Name: Mark Abrenio
Project: Udacity Project 1

Concrete Future Ideas:

My last review of this submitted project noted that I forgot to include my ideas for future improvements to this project. There are actually three things I really wanted to try to make myself learn more and to make the agent work better. The first thing I wanted to try was to attempt to implement the Double DQN algorithm to see if it helps to improve the agents ability to learn faster. I also want to do this so I can simply learn how to implement the Double DQN! I found some tutorials online and I already made the following change to my dqn_agent.py to attempt to implement DDQN. I will be testing it to see if it works later this week.

```python
DQN = False
DDQN = True #**Lets test it real quick!

if DQN:
    predicted_targets = self.qnetwork_local(states).gather(1, actions) #**effectively gets MAX q for multiple actions
    with torch.no_grad():
        #target_net_pred1 = self.qnetwork_target(next_states)
        target_net_pred = self.qnetwork_target(next_states).max(1)[0].unsqueeze(1) # ***TARGET NETWORK TO GET `MAXQ
    labels = rewards + (gamma * target_net_pred * (1 - dones))

if DDQN:
    next_actions = self.qnetwork_local(states).argmax(-1, keepdim=True)  # **effectively gets MAX q for multiple actions
    print(next_actions)
    with torch.no_grad():
        target_net_pred = self.qnetwork_target(next_states).gather(-1, next_actions)  # ***TARGET NETWORK TO GET `MAXQ
    print(target_net_pred)
    labels = rewards + (gamma * target_net_pred * (1 - dones))

loss = self.criterion(predicted_targets, labels).to(device)
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

```python
if DQN:
    predicted_targets = self.qnetwork_local(states).gather(1, actions) #**effectively gets MAX q for multiple actions
    with torch.no_grad():
        #target_net_pred1 = self.qnetwork_target(next_states)
        target_net_pred = self.qnetwork_target(next_states).max(1)[0].unsqueeze(1) # ***TARGET NETWORK TO GET `MAXQ
    labels = rewards + (gamma * target_net_pred * (1 - dones))

if DDQN:
    next_actions = self.qnetwork_local(states).argmax(-1, keepdim=True)  # **effectively gets MAX q for multiple actions
    print(next_actions)
    with torch.no_grad():
        target_net_pred = self.qnetwork_target(next_states).gather(-1, next_actions)  # ***TARGET NETWORK TO GET `MAXQ
    print(target_net_pred)
    labels = rewards + (gamma * target_net_pred * (1 - dones))
```

I am still trying to understand the exact mathematics behind why it works, but it doesn't look super hard to implement. It looks like you basically just replace grabbing the MAX value from your qtarget_network, to actually grabbing the values of the next actions that your local qnetwork_model thinks is best. I am not sure if I actually implemented this correctly yet but I am going to keep trying until I get it to work correctly.

I also think that the model could learn better if it were forced, via epsilon greedy exploration, to take "pseudo random" actions that more effectively put it on a path to more quickly discover where the yellow and purple bananas are. The way it is now when you have a high epsilon value the agent just keeps jumping around all over the place and can play whole games and not even bump into any bananas at all. I think a high epsilon value, but with a bias to have the agent move around in a more effective way would make the agent learn faster. I am going to try to implement this too to try to learn more.

I also want to implement solving the game in pixels rather than just using the provided numeric states. I think I would learn a lot more and it will help me more effectively build real world projects where I can implement computer vision.

Project:

For this project, we had to solve the Banana Navigation environment. My algorithm used to solve this challenge was very simple. It is just a standard DQN that is used with a very low GAMMA / Discount rate. In fact, my discount rate was zero!

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128         # minibatch size
#GAMMA = 0.99            # discount factor
GAMMA = 0.0
#TAU = 1e-3              # for soft update of target parameters
TAU = 1
#LR = 5e-4               # learning rate
LR = 0.0005
UPDATE_EVERY = 4        # how often to update the network
UPDATE_TARGET_EVERY = 10000

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

I started out trying to use a higher GAMMA value but that didn't work to well because the agent kept taking non optimal paths and it would run out of time. I also used a very small model for this DQN with a small amount of parameters.

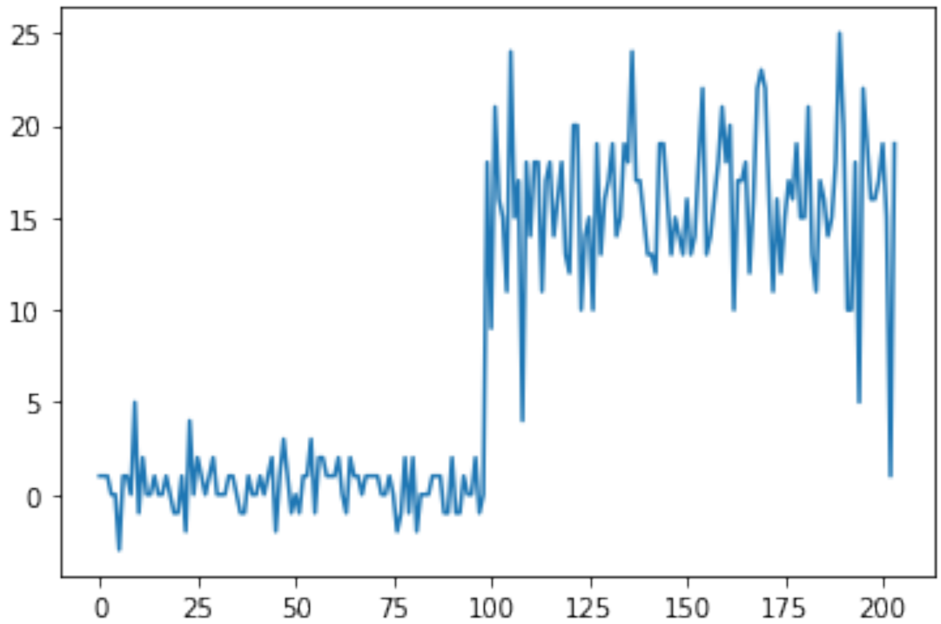
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_unit=32, fc2_unit=32):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        "*** YOUR CODE HERE ***"
        self.fc1 = nn.Linear(state_size, fc1_unit)
        self.fc2 = nn.Linear(fc1_unit, fc2_unit)
        self.fc3 = nn.Linear(fc2_unit, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

As illustrated its only got 3 linear layers. And the fc1 and fc2 layers only output 32 units each. There is really nothing very interesting about the model. I originally tried deeper models and they worked but it takes longer to train. And since the state space isn't really that big it just didn't seem necessary to have all of those parameters! Then I had to set the learning agent to only update the target network every 10000 steps. If the number of steps before updates was small, the agent never really got that good.

