John Murphy

CAP 5636

Reinforcement Learning

**Q1**. Value Iteration

Reading the question, it states we need to implement value iteration within the $ValueIterationAgents.py$

So a few things I noted was that we need to get the Q values since we are passed in an MDP. It also states that this isn't reinforcement and rather that it'll train better depending on the iterations. They also throw out some valuable information such as:

$$computeActionValues(state)$$

$$computeQValuefromValues(state, action)$$

$$util.Counter$$

Which we'll probably be diving into to implement into our Value iteration.

```
    discount factor.
"""
def __init__(self, mdp, discount = 0.9, iterations = 100):
    """
      Your value iteration agent should take an mdp on
      construction, run the indicated number of iterations
      and then act according to the resulting policy.

      Some useful mdp methods you will use:
          mdp.getStates()
          mdp.getPossibleActions(state)
          mdp.getTransitionStatesAndProbs(state, action)
          mdp.getReward(state, action, nextState)
          mdp.isTerminal(state)
    """
    self.mdp = mdp
    self.discount = discount
    self.iterations = iterations
    self.values = util.Counter() # A Counter is a dict with default 0
    self.runValueIteration()
```

So we do have some things instantiated for us such as our mdp, discounts, iterations and values

**Our first function is : $runValueIteration$**

Some things to dive into: $mdp$.

We'll need to go through all the states in our MDP, for N iterations.

```
for i in range(self.iterations):
    for j, state in enumerate(self.mdp.getStates()):
```

Next we have to check if we are in a terminal state or not and get our current states actions:

```
if not self.mdp.isTerminal(state):
    #get the current states actions

    curr_state_actions = []
```

Now we need to check in the mdp file where we could obtain those actions:

```
def getPossibleActions(self, state):
    """
    Return list of possible actions from 'state'.
    """
    abstract
```

This will give us all possible actions from the state we are.

So now we need to check all possible actions within our current state:

```
curr_state_actions = []
for k, actions in enumerate(self.mdp.getPossibleActions(state)):
```

From all possible actions, compute the Q values of all actions which we get from:

- `computeActionFromValues(state)` computes the best action according to the value function given by `self.values`.
- `computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by `self.values`.

We can grab $computeQValueFromValues$ function from self.values.

Now we append all Q values from that current state to our current state actions list.

```
for k, actions in enumerate(self.mdp.getPossibleActions(state)):
    curr_state_actions.append(self.computeQValueFromValues(state, actions))
```

Bug:

I forgot to keep track of all of our states not just the current state: this dawned on my while reading the hints for using Util.Counter from the website.

*Hint:* You may optionally use the `util.Counter` class in `util.py`, which is a dictionary with a default value of zero. However, be careful with `argMax`: the actual argmax you want may be a key not in the counter!

This way I can keep track of all the best actions from each state and use curr_actions_states for calculating solely the current state in which we are at.

```
for i in range(self.iterations):
    for j, state in enumerate(self.mdp.getStates()):
        global_counter = util.Counter()
```

From here, to get our V value, we need to get the MAX weight from the actions in our state:

$$\max_a \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma V_k(s') \right]$$

And since we traversed through all the possible actions we can just get the best weighted action to append to the global_counters state.

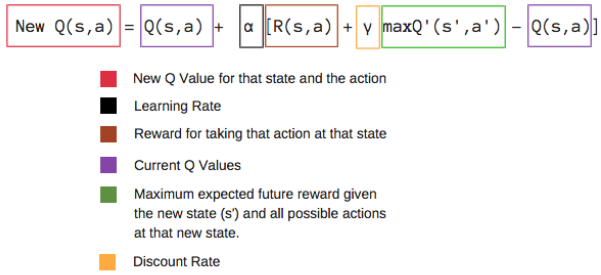
```
    global_counter[state] = max(curr_state_actions)
```

Finally, we set the current value to the max action at that state:

```
self.values = global_counter
```

**Second Function: *computeQValueFromValues***

In order to compute the Q values we need to calculate all actions weights.




So lets check how we can obtain all of our actions from each state.

```
def getTransitionStatesAndProbs(self, state, action):
    """
```

The reason as to why we use this function over mdp.getActions is because this also gives us the probability that we'll take that path since our MDP is non-deterministic.

