Learning algorithm

The training algorithm employed is similar to that used by Mnih, et al in *Playing Atari with Deep Reinforcement Learning*. There are three components – the agent, model and the training algorithm.

The agent is dubbed a "Deep Q Network Agent" because it uses a deep neural network to estimate the optimal q-values (action values) and thus optimal policy given a state (the "local" Q network). Furthermore, it uses a second deep neural network to select the action used for the target (the "target" Q network.) Unlike supervised learning, the target must be estimated and we do so by using this target Q network. In particular, we use Q-learning, a temporal difference method from classical reinforcement learning, to calculate the expected value of the reward for taking an action in a given state plus the discounted maximum action value associated with the next state. Concretely, the target y_j for a time step j in an episode is defined as

$$y_j = r_j + \gamma max_{a'} Q_{target}(s_{t+1}, a'; W^-)$$

where r_j is the reward generated by the environment taking action a_t at state s_t, γ is a discount factor for future rewards and W^- represents the parameters in the network. For the original DQN defined in the Mnih paper, the parameters W^- for the target Q network are fixed and only updated to reflect the local Q network's parameters once every number of predefined cycles. This prevent oscillations and divergence during training. We actually modify this slightly and update the parameters for both the local and target networks every number of predefined cycles, though the target network's parameters are only modified by a tiny amount, defined by a hyperparameter τ. This too prevents oscillations and divergence.

Once we calculate the target and use the local Q network to generate a prediction, we use mean squared error for the loss function

$$Loss(W) = \frac{1}{batch\_size} \sum_{i=0}^{batch\_size} (y_i - Q_{local}(s_i, a_i; W))^2$$

where each element in a batch corresponds to one interaction with the environment, i.e. in the present state feed a selected action to environment; obtain a reward, the next state, and indicator of whether or not the episode is over; and use these values to calculate the target and the prediction.

Each batch is constructed using "experience replay." That is, we generate a number of episodes and thus steps within those episodes and cache these, along with their results, into a buffer. A batch is generated by randomly sampling from this buffer using a uniform distribution. We try to mix up the example in this way because we use stochastic gradient descent (SGD) to update the parameters. SGD assumes examples are independently and identically distributed, which would clearly be violated if we simply fed examples to the network serially.

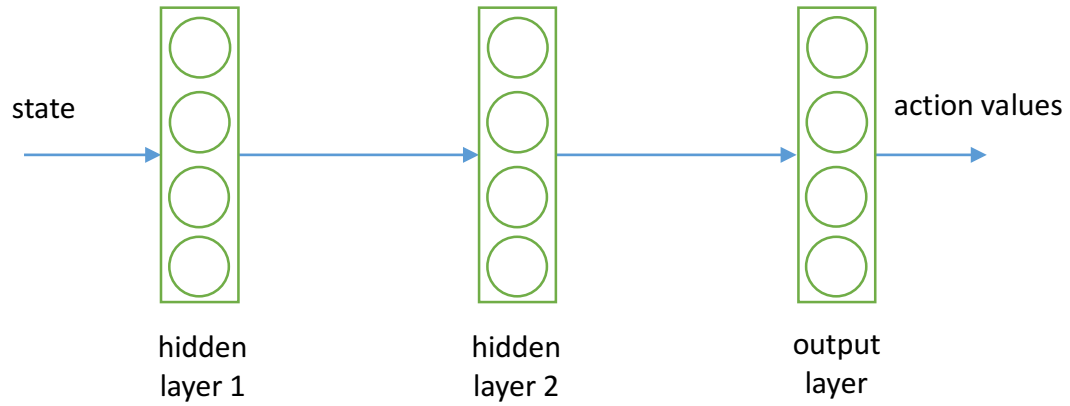
The full training algorithm is described in Algorithm 1 below.

