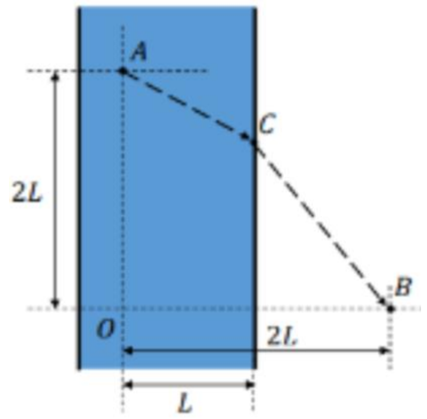
MAE 301: Applied Experimental Statistics

Fastest Route Optimization

Tanner Merry

Problem Statement

With the objective of displaying the application of statistics in the real world, a deep learning model will be used in an attempt to solve an optics problem. The optics problem can also be explained in mechanics perspective as follows.



A person starts off at point A on a boat. The subject must travel through the water and over land to get to point B as shown in figure. On land the subject is able to travel k times as fast as they can travel in water. Find the optimal point p, in which the subject should transition from water to land. The optimal point should be found using reinforcement learning methods.

Procedure

In order to grasp a base understanding of reinforcement and reinforcement learning methods similar projects were manipulated in order solve the optimization problem at hand. As a popular reinforcement learning example is a maze solver, a deep reinforcement learning project was utilized in the process of solving the given problem. The maze project will assist on the basis of setting up the environment, and developing a model for the project. A similar action space will be used as well, however, the maze is limited to the actions of up, down, left, and right, while there will be a 45 degree move in the action space as well. While referencing a maze application, the integration of blocking the path of the human with an obstacle will be integrated into the problem at hand. By integrating obstacles into the project, the optics problem transforms into the visual representative mechanics problem, as more question is added than just “What is the free path to point B?”

Setting Up the Environment:

```
k = 3 # this is the factor mu for the different speed in water

# Create the space using deductions for each space in the 10 by 10 grid
#    Note there is a greater deduction in the water, because it takes more
#    time to move in the water
space = np.concatenate(((-k)*np.ones((10,5)),(-1)*np.ones((10,5))),axis=1)
```

As the first block of code illustrates, the value of k represents the speed which the human can travel faster than in water. Beyond defining this factor, the special environment is created, as each space is given a value based on terrain. The spaces on the left half of the environment are all assigned a value of -k, while the spots on the right half of the environment are given a value of -1.

```
# Define the possible actions
LEFT = 0
UP = 1
RIGHT = 2
DOWN = 3
DOWNLEFT = 4
UPLEFT = 5
UPRIGHT = 6
DOWNRIGHT = 7
```

```
# Create a dictionary for the possible actions
actions_dict = {
    LEFT: 'left',
    UP: 'up',
    RIGHT: 'right',
    DOWN: 'down',
    DOWNLEFT: 'down-left',
    UPLEFT: 'up-left',
    UPRIGHT: 'up-right',
    DOWNRIGHT: 'down-right',
}

# number of defined actions
num_actions = len(actions_dict)
```

Next, the possible actions are defined. The possible actions include moving left, right, up, down, up-left, up-right, down-left, and down-right. Thus, the cursor will be able to move in 8 different directions. This is still limiting figuring the fact that real case allows for movement in a full 360 degrees.

```
# Starting exploration factor
epsilon = 0.1
```

The exploration factor, described by epsilon will be utilized when the training algorithm is finding a path from point A to point B. The value of 0.1 means there is a 10% chance that the next chosen move will be random, and the other 90% will be chosen based on the model parameters defined in the hidden layers of the neural network.