So we'll loop over all the states and probabilities of our actions from that state.

```
#Loop through all transition states
for i, (future, odds) in enumerate(self.mdp.getTransitionStatesAndProbs(state, action)):
```

Now lets follow the formula, we need to calculate:

$$current\ Q_{values} + L.R \times (Reward + Discount \times \max(\text{future}) - \text{curr})$$

```
#add the reward, decay(discount), with our odds and the val we got from before
curr_val = curr_val + ((self.mdp.getReward(state, action, future) + values[future] * discount) * odds)
```

Then we can append those to our global values and return the best value from a list of our global values.

```
    #append this to our global values
    global_vals.append(curr_val)

#return the best value
best = max(global_vals)
```

**Third Function: *computeActionFromValues***

Again from here, we need to iterate through all of our possible actions to get the best action. Similar to calculating the V values, they are tied hand in hand since the weight determines the action.

```
global_vals = []

#append all actions and weights to our val
for action in self.mdp.getPossibleActions(state):
    global_vals.append([action, self.computeQValueFromValues(state, action)])
```

So we'll have a similar setup again where we append all actions to global values to get the best action returned.

From here we can get solely the actions from the max weight and return the action since its setup in a tuple.

```
#Get the best "move" from the max weight
act = max(global_vals, key=lambda tup: tup[1])

return act[0]
```

Now we have our three functions setup time to test them out!

So I failed all the test cases and the error I got was this:

```
***    File "C:\Users\John Murphy\Desktop\School\FALL 2022\AI\HW 3\reinforcement-fixed\testCl
te
***      if not f(grades):
***    File "autograder.py", line 307, in <lambda>
***      return lambda grades: testCase.execute(grades, moduleDict, solutionDict)
***    File "reinforcementTestClasses.py", line 60, in execute
***      testPass, stdOutString, fileOutString = self.executeNIterations(grades, moduleDict,
)
***    File "reinforcementTestClasses.py", line 73, in executeNIterations
***      valuesPretty, qValuesPretty, actions, policyPretty = self.runAgent(moduleDict, n)
***    File "reinforcementTestClasses.py", line 132, in runAgent
***      policy[state] = agent.computeActionFromValues(state)
***    File "valueIterationAgents.py", line 146, in computeActionFromValues
***      act = max(global_vals, key=lambda tup: tup[1])
*** ValueError: max() arg is an empty sequence
***

### Question q1: 0/4 ###
```

Max arg is an empty sequence so now I need to see where I use max and pass nothing through.

FIX:

After spending some time I realize in the function: $computeActionfromValues$

I never checked if I was in a terminal state.

```
#Check if we are in a terminal state or not
if not self.mdp.isTerminal(state):
    global_vals = []
```

Now I check and if I am in a terminal state, return None.

```
return None
```

Now lets try again:

```
Q-Values at iteration 1 for action north are NOT correct.
   Student solution:
 q_values_k_1_action_north: """
          illegal      0.0000      illegal      0.0000      0.0000
          illegal      0.0000    _________      0.0000      0.0000
          illegal      0.0000      illegal      0.0000      0.0000
          illegal      0.0000    _________    _________     0.0000
          illegal      0.0000       0.0000      0.0000      0.0000
"""
```

I got a bunch of illegal actions so now its time to dive into the code again.

FIX:

```
For more details to help you debug, see test output file test_cases\q1\3-bridge.test_output
```

After spending some time with the debugger, and scrolling through the code line by line I realized that the global_counter in $runValueIteration$ was getting reset after every state rather than after every iteration.

I had:

```
for i in range(self.iterations):
    #Iterate through the MDP
    for j, state in enumerate(self.mdp.getStates()):
        global_counter = util.Counter()
```

When I needed the global_counter to be rest after each iteration

```
for i in range(self.iterations):
    global_counter = util.Counter()
    #Iterate through the MDP
    for j, state in enumerate(self.mdp.getStates()):
```

Now if we try the Q1 grader again …

We pass all test cases!

```
Question q1
===========

*** PASS: test_cases\q1\1-tinygrid.test
*** PASS: test_cases\q1\2-tinygrid-noisy.test
*** PASS: test_cases\q1\3-bridge.test
*** PASS: test_cases\q1\4-discountgrid.test

### Question q1: 4/4 ###
```

**Q2.**

For this, this is analyzing the MDP and seeing what best suits the AI to make better choices. Since we want our AI to go across a bridge and take a direct path, we want it to be **Deterministic** in nature, meaning we are going to get rid of the noise for it to cross the bridge without taking a left or right or explore other options which in our case would be either falling off or going backwards which won't help. Not only will that promote going back and forth since if we stay alive, its better than falling off, we may get stuck on the bridge from this, so its better to get rid of any noise in the first place.