```python
LR = 0.0005
UPDATE_EVERY = 4        # how often to update the network
UPDATE_TARGET_EVERY = 10000

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.criterion = torch.nn.MSELoss()

        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 10
        self.l_step = 1
                        #**STEP() SIMPLY RECORDS AND LEARNS
```

```python
def learn(self, experiences, gamma): #**learns and updates
    """Update value parameters using given batch of experience tuples.

    Params
    ======
        experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done) tuples
        gamma (float): discount factor
    """
    states, actions, rewards, next_states, dones = experiences

    ## TODO: compute and minimize the loss
    "*** YOUR CODE HERE ***"
    self.qnetwork_local.train()
    self.qnetwork_target.eval()
    predicted_targets = self.qnetwork_local(states).gather(1, actions)
    with torch.no_grad():
        #target_net_pred1 = self.qnetwork_target(next_states)
        target_net_pred = self.qnetwork_target(next_states).max(1)[0].unsqueeze(1) # ***TARGET NETWORK TO GET `MAXQ

    labels = rewards + (gamma * target_net_pred * (1 - dones))

    loss = self.criterion(predicted_targets, labels).to(device)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    # ------------------- update target network ------------------- #
    # **by default updates target network every time we learn. but lets
    # **try to only update target every so often
    self.l_step = (self.l_step + 1) % UPDATE_TARGET_EVERY
    #print("learning")
    if self.l_step == 0:
        #print("updating target model")
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

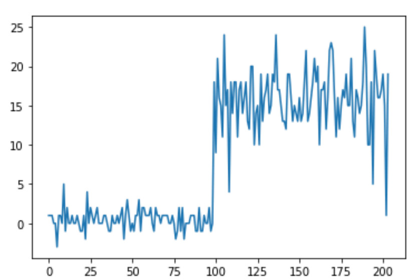
Note: It took the model a couple hundred games to learn the environment. But the code got interrupted before I could let it all train. Therefore in my submission the game only runs approximately 200 times. The first 100 steps it ran with a 90 percent epsilon rate so it could just find some more details about the environment. Then the last 100 step it was playing with epsilon of zero. Below is the chart of the training. As illustrated the first half doesn't go to well because epsilon is 90 percent and the model is still learning some stuff. But the last 100 games go much better because epsilon is turned to zero percent.



Also please note, the final model is saved as the checkpoint file `MODEL_CHECKPOINT.2004788.model.pt. This file can be found no the GitHub repo.`

In [3]:
```
print("Saving model")
print("Graph of Scores:")
my_agent.save_model()  #**The required checkpoint model
from matplotlib import pyplot as plt
plt.plot(all_scores) #**Well do a little score plot
plt.show()
# When finished, you can close the environment.

# In[ ]:


env2.close()
```

```
Saving model
Graph of Scores:
Model Saved: MODEL_CHECKPOINT.2004788.model.pt
```



