Rubiks Terning: Legeplads for Reinforcement Learning som værktøj til diskret optimering

Søren Winkel Holm, Anne Agathe Pedersen, Asger Laurits Schultz

1. april 2020

Indhold

1. Introduktion

1.1 Problemet og dets perspektiv

Reinforcement Learning og diskret optimering

Reinforcement Learning er en af de mest generelle læringsparadigmer i maskinlæring: Den enkle opsætning med en agent, et miljø og maksimering af kumulativ skalarbelønning indfører få antagelser om problemets natur og gør sammenligninger til menneskelig læring og fantasier om kunstig generel intelligens fristende. Når de helt store fremtidsperspektiver lægges til side, er der ofte praktiske mål som selvkørende biler, produktionsrobotter og andre autonome systemer, som bliver set som Reinforcement Learnings potentiale. Sådanne fysiske agenter i kontinuerte, dynamiske miljøer falder oplagt ind, når man tænker på Reinforcement Learnings formulering og dets familieforhold til kontrolteori, der har rig tradition for styring af fysiske systemer.

Det er dog unødvendigt begrænsende at se dette lovende læringssystem som dybt forbundet til det kontinuerte og fysiske: Mængden af praktiske beslutningsproblemer er alsidig og der findes vigtige opgaver med helt andre modelleringsudfordringer end bevægelse af et robotlegeme.

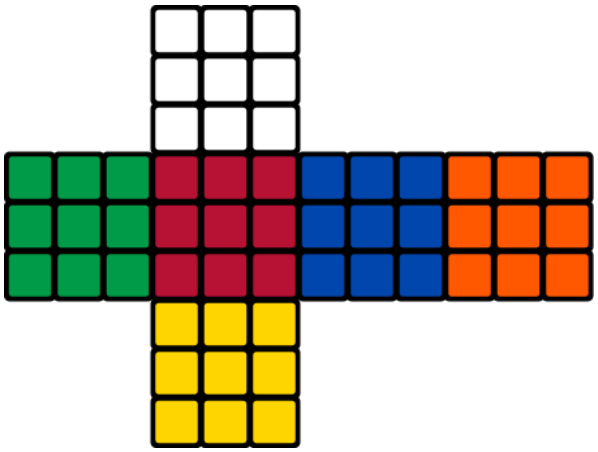
Diskret optimering er et dybt og velstuderet felt indenfor anvendt matematik og datalogi med klassiske problemer som *Traveling Salesman*, grundlæggende datastrukturer som grafer og matroider og vigtige ingeniørpraktiske anvendelser som operationsanalyse, computationel kemi og planlægningssystemer. Disse problemer, der kan udmynte sig i kombinatorisk optimering vha. heltalsprogrammering eller optimering under bibetingelser, har ofte dybe, matematiske strukturer beskrevet af abstrakt algebra, herunder gruppeteori.

I denne rapport vil Reinforcement Learnings generaliserbarhed undersøges ved at bruge det som løsningsmetode til et sådant diskret problem, der konventionelt løses med algoritmer fra gruppeteorien. Det større mål er altså at opnå erfaringer ved at bruge Reinforcement Learning på det kombinatoriske beslutningsproblem, perspektivere det til domænespecifikke løsningsstragier og undersøge udfordringerne med Reinforcement

Learning i et stort udfaldsrum.

Rubiks Terning

Rubiks terning er et lille, kombinatorisk legetøj opfundet af den ungarske billedhugger og professor i arkitektur Ernö Rubik i 1974. Med 350 mio. solgte terninger regnes den seksfarvede terning som det bedst sælgende legetøj og har fortsat en dedikeret fanskare organiseret i *World Cube Association*, der afholder *speedcuber*-arrangementer og deler praktiske løsningsalgoritmer [1].



Figur 1.1: Illustration of de seks farvede sider i en 3x3 Rubiks terning.[1.1]

Målet er ved en række enkle rotationstræk at få alle siderne til at være ens som på figur 1.1. Har man selv sat sig ned med en sådan terning, kan det opleves, at problemet virker enkelt til at starte med, men hurtigt fremstår overraskende svært, når hver handling påvirker flere sider på forskellige måder. Og uden en præcis løsningsvejledning, kan det forekomme umuligt at nå frem en helt udrullet terning.

Dette skyldes, at der kun er én måltilstand og et utroligt stort tilstandsrum (mere om det i kapitel 2). Uden en algoritme vil det formentlig tage et gennemsnitligt menneske utroligt lang tid at finde en måde at løse Rubiks terning på, når så stort et rum skal udforskes – det tog Rubik selv en måned at finde en løsningsalgoritme til problemet. Det er denne udforskning af udfaldsrummet, der bliver læringsagentens store opgave.

[1.1] Billede: Wikimedia Commons på `https://en.wikipedia.org/wiki/File:Rubik%27s_cube_colors.svg`

1.2 Nyeste litteratur

Siden Ernö Rubik opfandt sin berømte terning i 1974, er der blevet opfundet adskillige løsningsalgoritmer. Disse kan overordnet set inddeles i to grupper: Gruppeteoretisk funderede algoritmer, fx. Kociembas algoritme, og brute force-algoritmer med udgangspunkt i skræddersyede heurestikker. De forskellige algoritmer har forskellige fordele og ulemper; nogle skal være lette for mennesker at løse, mens andre er designet til såkaldte speed cubers, der skal løse terningen med så få træk som muligt; nogle er optimerede til lavt hukommelsesforbrug og andre til at kunne løses på kort tid af computere.

