Deep Q learning is achieved through two primary steps:

1. First the environment is **sampled** where actions and the outcomes from experiences are stored in a replay memory. The replay memory contains the observed experiences from those actions.

2. Second comes the **learn step** where a small batch of experiences is sampled randomly to train the agent to perform actions that maximize future rewards.
    a. This is where the term "deep" comes in as a neural network is used as a way to fit a non-linear function from action state pairs to Q-values via backpropagation across a fully connected neural net. In the case of the banana this was accomplished with 4 layers built in a neural net.

    b. Essentially, the neural net fits a non-linear function to predicted future Q-values based on previous observations. The non-linear function from the net then estimates the Q value of a specific action given a particular state. This allows the deep Q network to approximate the expected Q values rather than building an "infinitely" large Q-table.
        i. Given that for very large environments the practicality of creating a Q table diminishes very quickly so neural nets are very important if we want to create a practical reinforcement learning environment.

    c. Furthermore, though not demonstrated in this particular project, convolutional networks can be deployed rather than nets with only fully connected layers. This allows direct observation of visuals where the Deep-Q-Network uses pixel values from frames of an environment to connect visual observations with outcomes and create a function that approximates Q values from those observations rather than using velocity and ray based perception like we did in the banana environment.

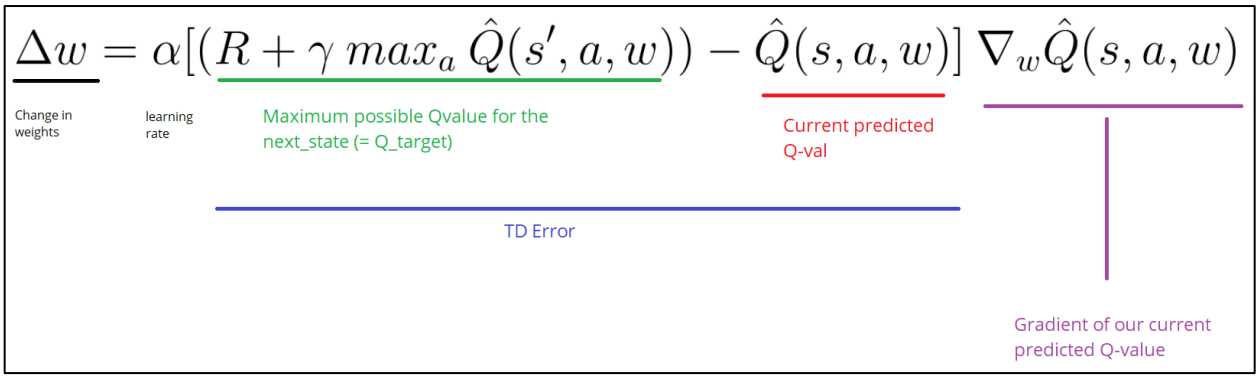


Fig. 1 - Back propagation through fully connected networks allows us to optimize a function that approximates Q values; Diagram reference: www.freecodecamp.com

Choosing Hyperparameter Values:

Fig. 2 - Variable Choice

```
BUFFER_SIZE = int(1e7)
BATCH_SIZE = 64
GAMMA = 0.98
TAU = 1e-3
LR = 0.0001
UPDATE_EVERY = 5
```

- Buffer Size – defines the size of the replay buffer and has impact on the overall "memory" of the agent
- Batch size – related to buffer size and how much is processed at a time while optimizing the weights of the neural network from the replay memory. Also avoids correlation by increasing variance.
- Gamma – discount rate used to put less value on future rewards
- Tau – soft update of target values, changing values gradually
- LR (alpha) – learning rate that determines the speed of gradient descent – gradient descent methods like adam (which was used in this instance) keep descent speed high at first and then slows it down after many iterations after gradient descent becomes difficult after getting closer to a minima.
- UPDATE_EVERY – refers to how often replay memory is updated.

Hyperparameters were tuned based on performance of the model as different combinations were tried. Additionally, prior experience in neural nets and working problems helped get an intuitive understanding of how tuning might impact the overall model. Furthermore, leveraging values used in papers helps quite a bit as well.

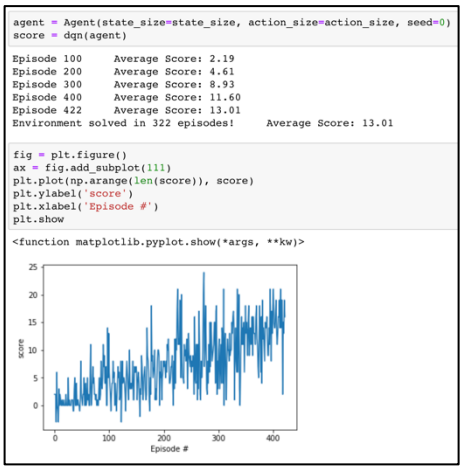
Neural Network Architecture, Outputs and Next Steps:

Fig. 3 – network as defined

```
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, seed):
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, action_size)

    def forward(self, state):

        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)

        return x
```

Network architecture was chosen based on a simple fully connected model with 4 layers. Deep networks enable better models to approximate Q values, however, this must also be balanced with any possible overfitting.

Fig. 4 - Model Outputs



Based on the deep Q network built during this exercise I was able to achieve benchmark scores within 322 episodes. This no doubt can be improved upon by using more sophisticated techniques such as double Q, dueling Q, or a combination of the two (rainbow). Possible next steps include evaluating the impact of implementing such variances in the algorithm and also implementing a convolutional model version where the agent can learn based on vision rather than just velocity and ray based perception.