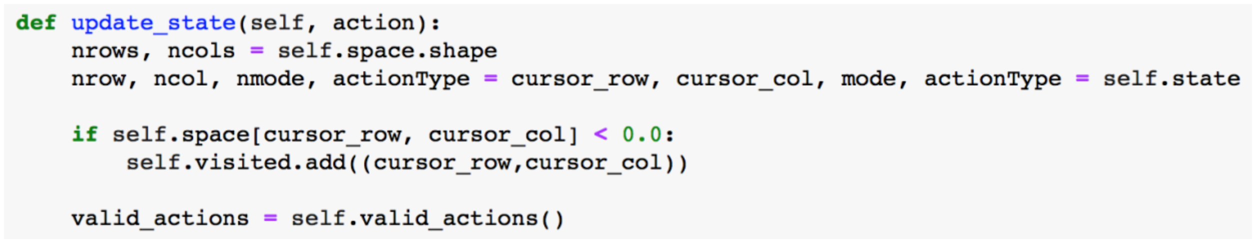
The Qspace Class:

```
class Qspace:
    def __init__(self, space, cursor = (0,0)): # Note that we start at 0,0
        self._space = np.array(space)
        nrows, ncols = self._space.shape
        self.target = ((nrows-1, ncols-1)) # Here is the point we are trying to get to
        self.free_cells = [(r,c) for r in range(nrows) for c in range(ncols) if self._space[r,c] != 0.0]
        self.free_cells.remove(self.target)
        if self._space[self.target] == 0.0:
            raise Expection("Invalid Space: target space is blocked!")
        self.reset(cursor)
```

As an instance of the class Qspace is created a few things are initially defined for the instance. First off, the original environment is assigned to the object, and the environment is dissected for its dimensions. Next, a target is defined within the environment in the final element of the array, and free cells are listed, excluding the target cell. The free cells come into play when 0's are placed in the space array in the environment development process. These 0's will be seen as obstructions, or obstacles. A 0, or obstacle, cannot be placed in the target cell. Finally, the reset method is called.

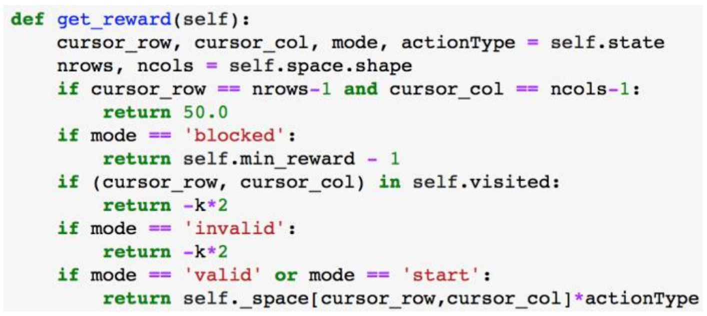
```
def reset(self, cursor):
    self.cursor = cursor
    self.space = np.copy(self._space)
    nrows, ncols = self.space.shape
    row, col = cursor
    self.space[row, col] = cursor_space
    self.state = (row, col, 'start', 1)
    self.min_reward =  (-k)*space.size
    self.total_reward = 0
    self.visited = set()
```

The reset method further sets the scene for the new instance of Qspace. As the code depicts, the cursor is further defined within the environment and the state of the initial state of the cursor is created. Next, the minimum reward is defined, in which each epoch cannot fall below, in order to speed up the learning process. Th minimum reward is defined as the value -k times the amount of spaces in the environment, as it is realistic to say that after visiting at least the number of squares in the environment, the path is not only non-optimal, but it is a waste of time to finish. The total reward for a path is also defined, and set to 0 to begin a given path. A list is then created to keep track of the squares that are visited within a given path.



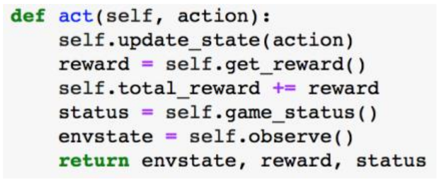
```python
def update_state(self, action):
    nrows, ncols = self.space.shape
    nrow, ncol, nmode, actionType = cursor_row, cursor_col, mode, actionType = self.state

    if self.space[cursor_row, cursor_col] < 0.0:
        self.visited.add((cursor_row,cursor_col))

    valid_actions = self.valid_actions()
```

