Tomi Tapio

Koneopetetun agentin hyödyntäminen vaikeustason balansointiin bullet hell -peligenressä

Tietotekniikan pro gradu -tutkielma

28. syyskuuta 2021

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Tomi Tapio

Yhteystiedot: tapio.tomi@gmail.com

Ohjaaja: Jonne Itkonen, Paavo Nieminen

Työn nimi: Koneopetetun agentin hyödyntäminen vaikeustason balansointiin bullet hell -peligenressä

Title in English: Using a machine learning agent for difficulty balancing in the bullet hell video game genre

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 84+0

Tiivistelmä:

Automatisoitu pelitestaus on vielä vähäistä peliteollisuudessa, ja koneoppimisalgoritmien nopea kehitys mahdollistaa koneopetetun agentin käyttämisen yhä paremmin pelimoottoreiden sisällä. Tutkimuksessa pelitestauksen osa-alueista keskitytään vaikeustason balansointiin. Tavoitteena oli tutkia, voiko agenttia käyttää bullet hell -genren pelin kenttien vaikeustasojen havainnointiin, jolloin tarve ulkopuolisille testaajille vähenisi. Tutkimusta varten toteutettiin bullet hell -genren prototyyppi, jonka kentissä agenttia simuloitiin. Opetetun agentin suoriutumista verrattiin käyttäjädataan, joka kerättiin käyttäjätesteillä. Tulokset osoittivat, että agentin opettaminen tällaisessa haasteellisessa genressä vaatii hyvin paljon opetusaikaa. Lisäksi kenttäkohtainen ylisovittaminen osoittautui suureksi haasteeksi. Tutkimuksen perusteella esitetään havaintoja agentin opettamisen tehostamiseksi, joiden katsotaan olevan hyödyksi pelikehittäjille, jotka ovat kehittämässä vaikeustasoltaan haastavaa peliä ja aikovat hyödyntää koneoppimista kehittämisessä.

Avainsanat: pelitestaus, koneoppiminen, bullet hell, vahvistusoppiminen, imitaatio-oppiminen

Abstract:

Automated game testing is still limited in the gaming industry and the rapid development of the machine learning algorithms makes it increasingly possible to use a machine learning agent inside the game engines. In this study, the core aspect of the game testing is a difficulty balancing. This thesis examines whether the agent could be used to detect the difficulty of the levels in the bullet hell genre, thereby reducing the need for the external testers. In this study, a machine learned agent was simulated in a prototype implemented for the study. The performance of the agent is compared to the user data collected through the user tests. The results showed that teaching the agent in a such a challenging genre requires a lot of teaching time. In addition, the level-specific overfitting proved to be a major challenge. The study presents the findings related to enhancing agent learning that are considered to be beneficial to a game developers who are developing a challenging game and intend to leverage a machine learning in the development.

Keywords: game testing, machine learning, bullet hell, reinforcement learning, imitation learning

Matemaattiset merkinnät

$\arg\max_a f(a)$	a:n arvo, kun funktio $f(a)$ on antaa suurimman tuloksen.
$\mathbb{R}$	Reaalilukujen joukko.
γ	Vähennyskerroin.
Φ	Aktivaatiofunktio.
s, s'	Agentin tila.
r	Agentille annettava palkinto.
a	Agentin toiminto.
t	Aika-askel.
s_t	Agentin tila aika-askeleella t.
a_t	Agentin toiminto aika-askeleella t.
r_t	Agentille annettava palkinto aika-askeleella t.
$\mathcal{S}$	Joukko kaikista tiloista.
$\mathcal{A}$	Joukko kaikista mahdollisista toiminnoista.
$\mathcal{A}(s)$	Joukko kaikista mahdollisista toiminnoista tilassa s.
$\mathcal{R}$	Joukko kaikista mahdollisista palkinnoista.
$\subset$	Alijoukko, kuten esimerkiksi $\mathcal{R} \subset \mathbb{R}$.
$\in$	Määritellyn joukon elementti, kuten esimerkiksi $s \in \mathcal{S}$ ja $r \in \mathcal{R}$.
$r(s,a)$	Odotettu välitön palkinto tilassa s toiminnolla a.
$r(s,a,s')$	Odotettu välitön palkinto siirtymällä $s \rightarrow s'$ toiminnolla a.
π	Agentin menettelytapa eli päätöksenteon sääntö.
$\pi(s)$	Menettelytapa tilassa s.
$v^{\pi}(s)$	Tilan s arvo menettelytavalla π.
$v^{*}(s)$	Tilan s arvo optimaalisella menettelytavalla.
θ	Menettelytavan vektoriparametri.
$\pi(a\|s,\theta)$	Toiminnon todennäköisyys tilassa s annetulla parametrilla θ.
π_{θ}	Parametria θ vastaava menettelytapa.
τ	Trajektori. Sisältää usein historiatiedon agentin toiminnoista, tiloista ja saaduista palkinnoista.
$\tau_E \sim \pi_{\theta_i}$	Trajektori τ_E on ote jakaumasta π_{θ_i}.