En ny tilgang til løsningsalgoritmer og diskrete problemer generelt er dyb reinforcement learning. Et forskerhold fra University of California Irvine har de sidste år gjort store fremskridt med netop denne fremgangsmåde som beskrevet i *Solving the Rubik's Cube Without Human Knowledge* (maj 2018) [2] og senest *Solving the Rubik's cube with deep reinforcement learning and search* (juli 2019) [3].

DeepCube, der introduceres i [2], er et dybt neuralt netværk, der lærer en heurestik til at styre en Monte Carlo-træafsøgning, så den kombinatoriske eksplosion undgås. DeepCube trænes med en approksimativ værdiiterations-algoritme, som forskerne betegner autodidaktisk iteration. Metoden formår at løse samtlige tilfældige testtilstande med en medianløsningslængde på 30 træk, hvilket er mindre end eller svarende til løsningsalgoritmer, der er baserede på domæneviden.

DeepCubeA [3] bygger videre på denne agent med en ændring i værdiiterationen, og en mere avanceret afsøgningsmetode, de kalder batch-vægtet A*-afsøgning. Disse ændringer, forstærkede DeepCubeA nok til at løse alle testkonfigurationer og generaliserer til andre diskrete spil, hvilket gav Irvine-holdet en plads til at præsentere algoritmen i *Nature* [3].

1.3 Rapportens mål

Motiveret af Rubiks terning som et miljø for eksperimentation i Reinforcement Learning på svære, kombinatoriske problemer, er det denne rapports mål at udforske mulighederne og begrænsningerne ved at bruge Reinforcement Learning på at løse Rubiks terning. Der ønskes altså at skabes en løsningsalgoritme, som ikke bruger problemspecifik viden og derfor selv skal lære problemet at kende og udvikle strategier gennem et Reinforcement Learning-paradigme. Den generelle form til agenten er på baggrund af litteratursammenligningen valgt til at være en grafsøgende agent, der bruger lærte,

dybe neurale netværk som heuristik til at afsøge tilstandsrummet frem til løsningen. For at tage udgangspunkt i en forholdsvis direkte løsningsmetode, er det et hovedmål for rapporten at genskabe metoderne (men ikke nødvendigvis resultaterne) for agenten *DeepCube* fra [2].

Der vil derfor implementeres et dybt neuralt neværk, der både approksimerer en værdi til hver tilstand og den optimale politik hertil. Dette netværk trænes ved brug af en værdiapproksimativ metode kaldet audodidaktisk iteration, der bruger en tidligere version af netværket samt en breddeførst søgning på dybde 1 til at approksimere værdi og politik til hver tilstand. Når netværket er færdigtrænet, bruges dette til at gennemføre Monte Carlo træsøgning frem til den færdige tilstand.

I implementationen af en sådan agent er der en del store udfordringer for konvergens som skyldes problemets kombinatoriske eksplosion, der beskrives i 2.1, hvilket gør det til et fokus for rapporten at optimere forudsætninger for læring og sammenligne detaljer i læringsparadigmer.

Konkret er målet med rapporten er at svare på følgende spørgsmål:

- Hvor god kan en Reinforcement Learning-algoritme blive til at løse Rubiks terning uden at have adgang til menneskelige heuristikker? Med *god* menes antallet af træk væk fra den løste tilstand, som algoritmen fast kan løse.

- Hvilke forhold er optimale i forhold til en stabil konvergens af værdifunktionen, det neurale netværk, i autodidaktisk iteration? Her tænkes på datarepræsentationen og design af træningsproceduren.

- Hvor tidseffektiv kan løsning og træning af RL i et problem med så stort udfaldsrum være? Her er både fokus på at lære den bedst mulige værdi-approkismation, så løsningstid minimeres, men også på at softwaredesign i træningsfasen gør det muligt for agenten at udforske så meget som muligt af udfaldsrummet indenfor overskuelig tid.

I følgende kapitel (2) vil datagrundlaget for opgaven blive beskrevet ved først at motivere problemets sværhedsgrad, indføre terminologi og notation samt vise tilstandsrummets størrelse ved brug af den gruppeteoretiske beskrivelse af terningen. Derefter vil den brugte datastruktur for terningen indføres og afbalancering mellem computationel effektivitet og semantisk passende repræsentation diskuteres.

I kapitel 3 vil metoderne for løsningen introduceres. Terningemiljøet gennemgås efterfulgt af en introduktion af Reinforcement Learning-opsætningen for problemet.

Så introduceres værdi- og politiknetværket samt træningsproceduren, autodidaktisk iteration, samt kostfunktion og optimering for dette. Beskrivelse af løsningsagenten vil så afsluttes med introduktion af Monte Carlo-træsøgningsalgoritmen. Afslutningsvist vil de gennemførte eksperimenter blive ridset op.