Algorithm 1 Training procedure for Deep Q Network (DQN)

Require:
The environment *env*
The *agent* interacting with the environment; agent is initialized with its local network *Q* and target network *Q'*; initialize these to have the same weights
C - how often to update the weights of target network Q'
τ - how much to change target network's parameters by
Discount factor γ
The maximum number of episodes *N*
The maximum number of time steps per episode *T*
ε - probability by which we choose a random action
Initialize:
Experience replay buffer *B* to capacity *n*
for episode = 1, N do
    state $s_t \leftarrow$ reset env
    for t = 1, T **do**
        With probability ε select a random action a_t otherwise select $a_t = \text{argmax}_a Q(s_t, a; W)$
        Execute in env, get reward r_t, next state s_{t+1}, and done flag
        Store example transition (s_t, a_t, r_t, s_{t+1}) in replay buffer B
        Every C steps, if the replay buffer B is large enough for a batch
            Generate batch of transitions (s_j, a_j, r_j, s_{j+1}) by uniformly sampling from buffer B
            Set target $y_j = r_j$ if episode terminates at s_{j+1}, otherwise $y_j = r_j + \gamma \text{max}_a Q'(s_{j+1}, a)$
            $\text{predicted\_q\_values}_j \leftarrow Q'(s_j, a_j)$
            Perform gradient descent step on $(y_j - \text{predicted\_q\_values}_j)^2$ with respect to the parameters of Q (the local network)
            Softly update the parameters of Q' such that $W^- := \tau * W + (1 - \tau) * W^-$

Model (Q Network)

Each model (target and local Q network) is simply a 3-layer fully connected network.



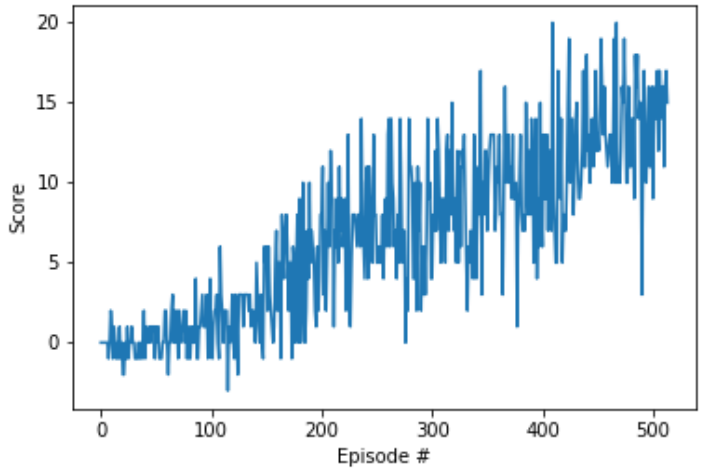
Each neural network takes the state of the banana environment and approximates action values for each action based on that state. The two hidden layers have ReLU activations. The output layer is a projection corresponding to the action-values.

Hyperparameters

Name	**Description**	**Value**
n	experience replay buffer size	100000
batch size	SGD batch size	64
γ	discount factor	0.99
C	how often to update both Q networks	4
τ	how much to change target network's parameters	0.001
lr	learning rate	0.0005
N	maximum number of episodes	1800
T	maximum number of time steps per episode	1000
ε start	the minimum value of epsilon	1.0
ε end	minimum value of epsilon (or end value)	0.01
ε decay	multiplicative factor (per episode) for decreasing epsilon	0.995
fc1_units	# of units in hidden layer 1	64
fc2_units	# of units in hidden layer 2	64
seed	seed to set random number generator for reproducible results	777

Results

Below is a plot of the average score (i.e. the sum of the rewards) per episode over a number of episodes.



The number of episodes needed to achieve a score greater than 13 (13.04) is 414 using the hyperparameters defined above.

Ideas for Future Work

Prioritized Experience Replay could be used to select transitions instead of using a uniform distribution for creating batches. This would allow the model to learn more readily from transitions that led to larger losses.

A dueling DQN could be used instead in which we have two densely connected networks that can be aggregated to produce the q values. One would estimate the value function, the other would estimate the advantage for each action. The advantage for an action is simply the extra gain (or conversely detriment) we get for taking an action given the value function for a given state. This helps with states in which actions associated with them do not impact the environment in any relevant way. This should in theory accelerate training.

The agent we have above couples action selection with value generation for the Q target network. This biases the target values to higher ones - the max value is always being selected! Double DQNs decouple the action selection from the Q target network value generation. The target network is used instead to select the next action; that selected action in turn is used to select the q value for the target network. This reduces the overestimation of the q values resulting in faster training.