The update_state method is used to take an action and move the agent. As the rest of the code within the method demonstrates the movement given a specific action, note that the diagonal movements multiply the reward by the square root of 2. This is a property of a 45-45-90 triangle, in which the hypotenuse is longer that either of the equivalent legs. If this factor is not included, then the cursor will always find the diagonal path from A to B to be the fastest, as the cursor might as well go down while moving sideways if there is no reward difference from the side to side movement. As the movement takes place, the new state of the cursor is defined.



```python
def get_reward(self):
    cursor_row, cursor_col, mode, actionType = self.state
    nrows, ncols = self.space.shape
    if cursor_row == nrows-1 and cursor_col == ncols-1:
        return 50.0
    if mode == 'blocked':
        return self.min_reward - 1
    if (cursor_row, cursor_col) in self.visited:
        return -k*2
    if mode == 'invalid':
        return -k*2
    if mode == 'valid' or mode == 'start':
        return self._space[cursor_row,cursor_col]*actionType
```

The get_reward method is defined to get the reward for the the action that had just taken place. Utilizing the updated state and the type of action (diagonal or side to side / up and down) the reward for the specific space is provided, as defined when the environment was created. If the space had already been created, a reward of -k*2 is provided, and if the space is 'blocked' the total reward will fall below the previously set minimum reward. The previously noted rewards are all deductions, which are not enjoyed by the learning algorithm. In order to converge to a certain path, a reward must be provided when the cursor gets to the target cell. This reward is set to 50, as this will distinguish a winning path from a losing one.



```python
def act(self, action):
    self.update_state(action)
    reward = self.get_reward()
    self.total_reward += reward
    status = self.game_status()
    envstate = self.observe()
    return envstate, reward, status
```

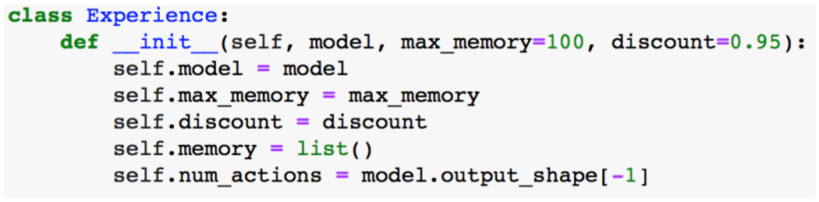
Next, the act method will take the action, call the update_state method to move the cursor. The act method then keeps track of the total reward as it also calls the get_reward method for the new state. The act method then returns the environment state, the reward for the new action, and the status of the game, utilizing the game_status and observe methods. The game_status method returns the status of the game and the observe method returns the environment state.

Another important method is the valid_actions method which keeps track of the actions that can be taken given the current state of the cursor. For example, the cursor cannot take the up action if it is on the top row. If actions are prohibited then, they will be removed from the available actions list which is used when training the model.

Additional Methods:

Other important methods which are used include show, test_path, and completion_check. The show method is for visualizing the cursor emerged in the environment canvas. The test_path method is used in the completion_check method to go through the environment and tell until there is a loss. The completion check makes sure that the objective is completed, as it may sound.

The Experience Method:



```
class Experience:
    def __init__(self, model, max_memory=100, discount=0.95):
        self.model = model
        self.max_memory = max_memory
        self.discount = discount
        self.memory = list()
        self.num_actions = model.output_shape[-1]
```

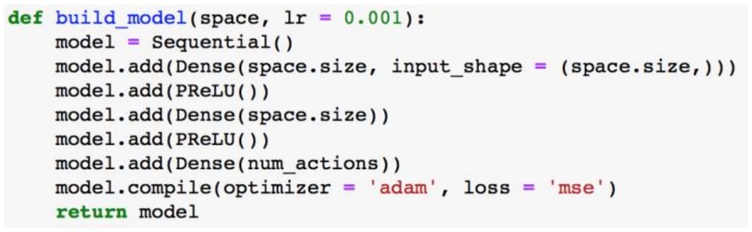
Beginning with the initializing method, the neural network model is assigned to the new instance of the Experience class, as well as the max_memory and the discount factor. The max_memeory is the maximum amount of episodes that will be remembered at once. The discount factor adjusts how we use the future reward in figuring the action to take in a particular state that will yield the highest reward. With a discount of 0.95, there will be a good amount of future transition figured into the value given an action and a state. The discount comes from the Bellman's equation which is a function of the actions and a state in which the function gives the value of a state given the instant reward and the maximum value for the next state and a given action, which is weighted by the discount, gamma.

