CS 445 Final Project : Reinforcement Learning with Flappy Bird

by Cole Juracek. Credit to Chuck Anderson for the general algorithm and information from lecture slides, as well as Stack Overflow for general Python/Pygame questions.

IMPORTANT NOTE

I cannot seem to get the game to display within the notebook. All of the code for the game is provided, however, and can be run outside of the notebook.

Introduction and Motivation

The general goal of this project is two-fold:

```
1. Code a Flappy Bird type game through Python
2. Control the box's movement through reinforcment learning to
teach it to go through the gap in the walls.
```

I have always wanted to code some sort of game in Python, and Pygame was the natural choice. It's fairly intuitive, and didn't take as long as I would have thought to code the game. In addition to this, I've thought the concept of reinforcement learning was very interesting. So to learn how to play a game based off nothing more than states, actions, and reinforcements seemed like a great choice for a final project.

The Game

The game itself will be familiar to anyone who has played Flappy Bird before. The goal is to get the box through the gap between the two scrolling walls. Collision with either the walls or the floor will result in a game over.

Implementation of the game was fairly simple, although it took a little while to learn Pygame's API. First, the game is initialized and the boilerplate code is handled. The game is advanced frame-by-frame at a rate of 30FPS. At each frame, the walls scroll forward, and new walls are initialized when these reach the end of the screen. The box's motion is handled by 'BoxClass.py', which uses equations for projectile motion to handle gravity. If at any point the space bar is pressed, the box will 'jump'.

- Again, the game will run without being displayed. Because of this, it will "crash" almost immediately due to falling down and hitting the floor. Copy out of the notebook to actually play.

In [54]:
```
import random
import pygame
```

```
import GameConstants
import BoxClass

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)

def play_game():

    pygame.init()

    game_display = pygame.display.set_mode((GameConstants.DISPLAY_WIDTH, GameCor
    pygame.display.set_caption('Flappy Bird')
    clock = pygame.time.Clock()

    flappy_box = BoxClass.Box(GameConstants.DISPLAY_WIDTH * 0.25, GameConstants.

    delta_x = -15
    wall_x_positions = [x for x in range(GameConstants.DISPLAY_WIDTH, 0, delta_x
    current_wall_position = 0
    top_wall_lengths = [x for x in range(50, 260, 30)]
    bottom_wall_lengths = [x for x in range(260, 50, -30)]
    index = random.randint(0, len(top_wall_lengths) - 1)
    top_wall_length, bottom_wall_length = top_wall_lengths[index], bottom_wall_l

    score = 0
    while True:
        for event in pygame.event.get():
            # If space down, make the box jump
            if event.type == pygame.KEYDOWN and event.key == pygame.K_SPACE:
                flappy_box.jump()

        game_display.fill(WHITE)

        # Draw the floor
        floor = pygame.draw.rect(game_display, BLACK, pygame.Rect((0, GameConsta
                                                                  (GameConstants

        # Draw the walls
        top_wall = pygame.Rect((wall_x_positions[current_wall_position], 0), (20
        top_wall = pygame.draw.rect(game_display, BLACK, top_wall)

        bottom_wall_start = GameConstants.DISPLAY_HEIGHT - floor.height - bottom
        bottom_wall = pygame.Rect((wall_x_positions[current_wall_position], bott
        bottom_wall = pygame.draw.rect(game_display, BLACK, bottom_wall)

        # Update flappy box
        flappy_box.update_position(game_display)
        flappy_box.increment_time()

        # Check for collision with floor + walls
        if flappy_box.rect.colliderect(floor):
            pygame.quit()
            print('GAME OVER. SCORE: ', score)
        elif flappy_box.rect.colliderect(top_wall) or flappy_box.rect.colliderec
            pygame.quit()
            print('GAME OVER. SCORE: ', score)

        # If walls at end of screen, create new walls
        current_wall_position += 1
        if current_wall_position > len(wall_x_positions) - 1:
```

```
            index = random.randint(0, len(top_wall_lengths) - 1)
            top_wall_length, bottom_wall_length = top_wall_lengths[index], bottc

        current_wall_position = current_wall_position % len(wall_x_positions)

        # Update graphics + advance 1 frame
        pygame.display.update()
        clock.tick(GameConstants.FRAMES_PER_SECOND)
        score += 1
```

In [55]:
```
play_game()
```

```
GAME OVER. SCORE:  25
```

```
---------------------------------------------------------------------------
error                                     Traceback (most recent call last)
<ipython-input-55-27da54639cdc> in <module>()
----> 1 play_game()

<ipython-input-54-787e16002907> in play_game()
     67
     68         # Update graphics + advance 1 frame
---> 69         pygame.display.update()
     70         clock.tick(GameConstants.FRAMES_PER_SECOND)
     71         score += 1

error: video system not initialized
```

A brief note: there will be some relaxed constraints on the game such that the reinforcement learning problem is simpler. Namely, the walls will not appear in random positions, but rather in one fixed position.

