En robot som utforsker et ukjent terreng

Karaktersatt oppgave 1 | DTE-2602 | H23

Almaz Ermilov
almaz.ermilov@uit.no

Table of Contents

Summary

Så vidt jeg forsto, var det viktigste i denne oppgaven å leke seg med og forstå parametrene som er beskrevet nedenfor, og også å finne balansen mellom alle disse parametrene for denne oppgaven ("Fine-Tuning"):

1. **Learning Rate (alpha)**:
- Bestemmer hvor mye ny informasjon som overstyrer gammel informasjon. En høy læringshastighet betyr at roboten lærer raskt (men kan potensielt ignorerere nyttig tidligere informasjon).
- Jeg valgte på slutten ***alpha=0.9***: under Q-læringsoppdateringene vil algoritmen bli sterkt påvirket av ny informasjon, men den vil ikke overskrive den gamle verdien fullstendig, vektet gjennomsnitt, med 90 % av den nye verdien og 10 % av den gamle verdien (**for høy verdi for et stabilt miljø kanskje, men derfor trenger jeg ikke så mange epoker til trening**).

2. **Discount Factor (gamma)**:
- bestemmer viktigheten av fremtidige belønninger. En faktor nær 1 verdsetter fremtidige belønninger nesten like høyt som umiddelbare belønninger, noe som fremmer langsiktig optimal politikk. En lavere diskonteringsfaktor gjør agenten kortsiktig og fokuserer mer på umiddelbare belønninger.
- Jeg valgte på slutten ***gamma=0.9***: ganske høy, noe som betyr at fremtidige belønninger anses som nesten like viktige som umiddelbare belønninger. Dette oppmuntrer algoritmen til å finne ruter som gir høy belønning på lang sikt, i stedet for bare å lete etter umiddelbare gevinster (**for høy verdi for et stabilt miljø, men derfor trenger jeg ikke så mange epoker til trening**).

3. **Exploration vs. Exploitation (epsilon)**:
- Fintuning av epsilon handler om å balansere exploration og exploitation. Høyere epsilon oppmuntrer til utforskning for å lære mer om miljøet.
- Jeg valgte ***epsilon=0.5***: Med en verdi på 0,5 velger roboten å utforske (velge en tilfeldig handling) halvparten av tiden og utnytte (velge den best kjente handlingen) den andre halvparten. Denne balansen kan være spesielt effektiv fordi den gjør det mulig for algoritmen å oppdage nye stier ("exploration") samtidig som den utnytter kunnskapen den allerede har tilegnet seg ("exploitation").

4. **Number of Episodes (num_episodes)**:
- Antall episoder påvirker hvor mye erfaring agenten opparbeider seg. For få episoder er kanskje ikke nok til at agenten lærer effektivt, mens for mange episoder kan være bortkastet eller føre til overtilpasning til omgivelsene.
- Jeg valgte på slutten ***num_episodes=100 (eller mye mindre pga store alpha og gamma)***: Selv om flere episoder generelt gir bedre læring, kan selv et lavere antall som 100 være tilstrekkelig hvis læringshastigheten (alpha) er høy, fordi Q-verdiene oppdateres mer aggressivt.

5. **Reward**:
Jeg valgte på slutten: ***self.rewards = {'M': -300, 'W': -500, 'L': 10, 'E': 1000}***
- 'M': -300 og 'W': -500: Straffen for fjell (M) og vann (W) gjør disse terrengene mye mer uønskede. Dette gjør at roboten i større grad unngår disse tilstandene, noe som fører til at det er mindre sannsynlig at ruten går gjennom dem.
- 'L': 10: Å holde vanlig landbelønningen (L) på 10 gir en liten positiv belønning som oppmuntrer roboten til å holde seg på land når det er mulig.
- 'E': 1000, Slutten (E): En så stor positiv belønning gir algoritmen et sterkt insentiv til å nå målet, noe som gjør sluttilstanden svært attraktiv. Dette kan fremskynde læringsprosessen av veien som fører til målet.

