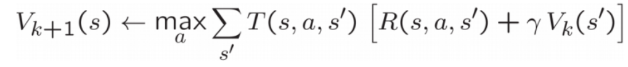
# Verslag labo 2

1. **Question 1: Value Iteration**

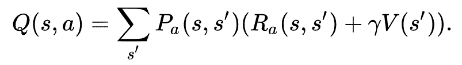
Om value iteration te kunnen uitvoeren, moeten we begrijpen wat er eigenlijk gebeurt in de formule:

De formule zegt dat je de maximum action moet nemen van de Q-value ofwel die som die er staat. In deze som staat er

1. T(s,a,s’): dit is de kans dat er over wordt gegaan van state s naar s’ via action a
2. R(s,a,s’): dit is de expected reward voor het overgaan van state s naar s’ via action a
3. yVk(s’): dit is de discounted (door de y) value van de volgende state s’ => deze discount zorgt ervoor dat het programma het in zo weinig mogelijk stappen zal proberen uitvoeren (aagezien het latere values steeds verkleint)

Value iteration berekent zo goed mogelijk (als k groter is, is de waarde beter geconverged) de optimale values Vk.

RunValueIteration zal de valueiteration uitvoeren en de code werkt als volgt: er zal stees over alle states gegaan worden en voor elke state worden steeds de actions bekeken en van die actions wordt steeds degene met de hoogste Qvalue bepaalt wat neerkomt op “de optimale value”. Enkel bij een terminal state zal de value niet bepaald worden want bij een terminal state moeten geen actions gebeuren. Deze optimale values worden opgeslagen in een tijdelijke array en uiteindelijk gestoken in self.values.

In computeQValueFromValues wordt op basis van de values in self.values. Dit wordt gedaan d.m.v. de formule:

Dit is hetzelfde als de som in de formule hierboven zonder het maximum ervan te nemen.

In computeActionFromValues wordt op basis van self.values steeds de optimale actions bepaald. Ik had dit eerst geprobeerd met een gewone dictionary maar dit werkte niet. Met de util.counter() die een extensie van de dictionary is, werkt dit uiteindelijk wel. In deze functie worden (Qvalue,action)-paren gemaakt voor een state waarvan later gewoon met de argmax() functie die de action bij de hoogste keywaarde neemt (wat neerkomt op action die hoort bij “de optimale value”).

1. **Question 2: Bridge Crossing Analysis**

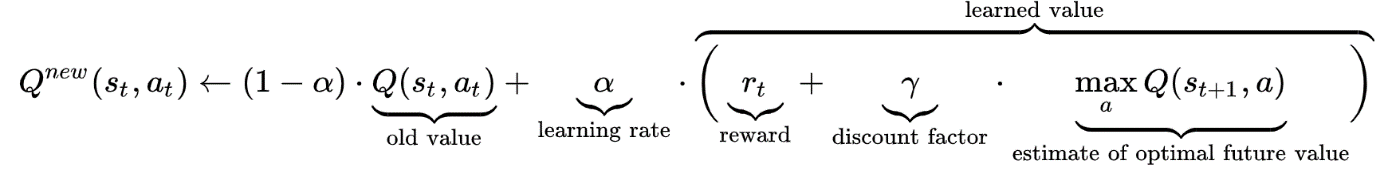
De bedoeling is om de brug over te steken door ofwel de discount ofwel de noise te veranderen. Het optimale policy wordt al gebruikt. Normaal zou het optimale policy altijd naar de hoogste reward moeten gaan (hangt ook wel af van de discount maar deze is niet zo hoog). Dus als we de noise naar 0 zetten, lukt het wel.

1. **Question 3: Policies**

Voor deze vraag is het belangrijk te weten wat de parameters betekenen. Discount en noise zijn hetzelfde als in vorige vraag => discount: als deze lager is, zullen meer stappen uiteindelijk een lager resultaat geven en als deze groter is, hebben meerdere stappen minder effect op de uiteindelijke totale beloning. Noise: hoe vaak dat de agent in een onbedoelde state terecht komt. De answerlivingreward geeft de reward voor elke move weer. In de code staan de parameters steeds met uitleg.

1. **Question 6: Q-Learning**

Q-Learning, in tegenstelling tot Value Iteration in opdracht 1, leert wel uit ervaring. In een Q-tabel wordt, voor elke combinatie van action en state, een Q-value bijgehouden. In het begin zijn deze waarden 0. Na een aantal keer spelen, wordt de tabel stelselmatig aangevuld. De Agent kiest zijn volgende actie door het zoeken van de hoogste Q-value in de huidige state.

In “QLearningAgent” werden een aantal methodes vervolledigd. “Learning” in Q-Learning gebeurt voornamelijk in update(state, action, nextState, reward). Hier krijgt een action, state paar een nieuwe Q-value aangewezen. De Agent kan deze later gebruiken voor het maken van een beslissing. De gebruikt formule is gebaseerd op deze van Wikipedia:

De laatste term in de vorige formule werd geïmplementeerd in “computeValueFromQValues”. De Agent gaat altijd naar de volgende state met de hoogste Q-Value. Elke Q-value, na het nemen van een actie, wordt in een lijst gestoken. Na het doorlopen van alle “legale” acties, kan de maximum Q-value van de lijst teruggegeven worden aan het programma. “ComputeActionfromQValues” geeft de actie terug, bijhorend bij de maximum Q-value.

1. **Question 7: Epsilon Greedy**

In de Q-Learning Agent ontbreekt één functie: “getAction(state)”. Tijdens het programma loopt, zijn er twee opties mogelijk. In epsilon keer van de gevallen, geeft de functie een willekeurige actie terug (uit “legalActions”). Epsilon kunnen we veranderen naar wens in de command line. “util.flipcoin(p)” geeft True terug met kans p (hier epsilon). In alle andere gevallen geven we de beste actie terug met “ComputeActionfromQValues”. Deze implementatie hebben we reeds gezien in vorige opdracht.

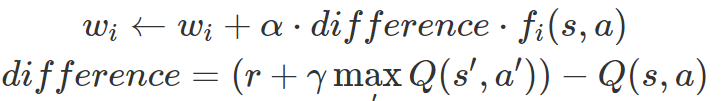
1. **Question 8: Bridge Crossing Revisited**

Na het uitproberen van enkele combinaties (epsilon en learning-rate) werd het redelijk snel duidelijk dit na 50 iteraties niet mogelijk zou zijn. Het leren van een “optimal policy” met een kans van 99% is echter niet haalbaar. Met deze reden, geven we de string “NOT POSSIBLE” terug.

1. **Question 9: Q-Learning and Pacman**

Deze opdracht past Q-Learning toe in de wereld van Pacman. We leren hieruit dat Q-Learning effectief is in een redelijk kleine omgeving. In een groot doolhof, waarin Pacman meestal wordt gespeeld, duurt Q-Learning lang. Het geeft hierbij ook een inefficiënt resultaat.

1. **Question 10: Approximate Q-Learning**

Voor deze opdracht moeten we de getQvalue en update van approximateQagent aanpassen. Voor de getQvalue moeten we gewoon de weights maal de features doen. De features kan men krijgen via de featExtractor. In update moeten we de weightvectors updaten volgens de formule:

Er is geprobeerd om met de keys van de features dictionary te werken om zo de bijbehorende weights te berekenen. Dit gaf helaas niet de gewenste uitkomst.