Answer: $Noise = 0.2 -> 0$

```
Question q2
===========

*** PASS: test_cases\q2\1-bridge-grid.test

### Question q2: 1/1 ###
```

**Q3.** This map has us on the edge of the cliff with varying scenarios/rewards we get

**3A. Prefer the close exit (+1), risking the cliff (-10)**

- We want this to be deterministic since we never want to fall off the cliff
- Hinder exploration to get toward the exit since the negative reward is so high
- Also have 0 living reward since we are "risking" the cliff
- Also add a little bit of a discount the longer we take to find the optimal path
    - $answerDiscount: 0.3$
    - $answerNoise: 0$
    - $answerLivingReward: 0.0$

Which has us passing!

**3B. Prefer the close exit (+1), but avoiding the cliff (-10)**

- This one is very similar to 3A except now we want to avoid the cliff
- I will try the same weights except now add a living reward as a negative
- This is to help promote to get to an exit as quickly as possible
    - $answerDiscount: 0.3$
    - $answerNoise: 0$
    - $answerLivingReward: -1$

```
** PASS: test_cases\q3\1-question-3.1.test
** FAIL: test_cases\q3\2-question-3.2.test
**     Policy not correct.
**         Student policy at (0, 2): south
**         Correct policy at (0, 2): north
**         Student policy:
**             E    E    S    S    S
**             N    .    S    E    S
**             S    .    X    .    X
**             E    E    N    E    N
**             X    X    X    X    X
**             Legend:  N,S,E,W at states which move north etc, X at states which exit,
**                      . at states where the policy is not defined (e.g. walls)
**         Correct policy specification:
**             E    E    S    _    _
**             N    _    S    _    _
**             N    _    _    _    _
**             N    _    _    _    _
**
**             _    _    _    _    _
**             Legend:  N,S,E,W for states in which the student policy must move north etc,
**                      _ for states where it doesn't matter what the student policy does.
**         Gridworld:
```

We failed 3B. time to tweak the weights. I think this time we need to increase the answer Discount.

- $answerDiscount: 0.6$
- $answerNoise: 0$
- $answerLivingReward: -1$

This still fails and it tells me:

```
**         Correct policy specification:
**             E    E    S    _    _
**             N    _    S    _    _
**             N    _    _    _    _
**             N    _    _    _    _
**
**             _    _    _    _    _
**             Legend:  N,S,E,W for states in which the student policy must move north etc,
**                      _ for states where it doesn't matter what the student policy does.
```

That it should be going north no matter what the students policy is, meaning we need to add some noise to allow for this to happen. I'm also going to tweak down the discount.

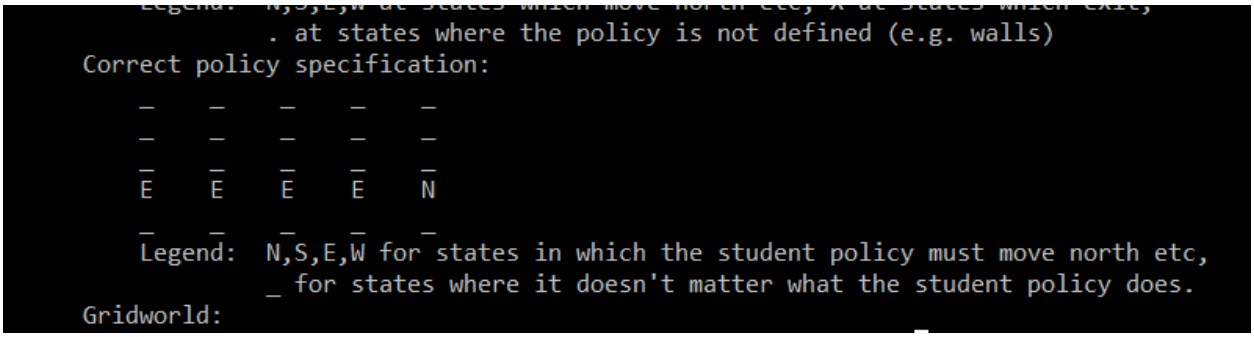
- $answerDiscount: 0.3$
- $answerNoise: 0.3$
- $answerLivingReward: -1$

Now we pass $3B$

### 3C. Prefer the distant exit (+10), risking the cliff (-10)

- Similar to 3A except now we want the distant exit
- We are still risking the cliff
- So we can have the same weights except now we can include a living reward
- This helps promote us to traverse further
- Since we still risk the cliff we add noise
    - $answerDiscount: 0.3$

- $answerNoise$: 0.3
- $answerLivingReward$: 1




We failed here so I'll try increasing the discount.