Kombinasjonen av disse parametrene og belønningsinnstillingene skapte et læringsmiljø der roboten raskt kunne lære seg en grei balanse mellom å utforske nye baner og utnytte kjente baner, samtidig som den i stor grad unngikk negativt terreng og siktet mot målet. Den høye læringshastigheten gjorde at roboten raskt kunne tilpasse seg de økte belønningene og straffene, noe som resulterte i en effektiv rute til målet.

Introduksjon

I denne oppgaven lager jeg en simulering av en robot som utforsker et ukjent terreng.

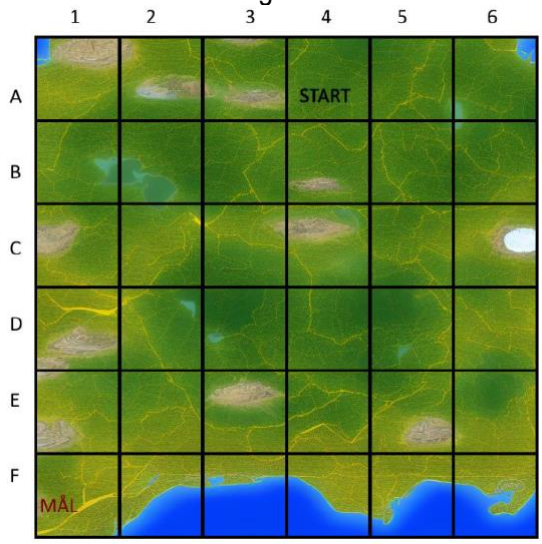
Målet med denne oppgaven: lære om Q-læring.
Q-læring er en modellfri forsterkningslæringsalgoritme for å lære verdien av en handling i en bestemt tilstand. (Jansson & Skjelvareid, 2023)

Problem: navigere en robot gjennom et rutenettbasert terreng med varierende risikonivåer.
Oppgaven er utformet for å hjelpe oss med å forstå hvordan vi kan bruke Q-læring på et praktisk problem.

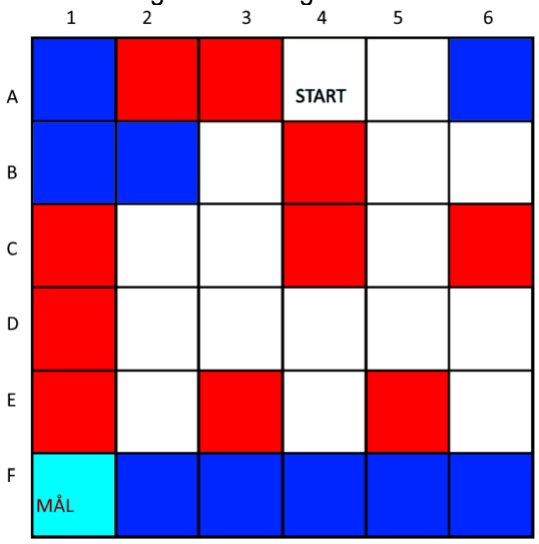
For enkelhets skyld er terrenget delt opp som et rutenett med 36 tilstander.
Roboten skal starte i A4 og skal prøve å finne en trygg vei frem til F1.

Om terrenget:

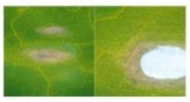
Terrenget



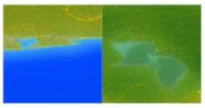
Forenklet figur av terrenget



Fjelltopper og bratt, ulendt terreng:



Vann og innsjøer:



Røde ruter er fjell og bratt terreng, og mørkeblå ruter er vann. Hvite ruter er forbundet med lav eller ingen risiko.

Roboten styres med et API som tillater følgende bevegelser: “opp”, “ned”, “høyre” og “venstre”. Hvis roboten står i rute “A4” og gjør bevegelsen “ned” havner den altså i rute “B4”.
Roboten kan ikke bevege seg diagonalt.