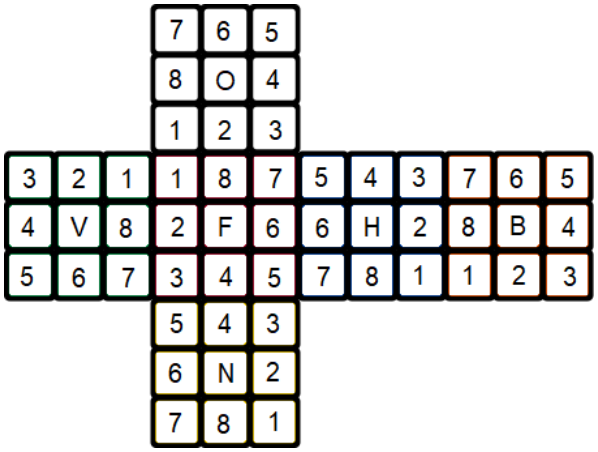
2. Terningen: Data og repræsentation

2.1 Matematikken bag tilstandsrummet

For at motivere sværhedsgraden af problemet, vil aspekter ved Rubiks terning blive beskrevet ud fra et gruppeteoretisk perspektiv. Dette afsnit bygger hovedsageligt på [4], men er tilpasset vores egen repræsentation og implementation af Rubiks terning.

Permutationer

Rubiks terning har seks sider og 26 cubies i alt; seks midtercubies med én udadvendt overflade, 12 kantcubies med to flader og otte hjørnecubies med tre flader. Hver side kan roteres $\frac{\pi}{2}$ radianer med eller mod urets retning. Fastholdes orientationen af Rubiks terning, vil positionen af alle centercubies også holdes fast, uafhængigt af hvordan siderne roteres. Således vil man kunne navngive hver side ud fra dens orientation, og vi har nu front F, bag B, op O, ned N, højre H og venstre V. De otte omkringliggende cubies på hver side nummereres som vist på Figur 2.1. Bemærk, at lige numre svarer til en kantcubie og ulige numre svarer til en hjørnecubie.



Figur 2.1: Repræsentation af Rubiks terning.

Ud fra kvart-drejnings-semantikken kan man foretage 12 forskellige handlinger på

Rubiks terning (13 hvis man medtager handlingen ikke at foretage nogen handling); hver side kan roteres én gang enten med eller mod uret. Notationsmæssigt betyder rotationen F, at frontsiden set fra observatørens perpektiv roteres $\frac{\pi}{2}$ radianer i urets retning. 2F svarer til en rotation af fronten på π radianer, og 3F = F' svarer til én rotation mod urets retning.

En rotation kan ses som en permutation af Rubiks terning. En permutation $f : A \to A$ en afbildning af mængden A på sig selv. En rotation kan skrives som en permutation bestående af fem disjunkte 4-cykler. Ud fra navngivningen i Figur 2.1 vil rotationen F kunne skrives som

$$F : (f_1 \quad f_7 \quad f_5 \quad f_3)(f_2 \quad f_8 \quad f_6 \quad f_4)(v_1 \quad o_3 \quad h_7 \quad n_5)(o_2 \quad h_6 \quad n_4 \quad v_8)(o_1 \quad h_5 \quad n_3 \quad v_7)$$

Ligeledes kan resten af permutationerne F', B, B', O, O', N, N', H, H', V og V' hver skrives som en kombination af fem disjunkte 4-cykler. Ikke at foretage en rotation navngives I.

Rubiks terning som en gruppe

Mængden $\{F, F', B, B', O, O', N, N', H, H', V, V', I\}$ er en permutationsgruppe $(\mathbb{G}, *)$ med gruppeoperatoren $*$ defineret som følger; $X*Y$ er udførelsen af handling X efterfulgt af handling Y. $\mathbb{G}$ opfylder forudsætningerne for en permutationsgruppe:

- $\mathbb{G}$ er associativ, eftersom $(X * Y) * Z = X * (Y * Z)$; at foretage handling X og Y efterfulgt af Z er det samme som at foretage handling X efterfulgt af Y og Z.
- $\mathbb{G}$ har et identitetselement I (ikke at foretage en handling), eftersom $I * X = X = X * I$.
- Hvis X er et element i $\mathbb{G}$ har X en invers X^{-1} således at $X * X^{-1} = I$. Dette er opfyldt, eftersom $X^{-1} = X'$.

Med denne viden om $\mathbb{G}$ og Rubiks terning kan vi nu beregne antallet af permutationer (ordenen af $\mathbb{G}$). Først beregnes alle permutationer, både lovlige og ulovlige. Dette svarer til alle kombinationer, hvor man fysisk tager en cubie og placerer den, som man vil. Der er otte pladser til hjørnecubies, som hver har tre måder at blive orienteret på. Der er 12 pladser til kantcubies, som hver har to måder at blive orienteret på. Eftersom

midtercubies er fikserede, fås

$$8! \cdot 3^8 \cdot 12! \cdot 2^{12} = 519.024.039.293.878.272.000 = 5{,}2 \cdot 10^{20}$$

permutationer. Der skal dog tages højde for ulovlige permutationer. Af disse er kun halvdelen af kantcubies placeret lovligt og en tredjedel af hjørnecubies placeret lovligt [4]. Endvidere er der i halvdelen af permutationerne blevet udskiftet et ulige antal cubies, hvilket er umuligt. Derfor her vi

$$\frac{8! \cdot 3^8 \cdot 12! \cdot 2^{12}}{2 \cdot 3 \cdot 2} = 43.252.003.274.489.856.000 = 4{,}3 \cdot 10^{19}$$

lovlige permutationer. Dette er ligeledes størrelsen på tilstandsrummet for problemet.

