Emil Keränen

Vertaileva tutkimus koneoppimisen hyödyntämisestä videopelien reitinhaussa

Tietotekniikan pro gradu -tutkielma

20. marraskuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Emil Keränen

Yhteystiedot: emil.a.keranen@student.jyu.fi

Ohjaaja: Tommi Kärkkäinen

Työn nimi: Vertaileva tutkimus koneoppimisen hyödyntämisestä videopelien reitinhaussa

Title in English: Comparative study of utilizing machine learning in video games' pathfinding

Työ: Pro gradu -tutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 50+0

Tiivistelmä: TODO: tiivistelmä suomeksi

Avainsanat: koneoppiminen, videopeli, reitinhaku, syvä vahvistusoppiminen

Abstract: TODO: In english

Keywords: machine learning, video game, pathfinding, deep reinforcement learning, Soft Actor Critic, Machine Learning Agents, Unity

Kuviot

Sisällys

1 Johdanto

Reitinhaku on robotiikan ja videopelien tekoälyn yksi suurimmista ongelmista, jota on tutkittu jo vuosikymmeniä (Abd Algfoor, Sunar ja Kolivand 2015; Cui ja Shi 2011). Reitinhaulla tarkoitetaan kahden pisteen välisen reitin selvittämistä. Useimmissa tapauksissa halutaan etsiä nopein ja tehokkain reitti, mutta ongelman haastavuuden vuoksi voidaan tyytyä myös epäoptimaalisiin ratkaisuihin (Rahmani ja Pelechano 2022). Reitinhakuongelma on ajan mittaan muuttunut lyhyimmän reitin löytämisestä myös reitin selvittämiseen muuttuvassa eli dynaamisessa alueessa. Erityisesti robottien suorittamaa dynaamisen alueen reitinhakua on tutkittu viime vuosina ja siten uudet vaatimukset, kuten turvallisuus ja tarkkuus, ovat nousseet prioriteetissa (Karur ym. 2021).

Usein perinteinen reitinhakuongelma pystytään ratkaisemaan A*-algoritmilla, jota käytetään etenkin videopeleissä, mutta myös robotiikassa (Abd Algfoor, Sunar ja Kolivand 2015; Botea ym. 2013; Cui ja Shi 2011). A*-algoritmi löytää optimaalisen reitin heuristiikkafunktion ansiosta. Heuristiikkafunktion avulla algoritmin ei tarvitse tutkia selvästi huonompia vaihtoehtoja, jolloin reitinhausta saadaan suorituskyvyltään kevyempää.

Videopeleissä reitinhaku ilmenee usein erillisten tekoälyagenttien toimintana. Näitä kutsutaan myös ei-pelaaja-hahmoiksi (engl. non-player-character, NPC) (Cui ja Shi 2011). Videopelien reitinhaussa suorituskyky nousee suurimmaksi ongelmaksi. Tietokoneen suorituskyky osoittautuu rajalliseksi vaativien grafiikka- ja fysiikkalaskelmien vuoksi. Joissain peleissä, kuten reaaliaikaisissa strategiapeleissä (engl. Real-Time Strategy, RTS), pelaajan ohjaamia liikkuvia hahmoja voi olla samanaikaisesti jopa satoja, joten esimerkiksi pelkkä A*-algoritmi sovellettuna jokaiselle agentille osoittautuu äärimmäisen haastavaksi laskennallisesti. Siksi vuosien mittaan on kehitetty ratkaisuksi erilaisia pelialueen esitystavan muokkauksia, algoritmien variaatioita ja lopulta jopa koneoppimisratkaisuja.

Koneoppimista ja sen menetelmiä on myös tutkittu viime vuosina hyvin paljon eri sovellusten parissa. Koneoppimisen avulla pystytään hyödyntämään aiempaa kokemusta uusissa toiminnoissa. Etenkin neuroverkkoja ja syväoppimista on pystytty hyödyntämään monissa erilaisissa ongelmissa luokittelusta robotiikkaan. Tässä tutkimuksessa on tarkoitus sovel-

taa Unity-pelinkehitystyökalun valmista ML-agents -pakettia reitinhakuagenttien luomiseen. ML-agents käyttää PyTorch-kirjastoa neuroverkkomallin luomiseen. ML-agents -paketin avulla agentteja voidaan opettaa reitinhakuun käyttäen syvää vahvistusoppimista ja siihen perustuvaa Soft Actor Critic -algoritmia. Tarkoitus on selvittää, pystyykö koneoppimisagentti suorittamaan reitinhakua yhtä tehokkaasti kuin A*-algoritmi. Tutkimuksessa pyritään myös huomioimaan mahdolliset tilanteet, joissa A*-algoritmilla olisi vaikeuksia suorittaa reitinhakua tehokkaasti. Agentin oppimista ja sen saamia palkkioita havainnoidaan Tensorboard-työkalun avulla.

Luvussa 2 kerrotaan yleisesti reitinhakuongelmasta videopeleissä pelialueiden esitystavan ja käytettyjen reitinhakualgoritmien muodossa. Luvussa 3 käsitellään yleisesti koneoppimista, sen eri paradigmoja ja syväoppimista. Luvussa 4 esitellään pelinkehitysalusta Unity, sen rakenne ja tässä tutkimuksessa käytetyt paketit. Luvussa 5 käydään läpi tutkimuksen empiirinen osuus menetelmien ja tutkimusasetelman kannalta. Luvussa 6 esitellään ja analysoidaan empiirisessä osuudessa saadut tulokset. Lopuksi luvussa 7 käydään läpi tutkimuksen yhteenveto lyhyesti.

2 Reitinhaku videopeleissä

Reitinhaulla tarkoitetaan yksinkertaisimmillaan reitin tai polun selvittämistä kahden pisteen välillä. Se on yksi videopelien tekoälyn ja myös robotiikan tunnetuimmista ja haastavimmista ongelmista, jota on tutkittu jo muutaman vuosikymmenen ajan (Abd Algfoor, Sunar ja Kolivand 2015; Cui ja Shi 2011). Reitinhakua esiintyy monissa eri peligenreissä, kuten roolipeleissä ja reaaliaikaisissa strategiapeleissä, joissa ei-pelaaja-hahmoja määrätään liikkumaan ennaltamäärättyyn tai pelaajan määräämään sijaintiin väistellen samalla vastaantulevia esteitä. Reitinhaun ja yleisesti tekoälyn on oltava realistista, jotta pelaaja pystyy syventymään videopelin maailmaan eikä pelaajan kokema immersio esty.

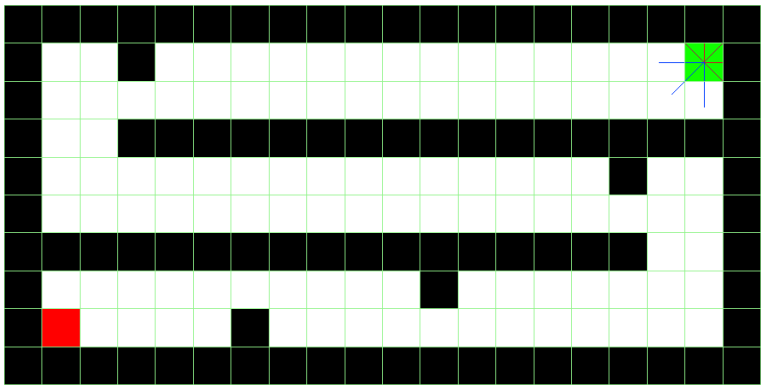
2.1 Pelialueen esitystavat

Pelialue esitetään reitinhakua varten aina graafina, joka koostuu solmuista ja kaarista (Lawande ym. 2022). Kaaret yhdistävät solmut toisiinsa ja mahdollistavat liikkumisen solmujen välillä. Graafista voidaan muodostaa erilaisia kokonaisuuksia, joita kutsutaan ruudukoiksi (engl. grid). Ruudukot voivat olla kuvioltaan säännöllisiä tai epäsäännöllisiä. Säännölliset ruudukot sisältävät esimerkiksi kolmioita, kuusikulmioita, neliöitä tai kuutioita riippuen onko kyseessä 2D- vai 3D-alue. Epäsäännölliset ruudukot voivat koostua esimerkiksi reittipisteistä (engl. waypoint) tai navigointiverkosta (engl. navigation mesh). Selvästi käytetyin esitystapa videopeleissä on 2D-neliöruudukko.

Yleisesti ruudukot sisältävät yksittäisiä vapaita ruutuja tai esteruutuja (engl. tile) (Botea ym. 2013). Vapaat ruudut muodostavat graafin, jolloin jokainen vapaa ruutu vastaa yhtä graafin solmuista. Vierekkäisiä ruutuja yhdistävät kaaret, joita pitkin reitinhaku ja liikkuminen tapahtuu. Solmujen vierekkäisyys voi tarkoittaa horisontaalista ja vertikaalista vierekkäisyyttä (neljä suuntaa) tai näiden lisäksi myös diagonaalista vierekkäisyyttä (kahdeksan suuntaa) (Abd Algfoor, Sunar ja Kolivand 2015; Botea ym. 2013). Kuva 1 havainnollistaa yksinkertaista ruudukkoaluetta, jossa liikkuminen tapahtuu neljään tai kahdeksaan suuntaan.

Vaikka neliöruudukkoa pidetään tunnetuimpana ja käytetyimpänä esitystapana, se voi kuitenkin osoittautua ongelmalliseksi tilanteessa, jossa hahmo voi liikkua myös diagonaalisesti,

jolloin kaikki vierekkäiset ruudut eivät ole saman etäisyyden päässä toisistaan. Ruudukossa, jossa ruudut ovat kooltaan 1x1, diagonaalinen etäisyys on $\sqrt{2}$, kun taas vertikaalinen ja horisontaalinen etäisyys on 1 (Panov, Yakovlev ja Suvorov 2018). Jos yhdellä aika-askeleella hahmo voi liikkua vain yhden ruudun verran, niin diagonaalinen liikkuminen voi olla vaikea toteuttaa.



Kuvio 1. Ruudukko, jossa mustat ruudut ovat esteruutuja ja valkoiset ruudut vapaita ruutuja. Vihreä ruutu on aloitusruutu ja punainen ruutu on maaliruutu.

Vähemmän tunnettuja pelialueen esitystapoja ovat kolmiointi ja kuusikulmiointi (Abd Algfoor, Sunar ja Kolivand 2015). Alueen kolmiointi ja siihen perustuvat TA*- ja TRA*-algoritmi ovat kuitenkin osoittautuneet moninkertaisesti nopeammiksi suurissa pelialueissa verrattuna A*-algoritmiin (Demyen ja Buro 2006). Kuusikulmioihin perustuvat pelialueet ja reitinhakualgoritmit ovat myös tuottaneet lupaavia tuloksia robotiikan tutkimuksessa sekä yleisesti suoriutuneet paremmin muistinkäytön ja ajankäytön suhteen verrattuna neliöruudukkoihin (Abd Algfoor, Sunar ja Kolivand 2015; Lawande ym. 2022). Epäsäännöllisistä ruudukoista navigointiverkkoa on käytetty suurimmaksi osin videopeleissä ja esimerkiksi Unity tarjoaa dokumentaatiossaan laajat ohjeet ja menetelmät navigaatioverkon luomiseen (Lawande ym. 2022; “Unity Docs” 2022).

2.2 Reitinhakualgoritmit

Graafin lyhyimmän polun ongelmaa ja eri reitinhakualgoritmeja on tutkittu jo vuosikymmenten ajan. Vanhimmat ja tunnetuimmat algoritmit, Dijkstran algoritmi ja A*-algoritmi, esiteltiin jo 50- ja 60-luvuilla ja ne pystyivät ratkaisemaan lyhimmän polun ongelman staattisessa

graafissa (Dijkstra ym. 1959; Hart, Nilsson ja Raphael 1968) . Uudemmat reitinhaun sovellukset, kuten itseohjautuvat autot ja robotit, toivat kuitenkin alkuperäiselle ongelmanratkaisulle lisää vaatimuksia. Lyhimmän polun löytämisen lisäksi reitinhakualgoritmin täytyy ottaa huomioon sovelluksesta riippuen reitin turvallisuus, tehokkuus ja mahdollisten esteiden väistäminen (Karur ym. 2021).

A*-algoritmi on selvästi tunnetuin videopelien ja robottien reitinhaussa nopeutensa ansiosta (Abd Algfoor, Sunar ja Kolivand 2015; Botea ym. 2013; Cui ja Shi 2011). A*-algoritmista on kehitetty jo monia eri variaatioita, jotka pyrkivät vastaamaan jatkuvasti kasvaviin vaatimuksiin. Tässä tutkimuksessa keskitytään tarkemmin A*-algoritmiin ja sen variaatioihin.

2.2.1 A*-algoritmi

A*-algoritmi on hyvin tunnettu paras-ensin -reitinhakualgoritmi, joka hyödyntää heuristista arviointifunktiota lyhimmän reitin etsimiseen (Cui ja Shi 2011; Duchoň ym. 2014). A*-algoritmia voidaan pitää käytetyimpänä graafien etsintäalgoritmina etenkin videopeleissä (Botea ym. 2013; Lawande ym. 2022).

Algoritmin toiminta tapahtuu seuraavasti: jokainen aloitussolmun vierekkäinen solmu arvioidaan kaavan

$$f(n) = h(n) + g(n)$$

mukaisesti, jossa n on solmu, $h(n)$ on heuristinen etäisyys solmusta n maalisolmuun ja $g(n)$ on todellinen etäisyys aloitussolmusta solmuun n. Näistä solmuista matalimman $f(n)$-arvon solmu käsitellään seuraavaksi, jolloin kyseisen solmun vierekkäisten solmujen $f(n)$-arvot lasketaan. Tämä prosessi jatkuu, kunnes maalisolmu saavutetaan. Heuristiikan ollessa nolla A*-algoritmista tulee Dijkstran algoritmi.

A*-algoritmilla on kolme esitettyä ominaisuutta (Hart, Nilsson ja Raphael 1968). Ensiksi A*-algoritmi löytää reitin, jos sellainen on olemassa. Toiseksi reitti on optimaalinen, jos heuristiikka on luvallinen eli arvioitu etäisyys on lyhyempi tai yhtä suuri kuin todellinen etäisyys. Viimeisenä mikään muu algoritmi samalla heuristiikalla ei käy läpi vähemmän solmuja kuin A*-algoritmi eli A* käyttää heuristiikkaa tehokkaimmalla mahdollisella tavalla (Cui ja Shi 2011; Hart, Nilsson ja Raphael 1968). Luvallisia heuristiikkoja ovat solmujen vierekkäi-

syydestä riippuen Euklidinen etäisyys, Manhattan-etäisyys, Chebyshev-etäisyys ja Octile-etäisyys (Duchoň ym. 2014; Botea ym. 2013). Manhattan-etäisyyttä käytetään pääasiassa neljän suunnan ja Octile- sekä Chebyshev-etäisyyttä kahdeksan suunnan vierekkäisyyksissä. Euklidista etäisyyttä voidaan käyttää tilanteessa, jossa agentti voi siirtyä seuraavaan soluun mistä kulmasta tahansa.