Terrenget som array:

```
self.terrain = [
    ['W', 'M', 'M', 'L', 'L', 'W'],  # A
    ['W', 'W', 'L', 'M', 'L', 'L'],  # B
    ['M', 'L', 'L', 'M', 'L', 'M'],  # C
    ['M', 'L', 'L', 'L', 'L', 'L'],  # D
    ['M', 'L', 'M', 'L', 'M', 'L'],  # E
    ['E', 'W', 'W', 'W', 'W', 'W']   # F
]
```

Hvor:
M er rødt terreng (fjell og bratt terreng)
W er mørkeblått terreng (vann)
L er hvitt terreng (lav eller ingen risiko)
S er start
E er slutt

Oppgavebeskrivelse

Besvarelsen på oppgaven består av to deler som skal leveres via repository på GitHub Classroom:

- En Python-fil med implementasjon.
- En rapport i form av et-PDF dokument (f.eks. generert fra MS Word eller LaTeX) med beskrivelse av din egen løsning. Rapporten bør være mellom 3 og 6 sider (A4, skriftstørrelse 11). Rapporten kan skrives på norsk eller engelsk.

Oppgaven er delt opp i underoppgaver:

1. Implementer klassen «Robot», og definer en R-matrise (reward-matrise) for kartet som er oppgitt. Gjør egne vurderinger angående hvilke verdier som bør brukes i matrisa. Gi en begrunnelse i rapporten for valgene du gjør.
2. Implementer metoden get_next_state_mc() som tilfeldig velger en av fire mulige «actions» for ruta roboten står i (opp, ned, høyre, venstre). Dersom roboten står i kanten av rutenettet og prøver å gå ut av kartet, skal den bli stående i samme rute.
3. Implementer metoden monte_carlo_exploration(), som kjører Monte Carlo-simulering av at roboten beveger seg tilfeldig i kartet (som beskrevet i oppgave 2). Metoden skal ha antall simuleringer som input-argument. Hver simulering starter med roboten i rute A4 og slutter når den når rute F1. Metoden skal beregne total belønning for hver simulering (basert på matrise i oppgave 1), og ta vare på den beste ruta og den høyeste belønningen den har funnet. Kjør simuleringen 100 ganger og beskriv den beste ruta roboten finner (gjerne med en figur).
4. Definer en Q-matrise for roboten, og implementer metoden q_learning(). Metoden skal la roboten utforske kartet og samtidig oppdatere Q-matrisa underveis. Metoden skal ha antall episoder som input-argument. For hver episode plasserer du roboten i en tilfeldig starttilstand, som slutter når den når F1 - måltilstanden. Velg en av to policy-varianter:

a. Enkel variant: Roboten velger 1 av 4 mulige bevegelser tilfeldig (samme som Monte Carlo i oppgave 3)
b. Mer kompleks/effektiv variant («epsilon-greedy»): Roboten velger tilfeldig bevegelse en andel av tida (f.eks. 80% av gangene), og velger alternativet som virker best (ut fra Q-matrisa) resten av gangene. Krever alternativ metode for å finne neste tilstand (get_next_state_ge()). Andelen tilfeldige valg (“epsilon”) bør være en input- parameter.

5. Implementer metoden greedy_path() som returnerer beste rute fra start til mål (basert på «trent» Q-matrise), med tilhørende total belønning.
6. Eksperimenter med å kjøre Q-learning-programmet med ulike antall episoder (og evt. ulike varianter av epsilon-greedy policy), og vis hvor god rute roboten finner for hvert eksperiment. Klarer den å finne den optimale ruta? Det kan hende du må justere reward- matrisa (se oppgave 1) for å få gode resultater. Sammenlign gjerne med resultatene i oppgave 3.

Solution

Oppgave 1

Robot-klassen initialiseres med:

