

PRACTICA 3

INTELIGENCIA ARTIFICIAL

José María Fernández Gómez

Pregunta 1 (4 puntos): Iteración de valor:

-------------------------CODIGO------------------------------

class ValueIterationAgent(ValueEstimationAgent):

def \_\_init\_\_(self, mdp, discount = 0.9, iterations = 100):

self.mdp = mdp

self.discount = discount

self.iterations = iterations

self.values = util.Counter() # A Counter is a dict with default 0

self.runValueIteration()

def runValueIteration(self):

# Write value iteration code here

"\*\*\* YOUR CODE HERE \*\*\*"

iterations=0 #Ponemos las iteraciones a 0

while iterations < self.iterations:

valsPi = util.Counter( )#Contador para saber los valores en funcion de la iteracion

states=self.mdp.getStates() #Todos los estados posibles

for state in states: #Recorremos todos los estados

if not self.mdp.isTerminal(state): #Si el estado no es terminal

valuesTer = util.Counter() #Inicializamos otro contador

actions = self.mdp.getPossibleActions(state)#Sacamos las posibles acciones

for action in actions:#Recorremos todas las acciones

valuesTer[action] = self.computeQValueFromValues(state, action)#Calculamos el q valor de cada accion

valsPi[state] = max(valuesTer.values())#Guardamos el valor maximo de cada estado

iterations += 1#Aumentamos las iteraciones

self.values = valsPi.copy() #Copiamos los valores para la siguiente iteracion

def getValue(self, state):

"""

Return the value of the state (computed in \_\_init\_\_).

"""

return self.values[state]

def computeQValueFromValues(self, state, action):

"""

Compute the Q-value of action in state from the

value function stored in self.values.

"""

"\*\*\* YOUR CODE HERE \*\*\*"

statesAndProbs=self.mdp.getTransitionStatesAndProbs(state,action)#Sacamos tuplas con el estado y la probabilidad de transicion

currVal=0#Inicializamos el valor a 0

for pair in statesAndProbs:#Recorremos los pares de estado y probabilidad

currVal+=pair[1]\*(self.mdp.getReward(state,action,pair[0])+self.discount\*self.values[pair[0]])#Calculamos el qValor con la formula

return currVal

def computeActionFromValues(self, state):

"""

The policy is the best action in the given state

according to the values currently stored in self.values.

You may break ties any way you see fit. Note that if

there are no legal actions, which is the case at the

terminal state, you should return None.

"""

"\*\*\* YOUR CODE HERE \*\*\*"

if self.mdp.isTerminal(state):#Si no hay acciones a seguir porque es estado terminal devolvemos none

return None

actions=self.mdp.getPossibleActions(state)#Extraemos las acciones posibles

if len(actions) == 0:#Si no hay acciones por lo que sea tambien devolvemos none

return None

values = util.Counter()#Contador con los valores que vamos a tener

for action in actions:#Parca cada accion que tengamos

values[action] = self.computeQValueFromValues(state, action)#Calculamos su q valor

return values.argMax()#Devolvemos la accion que hace que el valor sea maximo

def getPolicy(self, state):

return self.computeActionFromValues(state)

def getAction(self, state):

"Returns the policy at the state (no exploration)."

return self.computeActionFromValues(state)

def getQValue(self, state, action):

return self.computeQValueFromValues(state, action)

-------------------------END CODIGO-----------------------

Pregunta 2 (1 punto): Análisis de cruce de puentes

-------------------------CODIGO------------------------------

def question2():

answerDiscount = 0.9

answerNoise = 0

return answerDiscount, answerNoise

-------------------------END CODIGO--------------------

Tras probar varias posibilidades el valor 0 da el mejor resultado ya que prueba otros estados y no solo el del major qvalor