2.2.2 A*-algoritmin variaatiot

Reitinhakualgoritmeja toteutettiin alunperin valmiisiin ja tarkkoihin ympäristöihin, joka ei kuitenkaan ole verrattavissa reaalimaailman tilanteisiin, joissa ympäristö voi muuttua arvaamattomasti (Lawande ym. 2022). A*-algoritmia ja sen rajoitteita onkin tutkittu jo useita vuosikymmeniä, joka on mahdollistanut useiden eri variaatioiden kehittämisen. Useimmiten variaatiot keskittyvät korjaamaan yleisimpiä A*-algoritmin reitinhakuongelmia, kuten suoritustehon optimointia ja sopeutumista muuttuviin alueisiin (Stentz 1994).

Stentz (1994) ja Stentz ym. (1995) esittivät 90-luvulla kaksi A*-algoritmin variaatiota: D*-algoritmi (Dynamic A*) ja Focussed D*-algoritmi. Uudet variaatiot pyrkivät ratkaisemaan etenkin muuttuvan ja tuntemattoman alueen ongelmat robotiikan tutkimuksessa. Alkuperäinen D*-algoritmi teki mahdolliseksi reitin korjaamisen esteen tai muutoksen tullessa reitille. Focussed D*-algoritmi tehosti alkuperäisen D*-algoritmin toimintaa ajallisesti ja näin kehitti sen toimintaa entisestään.

Koenig ja Likhachev (2005) esittivät D* Lite -algoritmin, joka nimestään huolimatta ei varsinaisesti perustu suoraan D*-algoritmiin, vaan A*- ja LPA* (Lifelong Planning A*) -algoritmiin. D* Lite -algoritmi osoittautui yksinkertaisemmaksi ja hieman tehokkaammaksi kuin D*- ja Focussed D*-algoritmit, jonka vuoksi sen toteuttaminen ja soveltaminen oli helpompaa.

TODO: D* pohdinnat vielä tarkistukseen, muutama lause ehkä lisää?

2.3 Reitinhaun haasteet

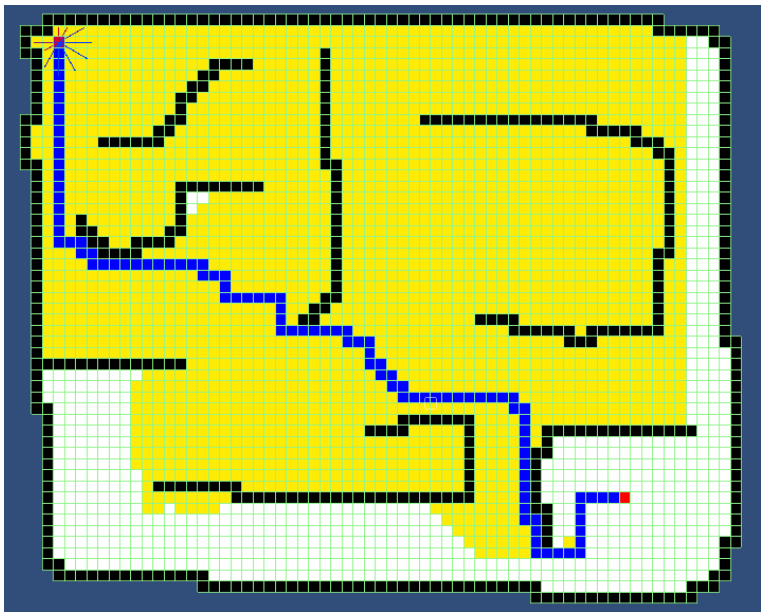
Yhden agentin staattisen ruudukkoalueen reitinhakuongelma on ratkaistavissa optimaalisesti heuristisilla reitinhakualgoritmeilla, mutta nykyään videopeleissä reitinhakuongelmat ovat monimutkaisempia ja saattavat vaatia useiden eri kriteerien täyttymisen. Suurimpia haasteita ovat esteiden järkevä väistäminen, optimaalisen reitin löytäminen ja suorituskykyvaatimusten minimointi (Abd Algfoor, Sunar ja Kolivand 2015; Cui ja Shi 2011). Näiden lisäksi reitinhakuongelmat pitävät sisällään esimerkiksi useamman agentin samanaikaista reitinhakua ja reaaliajassa muuttuvan alueen reitinhakua. Siksi videopeleissä onkin käytössä monia eri reitinhakualgoritmeja, joista osa on kehitetty toimimaan dynaamisissa ympäristöissä, osa staattisissa ympäristöissä ja osa molemmissa (Lawande ym. 2022). Reitinhakualgoritmin valinta on siis riippuvainen käyttötarkoituksesta. Nykytutkimus keskittyy pääasiassa monimutkaisten reitinhakuongelmien ratkaisemiseen ja algoritmien vertailuun erilaisissa reitinhakutilanteissa.

2.3.1 Suorituskyky

Reitinhakualgoritmin täytyy minimoida suoritustehon ja tallennustilan käyttö. Vaikka komponentit ovatkin kehittyneet vuosi vuodelta nopeasti, myös videopelit ovat monimutkaistuneet ja niiden laskennalliset vaatimukset kasvaneet. Reitinhakualgoritmeille varatut resurssit ovat videopeleissä rajatut, koska resursseja käytetään hyvin paljon myös graafisiin ja fysikaalisiin ominaisuuksiin (Lawande ym. 2022). Suoritustehoon liittyen etenkin muistinkäyttöä ja laskentatehoa pidetään yleisesti rajoittavina tekijöinä videopelien reitinhaussa (Botea ym. 2013). Ongelmia voidaan ratkaista erilaisilla algoritmeilla tai pelialueen esitystapaan liittyvillä ratkaisuilla (Botea ym. 2013; Cui ja Shi 2011).

Yksi suorituskykyyn liittyvistä ongelmista on reitinhakualgoritmien huono skaalautuvuus etenkin muistinkäytön suhteen. Jos agentin reitinhakuun sovelletaan A*-algoritmia suurella 1000x1000 pelialueella, muistiin joudutaan tallentamaan pahimmassa tapauksessa miljoona solmua (Cui ja Shi 2011; Duchoň ym. 2014). Muistinkäyttöä havainnollistetaan kuvassa 2, jossa A*-algoritmia on käytetty esteitä sisältävän alueen reitinhakuun. Kuvan keltaiset ruudut ovat tallennettu muistiin reitinhaun aikana. Kuvasta huomaa, että A*-algoritmi käy läpi

suuren määrän ylimääräisiä solmuja, koska alueella on esteitä. Ongelma moninkertaistuu, jos alueella liikkuu useita reitinhakuagentteja. Tässä tapauksessa A*-algoritmi osoittautuu riittämättömäksi ongelman ratkaisemisessa, joten tilanteeseen kannattaa soveltaa erilaisia menetelmiä tai algoritmien variaatioita.



Kuvio 2. Ruudukkoalue, jossa sininen reitti on A*-algoritmin laskema optimaalisin reitti. Keltaiset ruudut ovat käsitellyt solmut, jotka on tallennettu muistiin.

2.3.2 Dynaamisen alueen reitinhaku

Dynaamisen eli muuttuvan alueen reitinhaussa pyritään löytämään optimaalisin reitti jatkuvasti muuttuvassa alueessa (Lawande ym. 2022). Dynaamisen alueen reitinhakua esiintyy erityisesti robotiikassa, mutta myös videopeleissä. Robottien reitinhaussa ympäröivää tilaa mitataan erilaisten sensorien avulla, jonka perusteella reitinhakua suoritetaan (Rahmani ja Pelechano 2022). Videopeleissä pelialue on usein valmiiksi reitinhakuagentin tiedossa eikä erillisiä sensoreita käytetä pelialueen havainnoimiseen, ellei agentti pyri matkimaan ihmismäistä käyttäytymistä.

Videopeleissä pelialue voi muuttua toisten pelaajien, ei-pelaaja-hahmojen tai muuten vain liikkuvien esteiden takia. Esimerkiksi kilpa-ajoneuvopeleissä pelaaja voi kisata ei-pelaaja-hahmon kanssa, joka joutuu väistelemään sekä pelaajan ajoneuvoa että muita alueen liikku-

via esteitä (Sazaki, Primanita ja Syahroyni 2017). Jokainen alueen muutos rikkoo reitinhakugraafin rakenteen, jolloin graafi ja valittu reitti täytyy korjata. Graafin luominen voi olla isoissa pelialueissa kallis operaatio, joten jatkuva uudelleenluominen ja uuden reitin etsiminen ei välttämättä ole vaihtoehto suorituskyvyn kannalta. Nykyään videopelien reitinhaun tutkimuksissa painotetaankin dynaamisia reitinhakualgoritmeja.

Graafin jatkuva päivittäminen on tuonut erilaisia ideoita tutkimuksissa. Luvussa 2.2.2 mainittu D*-algoritmi on kehitetty juuri tuntemattoman ja dynaamisen alueen reitinhakuun. Myös koneoppimismenetelmiä on alettu soveltamaan dynaamisissa alueissa. Lei, Zhang ja Dong (2018) käyttivät syvää vahvistusoppimista koneoppimisagentin opettamiseen simulaatiossa ja testasi algoritmia myös reaalimaailman robotin reitinhaussa. Robotti käytti lidar-sensoreita lähiympäristön mallintamiseen ja onnistui muuttamaan reittiään esteiden tullessa eteen.

2.3.3 Moniagenttireitinhaku

Moniagenttireitinhaussa alueella on useampi kuin yksi agentti ja jokaisella niistä on oma aloitus- ja lopetuspisteensä graafissa. Jokaisella aika-askeleella agentti voi joko liikkua toiseen solmuun tai pysyä paikallaan nykyisessä solmussaan (Sharon ym. 2015; Stern ym. 2019). Tarkoituksena on samaan aikaan ratkaista moniagenttireitinhakuongelma ja minimoida reitinhaun kustannusfunktio (engl. cost function). Yleinen kustannusfunktio on *sum-of-costs*, joka on agenttien aika-askelten summa kun ne saapuvat kohteisiinsa. Kustannusfunktiona voi olla myös *makespan*, joka minimoi ajan kunnes viimeinen agentti on saapunut kohdesolmuun tai *fuel*, joka minimoi agenttien kulkeman matkan. Erityisenä huomiona fuel-kustannusfunktiossa on se, että paikallaan odottaminen ei nosta kustannusta.

Moniagenttireitinhaun rajoitteet voivat vaihdella tutkimusalasta riippuen, kuten esimerkiksi saavatko agentit kulkea samaa reittiä pitkin seuraten toisiaan, mutta yleensä perusrajoitteet ovat samat (Sharon ym. 2015; Stern ym. 2019). Perinteisesti agenttien täytyy liikkua lopetuspisteeseen törmäämättä toisiinsa matkalla ts. yhdellä solmulla ei saa olla samanaikaisesti enempää kuin yksi agentti. Kaksi agenttia eivät myöskään saa kulkea saman kaaren kautta samalla aika-askeleella. Moniagenttireitinhakuongelman sovelluksia esiintyy videopelien lisäksi esimerkiksi robotiikassa, ilmailussa, liikennesuunnittelussa ja itseohjautuvissa ajo-

neuvoissa, joten se on saanut viime vuosina paljon huomiota tutkimuksissa ja akateemisissa yhteisöissä.

Moniagenttireitinhakuongelmaan on sekä optimaalisia että epäoptimaalisia ratkaisuja (Sharon ym. 2015). Optimaaliset ratkaisut ovat luonteeltaan NP-kovia, koska agenttien lukumäärä kasvattaa ongelman tila-avaruutta eksponentiaalisesti. A*-algoritmi on esimerkiksi optimaalinen ratkaisu, mutta moniagenttireitinhaussa sen suoritusaika voi olla hyvin pitkä ja muistinkäyttö liian suurta. Tämän vuoksi ongelmissa, joissa agentien lukumäärä on suuri, käytetään usein epäoptimaalisia ratkaisuja.

3 Koneoppiminen

Koneoppiminen on tällä hetkellä yksi teknisten tutkimusalojen suosituimmista aihealueista. Sen ydin rakentuu kysymykselle, pystyykö tietokone jäljittelemään ihmismielen oppimisprosessia ja täten oppia automaattisesti kokemuksen kautta (Das ja Behera 2017; Jordan ja Mitchell 2015). Automaattisella oppimisella pyritään vähentämään manuaalista, tapaus tapauksen perään ohjelmoimista. Sen sijaan konetta opetetaan syöte-tuloste -parien avulla. Vuosikymmenten aikana koneoppimisen tutkimus on edistynyt hyvin paljon eikä loppua ole toistaiseksi näkymässä. Uudet sovellukset ja algoritmit, laskentatehon kasvu ja big data eli hyvin suuret, keskittyneet datamäärät ovat tuoneet tarpeen sekä teorian tutkimukselle että kehittyneille käytännön ratkaisuille.

3.1 Koneoppimisen perusteet

Koneoppiminen on tieteenalana yhdistelmä tietotekniikkaa ja tilastotieteitä. Tietotekniikka ja tietokoneet mahdollistavat ongelmanratkaisun. Suurista datajoukoista oppiminen, erilaisten ennusteiden tekeminen ja päätöksenteko vaativat sen sijaan tilastotieteellisiä menetelmiä (Das ja Behera 2017; Jordan ja Mitchell 2015). Myös neurotieteiden ja psykologian roolit ovat kasvamassa koneoppimisen tutkimuksessa. Esimerkiksi ihmisaivojen ja sitä kautta oppimisprosessin tutkiminen ja hyödyntäminen koneoppimisessa ovat tulevaisuudessa merkittäviä tutkimuksen kohteita (Das ja Behera 2017). Koneoppimisen sovellukset ovat nykyään hyvin laaja-alaiset. Tunnettuja esimerkkejä ovat konenäkö, puheentunnistus ja robotiikka. Konenäköä hyödynnetään etenkin terveydenhoidon alalla anomalioiden tunnistamiseen kuvissa. Mainonnassa koneoppimista käytetään personoitujen suositusten luomiseen ja markkinoinnissa erilaisissa ennusteissa.