Guds tal

Inde for Rubiks terning betegnes det mindste antal træk, der skal til for at løse en terning, som Guds tal. Betegnelsen *Guds tal* kommer af, at man er nødt til at være et alvidende væsen for at kunne løse enhver af de $4{,}3 \cdot 10^{19}$ permutationer på dette antal træk eller færre. Dette tal er 26 eller færre i kvart-drejnings-semantik og blev fundet tilbage i 2014 ved essensielt at bevise, at alle permutationer af Rubiks terning kan løses på 26 eller færre træk [5]. Fremgangsmåden var at dele alle permutationer op i $2{,}2 \cdot 10^9$ sæt med $2{,}0 \cdot 10^{10}$ permutationer i hver, som derefter vha. symmetri og sætdækning kunne reduceres til i alt $5{,}6 \cdot 10^7$ sæt, der skulle løses. For hvert sæt blev fundet (ved brug af særligt stor computerkraft) løsninger på længden 26 træk eller mindre, og Guds tal er derfor nu 26.

2.2 Repræsentation af terninger

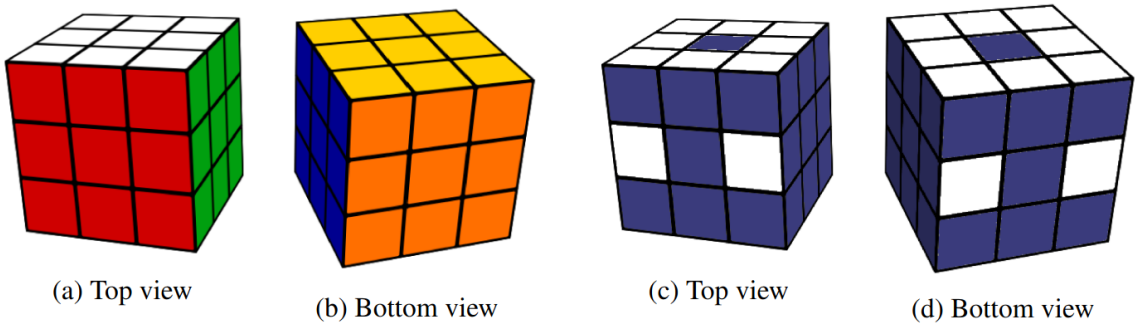
Der findes en lang række måder at repræsentere tilstanden af en terning på. For at opnå de bedste resultater, skal den valgte repræsentation

- undgå overflødighed
- bruge lidt hukommelse
- kunne foretage rotationer hurtigt

- fungere godt med neurale netværk

Den mest oplagte måde at repræsentere terningen på er en 6×9-tensor, hvor hver indgang er et tal svarende til en farve. Det er dog ikke nødvendigt at modellere det midterste kvadrat, så længe terningens orientering er fastlagt, hvilket reducerer repræsentationen til 6×8. Denne repræsentation bruger kun $6 \cdot 8 \cdot 4 = 192$ bytes, når tallene gemmes som 32 bit floating points. Det er også muligt at foretage rotationer hurtigt, men det er muligt at repræsentere en lang række tilstande, der ikke ligger i tilstandsrummet – fx at alle kvadrater på alle sider har samme farve. Med seks farver består det repræsenterbare tilstandsrum af $6^{6 \cdot 8} \approx 2{,}25 \cdot 10^{37}$ tilstande, langt større end de $4{,}3 \cdot 10^{19}$ lovlige tilstande. Repræsentationen er heller ikke velfungerende med neurale netværk, hvor én ud af 6-kodning er at foretrække. Ved at tilføje en ekstra akse til tensoren er en ud af 6-kodning muligt, men det kræver $6 \cdot 8 \cdot 6 \cdot 4 = 1152$ bytes og løser stadig ikke overflødighedsproblemet.

En anden, mindre intuitiv repræsentation er en 20×24-tensor. Den tager udgangspunkt i Rubiks terning som bestående af de 26 cubies, hvorefter man fjerner de seks midtercubies og har 20 kuber, der skal repræsenteres. Disse inkluderer 8 hjørnecubies og 12 sidecubies. Hjørnekuberne kan sidde 8 steder og have tre forskellige orienteringer, hvilket giver 24 mulige tilstande. Tilsvarende kan sidekuberne sidde 12 steder med to orienteringer, hvorfor de også har 24 mulige tilstande. Klistermærkerne på hver cubie har en unik farvekombination,[2.1] og det er derfor muligt at repræsentere hver kube ved blot ét af dens klistermærker. Repræsentationen er visualiseret på figur 2.2 og følger direkte repræsentationen i [2], hvorfra illustrationen også er taget.



Figur 2.2: Repræsentation af Rubiks terning. Der holdes styr på de hvide kvadrater på (c) og (d). [2]

[2.1] Det er muligt at overbevise sig selv om det ved at forestille sig den løste terning.

Denne repræsentation kan repræsentere $24^{20} \approx 4{,}02 \cdot 10^{27}$ tilstande – et stykke over tilstandsrummets størrelse, men stadig langt bedre end det omtalte alternativ. Til gengæld kræver det $20 \times 24 \times 4 = 1920$ bytes at gemme en tilstand, men rotationer kan foretages hurtigt. Endeligt er tilstande én af 24-kodede, hvilket gør det anvendeligt til neurale netværk. Samlet set vurderes denne repræsentation til bedre at opfylde kravene og det er den, der anvendes som datagrundlag i rapporten.

3. Metode

Al kode er tilgængelig på `https://github.com/sorenmulli/rl-rubiks`
De implementerede dybe Reinformcent Learning-modeller er alle implementationer af DeepCube fra [2] med nogle undtagelser, der beskrives i de følgende afsnit. Løsningsmodellen består af et neuralt netværk, der trænes med en Reinforcement Learning-procedure kaldet autodidaktisk iteration til at vurdere værdien af og den optimale handling til enhver tilstand for Rubiks terning. Dette netværk bliver så brugt som heuristisk for en Monte Carlo træsøgning som løsningsstrategi til terningen.