The remember method stores the episodes as memory and if the memory is exceeded then the oldest episode is deleted and the newest is added. The predict method will use the model to predict the next action given the current environmental status.

The qtrian Method:

The qtrain method is the heart of the reinforcement learning as it loops over the path finding process in trails called epochs. The qtrain method can be manipulated by the number of epochs set, the maximum memory and the data size. As the path follows the model, the previously mentioned exploration factor, epsilon is lowered to converge to a solution. As the qtrain method goes through epochs the arguments from the model are updated.

The Model:

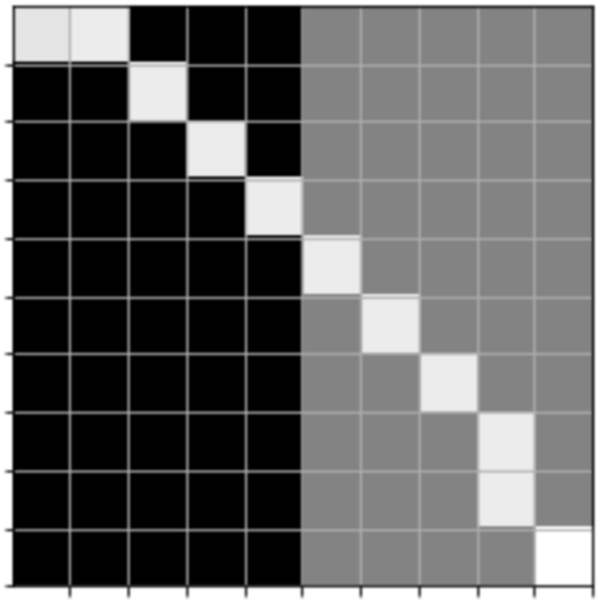


```
def build_model(space, lr = 0.001):
    model = Sequential()
    model.add(Dense(space.size, input_shape = (space.size,)))
    model.add(PReLU())
    model.add(Dense(space.size))
    model.add(PReLU())
    model.add(Dense(num_actions))
    model.compile(optimizer = 'adam', loss = 'mse')
    return model
```

In the build_model method the model is defined. Two layers were added to create a forward feeding neural network using a PReLU activation function. The model also uses the mean squared error as the loss function.

The Preliminary Results

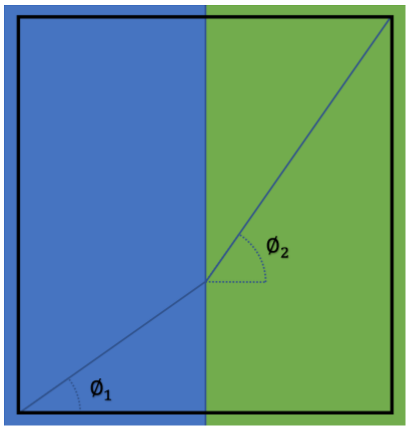
The results from the reinforcement learning algorithm are not the best at the preliminary stage. The limiting factor comes with the actions. For an improved learning algorithm, a new set of actions should be used involving a single step size, with a degree system from 0 to 360 with intervals. This would yield a better expected result. The current results, as pictured, do show a preference on where



point C should be located. This is important, as it shows the reinforcement learning model learned the physics behind the scenario. Further actions should be improved, and followed by the integration of obstacles to block the optimal path. After that the model should be adjusted in order to see if any improvements can be made.