Tietoteknisten laitteiden suosio ja sitä kautta datan räjähdysmäinen kasvu ovat olleet suuressa roolissa koneoppimisen hyödyntämisessä. Kyseisiä suuria datamassoja eli big dataa on mahdotonta tarkastella manuaalisesti, joten niiden käsittelyyn on otettu käyttöön koneoppimisen menetelmiä (Jordan ja Mitchell 2015). Koneoppimisalgoritmit pystyvät muokkaamaan palveluita vastaamaan jokaisen henkilökohtaisiin tarpeisiin personoidun datan avulla. Esimer-

kiksi mainoksia voidaan kohdentaa tietyille ryhmille ja vanhoja potilastietoja voidaan hyödyntää hoitotyypin valitsemiseen tietyille potilaille. Personoitu data tuo tosin tietoturvallisia haasteita. Henkilödatasta on pyrittävä tekemään anonyymia, jottei sitä pysty yhdistämään suoraan henkilöihin. Samalla kuitenkin data voi olla niin tarkkaa, että jokaisella sanotaan olevan oma digitaalinen sormenjälki suuressa datamassassa. Myös suorituskykyvaatimukset ovat nousseet datan kasvun myötä, joten algoritmit täytyy kehittää mukautuviksi.

Koneoppiminen perustuu koneen opettamiseen erillisen opetusdatan avulla (Das ja Behera 2017; Jordan ja Mitchell 2015). Opetusdatasta pyritään havaitsemaan piilossa olevia malleja tilastotieteellisten menetelmien avulla. Kohdatessaan uusia esimerkkejä ja uutta dataa kone pystyy muodostamaan arvion vastauksesta havaitun mallin perusteella. Opetusdata voi olla valmiiksi luokiteltua tai kokonaan luokittelematonta, joista luokiteltua dataa käytetään tavallisesti ohjattuun oppimiseen perustuvissa menetelmissä ja luokittelematonta dataa ohjaamattomaan oppimiseen perustuvissa menetelmissä.

Koneoppimisella yritetään perinteisesti ratkaista luokitteluongelmia, joissa datajoukosta voidaan päätellä, kuuluuko käsiteltävä asia tiettyyn luokkaan vai ei (Jordan ja Mitchell 2015). Esimerkiksi sähköposti voidaan luokitella roskapostiksi tai aidoksi sähköpostiksi riippuen mitkä ovat luokittelun vaihtoehdot. Koneoppimisen avulla kasvatetaan luokittelun tarkkuutta, joka on kyseisen luokitteluongelman tärkein mittari. Opetusdata voi koostua kokoelmasta erilaisia sähköposteja, jotka on valmiiksi luokiteltu roskapostiksi tai aidoksi sähköpostiksi. Syötteenä on siis sähköposti ja tulosteena *roskaposti* tai *aito*. Koneoppimisen avulla kone opetetaan tunnistamaan malleja ja kuvioita datassa, jolloin se pystyy myöhemmin tunnistamaan samoja kuvioita oikeissa tilanteissa.

3.2 Koneoppimisen paradigmat

Koneoppiminen voidaan jakaa eri osiin oppimistyylin mukaan. Oppimistyylejä ovat esimerkiksi ohjattu oppiminen, puoliohjattu oppiminen, ohjaamaton oppiminen, transduktio ja vahvistusoppiminen (Das ja Behera 2017). Näistä tunnetuimpia ovat ohjattu oppiminen, ohjaamaton oppiminen ja vahvistusoppiminen.

3.2.1 Ohjattu oppiminen

Käytetyin oppimistyyli on ohjattu oppiminen ja siihen liittyvät menetelmät (Jordan ja Mitchell 2015; Nasteski 2017). Ohjatun oppimisen tehtävät voidaan jakaa luokitteluun ja regressioon. Luokitteluongelmissa vastaukset ovat kategorisia, esimerkiksi "lintu", "koira"tai jokin kokonaisluku, ja regressiossa jatkuvia lukuarvoja, esimerkiksi hinta. Ohjatussa oppimisessa opetusaineisto tuodaan (x, y) pareina, jossa x on syöte ja y on tulos. Oppijan tavoitteena on approksimoida funktio, joka kuvaa kaikki syötteet niitä vastaaviin tuloksiin. Yksittäisessä luokittelutehtävässä oppija luo mahdollisimman tarkan ennusteen y syötteen x perusteella ja vertaa ennustetta oikeaan tulokseen. Jos ennuste epäonnistuu, niin mallia muokataan tarpeen mukaisesti. Yleisesti ohjatun oppimisen heikkoutena voidaan pitää luokitellun datan saatavuutta ja laatua (Das ja Behera 2017). Data täytyy esikäsitellä ja luokitella ennalta muilla menetelmillä, joka nostaa lopullista laskentakustannusta. Toisaalta jotkin menetelmät, kuten naiivi Bayes-luokittelija, eivät vaadi suuria datamääriä tuottaakseen tarkkoja vastauksia (Osisanwo ym. 2017). Naiivi Bayes-luokittelijan lisäksi tunnettuja ratkaisu-algoritmeja ovat päätöspuut ja tukivektorikoneet.

Päätöspuu (engl. decision tree) on tiedonlouhinnassa ja koneoppimisessa käytetty luokittelualgoritmi, jolla luokitellaan asioita ominaisuuksien perusteella (Nasteski 2017; Osisanwo ym. 2017). Päätöspuu koostuu juurisolmuista, oksasolmuista ja lehdistä. Juuri- ja oksasolmuissa sijaitsee yksittäisiä ominaisuuksia ja lehtisolmuissa mahdollisia tuloksia. Luokittelu alkaa juurisolmusta ja päätyy johonkin lehtisolmuun eli yksittäiseen tulokseen. Tarvittaessa päätöspuun yleistä arviointi- ja suorituskykyä voidaan parantaa karsimalla (engl. prune). Karsimisessa päätöspuun tarpeettomia solmuja poistetaan kokonaan käytöstä, jolla vähennetään esimerkiksi ylisovittamista.

Naiivi Bayes-luokittelija (engl. Naive-Bayes) on ohjatussa oppimisessa käytetty luokittelumenetelmä, jolla luokitellaan asioita piirteiden perusteella käyttäen Bayesin todennäköisyysteoreemaa (Nasteski 2017; Rish ym. 2001). Yksinkertaistamisen vuoksi naiivi Bayes-luokittelijassa piirteet oletetaan tilastollisesti riippumattomiksi toisistaan, jolloin tuloksena saadaan approksimaatio luokittelusta. Tästä huolimatta naiivi Bayes-luokittelija on osoittautunut tehokkaaksi menetelmäksi käytännön luokitteluongelmissa, kuten tekstin luokittelussa ja lääketieteellisessä diagnosoinnissa (Rish ym. 2001).

Tukivektorikone (engl. support vector machine, SVM) on yksi tunnetuimmista ja tehokkaimmista ohjatun oppimisen menetelmistä luokitteluun ja regressiotehtäviin (Cervantes ym. 2020; Osisanwo ym. 2017). Opetusaineistosta saaduista ominaisuuksista muodostetaan piirreavaruus (engl. feature space), josta luokat erotellaan toisistaan piirteiden perusteella. Luokkia erottelevaa rajapintaa kutsutaan tasoksi (engl. plane) ja etäisyyttä tasoa lähimpänä olevaan datapisteeseen marginaaliksi (engl. margin). Kaksi vektoria, jotka kulkevat tason molempien puolien lähimpien pisteiden kautta, muodostetaan yhdensuuntaisesti tasoon nähden ja näiden vektorien etäisyys toisistaan pyritään maksimoimaan, jolloin pyritään minimoimaan luokitteluvirhe ja parantamaan yleistämistä.

3.2.2 Ohjaamaton oppiminen

Ohjaamattomassa oppimisessa analysoidaan luokittelematonta dataa (Das ja Behera 2017; Jordan ja Mitchell 2015). Tarkoituksena on tehdä päätelmiä datan rakenteesta ja malleista ilman ulkopuolista ohjeistusta. Erityisen tärkeää on pyrkiä valitsemaan datasta ominaispiirteet, jotka ovat relevantteja datan tarkastelun kannalta. Liian suuri määrä piirteitä kasvattavat datajoukon ulottuvuutta eli dimensiota ja näin ollen tekee datajoukon rakenteesta monimutkaisen. Datajoukkoa pyritään strukturoimaan eri menetelmien avulla, jolloin datan tarkastelua pyritään selkeyttämään. Klusterointi ja dimension redusointimenetelmät ovat yleisiä datan strukturointimenetelmiä.

Klusteroinniksi kutsutaan menetelmiä, joissa datapisteet jaetaan ominaispiirteidensä perusteella klustereihin. Yksittäiset klusterit sisältävät siis ominaisuuksiltaan samankaltaiset datapisteet. *K-means* -klusterointi on tunnettu klusterointimenetelmä, jossa datapisteet n jaetaan K:hon klusteriin. Klusterin keskipistettä lähimpänä oleva datapiste toimii eräänlaisena prototyyppinä, johon muiden datapisteiden etäisyyksiä verrataan. Uusi datapiste katsotaan kuuluneeksi siihen klusteriin, jonka keskipisteeseen on pienin etäisyys.

Dimension redusointimenetelmät pyrkivät tihentämään datajoukkoa ja helpottamaan sen analysointia. Niiden periaatteena on vähentää datajoukossa tarkasteltavia ominaispiirteitä ja näin ollen vähentää datajoukon kokonaisulottuvuutta, jolloin datajoukon analysointi yksinkertaistuu. Uuteen datajoukkoon pyritään sisällyttämään keskeisimmät piirteet siten, ettei relevant-

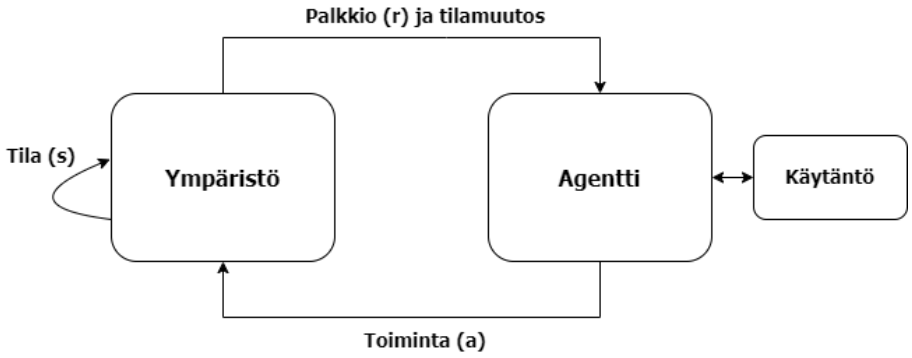
tia informaatiota katoaisi. Pääkomponenttianalyysi on yksi tunnetuimmista ja vanhimmista dimension redusointimenetelmistä, jossa olemassaolevasta datajoukosta poimitaan tärkeimmät piirteet ja luodaan niistä pääkomponentteja (engl. principal components) (Abdi ja Williams 2010).

3.2.3 Vahvistusoppiminen

Vahvistusoppiminen on kolmas suuri koneoppimisen paradigma, jossa oppijana toimii agentti ja agentin toimintaa ohjaavat palkkiot (Arulkumaran ym. 2017; Li 2018; Jordan ja Mitchell 2015). Vahvistusoppimisen opetusaineisto eroaa muiden paradigmojen opetusaineistoista siten, ettei tietylle syötteelle x anneta valmiiksi oikeaa tulosta y tai etsitä rakenteellisia malleja datajoukosta, vaan agentti saa jokaisesta suorittamastaan toiminnosta positiivisen tai negatiivisen palkkion. Palkkiot ohjaavat agentin oppimisprosessia ja käyttäytymistä ympäristössä. Toimintojen ja niistä saatujen palkkioiden lisäksi agentti on jatkuvassa vuorovaikutuksessa ympäristönsä kanssa siten, että agentti tarkkailee toimintojensa vaikutuksia ja ympäristön tilamuutoksia. Näin agentti pystyy muokkaamaan toimintojaan saamiensa palkkioiden perusteella. Vahvistusoppimistehtävän ideaaliratkaisu on palkkiot maksimoivan käytännön (engl. policy) selvittäminen.

Vahvistusoppiminen voidaan kuvata Markovin päätösprosessina (engl. Markov decision process, MDP) tilojen S, toimintojen A ja palkkiosignaalien r avulla (Arulkumaran ym. 2017). Jokaisella aika-askeleella t agentti suorittaa toiminnon a, jolloin ympäristön tila s muuttuu ja muutos viestitään agentille palkkiosignaalin r kautta. Palkkiosignaalin lisäksi ympäristö viestii agentille tilamuutoksesta, jolloin agentti saa tiedon uudesta tilasta $s + 1$. Palkkion ja uuden tilan perusteella agentti valitsee jälleen seuraavan toimintonsa. Markov-prosessi voi olla jaksollinen (engl. episodic), jolloin tila nollaantuu tietyn ajan jälkeen. Agentin lopullisena tavoitteena on saavuttaa paras mahdollinen käytäntö (engl. policy), joka maksimoi saadun palkkion määrän. Kuva 3 havainnollistaa vahvistusoppimisen toimintaa.

Oppiakseen optimaalisen käytännön agentin tekemät päätökset vaativat jonkin ennusteen tulevien toimintojen arvosta. Yleisesti tämän ennusteen laskemista kutsutaan nimellä *action-value function*, ja yksi tunnettu menetelmä sen laskemiseen on Q-oppiminen (engl. Q-learning).



Kuvio 3. Vahvistusoppimisympäristön oppimissilmukka yhdellä aika-askeleella.

Q-oppiminen on *model-free* menetelmä eli se ei tarvitse toimiakseen erillistä mallia ympäristöstä. Q-oppimisessa jokaisen tilan eri toimintoa arvioidaan Arulkumaran ym. (2017) esittämän kaavan

$$Q(s,a) = r_t + \gamma max_a Q^{\pi}(s_{t+1}, a)$$

mukaisesti, jossa s on tila, a on toiminto, r_t on tilan palkkio, γ on erillinen vähennys, joka määrittelee tulevien toimintojen tärkeyden aiempiin verrattuna ja $max_a Q^{\pi}(s_{t+1}, a)$ on ennakkoarvio tulevasta, optimaalisesta palkkiosta.

Funktiosta saatuja arvoja päivitetään erillisessä Q-taulussa. Oppimisepisodien aikana agentti pystyy ennakoimaan Q-taulun avulla jokaisen mahdollisen toimintonsa tulevaa lopullista palkkiota ja tarvittaessa päivittämään Q-taulun arvoja. Ongelmaksi osoittautuu kuitenkin uuden tutkimisen (engl. exploration) ja olemassa olevan tiedon hyödyntämisen (engl. exploitation) tasapainotus (Arulkumaran ym. 2017). *Epsilon-Greedy* on menetelmä, jossa tietyllä todennäköisyydellä ε agentti valitsee pienemmän arvon tilan, jolloin voidaan löytää uusia, parempia ratkaisuja haluttuun lopputulokseen. Agentti voi myös jatkaa aiemman käytännön hyödyntämistä todennäköisyydellä 1 - ε.