For at undersøge, hvad der er vigtigt for at få gode resultater, udføres der sammenlignende lærings-eksperimenterer og evalueringseksperimenter, hvor der ændres på detaljer i træningsparadigmet og tiden for træsøgningsmodellen.

3.1 Rubiks terning som miljø

En af de definerende karaktestikker ved reinforcement learning er agentens og miljøets interaktion. Det er derfor nødvendigt at have en velfungerende repræsentation af miljøet, på hvilken agenten kan interagere. Miljøet implementeres i `python` som en klasse, hvis centrale egenskab er rubiksterningens tilstand som beskrevet i afsnit 2.2. Miljøet initialiseres med den løste tilstand, hvorefter indbyggede metoder bruges til at foretage givne eller tilfældige rotationer. Derudover indeholder miljøklassen en metode til at teste for, om den pågældende tilstand er løst, samt en metode, der giver tilstanden i et mere læsevenligt format.

I designet af miljøklassen har der været fokus på optimering under forventningen om adskillige millioner, hvis ikke milliarder, simuleringer under træningen af agenten. Især rotationsmetoden har været tildelt særligt fokus, da rotationer udgør al interaktion med miljøet, hvorfor det er favorabelt, at de kan foretages hurtigt.

Med miljøet fastlagt er det nu muligt at påbegynde træningen af en agent, der kan interagere intelligent med miljøet og nærme sig eller finde den løste tilstand.

3.2 Den lærende agent

Opgaven med at udvikle løsningsmodellen er et veldefineret læringsproblem, da det formaliseres som at følge Reinforcement Learning-opsætningen ved at have følgende aspekter:

- *Agent*: Løsningsmodellen – som her kaldes DeepCube, selvom den også afprøves med ændringer i forhold til DeepCube i [2] – har rollen som lærende agent, der interagerer med miljøet. Den består af det dybe neurale netværk (DNN) samt træafsøgningen og beskrives i løbet af dette kapitel. Agenten er model-baseret, da den har en indre repræsentation af Rubiks terning for at træ-afsøge tilstande. Agenten er implementeret i `src/rubiks/post_train/agents.py`.

- *Miljø*: Rubiks terning, der modtaget handlinger a og returnerer tilstand $\mathbf{s}$. Miljøet er en fuldt-observerbar, deterministisk Markov beslutningsproces, da al information om ny tilstand $\mathbf{s}_{k+1}$ kun er afhængig af foregående tilstand og foregående handling a_k:

  $$\mathbf{s}_{k+1} = \mathbf{f}(\mathbf{s}_k, a_k)$$

  og da agentens observation indeholder al information om tilstanden. Miljøet har én afsluttende tilstand $\mathbf{s}_T$. Miljøet er beskrevet i afsnit 3.1 og implementeret i `src/rubiks/cube.py`

- *Handlinger*: Handlingsrummet A bestående af de tolv handlinger beskrevet i afsnit 3.1 er uafhængigt af tilstanden og tidsskridtet.

- *Belønning*: Belønningsfunktionen $R(\mathbf{s}_k)$ er kun afhængig af tilstanden $\mathbf{s}$ og returnerer skalarbelønningen på følgende måde:

  $$R(\mathbf{s}_k) = \begin{cases} 1 & : \mathbf{s}_k = \mathbf{s_T} \\ -1 & : \mathbf{s}_k \neq \mathbf{s_T} \end{cases}$$

  Belønningen er således valgt for at være i overenstemmelse med [2] og for at give minimal information om problemet til agenten, der så testes uafhængigt af menneskeudviklet heuristik.

Politik og værdi Målet med træning af Reinforcement Learning-agenten DeepCube er så godt som muligt at approksimere den optimale politik π^*, der maksimerer akkumuleret belønning v, til hvilken der opnås den optimale akkumulerede belønning v^*:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_{k=0}^{N-1} R\big(f(\mathbf{s_k}, \pi(\mathbf{s}_k))\big), v^* = \max_{\pi} \sum_{k=0}^{N-1} R\big(f(\mathbf{s_k}, \pi(\mathbf{s}_k))\big) \tag{3.1}$$

hvor $N = \min\{k_T, k_{\max} - 1\}$, k_T er tidsskridtet, når agenten løser terningen og $k_{\max}$ er en grænse på antal skridt, som defineres i eksperimentet.