Pregunta 3 (5 puntos): Políticas

-------------------------CODIGO------------------------------

def question3a():

answerDiscount = 0.9

answerNoise = 0.2

answerLivingReward = -2

return answerDiscount, answerNoise, answerLivingReward

# If not possible, return 'NOT POSSIBLE'

def question3b():

answerDiscount = 0.5

answerNoise = 0.4

answerLivingReward = -0.5

return answerDiscount, answerNoise, answerLivingReward

# If not possible, return 'NOT POSSIBLE'

def question3c():

answerDiscount = 0.9

answerNoise = 0

answerLivingReward = -3

return answerDiscount, answerNoise, answerLivingReward

# If not possible, return 'NOT POSSIBLE'

def question3d():

answerDiscount = 0.8

answerNoise = 0.4

answerLivingReward = -0.4

return answerDiscount, answerNoise, answerLivingReward

# If not possible, return 'NOT POSSIBLE'

def question3e():

answerDiscount = 0.5 #Siga buscando salidas

answerNoise = 0.5 #Explora salidas diferentes

answerLivingReward = -0.8 #Da igual si te mueres

return answerDiscount, answerNoise, answerLivingReward

# If not possible, return 'NOT POSSIBLE'

-------------------------END CODIGO-----------------------

Por prueba y error he deducido que el primer parámetro ayuda a encontrar la primera salida, el segundo hace que sigas explorando y el tercero lo mucho o poco que importa caerse por el acantilado

Pregunta 4 (1 punto): Iteración asincrónica de valor

-------------------------CODIGO------------------------------

def runValueIteration(self):

"\*\*\* YOUR CODE HERE \*\*\*"

value = 0#Inicializar valor

maxValue = 99999999999999999#Valor máximo inicializado a infinito

states = self.mdp.getStates()#Sacar todos los estados

iterations = self.iterations#Inicializar iteraciones

for i in range(iterations):#Pasar por todas las iteraciones

current = util.Counter()#Estado actual

length = len(states)#Numero de estados

state = states[i%length]

if not self.mdp.isTerminal(state):#Si no es un estado terminal

maxValues = []#Inicializamos el valor máximo a 0

else:

current[state] = 0

actions = self.mdp.getPossibleActions(state)#Extraemos y recorremos todas las #acciones

for action in actions:

value = 0

nextAction = next

statesAndProbabilities = self.mdp.getTransitionStatesAndProbs(state,action)

#Extraemos estados y probabilidades de estos

for nextAction, prob in statesAndProbabilities:# Recorremos esas tuplas

reward = self.mdp.getReward(state,action,nextAction)# Calculamos la recompensa

discount = self.discount#Descuento de accion

value = value + prob \* (reward + discount\*self.values[nextAction])#Calculamos valor con la formula

maxValues = maxValues + [value]# Actualizamos el valor acumulado del valor

if value > maxValue: #Actualizamos el valor máximo comparándolo con el actual

maxValue = value

length = len(maxValues)#Tamaño de la lista de valores

if length!=0:#Si no tenemos mas valores

current[state] = max(maxValues)#El estado actual tiene el máximo valor de todos

self.values[state] = current[state]

-------------------------END CODIGO-----------------------

Pregunta 5 (3 puntos): Iteración de valor de barrido priorizada

-------------------------CODIGO------------------------------

def runValueIteration(self):

"\*\*\* YOUR CODE HERE \*\*\*"

fringe = util.PriorityQueue()

states = self.mdp.getStates()#Obtener todos los estados

predecessors = {} # Crear un nuevo diccionario vacío para los predecesores

for tState in states:#Recorremos todos los estados

previous = set()# Inicializar el conjunto, sin elementos duplicados en el conjunto

for state in states:

actions = self.mdp.getPossibleActions(state)#Sacamos las acciones posibles para cada estado

for action in actions:#Recorremos todas las acciones

transitions = self.mdp.getTransitionStatesAndProbs(state, action)#Tuplas con las transiciones y sus probabliidades

for next, probability in transitions:#Recorremos cada elemento de ese conjunto de tuplas

if probability != 0:#Si no hay probabilidad de ir a ese estado

if tState == next:#Si ya hemos recorrido ese estado

previous.add(state)#Añadimos a recorridos

predecessors[tState] = previous#Añadimos al array de predecesores el estado que acabamos de recorrer