Vahvistusoppimiseen liittyy haasteita optimaalisen käytännön löytämisessä ja agentin satunnaisissa toiminnoissa (Arulkumaran ym. 2017). Optimaalinen käytäntö on mahdollista löytää vain jatkuvan yrittämisen kautta, ja ainoana oppimisen signaalina toimivat saadut palkkiot. Riippuen tehtävän haastavuudesta lukuisatkaan yritykset eivät välttämättä takaa optimaalisinta ratkaisua. Toiseksi agentin palkkioon johtavissa toiminnoissa saattaa olla paljon turhia toimintoja eli toisin sanoen agentti ei välttämättä tiedä, mitkä toiminnot oikeasti johtivat

palkkion saamiseen. Esimerkiksi robotin liikkuessa sisätiloissa sen havainnot riippuvat vahvasti siitä, kohtaako se matkalla umpikujia vai siirtyykö se satunnaisilla toiminnoilla suoraan kohteeseen. Jos palkkio annetaan maaliin pääsemisen yhteydessä, niin robotti voi olettaa, että umpikujaan päätyminen kuului palkkion saamiseen vaativien toimintojen joukkoon. Tätä ongelmaa kutsutaan nimellä *(temporal) credit assignment problem*.

3.3 Syväoppiminen

Koneoppimisen ja sitä kautta myös syväoppimisen tutkimus on ollut viime vuosina hyvin aktiivista (Pouyanfar ym. 2018). Syväoppimisella ollaan onnistuttu ratkaisemaan ongelmia, joita tekoälyn tutkimuksessa on yritetty ratkaista vuosia (LeCun, Bengio ja Hinton 2015; Pouyanfar ym. 2018). Erityisesti tuloksia on tullut kuvantunnistuksessa, puheentunnistuksessa ja erilaisissa luonnollisen kielen prosessoinnin tehtävissä. Suurimpia syväoppimisen tutkimuksen askeleita viime vuosikymmenillä ovat vastavirta-algoritmi (engl. backpropagation), konvoluutioneuroverkon (engl. Convolutional Neural Network, CNN) inspiraationa tunnettu neocogitron-neuroverkko, siitä hieman myöhemmin kehitetty takaisinkytketty neuroverkko (engl. Recurrent Neural Network, RNN) ja syvä uskomusverkko (engl. Deep Belief Network, DBM). Syvä uskomusverkko mahdollisti entistä syvemmän neuroverkon oppimisen, jonka seurauksena siirryttiin yleisesti käyttämään termiä syväoppiminen. Viime vuosien yksi tunnetuimpia syväoppimisen sovelluksia on muun muassa Googlen AlphaGo, joka onnistui päihittämään useita Go-pelin ammattilaisia pelin äärimmäisestä strategisesta haastavuudesta huolimatta. Tässä luvussa kerrotaan neuroverkkojen, syväoppimisen ja syvän vahvistusoppimisen toiminnan perusteet sekä esitellään Soft Actor Critic -algoritmi, jota myöhemmin sovelletaan tutkimuksen empiirisessä osuudessa.

3.3.1 Neuroverkkojen perusteet

Neuroverkkojen perusosiin kuuluvat perseptronit (engl. perceptron), jotka ovat keinotekoisia neuroneita (Nielsen 2015). Perseptronit koostuvat binäärisistä syötteistä $x_1, x_2, x_3, ..$, ja yksittäisestä binäärisestä tuloksesta. Syötteiden relevanttius määritellään erillisten painoarvojen (engl. weight) $w_1, w_2, w_3, ..$, perusteella. Syötteet ja painoarvot lasketaan painotettuna summana $\sum_j w_j x_j$ ja tulokseen lisätään neuronin vakiotermi (engl. threshold tai bias), jo-

ka on neuronille parametrina annettu luku ja määrittää onko lopullisena tuloksena 0 vai 1. Yksinkertaistettuna Nielsen (2015) kuvaa perseptronia "laitteeksi, joka tekee päätökset punnitsemalla todisteita".

Yleisesti neuroverkko koostuu syötekerroksesta, piilokerroksesta ja ulostulokerroksesta (Nielsen 2015). Piilokerros koostuu yhdestä tai useammasta perseptroni-kerroksesta. Ensimmäisellä perseptroni-kerroksella tehdään yksinkertaisimmat päätökset ja jokaisella seuraavalla kerroksella tehdään toinen toistaan monimutkaisempia päätöksiä. Kerrosten suuri lukumäärä mahdollistaa siis vaativienkin päätösten tekemisen.

Oppiminen on mahdollista painoarvoja muuttamalla, jolloin koko neuroverkon lopullinen tulos muuttuu (Nielsen 2015). Vaarana kuitenkin on, että pienet muutokset muuttavat neuroverkon tulosta radikaalisti vastakkaiseen suuntaan. Korvaamalla alkuperäiset, binääriset perseptronit Sigmoid-neuroneilla painoarvojen muutokset eivät aiheuta suurta muutosta tuloksessa. Sigmoid-neuronit käyttävät sigmoid-funktiota porrasfunktion sijaan, jolloin tulos voi olla mitä tahansa 0 ja 1 välillä.

Kuten koneoppimisessa, myös syväoppimisessa neuroverkon opettaminen tapahtuu opetusaineiston avulla (Nielsen 2015). Luokitteluongelman suoran ratkaisemisen sijaan neuroverkon opettamisessa keskitytään ensin painoarvojen ja vakiotermien muodostaman virhefunktion minimiarvon löytämiseen ja vasta sen jälkeen luokittelun tarkkuuteen. Virhefunktion minimin löytämiseen käytetään tunnettua optimointikeinoa, gradienttimenetelmää (engl. gradient descent) ja vastavirta-algoritmia (engl. backpropagation). Vastavirta-algoritmilla voidaan laskea kerralla gradianttimenetelmään tarvittavat osittaisderivaatat painoarvojen ja vakiotermien suhteen virhefunktiossa. Tuloksena saadaan virhefunktion gradientti, jonka jälkeen painoarvot ja vakiotermit päivitetään neuroneissa takaperin iteroiden viimeisestä kerroksesta lähtien ja menetelmä toistetaan.

Gradienttimenetelmällä pyritään yleisellä tasolla selvittämään virhefunktion minimi suorittamalla pieniä askeleita kohti globaalia tai lokaalia minimiä (Nielsen 2015). Askeleiden pituus määritellään oppimisnopeudella (engl. learning rate), joka on usein pieni vakioluku. Oppimisnopeus ei saa kuitenkaan olla liian pieni, koska muuten algoritmi toimii hyvin hitaasti. Oppimisnopeutta voidaan tehostaa myös käyttämällä stokastista gradienttimenetelmää, jossa

painoarvot ja vakiotermit päivitetään vasta pienen opetusaineistojoukon jälkeen. Tästä saatu tulos toimii hyvänä arviona todelliselle gradienttivektorille.

3.3.2 Syväoppimisen haasteet

Yksi suurimmista syväoppimisen haasteista on ylisovittaminen (engl. overfitting) (Nielsen 2015). Ylisovittamisessa neuroverkko oppii opetusaineiston liiankin täsmällisesti ja menettää sen johdosta kykynsä yleistää oppimaansa toisiin tapauksiin esimerkiksi testiesimerkkidatassa. Ylisovittamista ilmenee usein jos opetusaineiston luokittelun tarkkuus nousee lähelle sataa prosenttia, mutta testiesimerkkidatan luokittelun tarkkuus pysyy ennallaan. Ylisovittamista voidaan estää esimerkiksi neuroverkon pienentämisellä tai opetusaineiston kasvattamisella. Joskus opettaminen voidaan myös keskeyttää, jos testiesimerkkidatan tarkkuus ei tunnu enää kasvavan.

Ylisovittamista voidaan hillitä myös säännöstelyllä (engl. regularization), jossa pyritään pienentämään neuronien painoarvoja virhefunktion minimin löytämiseen (Nielsen 2015). Suurten painoarvojen pienet muutokset voivat aiheuttaa suuria muutoksia neuroverkon toimintaan, jolloin se voi ajautua ylisovittamaan opetusaineistoa. Säännöstely toisin sanoen tehostaa neuroverkon kykyä tehdä yleistyksiä. Yleisiä säännöstelymenetelmiä ovat L1- ja L2-säännöstelyt sekä neuronien poistaminen (engl. dropout). Neuronien poistaminen eroaa muista mainituista siten, ettei siinä keskitytä virhefunktion muokkaamiseen, vaan neuroverkon piilokerroksesta poistetaan hetkellisesti neuroneita. Neuroverkon toiminta pysyy muuten ennallaan, mutta vain osaa neuroneista käytetään ja niiden parametreja päivitetään. Seuraavalla iteraatiolla neuronit palautetaan ja jälleen osa piilokerroksen neuroneista poistetaan väliaikaisesti. Jokainen iteraatio käyttää opettamiseen rakenteeltaan erilaista neuroverkkoa, koska osa neuroneista on piilotettu. Tämän avulla saadaan luotua tilanne, jossa samaa neuroverkkoa opettaessa opetetaankin useita erilaisia neuroverkkoja ja näiden neuroverkkojen ylisovittamiset kumoavat toisensa.

3.4 Syvä vahvistusoppiminen

Yksi tekoälyn tutkimuksen suurimpia tavoitteita on luoda täysin autonominen ja ohjailtava agentti, joka pystyy toimimaan erilaisissa ympäristöissä samalla oppien ja kehittäen käyttäytymistään paremmaksi (Arulkumaran ym. 2017). Alunperin vahvistusoppimisella pystyttiin kehittämään oppivia agentteja, mutta ongelmaksi osoittautui ongelmien ja ratkaisujen suppeus, koska pelkällä vahvistusoppimisella ei pystytty käsittelemään ja hyödyntämään moniulotteista dataa, kuten robotin raakaa sensoridataa tai kuvan pikseleitä. Syväoppimisella onnistuttiin vastaamaan tähän vaatimukseen, ja pian syvän vahvistusoppimisen sovelluksilla pystyttiin jo pelaamaan Atari 2600:n videopelejä käyttämällä hyväksi kuvan pikselidataa.

3.4.1 Syvän vahvistusoppimisen menetelmät

Syvä vahvistusoppiminen hyödyntää syväoppimista ja neuroverkkoja suurten tila- ja havaintoavaruuksien tapauksissa (Arulkumaran ym. 2017; Li 2018). Se eroaa vahvistusoppimisesta pääasiassa funktioapproksimaation suhteen. Esimerkiksi tavallisessa Q-oppimisessa muodostetaan taulu tila-toiminta -pareista ja niiden Q-arvoista, mutta syvässä Q-oppimisessa (engl. deep Q-learning) taulu korvataan neuroverkolla. Syvässä Q-oppimisessa tila, esimerkiksi pikselidata, syötetään neuroverkolle ja tuloksena saadaan mahdolliset toiminnot ja niiden arvot. Oppiminen tapahtuu neuroverkon tapaisesti eli pyritään maksimoimaan esimerkiksi palkkiofunktio ja päivitetään neuroverkon painoarvoja sen mukaisesti. Syvän Q-verkon ongelmaksi voi muodostua epävakautta, jota voidaan korjata *Experience replay*-menetelmällä. Menetelmässä tallennetaan muistiin satunnaisesti aiempia kokemuksia, joita agentti voi myöhemmin käyttää oppimiseen. Menetelmän hyöty perustuu siihen, että opettamisesta pyritään poistamaan ajalliset korrelaatiot.

Arvofunktioiden approksimaation perustuvat menetelmät kohtaavat suuria haasteita moniulotteisissa tila- ja toimintoavaruuksissa, kuten robotiikassa esiintyvissä ongelmissa on tapana olla (Arulkumaran ym. 2017; Deisenroth, Neumann, Peters ym. 2013). Vaihtoehtoisina menetelminä ovat *policy search*-menetelmät, joissa arvofunktiomallin ylläpitämisen sijaan käytetään parametrisoituja käytäntöjä. Parametreja päivittämällä pyritään maksimoimaan odotettu tulos ja löytämään paras käytäntö. Policy search -menetelmät voivat olla *model-based*

tai *model-free* eli joko ylläpidetään mallia ympäristöstä tai opetellaan suoraan käytäntö. Usein käytännön opetteleminen on helpompaa kuin tarkan mallin oppiminen.

Arvofunktioita ja *policy search*-menetelmiä voidaan hyödyntää myös yhdessä, joita kutsutaan tekijä-kriitikko -menetelmiksi (engl. actor-critic) (Arulkumaran ym. 2017). Tässä kontekstissa sana tekijä viittaa käytäntöön ja kriitikko arvofunktioon. Tekijä käyttää kriitikolta saamaa palautetta oppimiseen. Toisin sanoen, tekijä saa ympäristöltä tilan ja valitsee sen perusteella toiminnon ja samaan aikaan kriitikko saa ympäristöltä tilan ja edellisen aika-askeleen palkkion. Kriitikko laskee tämän perusteella ennustevirheen (engl. temporal difference) ja päivittää itsensä ja tekijän.

3.4.2 Soft actor-critic

Soft actor-aritic (SAC) on Haarnoja, Zhou, Abbeel ym. (2018) kehittämä *model-free* ja *off-policy* syvä vahvistusoppimis-algoritmi. Se kehitettiin vastaamaan yleisiin vahvistusoppimis-algoritmien ongelmiin reaalimaailmassa, kuten opetusaineiston kompleksisuuteen (engl. sample complexity) ja hyperparametrien herkkyyteen (Haarnoja, Zhou, Hartikainen ym. 2018). Yksinkertaisetkin tehtävät saattoivat vaatia satoja tuhansia ellei jopa miljoonia opetusaskeleita ennen kuin sopiva lopputulos saavutettiin. Tämän lisäksi eri tehtävien välillä joutui viritttelemään monien eri hyperparametrien arvoja, jotta oppiminen olisi edes mahdollista. SAC pyrkii mahdollisimman satunnaisiin toimintoihin samalla maksimoiden odotetun palkkiosumman, jota kutsutaan myös nimellä *maximum entropy*. Tavallisesti parhaimman käytännön löytämisessä halutaan maksimoida saatu palkkio, mutta *maximum entropy* -periaatteessa yhtälöön lisätään erillinen entropia-termi. Entropian suuruutta kontrolloidaan erillisen kertoimen α avulla, joka voidaan asettaa myös nollaksi, jolloin yhtälö palauttaa jälleen maksimipalkkion antavan käytännön.

TODO: SAC testauksesta, suoriutui paremmin kuin muut DRL-algoritmit, ehkä lisää teoriataustaa?