Reinforcement Learning: An Overview

The general premise of reinforcement learning is to "reward" an agent with reinforcements for positive moves toward a goal. This is accomplished via 3 key components:

1. *state*: any information necessary to represent the agent's current state
2. *actions*: the set of available actions for the agent in a given state
3. *reinforcement*: represents how well received the current (state, action) pair is

Ideally, the agent would like to pick the action from a given state with the highest reinforcement. But not only should the agent account for the immediate future, it should also account for future states down the line. Because of this, the agent should pick a move based off of the summation of all returns from the future states. This function that predicts the sum of future reinforcements is known as the *Q-function*, and utilizing it in reinforcement learning is known as *Q-Learning*. The general idea is for the agent to examine all of the Q-values for its (state, action) pairs, and pick the move with the highest value.

We would like to update the Q-values to be more accurate over time as the agent explores the problem. There are two main approaches to this. The first is known as *Monte-Carlo*, where Q(state, action) is updated through the average over many trials. Another approach, and the one used for this project, updates Q(state, action) via the next state, action, and reinforcement in a process known as *temporal difference*:

$$ \begin{align*} Q(s_t,a_t) = Q(s_t,a_t) + \rho (r_{t+1} + Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \end{align*} $$

Finally, we run into the dilema of exploration vs. exploitation in reinforcement learning. This problem is stated as the following: should the agent attempt to learn as much as possible about the environment to fill its Q-table (exploration)? Or should the agent only pick the "greedy" action (the one with the highest Q-value) from each state (exploitation)? This issue is resolved by the epsilon parameter. During each iteration, decay epsilon by multiplying it by some fraction (epsilon "decay"). The lower epsilon becomes, the more likely the agent is to pick the greedy option. In effect, this will cause the agent to move from a policy of exploration toward exploitation over the course of its journey.

Reinforcement Learning Applied to Flappy Bird

For the game of flappy bird, we will define the state as the y-position, and the actions as {0, 1}, corresponding to not jumping and jumping respectively. The reinforcement will more or less be negligible except for the case when the box is colliding with a wall or floor. In this case, the reinforcement for the (state, action) pair will be assigned as low as possible; this corresponds to a game over and is the only real issue we're trying to avoid. If we note the agent is in the gap between the two walls, we will also provide a *positive* reinforcement.

There is no end state in this game; collisions with the floor and walls will not stop it. Each frame will be considered a new "iteration", and the box will move from exploration to exploitation over the course of around 1 minute.

The default parameters to the game are as follows:

- *epsilon*: 1.0 (standard)
- *epsilon_decay*: 0.999
  - While this might seem like a high value, it is being applied every frame, so epsilon will still decay in a reasonable time.
- *learning_rate*: 0.8
  - This is how much "weight" we should give to the previous Q-value based off the current temporal difference formula. A value of 1.0 will simply be the formula itself. We will choose a default of slightly less.

Games will be run for a length of two minutes, and the position of the box will be recorded on each frame. There is no means for the game stopping, so interrupting the kernel is necessary; the results will still be saved in 'y_positions'.

In [1]:
```
moves = []

import random
import pygame
import sys
import GameConstants
import BoxClass

BLACK = (0, 0, 0)
```

```
WHITE = (255, 255, 255)

y_positions = []
```

In [21]:

```
def learn_game(epsilon_decay=0.999, learning_rate=0.8, reinf_function=None):
    pygame.init()

    game_display = pygame.display.set_mode((GameConstants.DISPLAY_WIDTH, GameCor
    pygame.display.set_caption('Flappy Bird')
    clock = pygame.time.Clock()

    flappy_box = BoxClass.Box(GameConstants.DISPLAY_WIDTH * 0.25, GameConstants.

    delta_x = -15
    wall_x_positions = [x for x in range(GameConstants.DISPLAY_WIDTH, 0, delta_x
    current_wall_position = 0

    Q = {}  # Lookup table for Q-values
    epsilon = 1.0
    valid_moves = [0, 1]  # Where 0 = Do nothing
                          #       1 = Jump
    state = flappy_box.y_pos

    # Return a new move
    def make_move(state, move):
        if move == 1:
            flappy_box.jump()

        # Update flappy box
        flappy_box.update_position(game_display)
        flappy_box.increment_time()

        return flappy_box.y_pos

    num_moves = 0
    epsilon_counter = 0

    while True:
        for event in pygame.event.get():
            x = 4

        game_display.fill(WHITE)
        # Draw the floor
        floor = pygame.draw.rect(game_display, BLACK, pygame.Rect((0, GameConsta
                                                                   (GameConstants
        # Draw the walls
        top_wall = pygame.Rect((wall_x_positions[current_wall_position], 0), (20
        top_wall = pygame.draw.rect(game_display, BLACK, top_wall)
        bottom_wall = pygame.Rect((wall_x_positions[current_wall_position], 350)
        bottom_wall = pygame.draw.rect(game_display, BLACK, bottom_wall)

        # TODO: Consider default value; might want to make very positive vs. ver
        # Pick a move
        epsilon *= epsilon_decay
        if random.uniform(0, 1) < epsilon:  # Random move
            move = random.choice(valid_moves)
        else:  # Greedy move
            q_values = [Q.get((state, action), -100) for action in valid_moves]
            move = valid_moves[q_values.index(max(q_values))]
```

```
        new_state = make_move(state, move)

        # TODO: Consider different default value
        # If (state, move) tuple does not exist, add it to Q-table
        if (state, move) not in Q:
            Q[(state, move)] = -100

        # Check for collision with floor + walls
        # If there is a collision, we should assign with as much negative reinfo
        #   be a game over.
        if flappy_box.rect.colliderect(floor) or flappy_box.rect.colliderect(top
            Q[(state, move)] = -sys.maxsize
        # elif 200 <= state <= 300 and -10 <= flappy_box.y_velocity <= 10:
            # Q[(state, move)] = sys.maxsize
        # elif state == 0:
            # Q[(state, move)] = -sys.maxsize
        elif 225 <= state <= 275 and 175 <= top_wall.center[0] <= 225:
            Q[(state, move)] = 100

        # If walls at end of screen, create new walls
        current_wall_position += 1
        current_wall_position = current_wall_position % len(wall_x_positions)

        # Update graphics + advance 1 frame
        pygame.display.update()
        clock.tick(GameConstants.FRAMES_PER_SECOND)

        # TODO: Consider adding reinforcement
        # Update Q-table of the old (state, move) tuple by calculating the tempo
        # Reinforcement of -1
        if num_moves > 1 and reinf_function is None:
            Q[(old_state, old_move)] += (learning_rate * (Q[state, move] - Q[(ol
        elif num_moves > 1 and reinf_function is not None:
            Q[(old_state, old_move)] += (learning_rate * (reinf_function(state,

        # Update moves + states for next move
        old_state, old_move = state, move
        state = new_state
        num_moves += 1

        if epsilon_counter % 100 == 0:
            print('Epsilon: ', epsilon)
        y_positions.append(state)

        epsilon_counter += 1
```