Kuviot

Kuvio 1. Flow-kanava 5
Kuvio 2. Perseptronin yksinkertaistettu kuvaus 9
Kuvio 3. Painokertoimet matriisimuodossa (mukaillen Yasoubi, Hojabr ja Modarressi 2016) 10
Kuvio 4. Aktivaatiofunktion kuvaus 11
Kuvio 5. Erityyppisiä aktivaatiofunktioita 12
Kuvio 6. Swish-aktivaatiofunktio 13
Kuvio 7. Perseptroni 14
Kuvio 8. Oppimisnopeuden periaate gradienttimenetelmässä 16
Kuvio 9. Monikerroksisen neuroverkon rakenne 17
Kuvio 10. Ketjusäännön periaate 19
Kuvio 11. Yli- ja alisovittaminen 23
Kuvio 12. Markovin päätösprosessi 26
Kuvio 13. Agentin ja ympäristön vuorovaikutus 27
Kuvio 14. Häviöfunktion L^{CLIP} todennäköisyyssuhteen leikkaus 33
Kuvio 15. Sisäisen uteliaisuusmallin (ICM) toimintalogiikka 35
Kuvio 16. ICM yhdistettynä ulkoisiin palkintoihin 36
Kuvio 17. Generatiivisen kilpailevan verkon toimintaperiaate 37
Kuvio 18. Vahvistusoppimisen ja käänteisen vahvistusoppimisen toimintalogiikan eroavaisuus 39
Kuvio 19. GAIL-menetelmän toimintaperiaate 41
Kuvio 20. Syvän vahvistusoppimisen perusperiaate 43
Kuvio 21. Kuvankaappaus toteutetusta prototyypistä 53
Kuvio 22. Käyttäjätestien läpäisyprosenttien hajonta 65
Kuvio 23. Koetulokset 66
Kuvio 24. Kahden viimeisen kentän vihollisaaltokohtaiset tulokset 67
Kuvio 25. Agentin palkintomäärän kehitys opetuksen aikana 68

Taulukot

Taulukko 1. Prototyypin vihollistyyppien ominaisuudet 54
Taulukko 2. Prototyypin kentät 55
Taulukko 3. Agentin toiminnot 56
Taulukko 4. Agentille havainnoitavaksi annetut muuttujat 58
Taulukko 5. Ote lopullisista opettamiseen käytetyistä asetuksista ja hyperparametriarvoista 60
Taulukko 6. Palkintosignaalien konfiguraatiot 61

Sisällys

1 JOHDANTO ... 1

2 PELIEN HAASTEET JA VAIKEUSASTEIDEN MÄÄRITTELY ... 3
2.1 Pelin haasteet ... 3
2.2 Optimaalinen haaste ... 4
2.3 Vaikeuden skaalaus ... 6

3 KONEOPPIMINEN ... 8
3.1 Keinotekoinen neuroverkko ... 9
3.1.1 Aktivaatiofunktiot ... 11
3.1.2 Oppiminen neuroverkossa ... 13
3.1.3 Vastavirta-algoritmi ... 17
3.1.4 Yli- ja alisovittaminen ... 22
3.2 Vahvistusoppiminen ... 23
3.2.1 Markovin päätösprosessit ... 25
3.2.2 Arvofunktio ... 28
3.3 Proksimaalisen menettelytavan gradienttimetodi ... 30
3.4 Uteliaisuusperusteinen tutkiminen ... 34
3.5 Generatiivinen kilpaileva verkko ... 37
3.6 Imitaatio-oppiminen ... 38
3.6.1 Käänteinen vahvistusoppiminen ... 38
3.6.2 Generatiivinen kilpaileva imitaatio-oppiminen ... 40
3.7 Syväoppiminen ... 43
3.8 Koneoppiminen Unity-pelimoottorissa ... 44
3.9 Aihealueen tutkimukset ... 46

4 TUTKIMUS ... 50
4.1 Pelisuunnittelu ... 51
4.1.1 Prototyypin ydinmekaniikat ... 51
4.1.2 Prototyypin vihollistyypit ... 53
4.1.3 Prototyypin kentät ... 54
4.2 Agentin opettamiseen liittyvä kehitystyö ... 55
4.2.1 Agentin toiminnot ... 56
4.2.2 Agentille määritetyt havainnot ... 57
4.2.3 Ulkoiset palkinnot ... 58
4.2.4 Hyperparametrit ja palkintosignaalit ... 59
4.2.5 Agentin opetusympäristö ... 61
4.3 Tulokset ... 64
4.3.1 Kokonaisen prototyypin opettaminen ... 64
4.3.2 Kenttäkohtainen opettaminen ... 67
4.4 Pohdinta ... 68
4.4.1 Tutkimuskysymys 1 ... 69
4.4.2 Tutkimuskysymys 2 ... 69

4.4.3 Tutkimuskysymys 3 70
5 YHTEENVETO 72
LÄHTEET 73

1 Johdanto

Tutkimuksessa selvitetään, kuinka hyvin imitaatio-oppimisen ja vahvistusoppimisen avulla kehitettyä itsenäistä agenttia voidaan käyttää pelikehityksessä vaikeustason määrittämiseen kenttäsuunnittelun ja pelitestauksen tukena. Opetetun agentin toimintaa testataan nopeatempoisessa ja haastavassa ympäristössä. Tarkasteltavana genrenä toimii bullet hell -genre, joka valikoitui genreksi, koska haastavuus on oleellinen ominaispiirre genren peleissä. Tutkimuksellisessa osiossa toteutetaan prototyyppi, jossa on genren ydinpelimekaniikat toteutettuna. Pelimoottorina toimii Unity ja koneoppimisalgoritmeja käytetään ML-Agents työkalun avulla, joka on avoimen lähdekoodin projekti.