4 Unity

Unity on Unity Technologiesin kehittämä pelinkehitysalusta, joka sisältää oman renderöinti- ja fysiikkamoottorin sekä Unity Editor -nimisen graafisen käyttöliittymän (Juliani ym. 2018). Unityllä on mahdollista kehittää perinteisten 3D- ja 2D-pelien lisäksi myös esimerkiksi VR-pelejä tietokoneille, mobiililaitteille ja pelikonsoleille. Unitystä onkin vuosien mittaan tullut yksi tunnetuimmista pelinkehitysalustoista, jonka parissa työskentelee kuukausittain jopa 1.5 miljoonaa aktiivista käyttäjää ("Unity" 2022).

Viime vuosina Unityä on käytetty simulointialustana tekoälytutkimuksen parissa (Juliani ym. 2018). Unity mahdollistaa lähes mielivaltaisten tilanteiden ja ympäristöjen simuloinnin 2D ruudukkokartoista monimutkaisiin pulmanratkaisutehtäviin, joka on sen suurimpia vahvuuksia simulointialustana. Kehitystyö ja prototypointi ovat Unityllä myös erityisen nopeaa.

4.1 Unityn hierarkia

Tässä luvussa käsitellään Unityn hierarkiaa. Aliluvuissa käydään läpi hierarkian osat ylimmästä lähtien.

4.1.1 Unity-projekti

Unityn hierarkian ylin osa on projekti, jonka luomisesta kehitystyö aina alkaa. Unityssä on mahdollista luoda projekti valmiista pohjista, joita ovat esimerkiksi 2D-, 3D- ja VR-pohjat. Pohjien avulla projekteihin saa lisättyä suoraan suositeltavat, parhaita käytäntöjä mukailevat asetukset.

Unity-projekteja voidaan hallinnoida ja avata erillisellä Unity Hub -sovelluksella. Unity Hub kertoo muun muassa mitä Unityn versiota projekti tukee. Projekteja voi tarvittaessa siirtää (engl. migrate) toimimaan uusimmilla Unityn versioilla, mutta siirto voi aiheuttaa toiminnallisuuksien muutoksia tai virheitä projektissa.

4.1.2 Näkymät

Projektista seuraavana hierarkiassa ovat näkymät (engl. scene). Näkymät toimivat työskentelyalustoina projektissa. Projekti sisältää aina yhden tai useamman näkymän, koska ilman niitä mitään ei pysty luomaan. Tavallisesti yksittäinen näkymä kuvaa aina yhtä kenttää, tasoa tai aluetta pelissä, ja siirryttäessä toiselle alueelle Unity pystyy lataamaan ajon aikana uuden skenen. Näkymän lataaminen voi tosin viedä aikaa, joten lataus peitetään useimmiten latausruuduilla, jotka voivat myös olla omia, yksinkertaisia näkymiä. Yksinkertaisimmillaan peli voi kuitenkin sisältää vain yhden näkymän, joka muokkautuu ja jota muokataan ajon aikana.

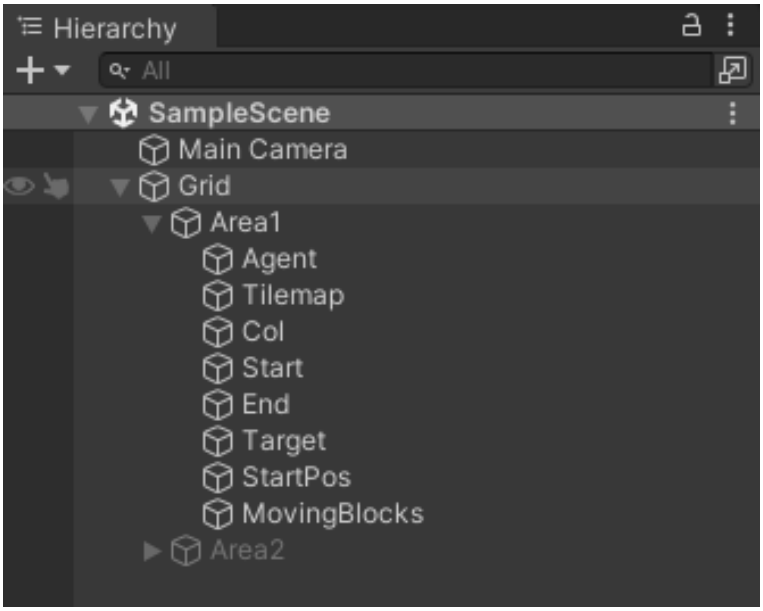
Projektin luonnin jälkeen Unity lisää siihen automaattisesti aloitusnäkymän, joka sisältää kamera- ja valonlähde-peliobjektin. Tätä näkymää voi lähteä muokkaamaan lisäämällä siihen erilaisia peliobjekteja, esimerkiksi maata ja erilaisia geometrisia muotoja.

4.1.3 Peliobjektit ja Prefabit

Peliobjektit (engl. GameObject) ovat tärkein osa Unityn pelinkehitysprosessia, koska kaikki peliin luotavat objektit ovat taustaltaan peliobjekteja. Peliobjektit eivät itsessään tee mitään tai näytä miltään, vaan ne toimivat säiliöinä komponenteille. Peliobjekteja voidaan järjestellä vanhempi-lapsi -periaatteella. Lapsiobjektit liikkuvat vanhemman mukana, jolloin niitä ei tarvitse liikutella näkymässä erikseen. Kuva 4 havainnollistaa esimerkkinäkymän peliobjekteja ja niiden vanhempi-lapsi -suhteita.

Prefabit ovat peliobjektien valmiita malleja, joita luodaan peliobjektien tavoin. Prefabit vähentävät toistuvaa työtä peliobjektien luomisen yhteydessä. Prefabeille voidaan lisätä komponentteja ja lapsiobjekteja kuten peliobjekteille. Kun prefabi lisätään näkymään, sen komponentteja ja arvoja voidaan muuttaa tarvittaessa ilman, että alkuperäisen prefabin arvot ja komponentit muuttuvat.

Peliobjekteja voi lisätä "GameObject-valikosta Unityn ylävalikosta joko tyhjinä objekteina tai valmiina kokonaisuuksina. Valmiita peliobjekteja ovat esimerkiksi erilaiset valonlähteet tai 3D-objektit, ja ne sisältävät automaattisesti tarvittavat komponentit.

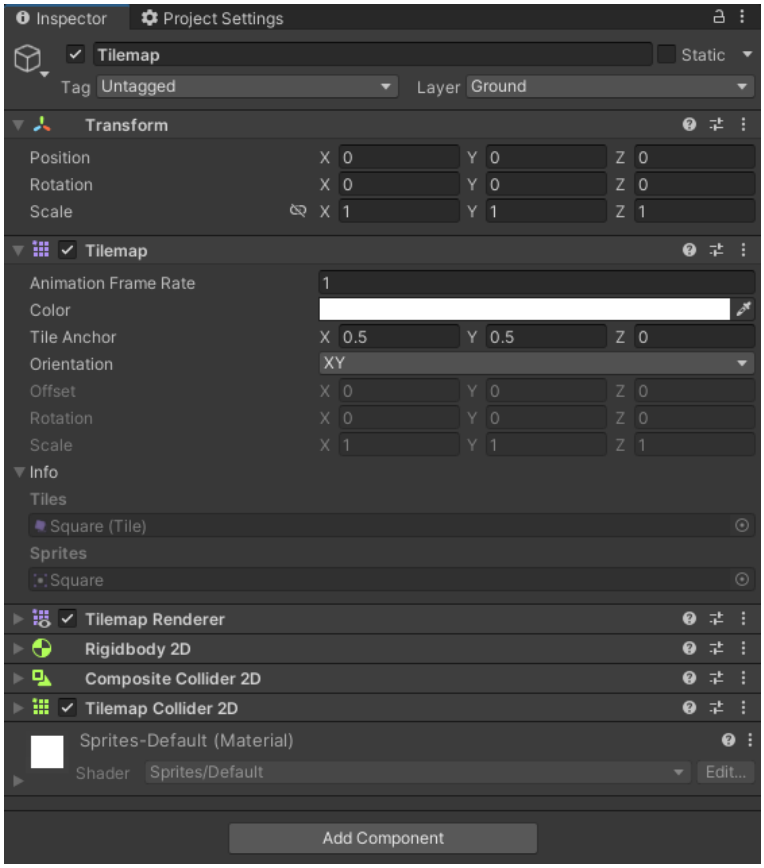


Kuvio 4. Lista peliobjekteista avoimessa näkymässä. Sisäkkäiset peliobjektit ovat lapsiobjekteja.

4.1.4 Komponentit

Komponentit antavat peliobjekteille ominaisuuksia ja toiminnallisuuksia kuten muodon, värin tai fysiikan. Komponentteja voi olla rajattomasti, mutta peliobjektilla on luonnin jälkeen vähintään Transform-komponentti, joka määrittää peliobjektin sijainnin, suunnan ja skaalan. Transform-komponenttia ei voi poistaa peliobjektilta. Esimerkiksi valmis 3D-objekti pallo (sphere) saa automaattisesti Mesh Filter-, Mesh Renderer- ja Sphere Collider-komponentit. Kaksi ensimmäistä komponenttia keskittyvät pallon graafisiin ominaisuuksiin ja Sphere Collider fyysisiin törmäysominaisuuksiin. Sphere Collider-komponentti asettuu automaattisesti pallon graafisen ulkomuodon kokoiseksi.

Usein komponenteilla on erilaisia arvoja, kuten koko tai väri, joita voi muokata käyttöliittymäelementtien avulla. Komponentit voivat sisältää myös viittauksia muihin peliobjekteihin, tiedostoihin tai assetteihin(käännös?). Esimerkiksi Sprite Renderer -komponenttiin voidaan lisätä viittaus kuvatiedostoon, jolloin Unity renderöi peliobjektin kohdalle lisätyn kuvan. Kuvassa 5 on lista peliobjektin komponenteista ja



Kuvio 5. Lista peliobjektin komponenteista.

4.1.5 Skriptit

Ohjelmoinnin merkitys Unityn käytössä tulee skripteistä. Skriptit ovat ohjelmakooditiedostoja, joita voi lisätä peliobjektiin komponentin tavoin, jos valmiit komponentit eivät riitä toiminnallisuuksiltaan. Skripteissä voi esimerkiksi määritellä ominaisuuksia ja arvoja, joita voi muokata komponenttilistauksessa tai ajon aikana. Unity tukee tällä hetkellä vain C-# -ohjelmointikieltä, mutta ennen myös Javascriptiin pohjautuvaa UnityScript-ohjelmointikieltä.

Unity tarjoaa skripteihin MonoBehaviour-pohjaluokan, joka mahdollistaa pelinkehityksen tärkeimmät osat eli Start()-aloitusfunktion ja Update()-päivitysfunktion. Start()-funktio ajetaan ennen yhtäkään päivitysfunktiota, joten siinä voidaan määrittää ja alustaa tarvittavat alkuarvot. Update()-funktio ajetaan joka ruudunpäivityskerralla, joten siihen sijoitetaan usein pelilogiikka ja mahdollisesti fyysiset toiminnallisuudet.

Skriptien avulla voidaan tehdä lähes kaikki samat asiat kuin Unity Editorissa. Peliobjekteja voidaan etsiä tagien tai nimien kautta ja niille voidaan lisätä ja poistaa komponentteja ajon aikana. Käyttämättömät peliobjektit voidaan tuhota tai asettaa epäaktiivisiksi jos niitä ei tarvita.

4.2 Machine Learning Agents

Machine Learning Agents on Unitylle kehitetty ilmainen koneoppimispaketti, joka mahdollistaa Unity Editorilla luotujen simulaatioympäristöjen ja Python APIn välisen vuorovaikutuksen (Juliani ym. 2018). ML-Agents SDK (Software Development Kit) tarjoaa kaikki toiminnallisuudet ja skriptit toimivan koneoppimisympäristön luomiseen. Kuva 6 havainnollistaa yksinkertaisen ML-Agents koneoppimisympäristön toimintaa.

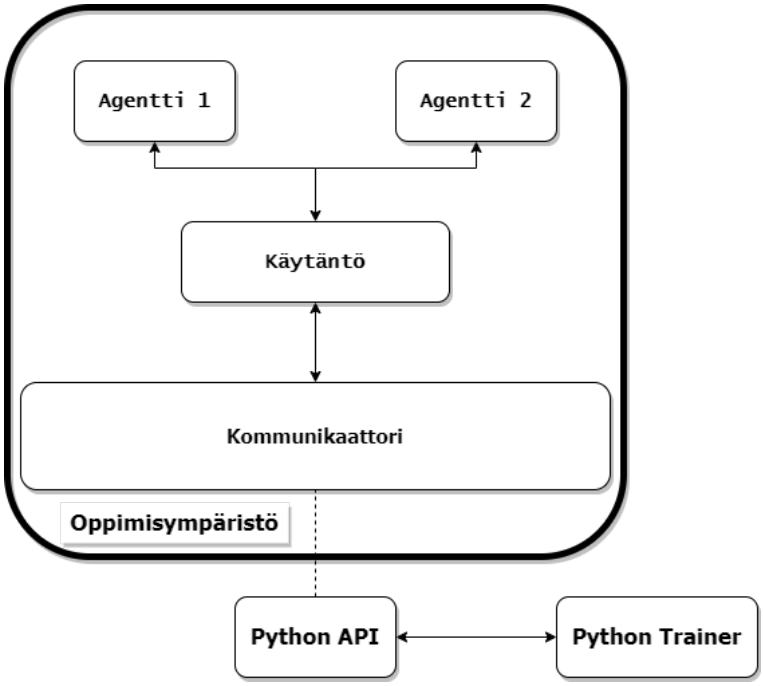
4.2.1 ML-Agents SDK

ML-Agents SDK sisältää kolme ydinosaa: sensorit, agentit ja akatemia. Agentti-komponentti voidaan lisätä suoraan Unityn peliobjektille, jolloin se pystyy keräämään havaintoja, suorittamaan toimintoja ja vastaanottamaan palkkioita. Sensorit mahdollistavat havaintojen keräämisen eri tavoin. Akatemia ylläpitää tietoa simulaation askelmäärästä ja ympäristön parametreista sekä ohjaa agenttien toimintaa.

Agentin käytäntö määritellään Behavior Name -nimikkeen avulla. Eri agenteilla voi olla sama käytäntö, jolloin agentit käyttävät kyseistä käytäntöä päätöksentekoon. Myös useiden erilaisten agenttien toiminta voidaan mahdollistaa erinimisillä käytännöillä.