In [3]:
```
learn_game()
```

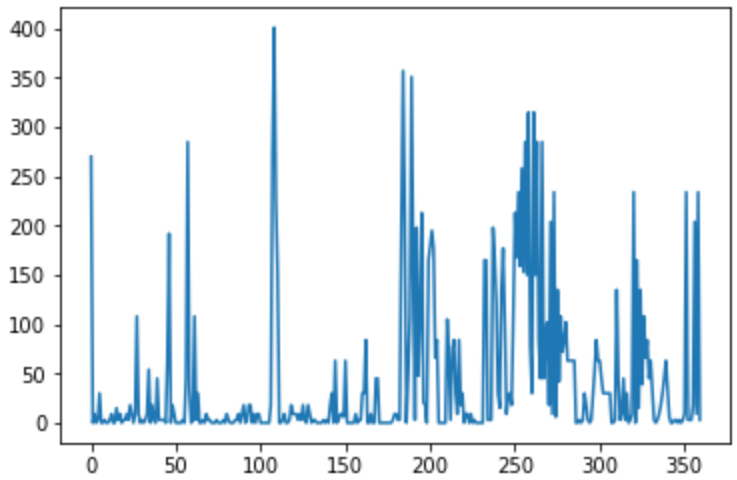
```
Epsilon:  0.999
Epsilon:  0.9038873549665959
Epsilon:  0.8178301806491574
Epsilon:  0.7399663251239436
Epsilon:  0.6695157201007336
Epsilon:  0.6057725659163237
Epsilon:  0.548098260578011
Epsilon:  0.4959150020176678
Epsilon:  0.44869999946146477
Epsilon:  0.4059802359226587
Epsilon:  0.36732772934619257
Epsilon:  0.33235524492954527
Epsilon:  0.3007124156643058
```

```
Epsilon:  0.2720822322326576
Epsilon:  0.2461778670932771
Epsilon:  0.22273980093919937
Epsilon:  0.2015332227394583
Epsilon:  0.18234567731717977
Epsilon:  0.1649849368967147
Epsilon:  0.14927707529619813
Epsilon:  0.13506472547210188
Epsilon:  0.12220550295922675
Epsilon:  0.11057057941158951
Epsilon:  0.10004339195341891
Epsilon:  0.09051847541007228
Epsilon:  0.08190040571973876
Epsilon:  0.07410284394064628
Epsilon:  0.06704767127628951
Epsilon:  0.060664206453048174
Epsilon:  0.05488849760960279
Epsilon:  0.049662681604038215
Epsilon:  0.04493440431994225
Epsilon:  0.04065629616391608
Epsilon:  0.03678549749984046
Epsilon:  0.03328322926552661
Epsilon:  0.030114404470033673
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-3-42455b02ff0d> in <module>()
----> 1 learn_game()

<ipython-input-2-c3f55d386d68> in learn_game(epsilon_decay, learning_rate, reinf
_function)
     79
     80         # Update graphics + advance 1 frame
---> 81         pygame.display.update()
     82         clock.tick(GameConstants.FRAMES_PER_SECOND)
     83

KeyboardInterrupt:
```

In [4]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [5]:
```
plt.plot(y_positions[0::10])
print(len(y_positions))
```

```
3593
```



We can definitely see an improvement over time. Ideally, we want to be around 250, as this is the middle of the gap between the two walls. Toward the beginning, the box is at 0 (top of the screen) the majority of the time, as multiple '0' actions are required to fall, while only one '1' action (jump) is needed to push back toward the top of the screen. However, the box eventually learns that multiple falling actions might be a good to escape the ceiling and head toward the gap.

Tweaking Parameters

There are only two real parameters to change here: the learning rate and epsilon decay. We'll try two radically different values for both, one higher and one lower, and examine the results.