- "*terrain*" som representerer omgivelsene roboten kan navigere i
- "*start_state*", som er startpunktet for roboten
- "*end_state*" som er destinasjonen roboten har som mål å nå
- "*rewards*" dict definerer belønningene for hver terrengtype.
- "*R_matrix*" opprettes basert på rutenettets 6x6 dimensjoner og inneholder belønningene for hver celle i rutenettet: den fylles med belønninger som tilsvarer terrengtypen for hver celle i rutenettet.
- "*Q_matrix*" initialiseres med de samme dimensjonene som R-matrisen, og alle verdier settes til null. Den lagrer de forventede belønningene for å utføre en handling i en gitt tilstand.

```python
class Robot:

    def __init__(self):
        self.terrain = [
            # 1    2    3    4    5    6
            ['W', 'M', 'M', 'L', 'L', 'W'],  # A
            ['W', 'W', 'L', 'M', 'L', 'L'],  # B
            ['M', 'L', 'L', 'M', 'L', 'M'],  # C
            ['M', 'L', 'L', 'L', 'L', 'L'],  # D
            ['M', 'L', 'M', 'L', 'M', 'L'],  # E
            ['E', 'W', 'W', 'W', 'W', 'W']   # F
        ]

        self.start_state = (0, 3) # Row A, Column 4 (A4)
        self.end_state = (5, 0)   # Row F, Column 1 (F1)

        # rewards dict for each type of terrain
        self.rewards = {'M': -100, 'W': -100, 'L': 10, 'E': 1000}

        # R-matrix 6x6 which is our grid
        self.R_matrix = [[0 for _ in range(6)] for _ in range(6)]

        # filling in the R-matrix based on the terrain and self.rewards
        for i in range(6):
            for j in range(6):
                terrain_type = self.terrain[i][j]
                self.R_matrix[i][j] = self.rewards.get(terrain_type, 0)

        # Q-matrix with the same dimensions as R-matrix
        self.Q_matrix = [[0 for _ in range(6)] for _ in range(6)]
```

Verdiene som er valgt for belønningene i R-matrisen, følger en logikk som er i samsvar med navigasjonsmålene:

```
self.rewards = {'M': -100, 'W': -100, 'L': 10, 'E': 1000}
```

(la oss begynne med det først)

Generell begrunnelse for de valgte verdiene:
- "M" og "W" har samme negative verdi fordi begge representerer dårlig terreng.
- "L" har en positiv verdi fordi det er godt terreng og bør være den foretrukne stien.
- "E" har en høyere positiv verdi for å motivere roboten til å nå dette målet fremfor alt annet.

Oppgave 2

Metoden definerer en dictionary med navnet "*actions*" som tilordner handlingsnavn til tilsvarende endringer i rutenettposisjonen.
"*current_state*" er en tupel som representerer robotens nåværende posisjon i rutenettet (rad og kolonne).
"*possible_actions*" er en liste for å holde oversikt over hvilke handlinger som kan utføres fra gjeldende tilstand uten å bevege seg utenfor rutenettet.

```python
def get_next_state_mc(self, current_state):
    actions = {
        'up': (-1, 0),
        'down': (1, 0),
        'left': (0, -1),
        'right': (0, 1)
    }

    row, col = current_state

    possible_actions = []
    for action, (dr, dc) in actions.items():
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 6 and 0 <= new_col < 6:
            possible_actions.append(action)

    if possible_actions:
        chosen_action = random.choice(possible_actions)
        dr, dc = actions[chosen_action]
        next_state = (row + dr, col + dc)
    else:
        next_state = current_state

    return next_state
```

Metoden itererer deretter over elementene i "*actions*":
- For hver handling beregnes den nye rutenettposisjonen (*new_row, new_col*) ved å bruke endringene på den gjeldende posisjonen.
- Den sjekker om den nye posisjonen er innenfor rutenettets grenser (dvs. at både rad- og kolonneindeksene er mellom 0 og 5, inkludert).
- Hvis den nye posisjonen er gyldig, legges handlingen til i listen over "*possible_actions*".

Etter å ha bestemt de mulige handlingene:
- Hvis det finnes noen mulige handlinger, velger metoden en tilfeldig ved hjelp av *random.choice(possible_actions)*, beregner den nye tilstanden ved å bruke den valgte handlingens endringer på gjeldende tilstand og returnerer denne nye tilstanden som "*next_state*".
- Hvis det ikke finnes noen mulige handlinger, settes "*next_state*" til "*current_state*" (roboten forblir i samme posisjon).

Metoden returnerer "*next_state*", som er robotens nye posisjon etter forflytningen.