Pelitestaus on jo pitkään tunnustettu haastavaksi tehtäväksi, joka perustuu peliteollisuudessa pääasiassa manuaaliseen pelitestaukseen. Systemaattiset ja automatisoidut testiratkaisut ovat edelleen suurelta osin vähäisesti käytössä (Zheng ym. 2019; Aleem, Capretz ja Ahmed 2016). Manuaalinen ad hoc -testaus on kallista ja tehotonta bugien löytämiseen laajemmissa peleissä. Tämän tuloksena moni bugi löytyy vasta julkaisun jälkeen pelaajien toimesta, jolloin niiden korjaaminen on pelinkehittäjille työläämpää ja pelifirmalle kalliimpaa. (Zheng ym. 2019)

Vaikeustasossa ilmenevät epäsuhdat ovat myös yksi peleissä esiintyvien bugien osa-alue (Zheng ym. 2019). Kehittäjälle itselleen vaikeustason arvioiminen on hyvin haastavaa, sillä kehittäjä oppii pelaamaan peliä kehityksen edetessä yhä paremmin. Pelitestaus ja kenttäsuunnittelu ovat aikaa vieviä prosesseja, ja koneoppimisen hyödyntämisellä pelitestaukseen liittyen on jo saatu merkittäviä kehitysaikaa lyhentäviä tutkimustuloksia (Gudmundsson ym. 2018). Lisäksi koneoppimisen avulla on pystytty esimerkiksi todentamaan pelin ne kohdat, joissa pelaajat luovuttavat pelin pelaamisen todennäköisimmin (Roohi ym. 2019).

Ideaalitilanne voisi olla, että uutta kehitettävää kenttää pystyisi nopeasti simuloimaan opetetulla, stabiilin taitotason agentilla ja havainnoida simuloinnin lopputuloksena kentän pelattavuus nopeasti, ilman välttämätöntä tarvetta ulkopuolisille pelitestaajille. Käyttämällä tietyn taitotason agenttia voitaisiin pelitestauksessa saada myös luotettavampia tuloksia verrattuna siihen, että pelitestaajina käytetään ihmisiä. Ihmisten osaaminen kehittyy pelin testaamisen

aikana, kun puolestaan agentin taitotason voidaan olettaa pysyvän stabiilimpana, varsinkin pidemmän aikavälin tarkastelujaksolla (Aponte, Levieux ja Natkin 2009).

Tässä pro gradu -tutkielmassa etsitään keinoja koneopetetun agentin käyttämiselle pelikehityksessä vaikeustason balansointiin. Tutkielmassa etsitään vastauksia seuraaviin tutkimuskysymyksiin:

- 1. Pystytäänkö koneopetetun agentin avulla päättelemään bullet hell -genren kentän suhteellinen vaikeustaso luotettavasti?
- 2. Onko opetetun agentin suoriutuminen bullet hell -genren pelaajien tasolla?
- 3. Miten ja millä menetelmillä koneopetettua agenttia voidaan hyödyntää bullet hell -genren pelitestauksessa, jotta sen hyödyntäminen on kehitysajan kannalta käytännöllistä?

Luotettavuudella tarkoitetaan tässä yhteydessä sitä, ovatko agentin pelitulokset kenttien vaikeustasojen mukaisia. Esimerkiksi vaikeimmassa kentässä agentin tulisi onnistua kentän läpäisyssä muita kenttiä harvemmin. Näin ollen kenttien vaikeustasot tulee olla myös määriteltävissä, jotta tutkimustuloksia voidaan pitää luotettavina. Luotettavuus mitataan vertaamalla agentin suoriutumista käyttäjädataan. Lisäksi tutkielman lopussa tuodaan esille konkreettisia huomioitavia asioita agentin opettamisen ja pelitestaukseen hyödyntämisen kannalta.

Luvussa 2 esitellään vaikeusasteen määritelmä pelisuunnittelun kannalta ja käydään läpi tämän tutkielman aihealueeseen liittyviä muita tutkimuksia. Luvussa 3 esitellään tämän tutkielman kannalta oleellisia koneoppimisen teorioita. Luvussa 4 käydään läpi agentin opettamiseen liittyvän kehitystyön vaiheet, tutkimustulokset sekä muut havainnot liittyen agentin hyödyntämiseen pelitestauksessa. Luvussa myös perustellaan tutkimukselliset ja pelisuunnitelmalliset ratkaisut suoritettavan kokeen suhteen. Lopuksi tutkimustulosten johtopäätökset esitellään luvussa 5.

2 Pelien haasteet ja vaikeusasteiden määrittely

Jotta agentin toimintaa voidaan jollakin tavalla arvioida suhteessa oletettuun vaikeustasoon, tulee vaikeustason määritelmä kuvailla tarkemmin. Tässä luvussa käydään läpi sitä, miten pelien vaikeusastetta voidaan määritellä ja mitata, jotta myös agentin toimintaa voidaan arvioida tarkemmin.