4.2.2 Python API ja PyTorch

Python APIa käytetään Unityllä tehdyn simulaatioympäristön ja koneoppimissilmukan käsittelyyn. APIn avulla tekijän ei tarvitse itse olla suoraan yhteydessä Pythonin koneoppimiskouluttajaan, vaan API tarjoaa helppokäyttöiset, valmiit menetelmät koneoppimissilmukan luomiseen. Tarkemmin APIsta ja sen toiminnasta voi lukea dokumentaatiosta (tähän linkki sivun alalaitaan tulevasta docsista?).



Kuvio 6. Lohkokaavio Unity ML-Agents toiminnasta.

PyTorch on avoimen lähdekoodin koneoppimiskehys, johon pohjautuen Unity ML-agents -paketin koneoppimiseen liittyvät toteutukset on tehty (Unity ML agents docs). PyTorch sisältää kaikki syvän koneoppimisen perusosat datan kanssa työskentelystä ja koneoppimismallin luomisesta mallin parametrien optimointiin ja oppimismallien tallentamiseen (PyTorch docs).

5 Tutkimuksen empiirinen osuus

Tässä kappaleessa käsitellään tutkimuksen empiiristä osuutta. Tutkimus toteutettiin empiirisenä vertailevana tutkimuksena. Vertailun kohteena olivat Unityn ML-agentin suorittama reitinhaku ja heuristiseen A*-algoritmiin perustuva reitinhaku. Kappaleessa 5.1 kuvaillaan tutkimusta yleisellä tasolla ja esitellään tutkimuksessa käytetyt työkalut. Kappaleessa 5.2 käydään läpi simulaatioympäristöt ja koneoppimiseen liittyvät konfiguraatiot. Lopuksi kappaleessa 5.4 käsitellään tulosten mittaamista ja käytettyjä suureita.

5.1 Tutkimuksen kuvaus

Tutkimuksen simulaatioalustana käytettiin Unityä, koska sen käyttö oli ennestään tuttua ja se soveltuu hyvin erilaisten ympäristöjen ja tilanteiden simuloimiseen. Tutkimusta varten Unity Hubissa luotiin valmis 2D-projekti ja projektiin lisättiin Unity Package Managerin kautta ML-Agents -paketti, jotta tarvittavat toiminnallisuudet saatiin käyttöön koneoppimista varten. Unityn valmiiden menetelmien avulla luotiin yksinkertaisia ruudukkoalueita, jonne sijoitettiin esteruutuja, kävelykelpoisia ruutuja ja alku- ja loppuruudut sekä joihinkin alueisiin liikkuvia esteitä. Reitinhaun vertailua varten luotiin A*-algoritmi. Agentille annettiin yhteensä neljä liikkumissuuntaa vaaka- ja pystysuunnassa, joten A*-algoritmin heuristiikaksi valittiin Manhattan-etäisyys. ML-agents -paketin avulla agentille annettiin *Behavior Parameters*, *Agent Script*, *Decision Requester* ja *Ray Perception Sensor 2D* -komponentit, jotka antavat tarvittavat toiminnallisuudet koneoppimista varten. Agentin opettamiseen käytetään syvään vahvistettuun oppimiseen perustuvaa SAC-algoritmia.

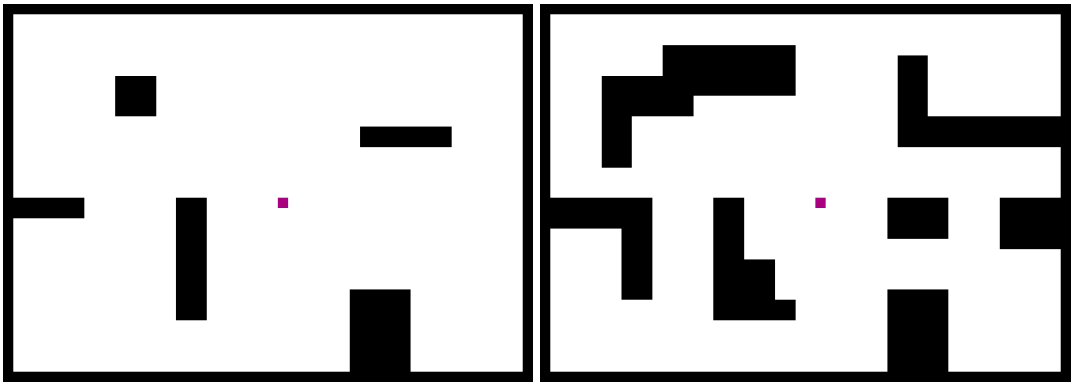
A*-algoritmia täytyi muokata tutkimukseen sopivaksi. Pelialueet voivat muuttua reaaliaikaisesti, jolloin perinteinen A*-algoritmi ei pysty reagoimaan muutoksiin välittömästi. Algoritmia muokattiin tutkimukseen siten, että esteen ilmestyessä laskelmoidulle reitille A*-algoritmi ajetaan uudestaan, mutta lähtöpisteeksi asetetaan agentin uusi sijainti. Uusi sijainti on agentin siihen asti kulkema matka, jolloin seuraava reitinhakuiteraatio vaatii lyhyemmän matkan ja on kevyempi muistinkäytön kannalta. Ratkaisu ei kuitenkaan ole järkevä suuremmissa pelialueissa, mutta soveltuu hyvin tämän tutkimuksen esimerkkeihin.

5.2 Tutkimusasetelma

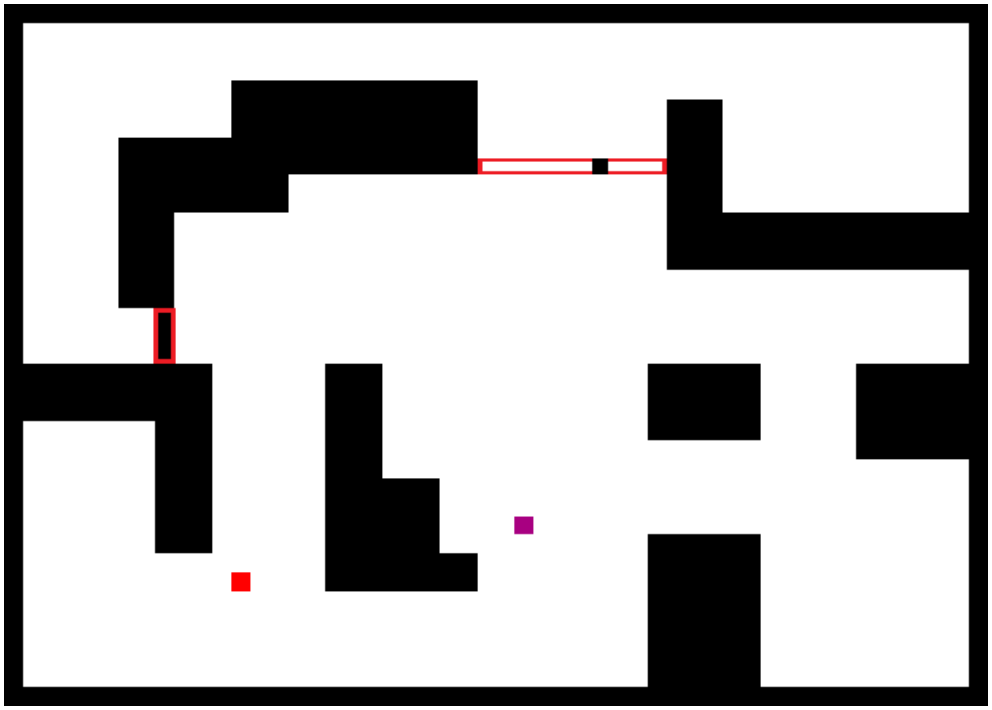
Tässä osiossa kerrotaan, miten Unityn simulaatioympäristö konfiguroitiin tutkimuksen eri osioita varten.

5.2.1 Oppimisympäristö

Tutkimuksessa toteutettiin kolme eri tason oppimisympäristöä: helppo, keskivaikea ja vaikea. Helppo ympäristö oli tehty tarkoituksella avaraksi ja alueella oli vain muutamia esteitä. Keskivaikeassa ympäristössä oli useampia esteitä ja seiniä ja alueelle pyrittiin tekemään sekä kapeita käytäviä että erillisiä huoneita. Kuvassa 7 helppo ja keskivaikea pelialue rinnakkain. Vaikea ympäristö on sama kuin keskivaikea, mutta ympäristöön tehtiin 1x1 suuruinen liikkuva este ja erillinen portti, joka aukeaa ja sulkeutuu kahden sekunnin välein. Liikkuva este etenee yhden sekunnin välein kuvan 8 punaisella merkityllä alueella ja toinen merkitty alue kuvaa portin sijaintia ja kokoa. Keskivaikean ympäristön koneoppimismallia käytettiin vaikean alueen testaamiseen, jotta voidaan testata agentin sopeutumista uusiin tilanteisiin. Koneoppimista varten kohderuutu valittiin Unityn valmiilla Random.Range()-funktiolla satunnaisesti vapaiden ruutujen seasta, mutta vaikean alueen testauksessa ruutuja valittiin esteiden läheisyydestä tai niiden toiselta puolelta.



Kuvio 7. Helppo ja keskivaikea pelialue. Violetti ruutu on agentti.



Kuvio 8. Vaikea pelialue, jossa on merkitty punaisella tasaisin väliajoin aukeava portti ja liikkuva este. Violetti ruutu on agentti ja punainen ruutu on kohderuutu.

5.2.2 Koneoppiminen

Agentilla voi olla käytössä sekä *Continuous Actions*- että *Discrete Actions*-taulukoita, jotka syötetään parametrina OnActionReceived()-funktioon. Continuous Actions -taulukon voivat olla mitä arvoja tahansa, mutta ne rajataan ennalta [-1,...,1] välille. Kyseiset arvot voidaan asettaa esimerkiksi agentin liikkumista varten, jolloin agentti pystyy oppimisen aikana kontrolloimaan arvoja. Toisena vaihtoehtona on Discrete Actions -taulukko, joka sisältää vain kokonaislukuja. Tässä tutkimuksessa agentilla on käytössä vain viiden elementin Discrete Actions -taulukko, jossa on arvot [0,1,2,3,4]. Agentin liikkuminen on toteutettu siten, että 0 on paikallaan pysyminen, 1 on liikkuminen vasemmalle, 2 on liikkuminen oikealle, 3 on liikkuminen alas ja 4 on liikkuminen ylös. Agentti pystyy liikkumaan kerralla yhden ruudun verran.

Unity ML-Agents mahdollistaa saman pelialueen rinnakkaisen opettamisen yksinkertaisella drag-and-drop -menetelmällä. Pelialueesta luodaan Prefab eli valmis malli, joita voidaan siirtää useampi kappale samalle näkymälle. Kaikki agentit pystyvät keräämään kokemuksia

samaan käytäntöön, kunhan jokaisella on sama Behavior Name -arvo. Opettaminen aloitetaan virtuaaliympäristössä komennolla *mlagents-learn <polku konfiguraatio-tiedostoon> –run-id=<opetustunnus>*. Opettaminen käynnistetään Unityssä Play-painikkeella. Yksi opetusjakso kestää 1000 askeletta tai kunnes agentti saapuu maaliin tai osuu esteeseen.

Agentin opettamisessa käytettiin samoja hyperparametreja eli konfiguraatio-tiedostoa kuin Unityn valmiissa koneoppimisesimerkissä nimeltä *FoodCollector*. FoodCollector-pelissä koneoppimisagentti pyrki keräämään pelialueelta vain tiettyjä ruokia ja oppi väistelemään kiellettyjä ruokia. Pelin ajatus oli hyvin lähellä tämän tutkimuksen esimerkkiä, joten tässä tutkimuksessa päädyttiin samoihin hyperparametreihin. Ainoastaan *max_steps* asetettiin miljoonaan, jolloin oppiminen pysäytettiin automaattisesti siihen. Myös *summary_freq*-arvoksi vaihdettiin 10000, jolloin opetusta pystyttiin seuraamaan tiheemmin aikavälein. Kokonaisuudessaan konfiguraatio-tiedoston sisältö löytyy liitteenä tutkimuksen lopusta luvusta 8 ja hyperparametrien kuvaukset Unity ML-agents GitHubista [1].

Opettamista voidaan seurata komentoriviltä, joka näyttää 10000 askeleen välein kuluneen ajan sekunneissa ja palkkioiden keskiarvon ja keskihajonnan. Tämän lisäksi tarkempaa kehittymistä voi seurata Tensorboardin avulla komennolla *tensorboard –logdir results –port 6006* ja menemällä selaimella sivulle localhost:6066. Tässä tutkimuksessa seurattiin agentin kumulatiivista palkkiota (engl. cumulative reward), entropian määrää mallissa eli kuinka satunnaisia agentin toiminnot ovat, kustannusfunktion häviön keskiarvoa (engl. policy loss) ja arvofunktion häviön keskiarvoa (engl. value loss). Näiden neljän mittarin perusteella voidaan arvioida agentin käytännön kehittymistä opetuksen aikana. Tässä tutkimuksessa kuvaajia ei tasoitettu ollenkaan.

Helpon tason ympäristössä opettamista tehtiin 500000 askeleen ajan, joka kesti 2712 sekuntia. Keskivaikea-tason ympäristössä opetettiin miljoonan askeleen ajan, joka kesti 5600 sekuntia. Molemmista alueista muodostui .nn-päätteinen neuroverkko-tiedosto, jonka pystyy asettamaan agentille käyttöön. Keskivaikea-tason luomaa neuroverkkoa sovellettiin vaikea-tason alueen reitinhakuun, jotta pystyttiin tarkastella agentin sopeutumista uusiin esteisiin.

1. https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md

5.3 Agentin havainnot ja palkkiot

Koneoppimisagentin oppimista ohjaavat sen tekemät havainnot ja sen saamat palkkiot. Havaintoja voi tallentaa suoraan skriptitiedostossa käsin *AddObservation()*-metodilla, jolle annetaan parametrina datatyyppi ja sen arvo. Tämän tutkimuksen reitinhakutehtävissä agentti havainnoi omaa ja kohteen sijaintia kaksiulotteisessa vektorimuodossa *(x,y)*, jolloin jokaisella askeleella neuroverkko saa tiedon näistä sijainneista. Tähän havaintoon käytetään Unityn omaa datatyyppiä *Vector2*, joka sisältää *x* ja *y* arvot *float*-tyyppisinä. Lisäksi agentti havainnoi sen etäisyyttä kohteeseen, jonka laskemiseen käytetään Unityn valmista Vector2.Distance()-funktiota. Agentille lisättiin myös kaksi *RayPerceptionSensor2D*-komponenttia, jotka lähettävät agentista osumatietoja keräämiä säteitä. Ensimmäinen sensori sisältää kaksitoista sädettä, jotka ovat pituudeltaan 15 yksikköä ja tunnistavat ainoastaan kohderuudun. Toinen sensori sisältää kahdeksan sädettä etäisyydeltään viisi yksikköä ja tunnistavat vain esteruutuja. Tunnistaminen määritellään erillisen *Detectable Tags*-kentän avulla, johon kirjoitetaan tunnistettavan kohteen tagi eli esteet ovat tagilla "Col"ja kohderuutu "End". Lisäksi sensorille annetaan *Ray Layer Mask*-valintalistaan tieto, mihin alueen kerroksiin säteet voivat fyysisesti osua ja mitkä kerrokset jätetään huomiotta. Tässä tutkimuksessa esteet on asetettu kerrokselle "Collision"ja kohderuutu kerrokselle "End". Näiden tietojen perusteella sensori lähettää automaattisesti säteiden osumadatan *float*-taulukkona neuroverkolle, jolloin sitä ei tarvitse itse toteuttaa skriptitiedostossa.