Changing epsilon decay

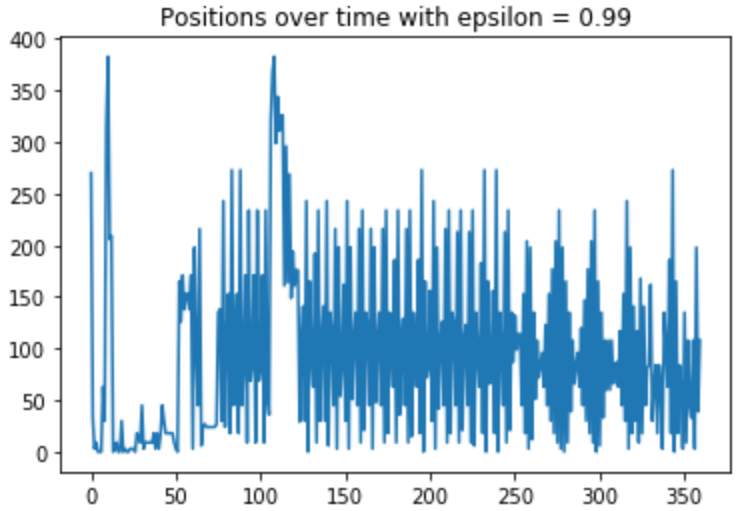
In [6]:
```
y_positions = []
learn_game(epsilon_decay=0.99) # Epsilon decays much quicker
```

```
Epsilon:  0.99
Epsilon:  0.36237201786049694
Epsilon:  0.13263987810938213
Epsilon:  0.0485504851305729
Epsilon:  0.017771047742294682
Epsilon:  0.006504778211990459
Epsilon:  0.0023809591983979563
Epsilon:  0.0008715080698656353
Epsilon:  0.00031900013925143135
Epsilon:  0.00011676436783668758
Epsilon:  4.2739534936551264e-05
Epsilon:  1.564405203775484e-05
Epsilon:  5.726228994379637e-06
Epsilon:  2.0959850054794266e-06
Epsilon:  7.67198298829217e-07
Epsilon:  2.808193895412968e-07
Epsilon:  1.0278897862871985e-07
Epsilon:  3.762409050455422e-08
Epsilon:  1.3771633935657849e-08
Epsilon:  5.0408634126267e-09
Epsilon:  1.8451190369623103e-09
Epsilon:  6.753732410271207e-10
Epsilon:  2.472084486464461e-10
Epsilon:  9.04862872405815e-11
Epsilon:  3.312090757179196e-11
Epsilon:  1.2123323343597234e-11
Epsilon:  4.4375284274692855e-12
Epsilon:  1.6242789197730948e-12
Epsilon:  5.945386158852978e-13
Epsilon:  2.176203615498405e-13
Epsilon:  7.965609044681476e-14
Epsilon:  2.91567052829197e-14
Epsilon:  1.067229709852063e-14
Epsilon:  3.906405893734998e-15
Epsilon:  1.429870895247362e-15
Epsilon:  5.233789915058403e-16
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-6-db22b6814b6a> in <module>()
      1 y_positions = []
----> 2 learn_game(epsilon_decay=0.99) # Epsilon decays much quicker
```

```
<ipython-input-2-c3f55d386d68> in learn_game(epsilon_decay, learning_rate, reinf
_function)
     79
     80         # Update graphics + advance 1 frame
---> 81         pygame.display.update()
     82         clock.tick(GameConstants.FRAMES_PER_SECOND)
     83

KeyboardInterrupt:
```

In [7]:
```
plt.plot(y_positions[0::10])
plt.title('Positions over time with epsilon = 0.99')
```

Out[7]:
```
<matplotlib.text.Text at 0x118d526d8>
```



As expected, this epsilon decay value brought epsilon down too quickly. The box seemed to learn that hugging the top of the screen wasn't optimal, but it very infrequently came between the gap in the wall. This is presumably because it didn't explore enough.