- $answerDiscount$: 0.6
- $answerNoise$: 0.3
- $answerLivingReward$: 1

We still fail and get the same output?

So after spending some time fiddling with the values, it doesn't want any noise even though we are risking the cliff, we still want to get rid of guess work because of the cliff and the living reward is what's helping us explore toward the farther goal.

- $answerDiscount$: 0.3
- $answerNoise$: 0
- $answerLivingReward$: 1

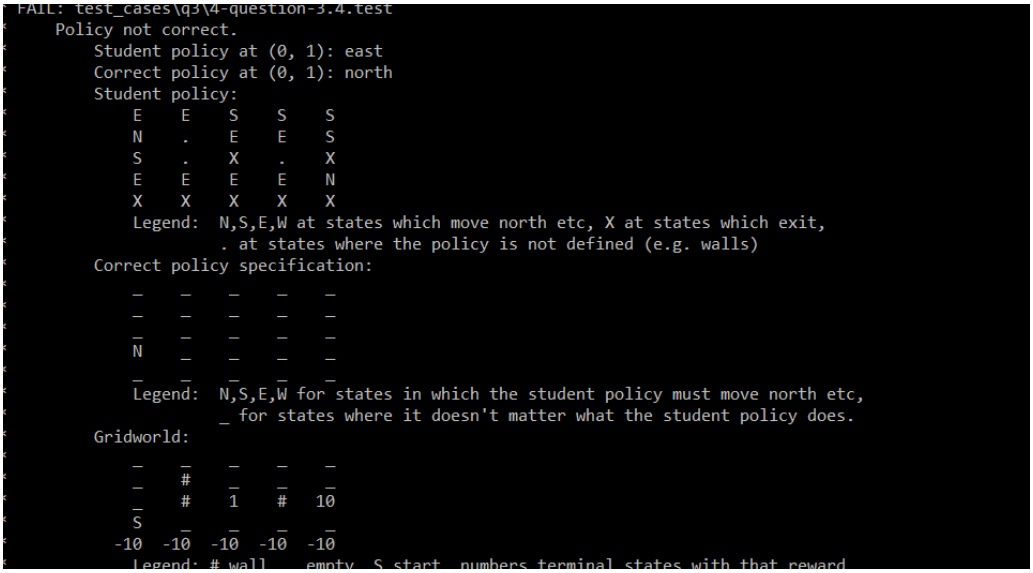
Since we allow for a positive living reward, we pass since we try to stay alive which not only promotes traversal, it also allows us to get to the further exit rather than finding the one closest and going there as quick as possible. The reason we still use a discount is to not exhaust exploration and allow the living reward to overcome the discount. This is a good balance.

Now we pass $3C$.

**3D. Prefer the distant exit (+10), avoiding the cliff (-10)**

- Similar to 3C except now we want to avoid the cliff which hinders exploration
- I'll start with the same weights as 3C as a base

- $answerDiscount: 0.3$
- $answerNoise: 0$
- $answerLivingReward: 1$




I kind of figured it wouldn't work so lets check to see what the error its giving me.

It's telling me there are some states in which it shouldn't matter if I go north or not, meaning its time to add some Noise to the answer.

- $answerDiscount: 0.3$
- $answerNoise: 0.3$
- $answerLivingReward: 1$

Now we pass $3D$ The noise allows us to explore the MDP with some randomness making it non deterministic. 3D ended up being the same as 3B except we added a positive reward rather than a negative to prefer the distant exit.

**3E. Avoid both exits and the cliff (so an episode should never terminate)**

- This one should probably have a high living reward
- No discount
- And no noise so we don't randomly go into a terminal state
  - $answerDiscount: 0$
  - $answerNoise: 0$
  - $answerLivingReward: 1$

And it worked because it keeps us alive with 0 discount and no noise to accidently let us leave.

```
Question q3
===========

*** PASS: test_cases\q3\1-question-3.1.test
*** PASS: test_cases\q3\2-question-3.2.test
*** PASS: test_cases\q3\3-question-3.3.test
*** PASS: test_cases\q3\4-question-3.4.test
*** PASS: test_cases\q3\5-question-3.5.test

### Question q3: 5/5 ###
```

We have passed all 5 test cases!

