 0.00025
    - update_every : 1 (for calculating loss)
    - target_network_update_freq : 4 (for updating target Q-network)
    - τ: 0.001 (soft update factor)
    - replay memory size: 100000
    - gradient clip norm: 5.0 (gradient clipping applied to all the algorithm variations)
    - exploration steps: 1000 (no op steps)
    - α: 0.7 (buffer priortization exponent)
    - β : (buffer important sampling exponent)
    - batch size: 32
    - Hidden Unit Size: 64 (fully connected layers)
    - Number hidden layers: 3

Results:

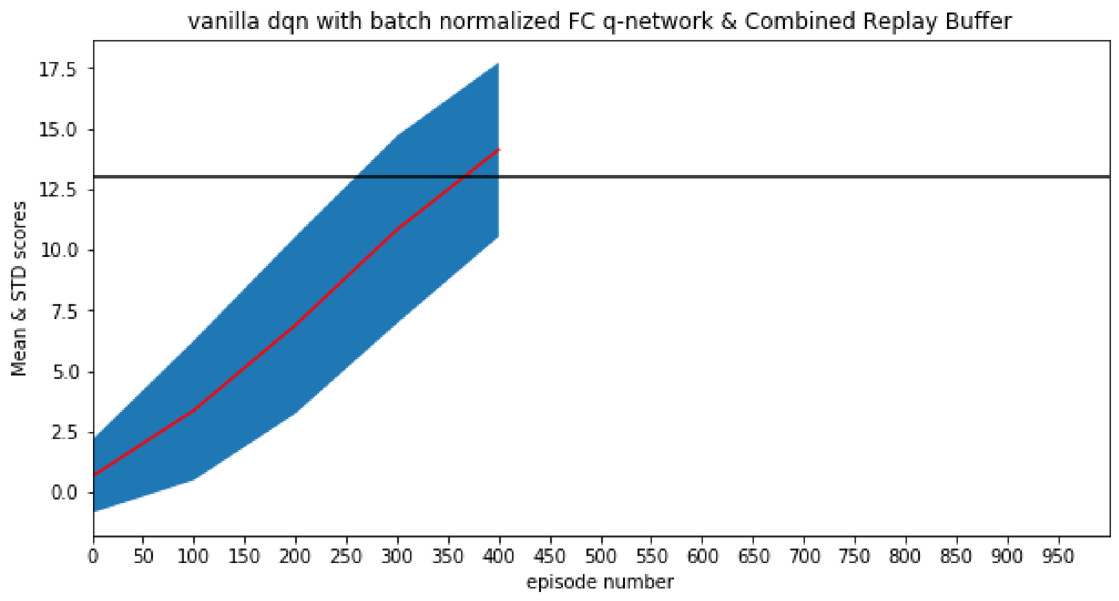
- **Vanilla DQN** : solves the task in the 553 episodes.