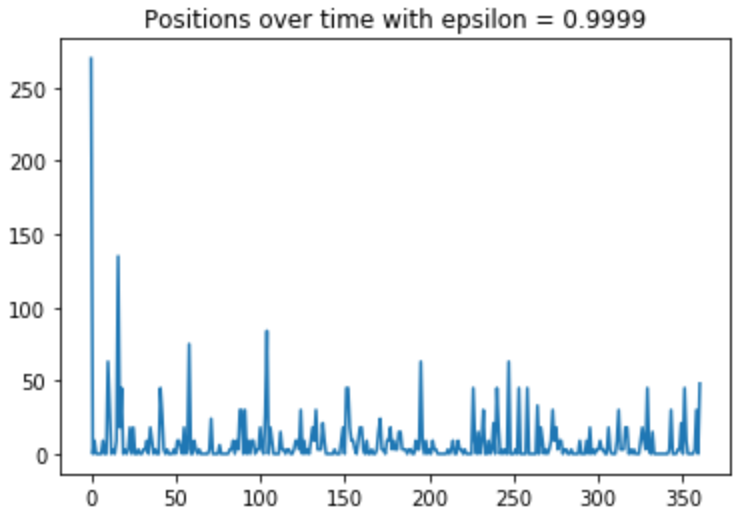
In [8]:
```
y_positions = []
learn_game(epsilon_decay=0.9999) # Epsilon decays much slower
```

```
Epsilon:  0.9999
Epsilon:  0.989950333757503
Epsilon:  0.9800996732739187
Epsilon:  0.9703470333764725
Epsilon:  0.9606914386955115
Epsilon:  0.9511319235669539
Epsilon:  0.9416675319357145
Epsilon:  0.9322973172600907
Epsilon:  0.9230203424170932
Epsilon:  0.9138356796087268
Epsilon:  0.9047424102692004
Epsilon:  0.89573962497306
Epsilon:  0.8868264233442354
Epsilon:  0.8780019139659949
Epsilon:  0.8692652142917918
Epsilon:  0.8606154505570021
Epsilon:  0.8520517576915366
Epsilon:  0.8435732792333273
Epsilon:  0.8351791672426676
Epsilon:  0.8268685822174137
Epsilon:  0.8186406930090225
Epsilon:  0.8104946767394292
Epsilon:  0.802429718718749
```

```
Epsilon:  0.7944450123638009
Epsilon:  0.78653975911744
Epsilon:  0.7787131683686925
Epsilon:  0.7709644573736867
Epsilon:  0.763292851177371
Epsilon:  0.7556975825360077
Epsilon:  0.7481778918404428
Epsilon:  0.7407330270401349
Epsilon:  0.7333622435679438
Epsilon:  0.7260648042656639
Epsilon:  0.7188399793103014
Epsilon:  0.7116870461410829
Epsilon:  0.7046052893871948
Epsilon:  0.6975940007962347
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-8-31cfa86d2989> in <module>()
      1 y_positions = []
----> 2 learn_game(epsilon_decay=0.9999) # Epsilon decays much slower

<ipython-input-2-c3f55d386d68> in learn_game(epsilon_decay, learning_rate, reinf
_function)
     79
     80         # Update graphics + advance 1 frame
---> 81         pygame.display.update()
     82         clock.tick(GameConstants.FRAMES_PER_SECOND)
     83

KeyboardInterrupt:
```

In [9]:
```
plt.plot(y_positions[0::10])
plt.title('Positions over time with epsilon = 0.9999')
```

Out[9]:
```
<matplotlib.text.Text at 0x11c0456a0>
```



At 2 minutes, epsilon was not able to decay to a value even close to 0. Because of this, it was frequently picking a random move even at the end, which more often than not caused the box to stay toward the top of the screen. This result could be interesting however; let's pick an epsilon in the middle and increase the time to 5 minutes:

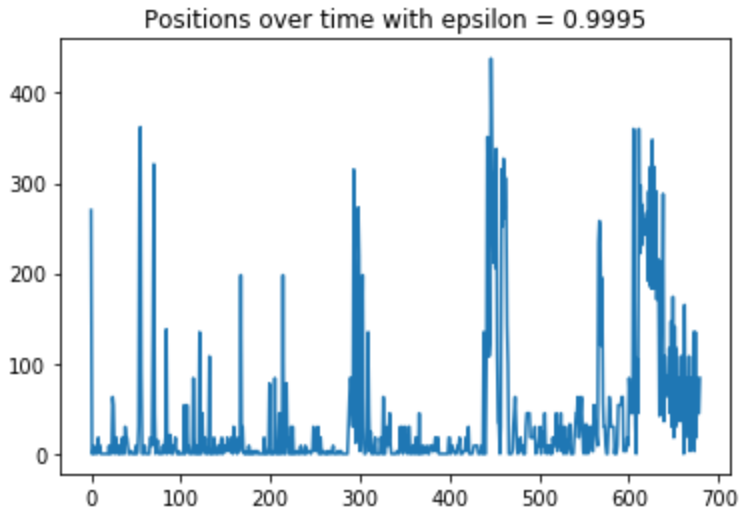
In [10]:
```
y_positions = []
learn_game(epsilon_decay=0.9995) # Epsilon decays a bit slower
```

```
Epsilon:  0.9995
Epsilon:  0.9507419214772129
Epsilon:  0.9043623824454068
Epsilon:  0.8602453518737936
Epsilon:  0.8182804590118382
Epsilon:  0.7783627172668048
Epsilon:  0.7403922615512425
Epsilon:  0.7042740984433092
Epsilon:  0.6699178685348912
Epsilon:  0.6372376203729679
Epsilon:  0.606151595428677
Epsilon:  0.5765820235561167
Epsilon:  0.5484549284291772
Epsilon:  0.5216999424696379
Epsilon:  0.4962501308035369
Epsilon:  0.47204182380537574
Epsilon:  0.4490144578112366
Epsilon:  0.427110423602305
Epsilon:  0.40627492227974177
Epsilon:  0.3864558281703322
Epsilon:  0.36760355841993914
Epsilon:  0.349670948948508
Epsilon:  0.3326131364562932
Epsilon:  0.31638744618611153
Epsilon:  0.30095328516083264
Epsilon:  0.2862720406290043
Epsilon:  0.2723069834645546
Epsilon:  0.25902317627889354
Epsilon:  0.24638738601553387
Epsilon:  0.23436800080856074
Epsilon:  0.22293495089695248
Epsilon:  0.21205963339689513
Epsilon:  0.20171484074388943
Epsilon:  0.19187469262562828
Epsilon:  0.1825145712353572
Epsilon:  0.1736110596837351
Epsilon:  0.16514188341511685
Epsilon:  0.15708585448169488
Epsilon:  0.14942281853608433
Epsilon:  0.14213360440974254
Epsilon:  0.1351999761510762
Epsilon:  0.12860458740324937
Epsilon:  0.12233093800755339
Epsilon:  0.11636333272377304
Epsilon:  0.11068684196427431
Epsilon:  0.10528726444358047
Epsilon:  0.10015109164999415
Epsilon:  0.09526547405038113
Epsilon:  0.09061818894356875
Epsilon:  0.08619760988193471
Epsilon:  0.08199267758468616
Epsilon:  0.0779928722700611
Epsilon:  0.07418818733723333
Epsilon:  0.07056910433207873
Epsilon:  0.06712656913417349
Epsilon:  0.06385196930544977
Epsilon:  0.06073711254383927
Epsilon:  0.0577742061880015
Epsilon:  0.054955837721862175
Epsilon:  0.052274956230188276
Epsilon:  0.04972485475880578
Epsilon:  0.04729915353532999
Epsilon:  0.04499178400842957
Epsilon:  0.04279697366569491
Epsilon:  0.04070923159212851
```

```
Epsilon:  0.03872333473312768
Epsilon:  0.0368343148275929
Epsilon:  0.03503744597847152
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-10-1e568cf58a91> in <module>()
      1 y_positions = []
----> 2 learn_game(epsilon_decay=0.9995) # Epsilon decays much slower

<ipython-input-2-c3f55d386d68> in learn_game(epsilon_decay, learning_rate, reinf
_function)
     33
     34     while True:
---> 35         for event in pygame.event.get():
     36             x = 4
     37

KeyboardInterrupt:
```