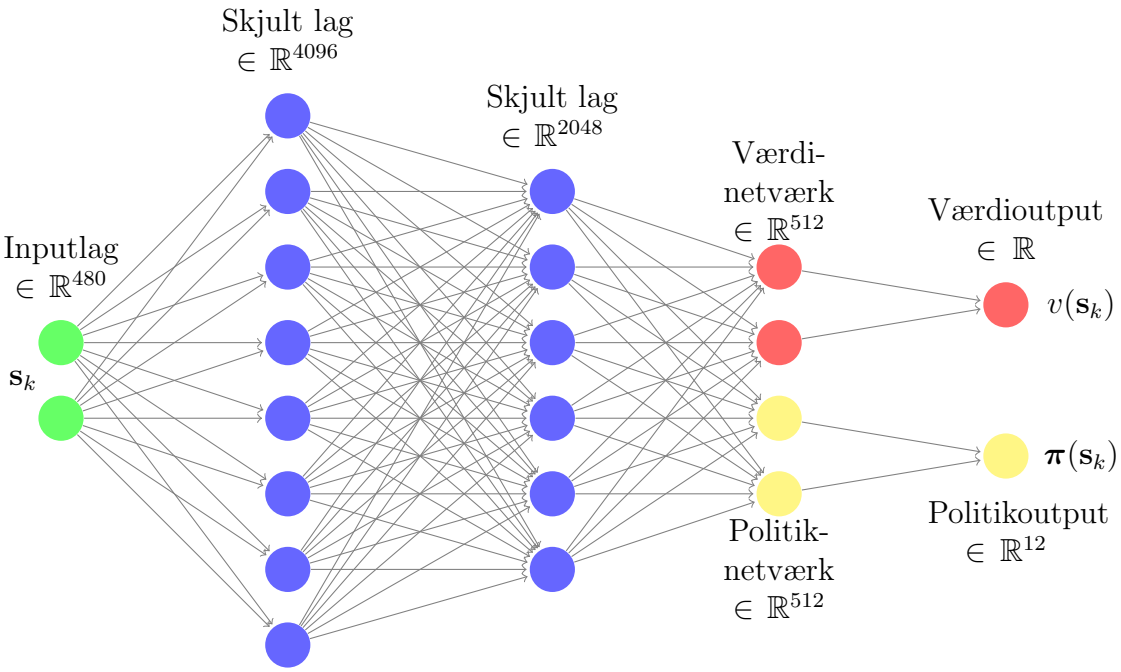
Til at bestemme $\widetilde{\pi^*} \approx \pi^*$ blev to sammenhængende afbildninger implementeret i DeepCube. Den ene komponent er en lært afbildning, der direkte approksimerer $\widetilde{\pi^*}$ ved at afbillede fra tilstanden $\mathbf{s}_k$ til en diskret sandsynlighedsfordeling, der approksimativt tilskriver hver mulig handling en sandsynlighed for at denne er den optimale. Denne funktion kaldes *politik-netværket* og repræsenteres fremover med symbolet $\boldsymbol{\pi}$:

$$\pi(\mathbf{s}_k)_i \approx \mathbb{P}(\pi^*(\mathbf{s}_k)_i = i), \quad i \in \{0,1,...,11\}$$

Desuden læres en funktion, der approksimerer *værdien* til hver tilstand forstået som den akkumulerede belønning der fås ved at følge politikken π herfra. Denne kaldes *værdi-netværket* og repræsenteres som v:

$$v(\mathbf{s}_k|\pi) \approx \sum_{j=k}^{N-1} R\big(f(\mathbf{s}_j, \pi(\mathbf{s}_j))\big)$$

Dybt neuralt netværk Disse to afbildninger, $\boldsymbol{\pi}, v$ læres altså i løbet af træningsprocessen og har centrale roller som dybde- og breddestyrende heuristikker i løsningsalgoritmen. For at repræsentere disse approksimationer blev et dybt neuralt netværk (DNN) brugt. Da disse afbildninger begge skal approksimere værdier relateret til (3.1), blev de udtrykt i det samme neurale netværk, der defor blev konstrueret med en todelt outputstruktur.



Figur 3.1: Illustration af det todelte neurale netværk brugt i *DeepCube*, hvor to skjulte lag skal resultere i hierakisk vidensopbygning brugbart for at repræsenterer løsningsstrategier.

Netværket består af lineære lag med bias. Arkitekturen, som er illustreret på figur 3.1, består af et 480-dimensionelt inputlag efterfulgt af to fuldt-forbundne skjulte lag på hhv. 4096 og 2048 neuroner. Derefter opdeles netværket i værdi-netværket og politik-netværket, begge med et skjult lag på 512 neuroner. Værdinetværket returnerer en skalarværdi $v(\mathbf{s}_k)$ og politiknetværket returnerer den tolvdimensionelle vektor $\boldsymbol{\pi}(\mathbf{s}_k)$. Denne struktur medfører ca. 25 millioner parametre i det dybe neurale netværk. Mellem hvert lag blev den elementvise ikkelineære aktiveringsfunktion *Eksponentiel lineær enhed* (eng: *Exponential Linear Unit*, ELU) placeret, som i nogle læringsproblemer er vist at accelerere læringen bl.a. ved at have en normaliserende effekt på aktiveringerne [6]

$$\mathrm{ELU}(x) = \max\{0, x\} + \min\{0, \mathrm{e}^x - 1\}$$

På det tolvdimensionelle politikoutput blev softmax-aktiveringen brugt for at fortolke den som en diskret sandsynlighedsfordeling:

$$\pi(\mathbf{s}_k)_i = \sigma(\mathbf{z})_i = \frac{\mathrm{e}^{z_i}}{\sum_{j=0}^{11} \mathrm{e}^{z_j}}$$

hvor $\mathbf{z} = (z_0, ..., z_i, ..., z_{11})$ er output fra politiknetværket. Det neurale netværk er implementeret i `src/rubiks/model.py`

Læring af værdi og politik: Autodidaktisk iteration

At opnå et politik/værdi-netværk med gode approksimationer, kræves et eksplorativt læringsparadime hvorigennem netværket opnår nok kendskab til tilstandsrummet S til at kunne finde generelle strategier. På trods af problemets determinisme og Markov-egenskab er der dog en stor udfordring i udforskningen af disse tilstande. Som beskrevet i afsnit 2.1, gør de mange mulige permutationer, at tilstandsrummet når en størrelse på $|S| \approx 4{,}3 \cdot 10^{19}$, hvilket gør, at enhver systematisk løsning af maksimeringsproblemet 3.1 kun kan kun kan tilskrive optimal værdi og politik til en forsvindende lille andel af tilstandsrummet. Der er derfor behov for et læringsprocedure, der kan observere approksimativ optimal politik og værdi til tilstande på kort tid samt vælge at udforske tilstande, der giver et brugbart billede af tilstandsrummet.