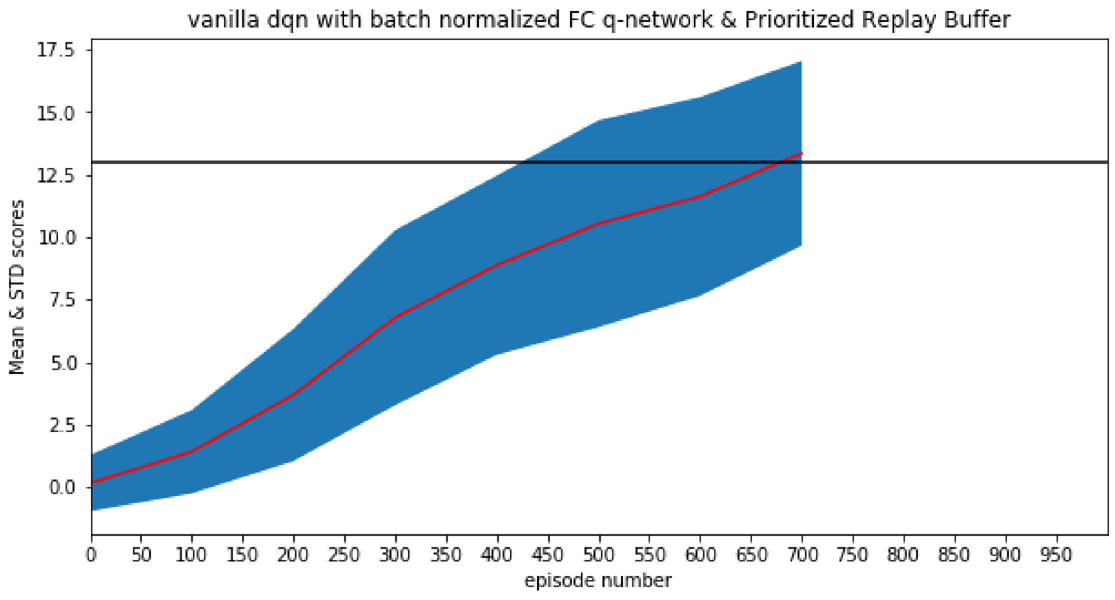
- **DQN with Batch Normalization**: solves the task in the 493 episodes.



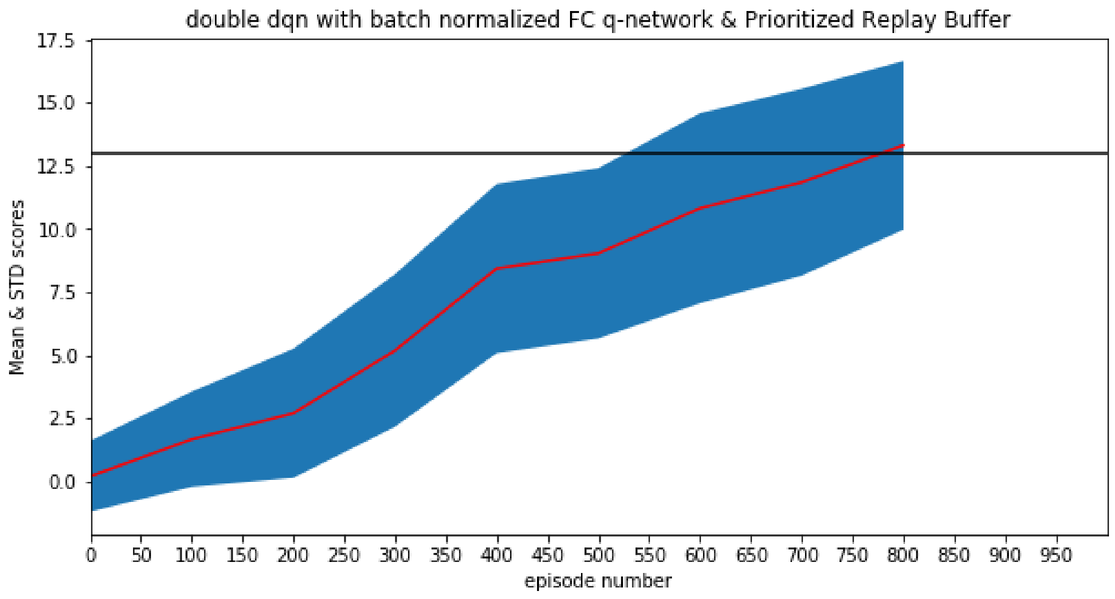
- **DQN with Batch Normalization and Combined Replay Buffer**: solves the task in the 462 episodes.



