The Navigation script has been created to train an Agent within the Banana Collector Unity environment to collect yellow bananas (providing a +1 score) and avoid blue bananas (providing a -1 score). The "BananaBrain" environment has an observation space size of 37 and a discrete action space with a size of 4. The training is considered complete when the Agent can achieve an average score of +13 when the score is averaged over the previous 100 episodes.

To accomplish this, the Agent learns using a Deep Q machine learning network. This network consists of two fully connected layers: one that is used to continuously produce actions (q_current) and a second that is only updated when the network learns from previous experiences (q_target). Starting at the same point in the environment during each episode, the agent uses the current state and the q_current network to predict an action out of the 4 possible actions. During early action decisions, it is beneficial to choose random actions to explore the environment more fully; therefore, a variable *epsilon* is used to bias action decisions towards random choice early in training, but as training progresses this variable is incrementally reduced until action decisions are dominated by those produced using the q_current network. The chosen action is then provided to the environment, which returns the next state, reward, and whether the episode is finished (collectively referred to as SARS for state, action, reward, next state). The agent 'steps' forward using the SARS information as well as two additional variables—*gamma* and *tau*—which will be discussed later. Once the agent has stepped forward, the current state is set to the value of the next state and the score is incremented based on the reward. The current state is used to determine the next action, and the cycle starts again.

Two aspects of this program allow for the agent to be trained to a high degree of accuracy over a relatively short period of time. The first is the use of a 'memory' container to store experiences (previous SARS information) that can then be drawn from to train the Deep Q network; whenever the Agent 'steps', the SARS information is stored in the memory container. The second is the learning process itself. After a pre-defined number of steps (*update_after*), a batch of experiences from the memory container (*batch_size*) is used to train the q_current network. The parameters of the q_target network are then updated using a weighted combination of the current parameters and the previous target parameters, with the weight variable *tau* between 0 and 1.

To further refine the training process, a Double Deep Q network (Double DQN) strategy has been applied in the learning step. Action selection is usually biased towards those actions with the highest reward, or Q-value, but these may not be the best actions overall; selection of a non-ideal action because it has the highest Q-value is known as the overestimation of action value. This leads to a high degree of noise in training, slowing the process down considerably. To avoid this, double DQN utilizes the two available networks to calculate a weighted target that is used in determining the loss in the network. First, the q_current network is used to predict the optimal actions given the next states from the experience batch. Next, those predictions are evaluated based on the output of the q_target network with the next states as inputs. Finally, new target values are calculated by summing the rewards from the experience batch and a weighted result of the prediction evaluation in the previous step, with the weight value defined as *gamma*.

The features detailed above were applied in the Navigation Deep Q network with the following hyperparameters:

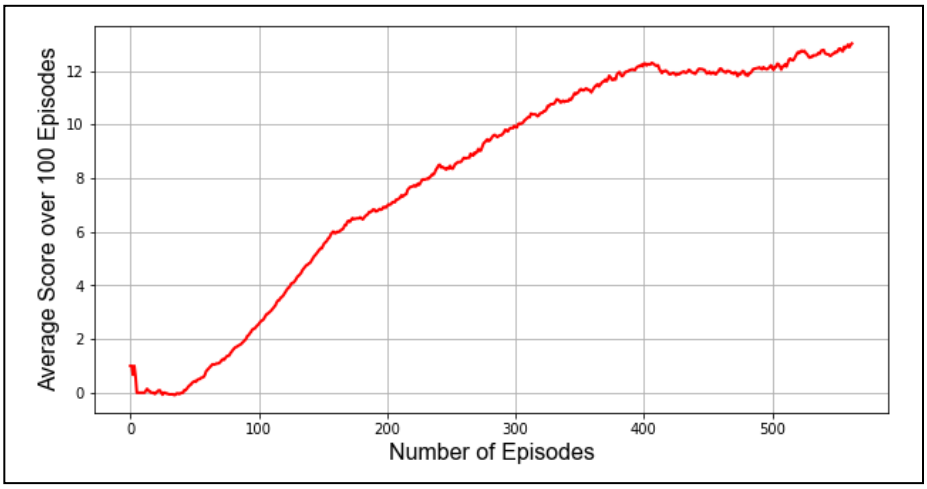
Variables

In [2]:

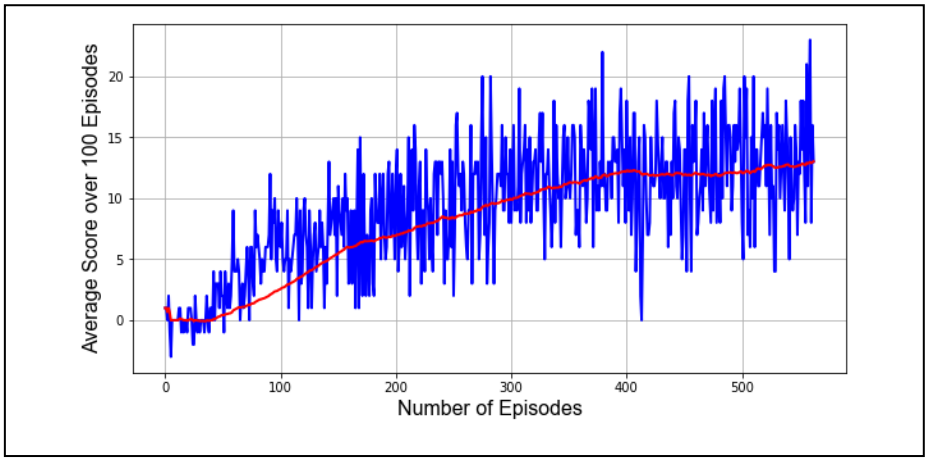
```
num_episodes = 1000          # Number of episodes
buffer_size = int(10e6)      # Size of "memory" for previous steps
batch_size = 32              # Size of random steps to pass to network
gamma = 0.99                 # Discount factor
tau = 10e-4                  # Scale factor to update target policy network
lr =  0.0005                 # Learning rate for current policy network
update = 4                   # Number of runs before learning
epsilon_start = 1.0          # Starting value for epsilon
epsilon_end = 0.05           # Ending value for epsilon
epsilon_decay = 0.992        # Epsilon decay rate
```

Both the q_current and q_target networks were 3-layer fully connected networks with 592, 296, and 4 nodes, respectively. The input to the q_current network was the state (37) and the output was a set of action predictions (4).

Training was completed in 563 episodes, with the progress of training depicted below (red line).



The amount of noise during training is evident in the variance in raw score over the course of training (blue) compared to the average score (red).



Several improvements may be implemented to optimize this network. The most evident has to do with the plateau of average scores at approximately episode 400, as can be seen in both graphs. By implementing a learning rate scheduler to reduce the learning rate when the average score stops changing, this bottleneck may be bypassed. The application of prioritized experience replay may decrease training time for the network, as those experiences that are most important to learn from—that is, those that are seen less frequently—may be chosen more often and the network optimized more rapidly.