2.1 Pelin haasteet

On olemassa monenlaisia määritelmiä siitä, mitä pelit ovat, mutta suurin osa niistä on samaa mieltä siitä, että säännöt ovat välttämätön ominaisuus peleissä. Salen ja Zimmerman (2003) määrittelevät pelin seuraavanlaisesti: "Peli on systeemi, jossa pelaajat sitoutuvat keinotekoisiin konflikteihin, joissa on määritelty säännöt, joista syntyy mitattava lopputulos". Juul (2003) määrittelee pelin puolestaan näin: "Peli on sääntöpohjainen formaali systeemi, jolla on muuttuvia ja mitattavissa olevia lopputuloksia, jotka ovat eriarvoisia. Pelaaja kykenee ponnisteluillaan vaikuttamaan lopputulokseen, pelaaja kokee olevansa liittynyt lopputulokseen, ja toiminnan seuraukset ovat toissijaisia ja neuvoteltavissa olevia". Kompleksiset systeemit yleensä sisältävät useita vuorovaikuttavia osia. Yksittäisten osien käyttäytyminen voi olla helposti ymmärrettävää, koska niiden säännöt voivat olla selkeitä. Kun kaikki osat yhdistetään, käyttäytyminen voi olla yllättävää ja vaikeasti ennustettavaa. (Adams ja Dormans 2012, s. 1)

Videopelisuunnittelijat käyttävät mieluummin sääntöjen sijasta termiä *pelimekaniikka*, sillä säännöt ymmärretään yleisesti tulostettuina ohjeina, joista pelaaja on tietoinen, kun taas pelimekaniikka on piilossa pelaajalta. Videopelien pelaajien ei tarvitse tietää, mitkä ovat pelin säännöt aloittaessaan pelaamisen; toisin kuin esimerkiksi lauta- ja korttipeleissä, videopeli opettaa säännöt pelin aikana. Termiä *ydinmekaniikka* käytetään usein indikoimaan pelimekaniikkoja, jotka ovat kaikkein vaikutusvaltaisimpia, vaikuttaen moniin pelin eri aspekteihin. (Adams ja Dormans 2012, s. 3–4)

Raph Koster (2004, s. 34–38) kertoo, että pelit ovat olemukseltaan pulmia, joita ratkomme. Yritämme oppia taustalla olevat kaavat ymmärtääksemme ne täysin, jolloin voimme hyödyn-

tää niitä tarvittaessa myöhemmin. Pelit ovat harjoitusta aivoillemme. Pelit, jotka epäonnistuvat tässä harjoittamisessa, muuttuvat tylsiksi. Samalla kun opimme lisää kaavoja, tarvitaan myös uutuuksia, jotta peli tuntuu koukuttavalta. Melkein kaikki pelit lopulta kaatuvat tähän, sillä ne ovat rajoittuneita formaaleja systeemejä. Kun jatkat pelaamista, tulet ennemmin tai myöhemmin ymmärtämään sen kaavat ja näin ollen tylsyyden saavuttaminen on usein väistämätöntä. Koster (2004, s. 40–42) esittääkin, että pelien tuottama hauskuus on sitä, kun opimme uusia kaavoja. Mikäli tällaista uuden oppimista ei ole, kohtaamme väistämättä tylsyyden.

Pelien tuottamaa haastetta voitaisiin siis yleisemmin sanottuna pitää uusien kaavojen oppimisen haasteena. Uudet haasteet usein myös pohjautuvat jo opittujen kaavojen hyödyntämiseen jollakin uudella tavalla, jotta peli edetessään tarjoaa jotakin uutta ja kiinnostavaa. Vaikeutta pelisuunnittelun yhteydessä voidaan kuvata yleisesti niin, että vaikeus koostuu useasta haasteesta, jonka pelaaja kohtaa pelissä (Aponte, Levieux ja Natkin 2009).

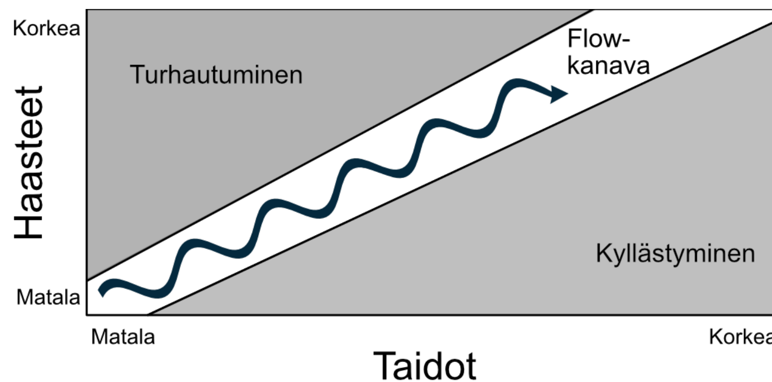
2.2 Optimaalinen haaste

Tunnettu psykologi Mihaly Csikszentmihalyi (1991) esitteli Flow-teorian, jonka taustalla oleva ajatus on se, että optimaalinen vaikeusasteen nousu henkilön suorittamalle aktiviteetille voi tuottaa henkilölle flow-tilaksi kutsutun ilmiön, jossa henkilö uppoutuu harjoitteeseen niin syvästi, että unohtaa ympärillään olevan fyysisen maailman. Flow on keskittymisen ja osallistumisen euforinen tila, jota usein väitetään yhdeksi miellyttävimmäksi ja arvokkaimmaksi kokemukseksi, jonka voi kokea. Flow-tilassa henkilö tulee niin kiintyneeksi aktiviteetin suorittamiseen, että suorittamiseen liittymättömät ajatukset ja käsitykset eivät pääse tietoisuuteen. (Csikszentmihalyi 1991)