The Next Step

With the grid format failing to produce a solution comparable to the expected solution, the polar action space will be used with the state space being altered to 2 dimensions, water or land. This is realistic because there should be a single direction in which the agent moves through the respective states, given a k value greater than 1.



After attempting to use such criteria with q learning instead of deep q learning, a problem was found in which the agent would become stuck if the optimal policy was between 90 and 360 degrees for either water or land. This becomes an issue, which led to a simplification due to the nature of the problem. The problem is to find the fastest path, which means we already know the general direction in which we are attempting to go. This allows us to reduce the action space to 1 to 89 degrees according to the new drawing. While adding this new parameter, an even simpler method for machine learning was introduced. The optimal policy was discovered by maximizing the total reward of a tested policies. The best policy was varied slightly producing a similar policy, which would either be better or worse. If better we then use this policy as our best policy, and continue the process until a solution is found. This method works great for the small state space and limited action space of the given problem. The reason that this method works is because there is only a single, unique solution to the problem, meaning it will not just get stuck at some path which has a local maximum total reward. In the end, the solution provided the new parameters and simplified method matches the analytical solution. This was found to be true because when k is 15, the first angle should be between 3 and 4, and the second should be between 62 and 63. The optimal policy was found to be 3 and 63 for water and land respectively. Another check was when k = 1. The optimal path was 45 degrees in both water and on land. This makes sense, as the fastest path when moving at a single speed is straight from start to finish. In the end, the machine learning application was found to be successful, in which the optimal path was found.

Final Code

```
import numpy as np

# Adjust the k factor, this is how much faster you move on land than in water
k = 15

# define the reward function given the two states, water and land
r = [-k,-1]

num_episode = 1000 # number of episodes
step_size = 0.01 # "Distance" per step
pi = np.pi # pi

count = 0 # counter for episodes

# create random best actions and very small best reward
# Note that the current knowledge is that the target is somewhere between 0 and 90 degrees
bestActions = [np.random.randint(0,90), np.random.randint(0,90)]
bestReward = -10000000

# learn the best path
while count <= num_episode:

  # Start at the bottom left corner in water each episode with no total reward
  cursorX = 0
  cursorY = 0
  s = 0
  totalReward = 0

  gameStatus = False # The status of the path

  # create set of test actions which slightly vary from the best actions
  # we can do this because we know there is single unique solution to our problem
  rand = np.random.randint(0,5)
  dec = np.random.rand()
  if dec > 0.5:
    tempOne = rand
  else:
    tempOne = -rand

  rand = np.random.randint(0,5)
  dec = np.random.rand()
```

```
if dec > 0.5:
   tempTwo = rand
else:
   tempTwo = -rand

testActions = [bestActions[0] + tempOne, bestActions[1] + tempTwo]

# make sure the test actions make sense
if testActions[0] <= 0:
   testActions[0] = 1
if testActions[0] >= 90:
   testActions = 89
if testActions[1] <= 0:
   testActions[1] = 1
if testActions[1] >= 90:
   testActions[1] = 89

# test the test policy
while gameStatus == False:

   # update cursor
   cursorX += step_size*np.cos(testActions[s]*pi/180)
   cursorY += step_size*np.sin(testActions[s]*pi/180)

   # Maintian the cursor within the 10 by 10 block
   if cursorX < 0:
      cursorX = 0
   elif cursorX > 10:
      cursorX = 10
   if cursorY < 0:
      cursorY = 0
   elif cursorY > 10:
      cursorY = 10

   # Define the current state
   if cursorX <= 5:
      s_next = 0
   else:
      s_next = 1

   # update the total reward for the current test policy
   totalReward += r[s]

   # see if the cursor hit the target
```

```
        if cursorX > 9.9 and cursorY > 9.9:
            gameStatus = True

        # update the state
        s = s_next

    # update best actions
    if totalReward > bestReward:
        bestReward = totalReward
        bestActions[0] = testActions[0]
        bestActions[1] = testActions[1]
    count += 1

print(bestActions)
```