**Q6.**

$$init$$

$$getQValue$$

$$computeValueFromQValues$$

$$computeActionFromQValues$$

$$getAction$$

$$update$$

Now its time to implement Q-Learning!

We will do this by adding code to the 6 functions from above.

**Function 1 : *init***

Here we are just initializing our q values which store our states and actions meaning we can store this within a dictionary.

```
"*** YOUR CODE HERE ***"
#add q values
self.Q = {}
```

**Function 2:** $getQValue$

```
"""
  Returns Q(state,action)
  Should return 0.0 if we have never seen a state
  or the Q node value otherwise
"""
"*** YOUR CODE HERE ***"
```

Here we are given information saying that if haven't seen the state yet, to just return 0.0 otherwise we'll return our state and action

So we can do an easy ternary statement that'll return Q's state action if its in self.Q

```
"*** YOUR CODE HERE ***"
return self.Q[state][action] if state in self.Q and action in self.Q else 0.0
```

**Function 3:** $computeValueFromQValues$

```
def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
```

Again we are given instructions to return the max action from our Q state/action where the max is over all legal actions.

So we first need to check if our action is even legal, which is given to us by:

```
  Functions you should use
    - self.getLegalActions(state)
      which returns legal actions for a state
"""
```

So now we can setup our legal actions from a current state

```
legal_actions = self.getLegalActions(state)
```

Lets see what legal_actions outputs:

BUG: I'm actually getting an error saying action is not defined even though it is passed through into my function?

FIX: the bug was that the action is coming from self.Q[state] not just self.Q itself, we need to be within a specific state to check what action is available to us.

```
and action in self.Q[state]
```

Back to what legal_actions, it wont even let me print out the output for it because we haven't implemented all the functions.

So from here, we need to make sure we are in a legal_action to ensure we're doing legal moves otherwise we'll need to return 0.0

```
if legal_actions:

else:
    return 0.0
```

Now we need to compute the max or best value and get our actions from the current state.

```
actions = self.getQValue(state, action)
```

BUG:

We need to loop through all legal actions:

FIX:

```
legal_actions = self.getLegalActions(state)
actions = []

if legal_actions:
    for action in legal_actions:
    actions.append(self.getQValue(state, action))
```

BUG: Float object is not iterable

```
Question q6
===========

[0.0]
*** FAIL: Exception raised: 'float' object is not iterable
***
```

No idea how to fix at the moment I'm going to go onto the next function but this was my end result for now.

```
    legal_actions = self.getLegalActions(state)
    actions = []

    if legal_actions:
        for action in legal_actions:
            actions.append(self.getQValue(state, action))

        return max(actions)

    else:
        return 0.0
```

**Function 4 :** *computeActionfromQValue*

This is similar to the prior function except now we want the action not the value and we return none if there is no legal action to take.

```
    "*** YOUR CODE HERE ***"
    legal_actions = self.getLegalActions(state)
    actions = []

    if legal_actions:
        for action in legal_actions:
            actions.append(self.getQValue(state, actions))
        return max(actions)
    else:
        return None
```

So we'll try this for now and see if it works.

Coming back after getting Q6 to finally work, this wasn't correct this was the correct final code:

```
"*** YOUR CODE HERE ***"
legal_actions = self.getLegalActions(state)
best = []
max_val = -999

for action in legal_actions:
    if max_val == self.getQValue(state,action):
        best.append(action)

    elif max_val < self.getQValue(state,action):
        max_val = self.getQValue(state,action)
        best = [action]


#return the best action
return max(best) if best is not [] else None
```

The main difference here is I was appending all of my actions when I only needed the "best" actions to be appended when the max value was the Q value otherwise to make the max value the current Q value. And then to return the best Q value from all the actions that we appeneded!

**Function 5 :** $getAction$

```
"""
  Compute the action to take in the current state.  With
  probability self.epsilon, we should take a random action and
  take the best policy action otherwise.  Note that if there are
  no legal actions, which is the case at the terminal state, you
  should choose None as the action.

  HINT: You might want to use util.flipCoin(prob)
  HINT: To pick randomly from a list, use random.choice(list)
"""
```

I first like to see what they say which is:

1. Compute the action to take with a probability of $self.epsilon$
2. Otherwise take a random action
3. Otherwise take the best policy action

Then they give us some hints such as using $util.flipCoin$ and $random.choice(list)$

Now we can start.

First check if its even a legal action.