Tärkeä edellytys flow-tilalle on se, että henkilön taito ja aktiviteetin tuoma haaste kohtaavat. Mikäli haasteet ovat suurempia kuin henkilön taidot, johtaa aktiviteetin suorittaminen henkilön turhautumiseen. Mikäli puolestaan henkilön taidot ovat suurempia kuin haaste, aktiviteetin suorittaminen johtaa henkilön välinpitämättömyyteen suorittamista kohtaan. Flow-tunteen tuottava aktiviteetti on yleensä niin antoisa, että henkilö haluaa suorittaa aktiviteetin itsensä vuoksi, riippumatta siitä, mitkä ovat suorittamisen seuraukset eli riippumatta siitä,

mitä henkilö saavuttaa aktiviteetin suorittamisen johdosta. (Csikszentmihalyi 1991)

Flow-teoria on myös keskeisessä roolissa pelisuunnittelussa ja flow-tilaa pidetään tavoitteena, joka pelaajalle halutaan pelin avulla tuottaa (Schell 2020, s. 145). Flow-aktiviteettien tulee pysyä kyllästymisen ja turhautumisen välisellä kapealla kaistalla. Tätä kutsutaan flow-kanavaksi, joka on kuvattu kuviossa 1. Flow-kanava kertoo, että vaikeustaso voi hetkellisesti myös helpottua, mutta pidemmällä tähtäimellä turhautumisen ja kyllästymisen välttämiseksi on tärkeää vaikeuden nouseminen kuvion mukaisesti.



Kuvio 1. Flow-kanava (mukaillen Schell 2020, s. 147)

Schell (2020, s. 145) esittää ydinkomponentit, jotka pelistä tulee löytyä, jotta pelaajalle on mahdollista luoda flow-tilan saavuttava aktiviteetti:

- **Selkeät tavoitteet**: Tavoitteiden ollessa selkeitä, voimme pysyä helpommin keskittyneenä tehtävän suorittamiseen. Tavoitteiden ollessa epäselvät, emme ole tietoisia siitä, ovatko suoritetut toiminnot hyödyllisiä.
- **Ei häiriötekijöitä**: Keskeytykset vievät keskittymisen tehtävästä. Ilman keskittymistä, ei ole mahdollista saavuttaa flow-tilaa.
- **Suora palaute**: Joka kerta kun toteutamme toiminnon, meidän tulee odottaa ennen kuin tiedämme toiminnon seuraukset. Odotusajan kasvaessa menetämme samalla keskittymiskykyä tehtävän suorittamiseen. Palautteen ollessa välitöntä, pystymme helposti pysymään keskittyneenä.
- **Jatkuva haaste**: Haasteen tulee olla sellainen, jonka uskomme pystyvämme suorittamaan. Mikäli alamme uskomaan, ettemme pysty suorittamaan sitä, turhaudumme ja

alamme etsimään palkitsevampaa aktiviteettiä. Mikäli haaste puolestaan on liian helppo, tässäkin tapauksessa kyllästymme ja alamme etsimään palkitsevampia aktiviteettejä.

Sweetser ja Wyeth (2005, s. 3–4) tiivistävät flow-tilan muodostumisen edellytykseksi sen, että pelaajalla tulee olla kontrollin tunne tehtävän suorittamisesta. Kontrollin tunteen muodostuminen pelissä on mahdollista, kun pelaaja on tarpeeksi taitava, kun tehtävällä on selkeä tavoite ja kun pelaaja saa toiminnon suorittamisesta välittömän palauteen. Parhaimmassa tapauksessa pelaajalle syntyy vahva immersion tunne pelatessaan, jolloin pelaaja uppoutuu peliin niin, että muut arkipäiväiset asiat unohtuvat. (Sweetser ja Wyeth 2005, s. 3–4)

Vaikeustason kehittyminen on usein identifioitu tärkeimmäksi näkökulmaksi hyvässä pelisuunnittelussa (Sweetser ja Wyeth 2005, s. 3–4). Pelin kohderyhmää vastaava vaikeustaso ja vaikeustason sopiva kehittyminen pelin edetessä on siis yksi osa-alue pelisuunnittelussa, mutta sen roolia pidetään usein kriittisimpinä. Vaikeustason kehittyminen flow-käyrän mukaisesti on tärkeää, jotta pelaajat pysyvät pelin parissa luovuttamatta pelaamista turhautumisen tai kyllästymisen vuoksi. Flow on kuitenkin hyvin hankala asia testata, sillä sitä ei pysty todentamaan lyhyellä aikavälillä, vaan pelaajaa tulee tarkkailla pidempiaikaisesti (Schell 2020, s. 149).