Oppgave 3

Monte_carlo_exploration-metoden utfører følgende trinn:

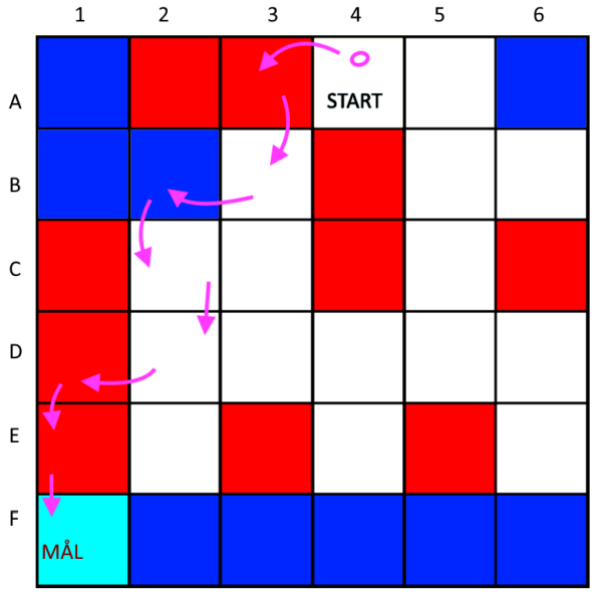
- Kjører flere simuleringer der roboten beveger seg tilfeldig fra start- til sluttposisjonen.
- I hver simulering sporer den den totale belønningen for den valgte banen.
- Sammenligner belønningen for hver bane for å finne den med høyest belønning.
- Returnerer den beste banen og den totale belønningen etter at alle simuleringene er fullført.

```python
def monte_carlo_exploration(self, num_simulations):
    best_route = None
    highest_reward = float('-inf')

    for simulation in range(num_simulations):
        current_state = self.start_state
        total_reward = 0
        route = [current_state]

        while current_state != self.end_state:
            next_state = self.get_next_state_mc(current_state)
            total_reward += self.R_matrix[next_state[0]][next_state[1]]
            route.append(next_state)
            current_state = next_state

            if current_state == self.end_state:
                break

        if total_reward > highest_reward:
            best_route = route
            highest_reward = total_reward

    return best_route, highest_reward
```

Jeg simulerte det 100 ganger og fikk det
(med self.rewards = {'M': -200, 'W': -100, 'L': 10, 'E': 100}):

Best Route from Monte Carlo Exploration:
[(0, 3), (0, 2), (1, 2), (1, 1), (2, 1), (3, 1), (3, 0), (4, 0), (5, 0)]
Highest Reward from Monte Carlo Exploration: -570

Så det ble -200 +10 -100 +10 +10 -200 -200 +100, se figuren:



Oppgave 4b

Endret i *__init__*:
- Endret Q-matrisen (**36x4**, num_states x num_actions) som skal oppdateres under Q-læringsprosessen.

q_learning:
- Gjennomfører episoder med utforskning, og oppdaterer Q-matrisen etter hvert trinn basert på belønningen og de estimerte fremtidige belønningene.
- Bruker en epsilon-greedy-strategi der roboten velger en tilfeldig handling med sannsynlighet epsilon (utforskning) og den best kjente handlingen med sannsynlighet 1-epsilon (utnyttelse).
- Q-verdiene oppdateres ved hjelp av Bellman-ligningen, som tar hensyn til læringshastigheten (alfa) og diskonteringsfaktoren (gamma).

get_next_state_eg:
- Velger neste tilstand under Q-læringsprosessen.
- Velger en tilfeldig handling med sannsynlighet epsilon.
- Med sannsynligheten 1-epsilon velges handlingen med høyest Q-verdi for gjeldende tilstand.