The final 100 scores are displayed below. These scores are reported on the Jupiter notebook as well.

```
GAME NUMBER: 101
Score: 9.0
GAME NUMBER: 102
Score: 21.0
GAME NUMBER: 103
Score: 16.0
GAME NUMBER: 104
Score: 15.0
GAME NUMBER: 105
Score: 11.0
GAME NUMBER: 106
Score: 24.0
GAME NUMBER: 107
Score: 15.0
GAME NUMBER: 108
Score: 17.0
GAME NUMBER: 109
Score: 4.0
GAME NUMBER: 110
Score: 18.0
GAME NUMBER: 111
Score: 14.0
GAME NUMBER: 112
Score: 18.0
GAME NUMBER: 113
Score: 18.0
GAME NUMBER: 114
Score: 11.0
GAME NUMBER: 115
Score: 17.0
GAME NUMBER: 116
Score: 18.0
GAME NUMBER: 117
Score: 14.0
GAME NUMBER: 118
Score: 16.0
GAME NUMBER: 119
Score: 18.0
GAME NUMBER: 120
```

```
Score: 13.0
GAME NUMBER: 121
Score: 12.0
GAME NUMBER: 122
Score: 20.0
GAME NUMBER: 123
Score: 20.0
GAME NUMBER: 124
Score: 10.0
GAME NUMBER: 125
Score: 14.0
GAME NUMBER: 126
Score: 15.0
GAME NUMBER: 127
Score: 10.0
GAME NUMBER: 128
Score: 19.0
GAME NUMBER: 129
Score: 13.0
GAME NUMBER: 130
Score: 16.0
GAME NUMBER: 131
Score: 17.0
GAME NUMBER: 132
Score: 19.0
GAME NUMBER: 133
Score: 14.0
GAME NUMBER: 134
Score: 15.0
GAME NUMBER: 135
Score: 19.0
GAME NUMBER: 136
Score: 18.0
GAME NUMBER: 137
Score: 24.0
GAME NUMBER: 138
Score: 17.0
GAME NUMBER: 139
Score: 17.0
GAME NUMBER: 140
Score: 15.0
```

GAME NUMBER: 141
Score: 13.0
GAME NUMBER: 142
Score: 13.0
GAME NUMBER: 143
Score: 12.0
GAME NUMBER: 144
Score: 19.0
GAME NUMBER: 145
Score: 19.0
GAME NUMBER: 146
Score: 16.0
GAME NUMBER: 147
Score: 13.0
GAME NUMBER: 148
Score: 15.0
GAME NUMBER: 149
Score: 14.0
GAME NUMBER: 150
Score: 13.0
GAME NUMBER: 151
Score: 16.0
GAME NUMBER: 152
Score: 13.0
GAME NUMBER: 153
Score: 14.0
GAME NUMBER: 154
Score: 18.0
GAME NUMBER: 155
Score: 22.0
GAME NUMBER: 156
Score: 13.0
GAME NUMBER: 157
Score: 14.0
GAME NUMBER: 158
Score: 16.0
GAME NUMBER: 159
Score: 18.0
GAME NUMBER: 160
Score: 21.0
GAME NUMBER: 161

```
Score: 18.0
GAME NUMBER: 162
Score: 20.0
GAME NUMBER: 163
Score: 10.0
GAME NUMBER: 164
Score: 17.0
GAME NUMBER: 165
Score: 17.0
GAME NUMBER: 166
Score: 18.0
GAME NUMBER: 167
Score: 12.0
GAME NUMBER: 168
Score: 16.0
GAME NUMBER: 169
Score: 22.0
GAME NUMBER: 170
Score: 23.0
GAME NUMBER: 171
Score: 22.0
GAME NUMBER: 172
Score: 16.0
GAME NUMBER: 173
Score: 11.0
GAME NUMBER: 174
Score: 16.0
GAME NUMBER: 175
Score: 12.0
GAME NUMBER: 176
Score: 15.0
GAME NUMBER: 177
Score: 17.0
GAME NUMBER: 178
Score: 16.0
GAME NUMBER: 179
Score: 19.0
GAME NUMBER: 180
Score: 15.0
GAME NUMBER: 181
Score: 15.0
```

```
GAME NUMBER: 182
Score: 21.0
GAME NUMBER: 183
Score: 13.0
GAME NUMBER: 184
Score: 11.0
GAME NUMBER: 185
Score: 17.0
GAME NUMBER: 186
Score: 16.0
GAME NUMBER: 187
Score: 14.0
GAME NUMBER: 188
Score: 15.0
GAME NUMBER: 189
Score: 18.0
GAME NUMBER: 190
Score: 25.0
GAME NUMBER: 191
Score: 20.0
GAME NUMBER: 192
Score: 10.0
GAME NUMBER: 193
Score: 10.0
GAME NUMBER: 194
Score: 18.0
GAME NUMBER: 195
Score: 5.0
GAME NUMBER: 196
Score: 22.0
GAME NUMBER: 197
Score: 19.0
GAME NUMBER: 198
Score: 16.0
GAME NUMBER: 199
Score: 16.0
GAME NUMBER: 200
WRITING MODEL.
Model Saved: MODEL_CHECKPOINT.6652469.model.pt
Scores 13 or above: 84
Scores below 13: 16
```

```
Average Score over Last 100: 15.94
```