2.3 Vaikeuden skaalaus

Yksi perustavanlaatuinen haaste pelikehityksessä on luoda hyvin muotoiltu vaikeuskäyrä, joka mahdollistaa flow-tilan syntymisen. Yksi pelisuunnittelun ydinelementti onkin pyrkiä tekemään pelin vaikeusasteen kehittymisestä sellainen, että pelaaja kokee pelin edetessä tarpeeksi haastetta, mutta ei kuitenkaan liikaa. Pelin vaikeudesta ei ole sellaista selkeää ja yleisesti hyväksyttyä määritelmää, jolla sitä voisi käyttää mitattavana parametrina. Pelin vaikeusasteen säätö on subjektiivinen ja iteratiivinen prosessi, jossa kenttä- ja pelisuunnittelijat luovat sarjan haasteita ja asettavat niiden parametrit vastaamaan heidän valitsemaansa vaikeuskäyrää. Sopivien haastesarjojen ja haasteeseen vaikuttavien parametrien arvojen hienosäätö perustuu pääosin pelitestaamiseen, jonka suorittavat pelisuunnittelijat. Pelitestaus on pelisuunnittelijoille hyvin aikaa vievää ja suunnittelijan on hyvin hankala arvioida haasteen

vaikeutta monen tunnin pelitestaamisen jälkeen. (Aponte, Levieux ja Natkin 2009)

Aponte, Levieux ja Natkin (2009) esittävät, että on olemassa kaksi teoreettista tapaa, joilla voidaan mitata pelin vaikeus. Ensimmäinen tapa on löytää pelin rakenteeseen liittyvä matemaattinen ilmaisu ja ratkaista haasteparametreihin liittyvät yhtälöt. He kuitenkin toteavat, että pelin kompleksisuus ja näin ollen myös haasteeseen vaikuttavien parametrien yhteisvaikutuksena syntyvä kompleksisuus osoittavat, ettei tällainen menetelmä ole yleensä käyttökelpoinen.

Käyttökelpoisempi tapa pelin vaikeustason mittaamiseen on pelin kokeminen ja haasteparametrin määrittäminen. Pelin kokemiseen voidaan käyttää joko oikeaa tai synteettistä pelaajaa. Oikean pelaajan käyttämisen hyötynä on se, että silloin demonstroidaan ihmisen käyttäytymistä. Haittana voidaan pitää sitä, että oikea pelaaja pelaa hitaasti, väsyy pelatessaan ja hänen käyttäytymisensä voidaan ymmärtää vain pelin käyttöliittymän kautta. Synteettinen pelaaja on puolestaan väsymätön ja pelaa nopeasti. Synteettisen pelaajan käyttäytyminen voidaan myös täysin ymmärtää. Synteettinen pelaaja voidaan mallintaa käyttämiseltään tietyn pelaajatyypin käyttäytymistä jäljitteleväksi, kuten riskejä ottavaksi tai varovaiseksi pelaajaksi. (Aponte, Levieux ja Natkin 2009)

Aponte, Levieux ja Natkin (2009) todentavat tämän *Pac-Man*-esimerkillä, jossa he loivat synteettisen pelaajan pelaamaan kyseistä peliä. Heidän hyvin yksinkertaisessa demonstraatiossaan vaikeusastetta voidaan skaalata muuttamalla pelaajan nopeutta. Tällä tavalla voidaan todeta, että synteettinen pelaaja saavuttaa keskimääräisesti paremman pistetuloksen, kun nopeusparametrin arvoa nostetaan, mikä puolestaan kertoo sen, että näin vaikeustaso helpottuu. Tämän esimerkin tulos on varmasti useimmille pelinkehittäjille itsestään selvää. Vaikeusasteeseen liittyy usein monia parametreja, jolloin niiden yhteisvaikutus vaikeustasoon on hyvin kompleksinen.

Tässä tutkielmassa tutkitaan, pystytäänkö synteettistä pelaajaa eli agenttia hyödyntämään vaikeustason balansointiin erittäin haasteellisessa peligenressä koneoppimista hyödyntäen. Tällainen automatisoitu ratkaisu vaikeustasoerojen havainnoimiseksi mahdollistaisi vaikeuskäyrän muotoilemisen agentin toiminnan perusteella. Koneoppimismenetelmiä hyödyntäen agenttia voidaan opettaa käyttäytymään ilman itse käyttäytymisen ohjelmointia.

3 Koneoppiminen

Koneoppiminen on tekoälyn yksi osa-alue, jonka avulla voidaan ratkaista tehtäviä, jotka ovat mahdottomia tai liian hankalia toteutettavaksi perinteisiä ohjelmointikieliä käyttäen (Campesato 2020, s. 24). Murphy (2012, s. 1) määrittelee koneoppimisen tarkoittavan oppimista annetun datan pohjalta: koneoppiminen voidaan määritellä joukoksi metodeja, jotka voivat automaattisesti tunnistaa datan sisällä toistuvat kaavat (*engl. patterns*). Tämän perusteella pystytään esimerkiksi ennustamaan tulevaisuuden dataa tai toteuttamaan muunlaista päätöksentekoa (Murphy 2012, s. 1). Huolimatta koneoppimisalgoritmien suuresta vaihtelevuudesta, data on yleensä tärkeämpää kuin valittu algoritmi. Koneoppimisen suhteen moni ongelma voi johtua datasta: dataa on vähän, data on huonolaatuista, data voi olla virheellistä tai oleellinen data voi puuttua (Campesato 2020, s. 24).