In [11]:
```
plt.plot(y_positions[0::10])
plt.title('Positions over time with epsilon = 0.9995')
```

Out[11]:
```
<matplotlib.text.Text at 0x11c0d7a20>
```



Toward the very end, the agent seemed to be performing at its best compared to other epsilon values. However, we did also train this for 5 minutes vs. 2 minutes. If anything, this just shows that longer training times might be necessary for optimal results.

Changing the learning rate

I don't expect changing the learning rate will have as big of an effect as changing the epsilon decay, as the Q-values being altered are very large. However, we will see:

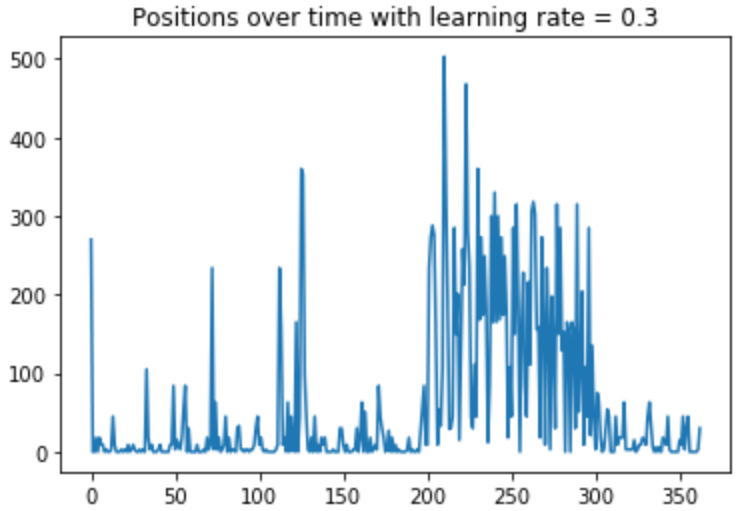
In [12]:
```
y_positions = []
learn_game(learning_rate=0.3) # Learn less from the immediate future
```

```
Epsilon:  0.999
Epsilon:  0.9038873549665959
Epsilon:  0.8178301806491574
Epsilon:  0.7399663251239436
Epsilon:  0.6695157201007336
Epsilon:  0.6057725659163237
Epsilon:  0.548098260578011
Epsilon:  0.4959150020176678
```

```
Epsilon:  0.44869999946146477
Epsilon:  0.4059802359226587
Epsilon:  0.36732772934619257
Epsilon:  0.33235524492954527
Epsilon:  0.3007124156643058
Epsilon:  0.2720822322326576
Epsilon:  0.2461778670932771
Epsilon:  0.22273980093919937
Epsilon:  0.2015332227394583
Epsilon:  0.18234567731717977
Epsilon:  0.1649849368967147
Epsilon:  0.14927707529619813
Epsilon:  0.13506472547210188
Epsilon:  0.12220550295922675
Epsilon:  0.11057057941158951
Epsilon:  0.10004339195341891
Epsilon:  0.09051847541007228
Epsilon:  0.08190040571973876
Epsilon:  0.07410284394064628
Epsilon:  0.06704767127628951
Epsilon:  0.060664206453048174
Epsilon:  0.05488849760960279
Epsilon:  0.049662681604038215
Epsilon:  0.04493440431994225
Epsilon:  0.04065629616391608
Epsilon:  0.03678549749984046
Epsilon:  0.03328322926552661
Epsilon:  0.030114404470033673
Epsilon:  0.027247276679492435
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-12-b67d4316191e> in <module>()
      1 y_positions = []
----> 2 learn_game(learning_rate=0.3) #b

<ipython-input-2-c3f55d386d68> in learn_game(epsilon_decay, learning_rate, reinf
_function)
     79
     80         # Update graphics + advance 1 frame
---> 81         pygame.display.update()
     82         clock.tick(GameConstants.FRAMES_PER_SECOND)
     83

KeyboardInterrupt:
```

In [13]:
```
plt.plot(y_positions[0::10])
plt.title('Positions over time with learning rate = 0.3')
```

Out[13]:
```
<matplotlib.text.Text at 0x11c3ea588>
```



Contrary to my prediction, this actually seemed to have a substantial effect, as we can see from the graph. The box tended to hover around 250 a good amount of time toward the end.