for state in states:#Volvemos a iterar sobre los estados

if self.mdp.isTerminal(state) == False:#Si el estado actual no es terminal

current = self.values[state]#Extraemos el valor del estado actual

qValues = []#Inicilizamos el array de qvalores vacio

actions = self.mdp.getPossibleActions(state)#Extraemos las accione sposibles

for action in actions:#Recorremos esas acciones

tempValue = self.computeQValueFromValues(state, action)#Calculamos los qvalores para esas acciones

qValues = qValues + [tempValue]#Añadimos los qvalores a la lista

maxQvalue = max(qValues)#Extraemos la accion que maximiza ese qvalor

diff = current - maxQvalue#Calculamos el valor real restandole el mayor qvalor al valor actual

if diff > 0:#Si el valor real es mayor que 0

diff = diff \* -1#Lo multiplicamos por -1 para convertirlo a negativo

fringe.push(state, diff)#Añadimos el estado actual a la cola de prioridad que es nuestra franja de trabajo

for i in range(0, self.iterations):

if fringe.isEmpty():#sI NO TENEMOS ESTADOS DONDE TRABAJAR

break#Paramos las iteraciones

s = fringe.pop()#Extraemos el primero elemento de la cola

if not self.mdp.isTerminal(s):#Si no es un estado terminal

values = []#Inicializamos lista de valores

for action in self.mdp.getPossibleActions(s):#Recorremos las acciones posibles

value = 0

for next, prob in self.mdp.getTransitionStatesAndProbs(s, action):#Recorremos cada estado siguiente y la probabilidad que este

reward = self.mdp.getReward(s, action, next)

value = value + (prob \* (reward + (self.discount \* self.values[next])))#Calculamos el valor del estado con la formula

values.append(value)#Añadimos a la lista de valores

self.values[s] = max(values)#Nos quedamos con el valor mas grande

for previous in predecessors[s]:#Recorremos los estados anteriores al actual

current = self.values[previous]#Extraemos los valores de ese estado anterior

qValues = []

for action in self.mdp.getPossibleActions(previous):#Vemos las acciones del estado anterior

qValues += [self.computeQValueFromValues(previous, action)]#Calculamos los qvalores de el predecesor

maxQ = max(qValues)#Nos quedamos con el valor maximo

diff = abs((current - maxQ))#Obtenemos la diferencia de lo extraido y lo calculado

if (diff > self.theta):

fringe.update(previous, -diff)#Actualizamos la franja de trabajo con el estado anterior dandole como prioridad la diferencia calculada previamente

-------------------------END CODIGO-----------------------

Pregunta 6 (4 puntos): Q-Learning y Pregunta 7 (2 puntos): Epsilon Greedy

-------------------------CODIGO------------------------------

def \_\_init\_\_(self, \*\*args):

"You can initialize Q-values here..."

ReinforcementAgent.\_\_init\_\_(self, \*\*args)

self.qVals = util.Counter()#Inicilizamos

"\*\*\* YOUR CODE HERE \*\*\*"

def getQValue(self, state, action):

"""

Returns Q(state,action)

Should return 0.0 if we have never seen a state

or the Q node value otherwise

"""

"\*\*\* YOUR CODE HERE \*\*\*"

return self.qVals[(state, action)]#Devolver los qvalore de cada estado y accion posible

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.

"""

"\*\*\* YOUR CODE HERE \*\*\*"

actions = self.getLegalActions(state)#Sacamos todas las acciones posibles

values = []#Lista con los valores

if len(actions) == 0:#Si no tenemos acciones posibles devolvemos 0 como valor

return 0

else:

for action in actions:#Recorremos cada accion

values.append(self.getQValue(state, action))#Añadimos a nuestra lista de valores el calculo del q valor de cada accion

return max(values)#Devolvemos ese qvalor maximo

def computeActionFromQValues(self, state):

"""

Compute the best action to take in a state. Note that if there

are no legal actions, which is the case at the terminal state,

you should return None.