Koneoppiminen voidaan jakaa pienempiin osa-alueisiin. Sutton ja Barto (2018, s. 2–3) ja Murphy (2012, s. 2) esittävät koneoppimisen päätyypeiksi:

- ohjattu oppiminen
- ohjaamaton oppiminen
- vahvistusoppiminen.

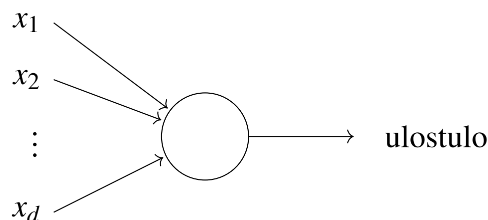
Ohjattu oppiminen tarkoittaa, että datalla on jokin tunniste, jolla voidaan identifioida ja luokitella sen sisältö. Ohjaamaton oppiminen sisältää merkitsemätöntä dataa, jonka käsittelyyn tyypillisesti tarvitaan klusterointia, jotta datasta saadaan johdettua jokin haluttu lopputulos. Nämä päätyypit voivat kuitenkin myös yhdistyä eri menetelmissä. Esimerkiksi Campesato (2020, s. 25–26) esittää vielä puoliohjatun oppimisen yhtenä päätyyppinä. Puoliohjattu oppiminen on yhdistelmä ohjatusta ja ohjaamattomasta oppimisesta: jotkin datapisteet ovat luokiteltuja ja jotkin taas eivät. (Campesato 2020, s. 25–26)

Tässä luvussa esitellään tämän tutkielman kannalta olennaisimmat koneoppimismenetelmät ja käytettävät algoritmit. Luvussa 3.1 käydään läpi neuroverkkojen toiminnan perusperiaatteet. Vahvistusoppiminen, jota käytettiin suurelta osin agentin opettamisessa, esitellään luvussa 3.2. Luku 3.3 kertoo tutkimuksessa käytetyn gradienttimetodin toiminnan. Luvuissa 3.4 , 3.5 ja 3.6 käydään läpi myös muut agentin opettamisessa käytetyt koneoppimismene-

telmät. Luvussa 3.7 esitellään syväoppimisen käsite, joka myös yhdistää edellisten lukujen osa-alueita isommaksi kokonaisuudeksi. Luku 3.8 käsittelee tutkimuksessa käytetyn Unity-pelimoottorin ominaisuuksia koneoppimisen kannalta. Lopuksi muita aihealueen tutkimuksia esitellään luvussa 3.9, jossa käydään läpi tutkimusten lähestymistapaa koneoppimisen hyödyntämiseen sekä käytettyjä koneoppimismenetelmiä. Suoranaisesti bullet hell -genreen tai muuhun hyvin nopeatempoiseen ja haastavaan genreen kohdistettuja vaikeustason balansointia käsitteleviä tutkimuksia ei etsinnästä huolimatta löytynyt, mikä on ymmärrettävää, sillä tutkimusalue on verrattaen uusi.

3.1 Keinotekoinen neuroverkko

Yksinkertaisinta keinotekoista neuroverkkoa kutsutaan nimellä perseptroni (*engl. perceptron*). Tämä neuroverkko sisältää yhden syötekerroksen ja yhden ulostulon. Kuviossa 2 on esitetty yksinkertaistettu kuvaus perseptronin toiminnasta, jossa perseptroni vastaanottaa syötteitä $\overline{X} = [x_1 \ldots x_d] \in \mathbb{R}^d$. Syötteitä on d kappaletta ja ne kuvaavat sisääntulotasoa.



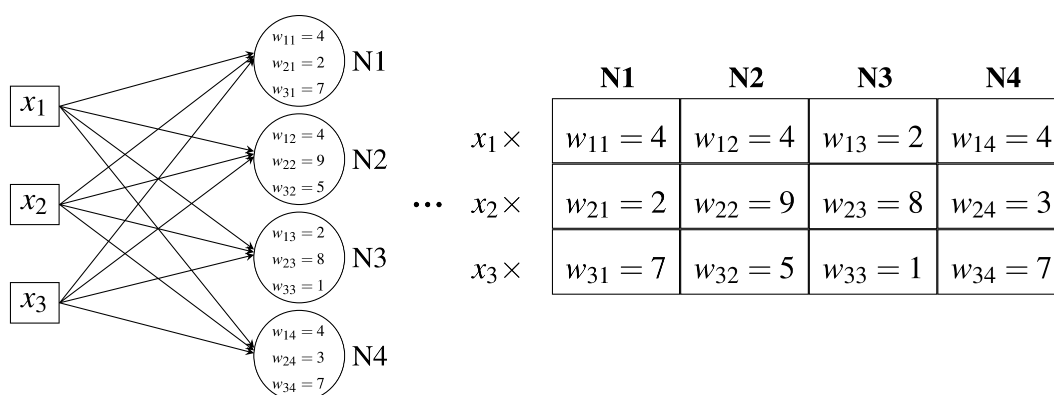
Kuvio 2. Perseptronin yksinkertaistettu kuvaus