Agentin saamat palkkiot ovat vahvistusoppimisen ydin ja tässä tapauksessa agenttia palkitaan sen nopeuden ja yleisen suoriutumisen perusteella. Jokaisen agentin päätöksen yhteydessä annetaan pieni negatiivinen palkkio, jolloin agenttia ohjataan suorittamaan tehtävä niin nopeasti kuin mahdollista. Palkkio määritellään skriptitiedostossa *AddReward(-1/MaxStep)*, jossa *MaxStep* on yhden opetusjakson maksimipituus eli kuinka monta päätöstä agentti voi yhden jakson aikana tehdä. Tässä tutkimuksessa pelialueet ovat melko pieniä, joten maksimipituudeksi asetettiin 1000. Jos agentti ei saa suoritettua tehtävää ja maksimi askelmäärä saavutetaan, agentti saa palkkioksi yhteensä -1. Jos agentti osuu kohteeseen, sille annetaan palkkioksi +1 eli *AddReward(1)*. Jos agentti yrittää liikkua matkalla esteruutuun, sille annetaan palkkioksi -1 ja jakso opetusjakso pysäytetään. Tällä halutaan estää agentin turha liikkuminen kohti esteitä.

5.4 Tulosten mittaaminen

A*-algoritmin reitin selvittämistä ja koneoppimisagentin toimintaa on vaikea mitata esimerkiksi ajallisesti, koska agentti ei laske reittiä ennalta toisin kuin A*-algoritmi. Tässä tutkimuksessa tarkastellaan molempien vaihtoehtojen reitin optimaalisuutta eli tekeekö koneoppimisagentti turhia liikkeitä verrattuna A*-algoritmin optimaaliseen reittiin. Dynaamisissa alueissa tutkitaan koneoppimisagentin tekemiä ratkaisuja A*-algoritmin ratkaisuihin ja selitetään sanallisesti niiden eroja.

Tensorboardin kuvaajista kumulatiivinen palkkio kertoo agentin jaksojen palkkioiden keskiarvon, jonka tulisi kasvaa oppimisen yhteydessä. Entropia-kuvaaja esittää agentin toimintojen satunnaisuutta. Aluksi entropian kuuluisi olla suurta, mutta agentin oppiessa entropian tulisi tasaisesti laskea. SAC-algoritmin luonteeseen kuuluu entropian maksimointi, joten entropian kuuluisi hieman vaihdella agentin selvitellessä uusia tiloja. Käytännön virhefunktio kuvaa agentin käytännön muuttumista ja sen tulisi laskea onnistuneen oppimisen aikana. Arvofunktio kuvaa mallin kykyä ennustaa tilojen arvoja. Sen tulisi aluksi kasvaa oppimisen aikana ja lopuksi laskea kun palkkio tasoittuu.

6 Tulokset ja johtopäätökset

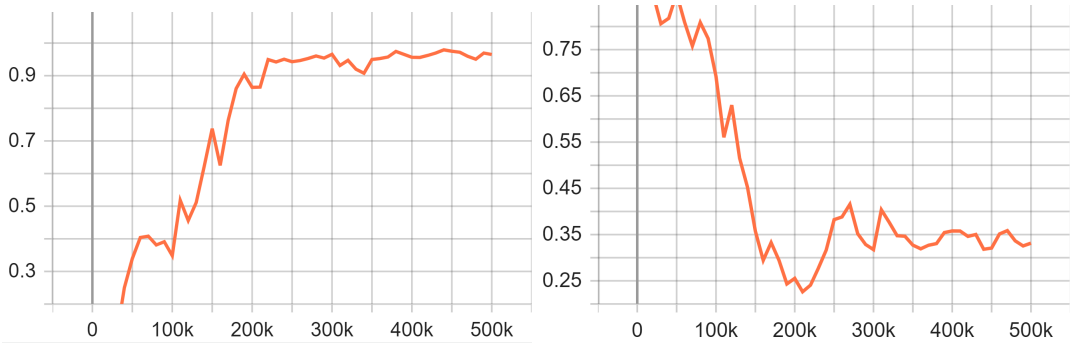
Tässä kappaleessa esitellään tutkimuksen tulokset. Kappaleessa 6.1 käsitellään helpon pelialueen koneoppimisen tuloksia. Kappaleessa 6.2 selitetään keskivaikean pelialueen koneoppimisen tulokset. Kappaleessa 6.3 kerrotaan keskivaikean neuroverkkomallin sopeutumisesta vaikean pelialueen dynaamisuuteen. Lopuksi kappaleessa 6.4 esitellään A*-algoritmin ja koneoppimisagentin vertailun tulokset.

6.1 Helppo pelialue

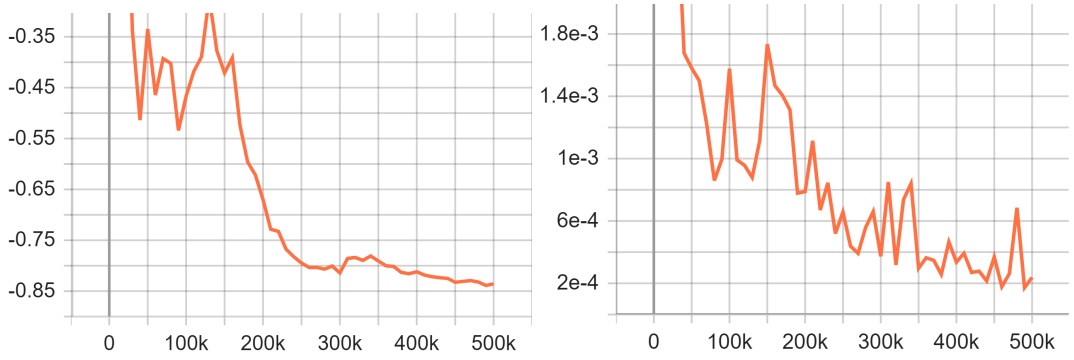
Koneoppimisagentin opettaminen aloitettiin helposta pelialueesta. Kuvasta 9 huomataan, että agentin kumulatiivinen keskiarvopalkkio nousi tasaisesti ylöspäin kohti maksimia eli yhtä, kunnes noin 220000 askeleen jälkeen kehitystä ei suuremmin tapahtunut. Aluksi agentti testasi satunnaisesti toimintoja ja sensoridataa huomatakseen eri toimintojen ja palkkioiden merkityksen. Seiniin osuminen aiheuttaa tässä tapauksessa ison menetyksen palkkiossa, joten agentti oppi nopeasti välttelemään osumia. Satunnaiset kontaktit maaliruutujen kanssa saivat agentin huomaamaan sijainti- ja sensoridatan merkityksen positiivisen palkkion saamiseen, joten agentti alkoi pyrkimään kohti lähimpiä maaliruutuja. Alueen kauimmaiset maaliruudut eivät kuitenkaan olleet helppoja tapauksia, joten niiden oppimiseen meni useita kymmeniä tuhansia opetusaskeleita. Lopulta agentti oppi kiertämään esteet ja saavutti jopa kauimmaiset maaliruudut vaivattomasti. Opettamista ei kuitenkaan kannattanut jatkaa liian pitkään, jotta liialliselta ylisovittamiselta vältyttäisiin. Opettaminen lopetettiin siksi ennenaikaisesti 500000 askeleen kohdalla.

Kuvasta 9 näkee myös, että agentin entropian määrä laski aluksi nopeasti. Onnistuneen oppimisen aikana entropian tulisi laskea samalla, kun palkkioiden määrä kasvaa. Helpon alueen tapauksessa tämä toteutui, joka viestii onnistuneesta oppimisesta. Kuvasta 10 nähdään agentin käytännön virhefunktion ja arvofunktion keskiarvohäviö. Käytännön virhefunktio laskee nopeasti 150000 askeleen kohdalla, kun agentti saa käsityksen oikeasta toiminnasta. Arvofunktion kuvaajassa esiintyy piikkimäisiä nousuja ja laskuja, mutta pitkällä aikavälillä sekin laskee tasaisesti. Kokonaisuudessaan agentti oppi maksimoimaan palkkion määrän ja onnis-

tui suurimmaksi osin ratkaisemaan annetut reitinhakuongelmat.



Kuvio 9. Vasemmalla helpon tason kumulatiivinen palkkio ja oikealla entropian kehitys 500000 askeleen aikana.

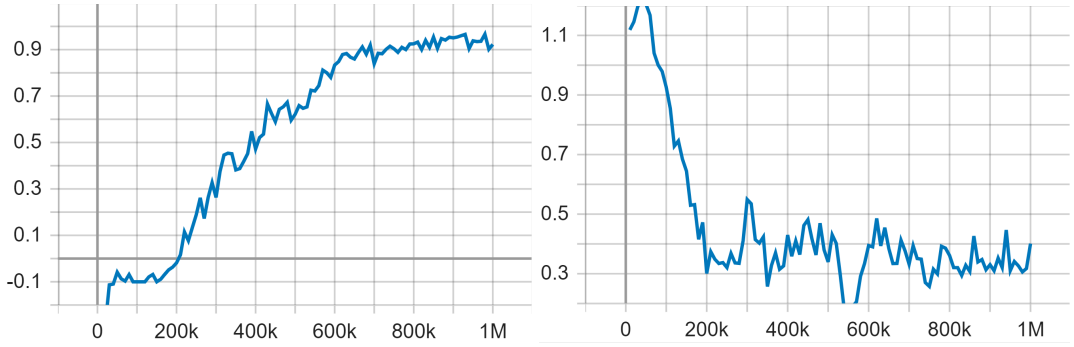


Kuvio 10. Vasemmalla helpon tason käytännön virhefunktion kehitys ja oikealla arvofunktion kehitys 500000 askeleen aikana.

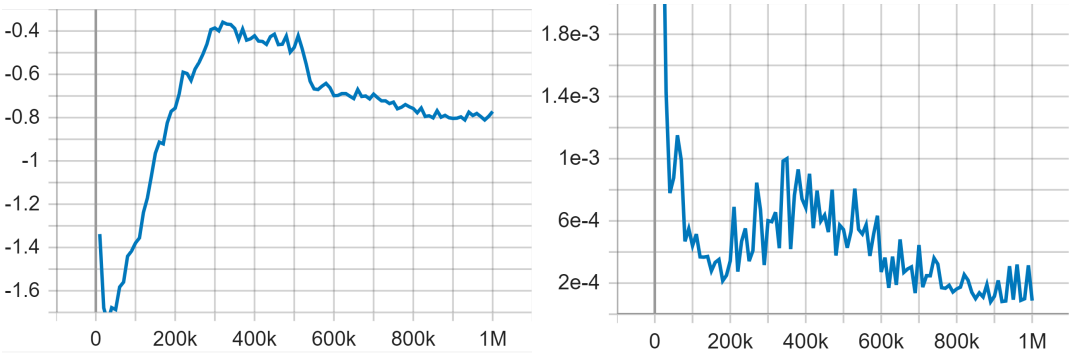
6.2 Keskivaikea pelialue

Keskivaikeassa alueessa koneoppimisagentti kehittyi selvästi hitaammin, joka näkyy myös kuvasta 11. Vasta 200000 askeleen kohdalla agentti alkoi saamaan keskiarvoltaan positiivista palkkiota. Oppiminen oli kuitenkin tehokasta 600000 askeleeseen asti, jonka jälkeen se hidastui merkittävästi saavuttaessaan noin 0.9 arvon. Keskivaikean alueen lisääntyneet esteet tekivät alun oppimisesta hidasta, koska agentti pyrki aluksi maksimoimaan palkkion määrän pysymällä kaukana esteistä. Agentti liikkui pitkään vain aloitusruudun ympärillä, mutta entropian kasvaessa se onnistui liikkumaan myös pidemmälle alkuruudusta. Entropian kuvaajasta huomaa, että entropian määrä laski nopeasti 200000 askeleeseen asti, jonka jälkeen se vaihteli loppuajan 0.3 ja 0.5 välillä.

Kuvan 12 virhefunktion keskiarvo kasvaa jostain syystä alussa 300000 askeleeseen asti, jonka jälkeen se laskee hitaasti. Alussa oppimisessa on selvästi ollut hankalaa ja agentti ei vältämättä ole pystynyt oppimaan tehokkaasti, kuten palkkion kehityksestä myös huomattiin. Arvofunktio sen sijaan laskee alussa jyrkästi ja nousee sen jälkeen hieman 200000 ja 400000 askeleen välillä ja lopuksi laskee hitaasti ja tasoittuu opetuksen loppuun mennessä.



Kuvio 11. Vasemmalla keskivaikean tason kumulatiivinen palkkio ja oikealla entropian kehitys miljoonan askeleen aikana.



Kuvio 12. Vasemmalla keskivaikean tason käytännön virhefunktion kehitys ja oikealla arvofunktion kehitys miljoonan askeleen aikana.