Til træningen af politik/værdi-netværket blev en iterativ metode, der tilskriver tilstande værdi ud fra den til hver tid nyeste version af netværket brugt. Metoden er præsenteret i [2, 4.1], hvor den kaldes *Autodidaktisk iteration* (ADI) og tager udgangspunkt i den løste Rubiks terning $\mathbf{s}_T$. For denne terning tages K tilfældige træk og for hvert træk gemmes tilstanden opnået ved disse træk samt antallet af tilfældige træk hørende til tilstanden $D(\mathbf{s}_i)$.

Der er altså nu skabt K tilstande $\mathbf{s}_i, i \in \{0..K-1\}$. For hver af disse skal der tilskrives de bedst mulige observationer for tilstandens værdi y_v og politik $y_{\boldsymbol{\pi}}$, som det ønskes at politik/værdi-netværket skal approksimere. For at finde disse, udføres nu alle tolv mulige handlinger til den pågældende tilstand. For hver af disse tolv subtilstande $f(\mathbf{s}_i, a), a \in A$, der her fremkommer, evalueres det nyeste *værdi*-netværk og tolv subværdier $v(a), a \in A$ haves nu for den pågældende tilstand. Disse tolv subværdier bruges nu til at tilskrive værdi og politik til forældertilstanden $\mathbf{s}_i$.

Til tilstanden $\mathbf{s}_i$ tilskrives nu værdi og politik ved følgende formler:

$$y_{vi} = \max_{a \in A} \; R\big(f(\mathbf{s}_i, a)\big) + v(a) \tag{3.2}$$

$$y_{\boldsymbol{\pi} i} = \operatorname*{argmax}_{a \in A} \; R\big(f(\mathbf{s}_i, a)\big) + v(a) \tag{3.3}$$

Der gemmes altså K observationer af værdi/politik-par til, der skal tilnærme sig den optimale værdi og handling til hver af dem ved at bruge værdinetværket for den følgende

tilstand og belønningsfunktionen R. Der genereres et større antal observationer ved at spille L spil, så der opnås $N = L \cdot K$ træningseksempler. Træningssættet X siges at bestå af alle opnåede tilstande x_i, således at $X = \{x_i\}_{i=1}^{N}$

Ovenover er autodidaktisk iteration beskrevet, som præsenteret i [2]. En alternativ regel for tilskrivelse af værdi er dog foreslået af Max Lapan [7] for at forbedre stabiliteten af træningen. Denne ændrer formel (3.2) til at være

$$y_{v_i} = \begin{cases} 0 & \text{hvis } \mathbf{s}_i \text{ er den vindende tilstand} \\ \max_{a \in A} \; R\big(\mathbf{f}(\mathbf{s}_i, a)\big) + v(a) & \text{ellers} \end{cases} \tag{3.4}$$

I træningen blev der spillet $L = 10000$ spil i autodidaktisk iteration. Dybden K blev undersøgt i eksperimenter.

Opdatering af model

Ovenfor blev en procedure til at generere N datapunkter, der hver indeholder en tilstand $\mathbf{s}_i$, tilsigtede observationer af værdi og politik $(y_{vi}, y_{\boldsymbol{\pi} i})$ samt antallet af tilfældige træk taget for at nå frem til tilstanden $D(\mathbf{s}_i)$. Disse blev brugt til at opdatere modellen i et Supervised Learning-paradigme, hvor værdinetværket løser et regressionsproblem med y_{vi} som mål og politiknetværket løser et klassifikationsproblem med $y_{\boldsymbol{\pi} i}$ som mål.

Som kost-funktion på netværkets forudsigelse af værdi og politik $(v(\mathbf{s}_i), \boldsymbol{\pi}(\mathbf{s}_i))$ blev følgende funktion brugt:

$$\mathcal{L}(x_i) = \underbrace{\frac{1}{D(\mathbf{s}_i)}}_{\text{Antal træk}} \Bigg(\underbrace{(x_{vi} - y_{vi})^2}_{\text{Kvadreret afvigelse}} \quad \underbrace{- \log \frac{\mathrm{e}^{\boldsymbol{\pi}(\mathbf{s}_i)_j}}{\sum_{k=0}^{11} \mathrm{e}^{\boldsymbol{\pi}(\mathbf{s}_i)_k}}}_{\text{Krydsentropi-kost}} \Bigg), \text{ hvor } j = y_{\boldsymbol{\pi} i} \tag{3.5}$$

Kosten til hvert datapunkt består altså en sum af kvadreret afvigelse for regressionsproblemet på værdien og 12-klasse krydsentropikost (eng: *Cross Entropy Cost*) for klassifikationsproblemet på politikken. Denne sum blev dog vægtet med $1/D(\mathbf{s}_i)$, som giver højere betydning til tilstande, som blev genereret få træk væk fra måltilstanden og lavere vægt til tilstande langt fra målstadiet. Dette er indført af [2] for konvergensstabilitet ved at vægte tilstande, hvor udfaldsrummet er mindre komplekst.