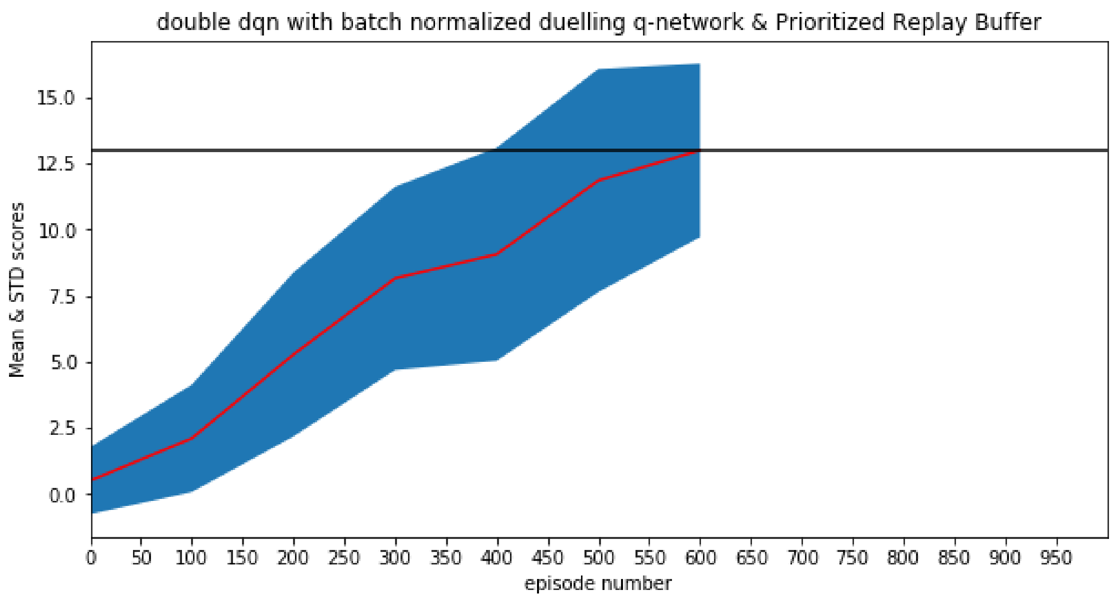
- **DQN with Batch Normalization and Prioritized Experience Replay Buffer**: solves the task in the 786 episodes.



- **Double DQN with Batch Normalization and Prioritized Experience Replay Buffer**: solves the task in the 899 episodes.



- **Double DQN with Batch Normalization, Dueling Q-network, and Prioritized Experience Replay Buffer**: solves the task in the 636 episodes.



Discussion and Future work:

- We we could see the improvements which applied to the deep learning part of the agent enables it to reach the expected results of 13 averaged over past 100 episode much faster. In case of the prioritized experience replay, double DQN , Dueling DQN it takes longer to solve the task. the reason for this decrease in performance lays in the hyper parameters chosen for our experiment. It is a very delicate task to find suitable hyper parameters which fit the algorithm the best and boost its performance.
- As we could expect the adding batch normalization layer to the sequential feed forward neural network improves the performance due to regularization effect. During the training phase each batch of samples has its own mean and standard deviation, which effect distribution of the activation values. Batch normalization set mean of all activation to zero.
- The effect of combined replay memory on the performance boost is tangible. However it the original paper it has been discussed that it mainly used for reducing the replay memory size and to reduce the effect of this hyper parameter.
- Prioritized experience replay and double DQN did not improve the performance, specifically the fast convergence. We keep all the hyper parameters the same to make results comparable. It might be necessary to change or tune the hyper parameters accordingly in other to gain performance boost. Generally speaking the authors of each of this improvement algorithms have shown that their method would increase the performance in majority of the benchmark tests.
- Dueling DQN gets relatively better performance because it directly effect the Q-network model not the Q-learning algorithm. We could conclude that most of the change done Q-network are less sensitive to hyper parameters and improve the performance of the Q-leaning algorithm sometimes with high margin.
- In the future in order to achieve high performance on each algorithm we could use bayesian hyperparameter optimization and search for some of the key parameters which increase the quality of the results.
- Further improvement could be done on the DQN algorithm:
    - Noisy Network for Exploration
    - Distributional DQN

    - multi-step bootstrap targets
    - Rainbow.
- To take one step further we could learn from raw pixels instead of the retrieved features.