"""

"\*\*\* YOUR CODE HERE \*\*\*"

actions = self.getLegalActions(state)#Sacamos las acciones posibles

allActions = []

if len(actions) == 0:#Si no tenemos acciones posibles devolvemos 0 como valor

return None

else:

for action in actions:#Recorremos cada accion

allActions.append((self.getQValue(state, action), action))#Metemos en una lista el qvalor y la accion que representa ese qvalor

bestActions = [pair for pair in allActions if pair == max(allActions)] #Nos quedamos con la accion que mejor nos convenga

bestActionPair = random.choice(bestActions)#De las mejore acciones elegimos una al azar para explorar

return bestActionPair[1]#Una vez explorado nos quedamos con la mejor

def getAction(self, state):

"""

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)

"""

legalActions = self.getLegalActions(state)#Sacamos las acciones

p = self.epsilon#inicializamos epsilon

if util.flipCoin(p):#Aleatoriamente elegimos si exploramos o si nos quedamos con la mejor accion

return random.choice(legalActions)#Devolvemos accion aleatoria de todas las disponibles

else:#Si no devolvemos la mejor accion

return self.computeActionFromQValues(state)

def update(self, state, action, nextState, reward):

"""

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

"""

"\*\*\* YOUR CODE HERE \*\*\*"

qSa = self.getQValue(state, action)#Sacamos el qvalor de la accion

sample = reward + self.discount\*self.computeValueFromQValues(nextState)#Calculamos la muestra

self.qVals[(state, action)] = (1-self.alpha)\*qSa + self.alpha\*sample#Actualizamos la lista de qvalores con la formula

-------------------------END CODIGO-----------------------

Pregunta 8 (1 punto): Cruce de puentes revisitado

-------------------------CODIGO------------------------------

def question8():

answerEpsilon = None

answerLearningRate = None

return 'NOT POSSIBLE'

-------------------------END CODIGO-----------------------

No ha sido posible encontrar ninguno de estos parámetros para que devolviera la política optima con solo 50 iteraciones

Pregunta 9 (1 punto): Q-Learning y Pacman y Pregunta 10 (3 puntos): Q-Learning aproximado

-------------------------CODIGO------------------------------

def getAction(self, state):

"""

Simply calls the getAction method of QLearningAgent and then

informs parent of action for Pacman. Do not change or remove this

method.

"""

action = QLearningAgent.getAction(self,state)

self.doAction(state,action)

return action

class ApproximateQAgent(PacmanQAgent):

"""

ApproximateQLearningAgent

You should only have to overwrite getQValue

and update. All other QLearningAgent functions

should work as is.

"""

def \_\_init\_\_(self, extractor='IdentityExtractor', \*\*args):

self.featExtractor = util.lookup(extractor, globals())()

PacmanQAgent.\_\_init\_\_(self, \*\*args)

self.weights = util.Counter()

def getWeights(self):

return self.weights#Devolvemos los pesos

def getQValue(self, state, action):

"""

Should return Q(state,action) = w \* featureVector

where \* is the dotProduct operator

"""

"\*\*\* YOUR CODE HERE \*\*\*"

features = self.featExtractor.getFeatures(state,action)#Devolvemos las caracterisiticas de cada estado y accion posible

weights = self.getWeights()#Llamamos a esos pesos para evaluar

dotProduct = features\*weights#Miramos si el peso y las caracteristicas del estado nos compensa para ir a el

return dotProduct

def update(self, state, action, nextState, reward):

"""

Should update your weights based on transition

"""

"\*\*\* YOUR CODE HERE \*\*\*"

difference = reward + self.discount\*self.computeValueFromQValues(nextState) - self.getQValue(state, action)

weights = self.getWeights()

if len(weights) == 0:#Si no tenemos pesos en nuestra lista

weights[(state,action)] = 0#Metemos 0 para el peso de ese estado

features = self.featExtractor.getFeatures(state, action)#Extraemos las caracterisiticas de el estado

for key in features.keys():#Recorremos esas caracteristicas

features[key] = features[key]\*self.alpha\*difference

weights.\_\_radd\_\_(features)#Recalculamos el peso con la suma de esas caracterisiticas ajustadas

self.weights = weights.copy()#Actualizamos los pesos globales

def final(self, state):

"Called at the end of each game."

PacmanQAgent.final(self, state)#Llamamos al constructor

if self.episodesSoFar == self.numTraining:#Revisa si ya hemos hecho todos los entrenamientos

pass

-------------------------END CODIGO-----------------------