Til at minimere denne kostfunktion, blev der brugt minibatchlæring på det dybe neurale netværk med batches på størrelse 50. Optimeringen foregik med algoritmen RMSProp, der tilpasser læringsraten ved at dividere den med et rullende gennem-

snit af gradientmagnituden [8]. Til optimeringen blev brugt `pyTorch` hvis indbyggede `autograd`-modul foretog denne gradientbaserede parameteropdatering.

Træningen af det dybe neurale netværk foregik ved at udføre autodidaktisk iteration til at generere de N træningseksempler efterfulgt af træning på dette datasæt. En sådan træning kaldes en *udrulning* af netværket og der blev gennemført 5000 udrulninger.

3.3 Løsningsalgoritme og Monte Carlo Træsøgning

Efter at have trænet det neurale netværk anvendtes Monte Carlo Træsøgning (MCTS) for at kunne løse en terning fra en *scrambled* tilstand s_0. MCTS er en heuristisk søgealgoritme. Som det ligger i navnet, bygger MCTS et træ af tilstande, hvor roden er s_0 og dens børneknuder er de resulterende tilstande efter hver af de 12 mulige handlinger. MCTS er en effektiv søgealgoritme, fordi den sørger for at ekspandere den mest lovende bladknude først.

En måde at vælge en knude at ekspandere er ved brug af den øvre konfidensgrænse en øvre konfidensgrænse (*upper confidence bound*, UCB), introduceret af [9]. For hver bladknude t beregnes UCB

$$UCB(t) = \frac{r_t}{n_t} + \sqrt{\frac{2 \ln n_s}{n_t}}$$

og bladknuden med den højeste UCB vælges til at blive ekspanderet. s er forælderknuden til t, og $\frac{r_t}{n_t}$ er belønningen for t. n_t og n_s er hvor mange gange henholdsvis t og s er blevet spillet. Første led af formlen korresponderer med udnyttelse, da den vil være høj for træk, der leder til den løste tilstand. Andet led korresponderer med udforskning, da den vil være høj for træk, der kun er blevet simuleret få gange.

Efter at have valgt en bladknude, genereres en barneknude til denne. Derefter genereres en række tilfældige træk, der enten resulterer i, at terningen løses, eller at den ikke løses (*rollout*). Resultatet af *rollout*'en tilbagepropageres ved at opdatere informationen i knuderne på stien fra barneknuden tilbage til s_0.

Der findes også andre, mere specialiserede, politikker for at udvælge en bladknude. I [2] og [10] vælges en handling a til tiden t fra tilstand s ud fra politikken

$$A_t = \text{argmax}_a \frac{cP_{st}(a)\sqrt{\sum_{a'} N_{st}(a')}}{1 + N_{st}(a)} + W_{st}(a) - L_{st}(a)$$

hvor $N_s(a)$ er antallet af gange a er blevet taget fra s, $W_s(a)$ er den maksimale værdi af a fra s, $L_s(a)$ er det nuværende virtuelle tab for a fra s, $P_s(a)$ er en prior-sandsynlighed af a fra s og c er en udforsknings-hyperparameter.

3.4 Eksperimenter

Der blev trænet to forskellige agenter. Én, hvor dybden af tilfældige træk i autodidaktisk iteration K var 10 og én hvor dybden var 100.

For hver af disse agenter blev gennemført to evalueringseksperimenter:

- **Terningedybde**: Afsøgningstiden med MCTS blev holdt konstant på 10 minutter og for hver dybde $d \in \{1,2,...,20\}$ blev der spillet 100 spil, som hver blev generet ved d tilfældige træk. Der blev så gemt andelen af spil, der blev løst på denne tid samt længden af løsningerne.
- **Afsøgningstid**: Antallet af tilfældige træk blev holdt konstant $d = 10$ og for hver tidsgrænse $t \in \{2\,\text{min}, 4\,\text{min}, ..., 10\,\text{min}\}$ blev der spillet 100 spil, hvor MCTS blev afbrudt efter t og andelen af løste spil og længden af løsningerne blev gemt.

4. Litteratur

[1] Wikipedia Contributors. Rubik's cube. *Wikipedia, The Free Encyclopedia*, March 2020. Hentet på: https://en.wikipedia.org/wiki/Rubik's_Cube on 10/3/2020.

[2] F. Agostinelli, S McAleer, A. Shmakov, et al. Solving the rubik's cube without human knowledge. *CoRR*, abs/1805.07470, May 2018.

[3] F. Agostinelli, S McAleer, A. Shmakov, et al. Solving the rubik's cube with deep reinforcement learning and search. *Nat Mach Intell*, 1:356–363, July 2019.

[4] Hannah Provenza. Group theory and the rubik's cube.

[5] Tomas Rokicki. Towards god's number for rubik's cube in the quarter-turn metric. *The College Mathematics Journal*, 45(4):242–242, 2014.

[6] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[7] M. Lapan. Reinforcement learning to solve rubik's cube (and other complex problems!). *Medium*, January 2019. Hentet på: https://medium.com/datadriveninvestor/reinforcement-learning-to-solve-rubiks-cube-and-other-complex-problems på 16/2/2020.

[8] G. Hinton. Overview of mini-batch gradien descent. September 2016. Fra kursus *Neural Networks for Machine Learning*. Hentet på: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf på på 18/3/2020.

[9] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[10] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with approximate policy iteration. In *International Conference on Learning Representations*, 2019.