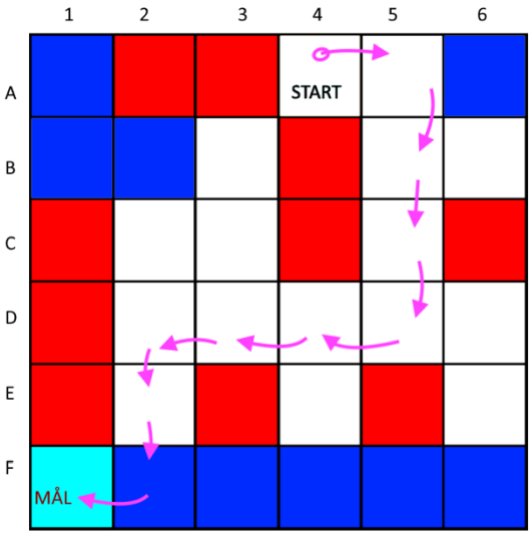
Oppgave 5 og 6

Greedy Path (*greedy_path*):
- Etter at Q-matrisen er etablert, finner denne metoden den beste ruten fra start- til sluttilstanden basert på de høyeste Q-verdiene.
- bruker en grådig strategi og velger alltid handlingen med høyest Q-verdi for hvert trinn.

Parametere som jeg brukte (etter har gjort masse eksperimenter):

```
num_episodes=100, epsilon=0.5, alpha=0.9, gamma=0.9
self.rewards = {'M': -300, 'W': -300, 'L': 10, 'E': 1000}
```

Best Route (en av kjøringer) som jeg fikk var:



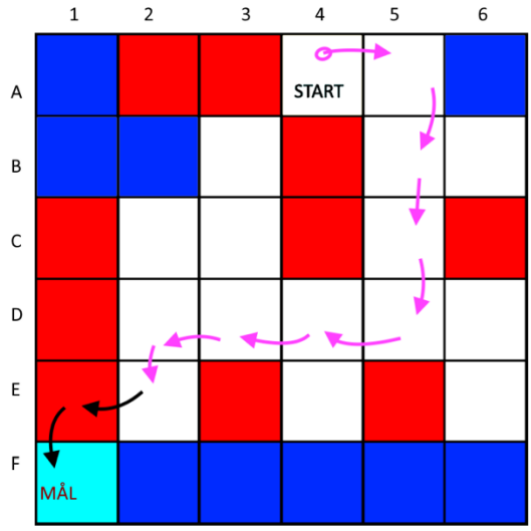
Best Route from Greedy Path based on Q-Matrix:
[(0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (3, 3), (3, 2), (3, 1), (4, 1), (5, 1), (5, 0)]
Total Reward from Greedy Path: 780

Ruten er vist på bildet til høyre.
Jeg liker denne ruten.

Det er mange parameter til å eksperimentere her, det var vanskelig å få balanse mellom disse parameterene (epochs, epsilon, alpha, gamma, rewards dict).

Jeg har prøvd mange forskjellige varianter og alternativer. Fjell og vann har ulik risiko, men det kan være fornuftig å gi mer straff for vann enn for fjellterreng (vann krever mer energi).

Så jeg endret det for vann i
self.rewards = {'M': -300, 'W': -500, 'L': 10, 'E': 1000}



Best Route from Greedy Path based on Q-Matrix:
[(0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (3, 3), (3, 2), (3, 1), (4, 1), (4, 0), (5, 0)]
Total Reward from Greedy Path: 780

Ruten er vist på bildet til høyre.
Jeg liker denne ruten også.

num_episodes=100, epsilon=0.5, alpha=0.9, gamma=0.9,
self.rewards = {'M': -300, 'W': -500, 'L': 10, 'E': 1000}

Kombinasjonen av disse parametrene og belønningsinnstillingene skapte et læringsmiljø der roboten raskt kunne lære seg en grei balanse mellom å utforske nye baner og utnytte kjente baner, samtidig som den i stor grad unngikk negativt terreng og siktet mot målet. Den høye læringshastigheten gjorde at roboten raskt kunne tilpasse seg de økte belønningene og straffene, noe som resulterte i en effektiv rute til målet.

Bibliography

Jansson, A. D., & Skjelvareid, M. (2023). *Modul 5: Reinforcement learning.* From Canvas: https://uit.instructure.com/courses/30442/modules/items/885185

Amini, A. (2023). *MIT Introduction to Deep Learning 6.S191: Lecture 5* . From YouTube: https://www.youtube.com/watch?v=AhyznRSDjw8

ADL. (2018, September 3). *An introduction to Q-Learning: reinforcement learning*. From freeCodeCamp: https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/