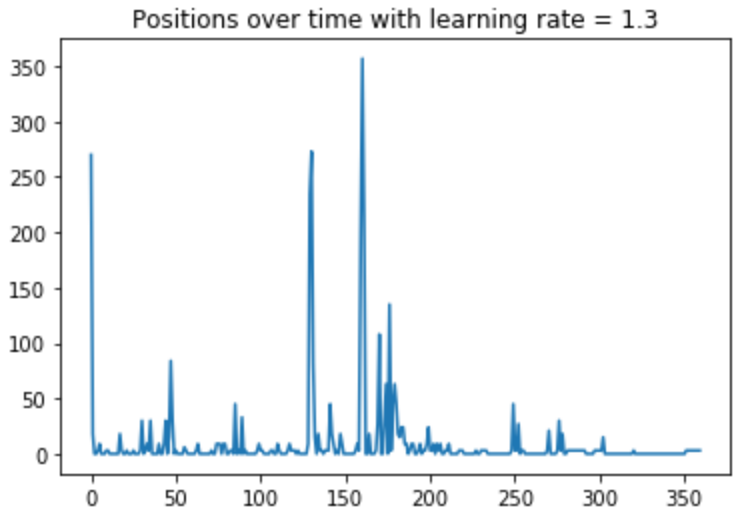
In [14]:
```
y_positions = []
learn_game(learning_rate=1.3) # "Learn" more from the immediate future
```

```
Epsilon:  0.999
Epsilon:  0.9038873549665959
Epsilon:  0.8178301806491574
Epsilon:  0.7399663251239436
Epsilon:  0.6695157201007336
Epsilon:  0.6057725659163237
Epsilon:  0.548098260578011
Epsilon:  0.4959150020176678
Epsilon:  0.44869999946146477
Epsilon:  0.4059802359226587
Epsilon:  0.36732772934619257
Epsilon:  0.33235524492954527
Epsilon:  0.3007124156643058
Epsilon:  0.2720822322326576
Epsilon:  0.2461778670932771
Epsilon:  0.22273980093919937
Epsilon:  0.2015332227394583
Epsilon:  0.18234567731717977
Epsilon:  0.1649849368967147
Epsilon:  0.14927707529619813
Epsilon:  0.13506472547210188
Epsilon:  0.12220550295922675
Epsilon:  0.11057057941158951
Epsilon:  0.10004339195341891
Epsilon:  0.09051847541007228
Epsilon:  0.08190040571973876
Epsilon:  0.07410284394064628
Epsilon:  0.06704767127628951
Epsilon:  0.060664206453048174
Epsilon:  0.05488849760960279
Epsilon:  0.049662681604038215
Epsilon:  0.04493440431994225
Epsilon:  0.04065629616391608
Epsilon:  0.03678549749984046
Epsilon:  0.03328322926552661
Epsilon:  0.030114404470033673
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-14-ccfa1b6d11e7> in <module>()
```

```
      1 y_positions = []
----> 2 learn_game(learning_rate=1.3) # Epsilon decays much quicker

<ipython-input-2-c3f55d386d68> in learn_game(epsilon_decay, learning_rate, reinf
_function)
     79
     80         # Update graphics + advance 1 frame
---> 81         pygame.display.update()
     82         clock.tick(GameConstants.FRAMES_PER_SECOND)
     83

KeyboardInterrupt:
```

In [15]:
```
plt.plot(y_positions[0::10])
plt.title('Positions over time with learning rate = 1.3')
```

Out[15]:
```
<matplotlib.text.Text at 0x11c7ba198>
```



This learning rate seemed to make the agent perform much worse. It was giving too much weight in the temporal difference formula. We do not see the agent come close to converging even after two minutes.

Adding a Reinforcement Function

Previously, we were not that concerned with what the box was doing when it was far away from the walls; we only cared about what happened when it was immediately around the walls. However, a simple reinforcement function might help the box stay near the center, such that it might converge quicker on good Q-values:

In [22]:
```
# If the box is in the top half of the screen, favor positive velocities
# Else, it's in the bottom half, so we should favor negative velocities
def reinforcement2(state, box):
    if state > 250:  # Bottom half, favor negative velocities
        return box.y_velocity * -1
    else:  # Top half
        return box.y_velocity
```

One might think that reinforcing based off of position would be a simple idea. However, this turned out to be more complicated than initially thought. If the box is in the top half and travelling on an upward arc, a move of 0 (doing nothing) would be desired. Yet all the algorithm