Next lets check the coin flip with epsilon if its true, return a random choice otherwise return the best policy

So after along time I got the code fixed up to the end result being :

```
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    #get a random choice from the choice of legal actions
    if not legalActions:
        return action

    elif util.flipCoin(self.epsilon):
        return random.choice(legalActions)
    else:
        return self.getPolicy(state)
```

**Function 6 :** *update*

```
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here

      NOTE: You should never call this function,
      it will be called on your behalf
    """
```

In this function we need to:

- Have our reward transition and next state from our current state and action

So lets first get the Qvalue from the current state and action

```
    "*** YOUR CODE HERE ***"
    Qvalues = self.getQValue(state, action)
```

Now we can check if the state is within Q then we apply the formula otherwise we make the action there 0.

```
    #Initilize the state's action to 0
    if state not in self.Q:
        self.Q[state] = {action: 0.0}
    else:
        self.Q[state][action] = Qvalues + self.alpha * (reward + self.getValue(nextState) * self.discount - Qvalues)
```

BUG:

```
    if not f(grades):
  File "autograder.py", line 307, in <lambda>
    return lambda grades: testCase.execute(grades, moduleDict, solutionD
  File "reinforcementTestClasses.py", line 451, in execute
    testPass, stdOutString, fileOutString = self.executeNExperiences(gra
dPolicy)
  File "reinforcementTestClasses.py", line 464, in executeNExperiences
    valuesPretty, qValuesPretty, actions, policyPretty, lastExperience =
  File "reinforcementTestClasses.py", line 539, in runAgent
    policyPretty = self.prettyPolicy(policy)
  File "reinforcementTestClasses.py", line 568, in prettyPolicy
    return self.prettyPrint(policy, '{0:10s}')
  File "reinforcementTestClasses.py", line 557, in prettyPrint
    row.append(formatString.format(elements[(x,y)]))
ValueError: Unknown format code 's' for object of type 'float'
```

Unknown format.

This was due to the forth function on how I was appending and when I was appending.

The final result of the final function was:

```
"*** YOUR CODE HERE ***"
Qvalues = self.getQValue(state, action)

if state not in self.Q:
    self.Q[state] = {action: 0.0}

self.Q[state][action] = Qvalues + self.alpha * (reward + self.getValue(nextState) * self.discount - Qvalues)
```

BUG: Before I fixed it I had the final line indented in the if statement when it needed to be outside to update the state/action of self.Q.

```
Question q6
===========

*** PASS: test_cases\q6\1-tinygrid.test
*** PASS: test_cases\q6\2-tinygrid-noisy.test
*** PASS: test_cases\q6\3-bridge.test
*** PASS: test_cases\q6\4-discountgrid.test

### Question q6: 4/4 ###
```

We are now finished.

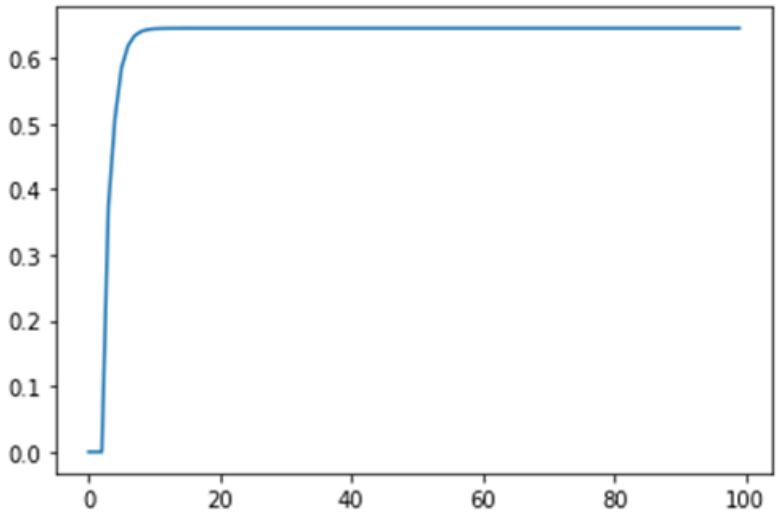
Q2. Using the command: $python\ gridworld.py\ -a\ value\ -i\ 100\ -k\ 10$

STATE = (0,2)

This is what the V values look over 100 iterations. It diverges towards .64 the top left corner

Where Y is the weight value

And X is the iterations





Which is exactly what we get here in the top left corner.

Q3. On state(1,2) after 10 iterations.