6.3 Vaikea pelialue

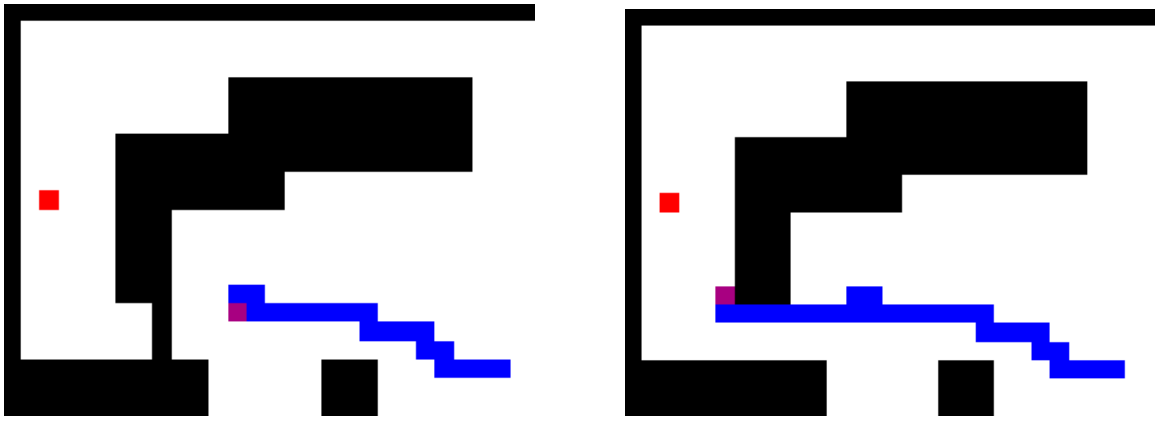
Keskivaikean tason opetuksen luomaa neuroverkko-tiedostoa käytettiin vaikean tason reitinhakuongelmiin, jolloin pystytään arvioimaan agentin sopeutumista dynaamisen alueen reitinhakuun. Agentti kohtasi reitillään uusia yllättäviä esteitä, joita ei opetusprosessissa esiintynyt. Reitinhakua testattiin aluksi portin toiselle puolelle. Agentti liikkui oppimansa mukaan kohti aukinaista porttia, mutta portin sulkeutuessa agentti pysähtyi muutaman ruudun

päähän odottamaan. Kun portti avautui uudestaan, agentti eteni kohderuutuun normaalisti. Kuvasta 13 nähdään agentin päätöksentekoa portin ollessa auki ja kiinni. Kuvasta myös huomataan, että agentti liikkui turhaan pystysuunnassa ennen porttia, joka kertoo epäoptimaalisesta reitistä. Optimaalisin vaihtoehto olisi siirtyä heti portin viereen odottamaan. Seuraavaksi testattiin agentin sopeutumista liikkuvaan esteeseen. Agentti varoi estettä monen ruudun päästä, joka johtui todennäköisesti liian pitkistä sensorisäteistä. Monissa tapauksissa agentti ei uskaltanut lähteä kiertämään estettä, vaan jäi liikkumaan esteen väärälle puolelle sivusuunnassa. Joissain tapauksissa agentti väisti esteen ja eteni kohderuuduun.

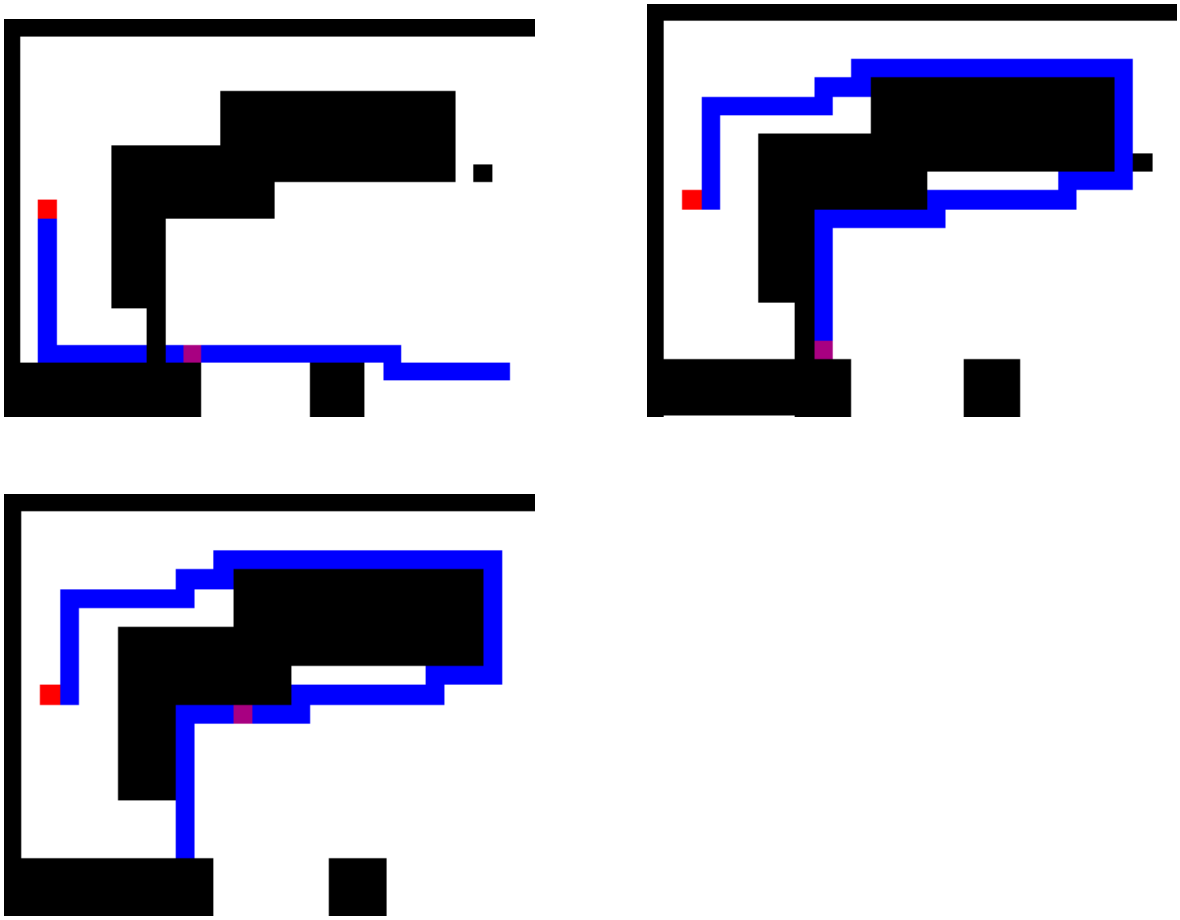
6.4 A*-algoritmin ja koneoppimisagentin vertailu

A*-algoritmi suoriutuu staattisista pelialueista moitteettomasti, joten helpon ja keskivaikean pelialueen vertailua oli turha toteuttaa. Vaikean pelialueen reitinhakutehtävissä hyödynnettiin portin olemassaoloa, joten kohderuutu asetettiin portin toiselle puolelle. Luvussa 6.3 selitettiin agentin toimintaa portin ympärillä. Kuvasta 14 nähdään, että A*-algoritmi päätyi sen sijaan valitsemaan aluksi portin läpi kulkevan reitin, mutta kohdatessaan sulkeutuneen portin se vaihtoi kokonaan reittiään. Kiertoreitin kulkeminen vei tässä tapauksessa selvästi pidemmän ajan, jolloin koneoppimisagentti päätyi suorittamaan reitinhakutehtävän nopeammin.

Liikkuvan esteen suhteen A*-algoritmi toimi selvästi paremmin kuin koneoppimisagentti. Tutkimuksessa toteutettu A* variaatio onnistui heti vaihtamaan reittiä kohdatessaan esteen, jolloin se pääsi kiertämään sen välittömästi. Ääritapauksissa A* valitsi kiertoreitin samalta puolelta johon este oli liikkumassa, jolloin este liikkui heti uudestaan reitille ja A* joutui laskemaan reitin jälleen uudestaan. Agentti sen sijaan oli liian varovainen liikkuvan esteen suhteen, kuten luvussa 6.3 selitettiin. Pahimmillaan agentti ei pystynyt suorittamaan reitinhakutehtävää ollenkaan.



Kuvio 13. Agentti osasi odottaa portin avautumista ja käytti portin läpi kulkevaa reittiä kohteeseen. Siniset ruudut kuvaavat agentin kulkemaa reittiä.



Kuvio 14. A*-algoritmin valitsema reitti portin ollessa auki ja kiinni. Siniset ruudut kuvaavat A*-algoritmin laskemaa reittiä.

7 Yhteenveto

Lähteet

Abd Algfoor, Zeyad, Mohd Shahrizal Sunar ja Hoshang Kolivand. 2015. “A comprehensive study on pathfinding techniques for robotics and video games”. *International Journal of Computer Games Technology* 2015.

Abdi, Hervé, ja Lynne J Williams. 2010. “Principal component analysis”. *Wiley interdisciplinary reviews: computational statistics* 2 (4): 433–459.

Arulkumaran, Kai, Marc Peter Deisenroth, Miles Brundage ja Anil Anthony Bharath. 2017. “A brief survey of deep reinforcement learning”. *arXiv preprint arXiv:1708.05866.*

Botea, Adi, Bruno Bouzy, Michael Buro, Christian Bauckhage ja Dana Nau. 2013. “Pathfinding in games”. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Cervantes, Jair, Farid Garcia-Lamont, Lisbeth Rodriguez-Mazahua ja Asdrubal Lopez. 2020. “A comprehensive survey on support vector machine classification: Applications, challenges and trends”. *Neurocomputing* 408:189–215.

Cui, Xiao, ja Hao Shi. 2011. “A*-based pathfinding in modern computer games”. *International Journal of Computer Science and Network Security* 11 (1): 125–130.

Das, Kajaree, ja Rabi Narayan Behera. 2017. “A survey on machine learning: concept, algorithms and applications”. *International Journal of Innovative Research in Computer and Communication Engineering* 5 (2): 1301–1309.

Deisenroth, Marc Peter, Gerhard Neumann, Jan Peters ym. 2013. “A survey on policy search for robotics”. *Foundations and Trends® in Robotics* 2 (1–2): 1–142.

Demyen, Douglas, ja Michael Buro. 2006. “Efficient triangulation-based pathfinding”. Teoksessa *Aaai,* 6:942–947.

Dijkstra, Edsger W, ym. 1959. “A note on two problems in connexion with graphs”. *Numerische mathematik* 1 (1): 269–271.

Duchoň, František, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico ja Ladislav Jurišica. 2014. “Path planning with modified a star algorithm for a mobile robot”. *Procedia Engineering* 96:59–69.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel ja Sergey Levine. 2018. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. Teoksessa *International conference on machine learning,* 1861–1870. PMLR.

Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel ym. 2018. “Soft actor-critic algorithms and applications”. *arXiv preprint arXiv:1812.05905.*

Hart, Peter E, Nils J Nilsson ja Bertram Raphael. 1968. “A formal basis for the heuristic determination of minimum cost paths”. *IEEE transactions on Systems Science and Cybernetics* 4 (2): 100–107.

Jordan, Michael I, ja Tom M Mitchell. 2015. “Machine learning: Trends, perspectives, and prospects”. *Science* 349 (6245): 255–260.

Juliani, Arthur, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar ym. 2018. “Unity: A general platform for intelligent agents”. *arXiv preprint arXiv:1809.02627.*

Karur, Karthik, Nitin Sharma, Chinmay Dharmatti ja Joshua E Siegel. 2021. “A survey of path planning algorithms for mobile robots”. *Vehicles* 3 (3): 448–468.

Koenig, Sven, ja Maxim Likhachev. 2005. “Fast replanning for navigation in unknown terrain”. *IEEE Transactions on Robotics* 21 (3): 354–363.

Lawande, Sharmad Rajnish, Graceline Jasmine, Jani Anbarasi ja Lila Iznita Izhar. 2022. “A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games”. *Applied Sciences* 12 (11): 5499.

LeCun, Yann, Yoshua Bengio ja Geoffrey Hinton. 2015. “Deep learning”. *nature* 521 (7553): 436–444.

Lei, Xiaoyun, Zhian Zhang ja Peifang Dong. 2018. “Dynamic path planning of unknown environment based on deep reinforcement learning”. *Journal of Robotics* 2018.

Li, Yuxi. 2018. “Deep reinforcement learning”. *arXiv preprint arXiv:1810.06339.*

Nasteski, Vladimir. 2017. “An overview of the supervised machine learning methods”. *Horizons. b* 4:51–62.

Nielsen, Michael A. 2015. *Neural networks and deep learning.* Nide 25. Determination press San Francisco, CA, USA.

Osisanwo, FY, JET Akinsola, O Awodele, JO Hinmikaiye, O Olakanmi ja J Akinjobi. 2017. “Supervised machine learning algorithms: classification and comparison”. *International Journal of Computer Trends and Technology (IJCTT)* 48 (3): 128–138.

Panov, Aleksandr I, Konstantin S Yakovlev ja Roman Suvorov. 2018. “Grid path planning with deep reinforcement learning: Preliminary results”. *Procedia computer science* 123:347–353.

Pouyanfar, Samira, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen ja Sundaraja S Iyengar. 2018. “A survey on deep learning: Algorithms, techniques, and applications”. *ACM Computing Surveys (CSUR)* 51 (5): 1–36.

Rahmani, Vahid, ja Nuria Pelechano. 2022. “Towards a human-like approach to path finding”. *Computers & Graphics* 102:164–174.

Rish, Irina, ym. 2001. “An empirical study of the naive Bayes classifier”. Teoksessa *IJCAI 2001 workshop on empirical methods in artificial intelligence,* 3:41–46. 22.

Sazaki, Yoppy, Anggina Primanita ja Muhammad Syahroyni. 2017. “Pathfinding car racing game using dynamic pathfinding algorithm and algorithm A”. Teoksessa *2017 3rd International Conference on Wireless and Telematics (ICWT),* 164–169. IEEE.

Sharon, Guni, Roni Stern, Ariel Felner ja Nathan R Sturtevant. 2015. “Conflict-based search for optimal multi-agent pathfinding”. *Artificial Intelligence* 219:40–66.

Stentz, Anthony. 1994. “Optimal and efficient path planning for partially-known environments”. Teoksessa *Intelligent unmanned ground vehicles,* 203–220. Springer.

Stentz, Anthony, ym. 1995. “The focussed dˆ* algorithm for real-time replanning”. Teoksessa *IJCAI,* 95:1652–1659.

Stern, Roni, Nathan R Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Satish Kumar ym. 2019. “Multi-agent path-finding: Definitions, variants, and benchmarks”. Teoksessa *Twelfth Annual Symposium on Combinatorial Search.*

“Unity”. 2022. Viitattu 4. huhtikuuta 2022. https://unity.com/.

“Unity Docs”. 2022. Viitattu 7. syyskuuta 2022. https://docs.unity3d.com/2023.1/Documentation/Manual/nav-InnerWorkings.html.

8 Liitteet

```
behaviors:
  GraduTest:
    trainer_type: sac
    hyperparameters:
      learning_rate: 0.0003
      learning_rate_schedule: constant
      batch_size: 256
      buffer_size: 2048
      buffer_init_steps: 0
      tau: 0.005
      steps_per_update: 10.0
      save_replay_buffer: false
      init_entcoef: 0.05
      reward_signal_steps_per_update: 10.0
    network_settings:
      normalize: false
      hidden_units: 256
      num_layers: 1
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 1000000
    time_horizon: 64
    summary_freq: 10000
    threaded: false
```

- ehkä agentscript tiedosto? miten saada pienemmällä fontilla?