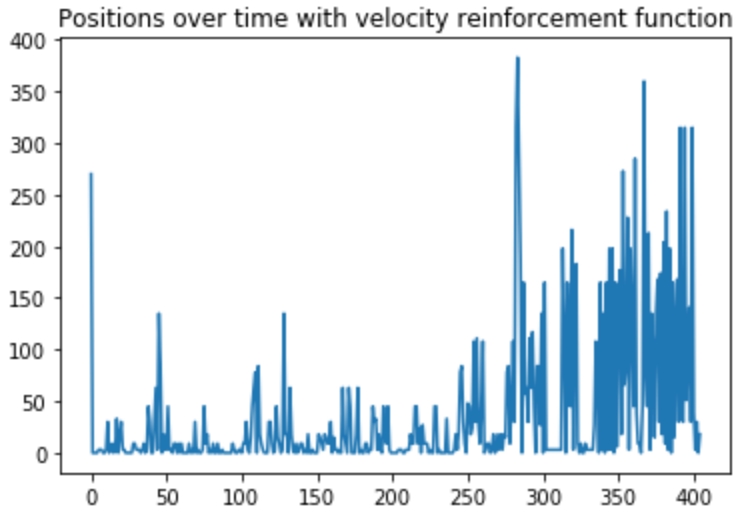
would see is a "nothing" move that moved the box upward. Because of this, favorable moves might be given bad values, so velocity was chosen instead.

In [23]:
```
y_positions = []
learn_game(reinf_function=reinforcement2)
```

```
Epsilon:  0.999
Epsilon:  0.9038873549665959
Epsilon:  0.8178301806491574
Epsilon:  0.7399663251239436
Epsilon:  0.6695157201007336
Epsilon:  0.6057725659163237
Epsilon:  0.548098260578011
Epsilon:  0.4959150020176678
Epsilon:  0.44869999946146477
Epsilon:  0.4059802359226587
Epsilon:  0.36732772934619257
Epsilon:  0.33235524492954527
Epsilon:  0.3007124156643058
Epsilon:  0.2720822322326576
Epsilon:  0.2461778670932771
Epsilon:  0.22273980093919937
Epsilon:  0.2015332227394583
Epsilon:  0.18234567731717977
Epsilon:  0.1649849368967147
Epsilon:  0.14927707529619813
Epsilon:  0.13506472547210188
Epsilon:  0.12220550295922675
Epsilon:  0.11057057941158951
Epsilon:  0.10004339195341891
Epsilon:  0.09051847541007228
Epsilon:  0.08190040571973876
Epsilon:  0.07410284394064628
Epsilon:  0.06704767127628951
Epsilon:  0.060664206453048174
Epsilon:  0.05488849760960279
Epsilon:  0.049662681604038215
Epsilon:  0.04493440431994225
Epsilon:  0.04065629616391608
Epsilon:  0.03678549749984046
Epsilon:  0.03328322926552661
Epsilon:  0.030114404470033673
Epsilon:  0.027247276679492435
Epsilon:  0.024653121969839265
Epsilon:  0.022305951160147018
Epsilon:  0.02018224944360293
Epsilon:  0.018260740807661956
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-23-2852cf01dc52> in <module>()
      1 y_positions = []
----> 2 learn_game(reinf_function=reinforcement2)

<ipython-input-21-929daaec0f13> in learn_game(epsilon_decay, learning_rate, rein
f_function)
     79
     80         # Update graphics + advance 1 frame
---> 81         pygame.display.update()
     82         clock.tick(GameConstants.FRAMES_PER_SECOND)
     83

KeyboardInterrupt:
```

In [24]:
```
plt.plot(y_positions[0::10])
plt.title('Positions over time with velocity reinforcement function')
```

Out[24]:
```
<matplotlib.text.Text at 0x11ca22d68>
```



Overall, adding the reinforcement function caused our agent to perform the best. We can see a definitive progression toward 250. But most importantly, once the agent learned this was a good place to be, it didn't seem to regress back toward constantly hugging the top.

Results

The two parameters seemed to have a large effect on the performance of the agent, although the default parameters performed at an acceptable level. The default value of epsilon seemed to perform best at the two minute mark, although I have little doubt that epsilon = 0.9999 would outclass this given noticeably longer times (10+ minutes). The agent also seemed to respond well to a lower-than-default learning rate, where the Q-values were updated in a less extreme manner. Finally, the simple reinforcement function I've implemented seemed to help more than not, and gave the agent an idea for where to aim even when near no walls.

Conclusion and Challenges Faced

Overall, I would say the assignment was a success. It was a great way to apply machine learning concepts to a relevant example. It taught me how to look for such abstract concepts as "state" and "action" when they weren't explicitly defined.

The main challenge of the assignment was not the machine learning; it was working with Pygame. I've never coded a game in it before, so I wasn't really sure how to translate my thoughts to code. Luckily, Pygame has good documentation and the common questions that I needed had been answered before. Moving to the machine learning side, the main issue I faced was debugging. When the game wasn't working as desired, it was hard to tell if it was my fault in the game logic, the algorithm, or how I had defined Q-learning. The only real means I had of examining it was to watch the game unfold on screen.

Moving forward, I would like to get the reinforcement learning working with random walls, and perhaps implement a better reinforcement function such that the solution might converge faster.