Perseptronialgoritmin kehitti Frank Rosenblatt (1958) 1950-luvulla ja sitä pidetään yhtenä ensimmäisistä kehitetyistä koneoppimisen algoritmeista (Murphy 2012, s. 266). Rosenblatt (1958) esitti yksinkertaisen tavan ulostulon laskemiseen: painokertoimet, $\overline{W} = [w_1 \ldots w_d] \in \mathbb{R}^d$. Syöte kerrotaan painokertoimella, jolloin painokertoimien avulla voidaan määrittää syötteen tärkeys. Tarkemmin sanottuna painokertoimella määritellään, kuinka paljon syöte otetaan huomioon ulostuloarvon laskemissa. Neuronin tuottama ulostulo määritetään painotetun summan $\sum_j w_j x_j$ perusteella, jossa siis sisääntuloarvot $[x_1 \ldots x_d]$ kerrotaan sisääntuloa vastaavan painoarvon $[w_1 \ldots w_d]$ kanssa ja summataan yhteen.

Laskemisessa hyödynnetään vakiotermiä (*bias*), jota kuvataan merkinnällä b, jotta neuronin tuottama ulostulo voidaan tarkemmin määritellä. Vakiotermin arvo määrittää kynnysarvon, joka kertoo milloin neuronin ulostuloarvo muuttuu. Perseptroni on binäärinen luokittelija eli se voi tuottaa tuloksen, joka on toinen kahdesta määritellystä tulosarvosta. Yhtälössä (3.1) perseptroni antaa tulokseksi 0 tai 1. Yhtälöstä käytetään nimitystä porrasfunktio. Mikäli painotettu summa ylittää kynnysarvon, niin tulos on 1. Summan ollessa pienempi tai yhtä suuri kuin kynnysarvo, tulos on 0 (Nielsen 2015):

$$\text{ulostulo} = \begin{cases} 0 & \text{jos } \sum_{j=1}^{d} w_j \cdot x_j + b \leq 0 \\ 1 & \text{jos } \sum_{j=1}^{d} w_j \cdot x_j + b > 0 \end{cases} \tag{3.1}$$

Yhtälössä (3.1) w_j ovat painokertoimet, x_j ovat syötevektorin $\bar{X}$ komponentit ja vakiotermi b on reaaliluku. Yleisimmissä tapauksissa kuitenkin painoarvot ovat matriisimuodossa (Campesato 2020, s. 104). Kuviossa 3 on kuvattu yksinkertainen neuroverkko, jossa painoarvot ovat matriisimuodossa. Matriisin koko määräytyy neuroneiden ja syötteiden määrän mukaan. Kuviossa matriisin sarake vastaa yksittäisen neuronin painoja. Yksittäisen syötteen tärkeys muodostuu, kun syöte kerrotaan neuronin vastaavalla painoarvolla eli toisin sanoen matriisin rivillä kuten kuviossa 3 nähdään. (Yasoubi, Hojabr ja Modarressi 2016)

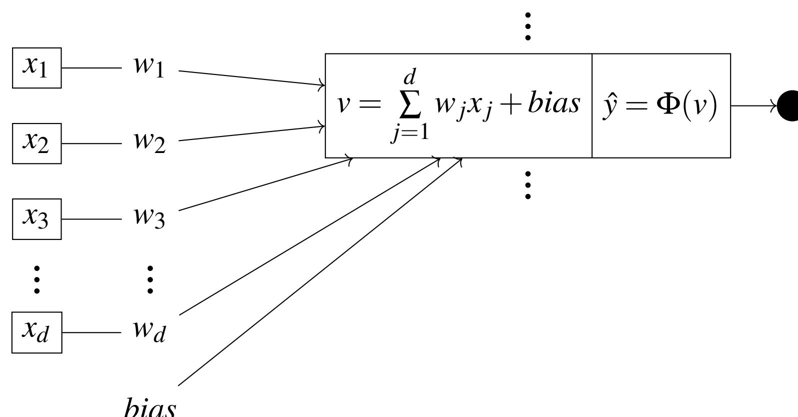


	N1	**N2**	**N3**	**N4**
$x_1 \times$	$w_{11} = 4$	$w_{12} = 4$	$w_{13} = 2$	$w_{14} = 4$
$x_2 \times$	$w_{21} = 2$	$w_{22} = 9$	$w_{23} = 8$	$w_{24} = 3$
$x_3 \times$	$w_{31} = 7$	$w_{32} = 5$	$w_{33} = 1$	$w_{34} = 7$

Kuvio 3. Painokertoimet matriisimuodossa (mukaillen Yasoubi, Hojabr ja Modarressi 2016)

3.1.1 Aktivaatiofunktiot

Aktivaatiofunktiot määrittävät siis sen, mille arvovälille neuronin tulos sidotaan. Kuviossa 4 on kuvattu neuroni, johon vaikuttaa painokertoimien lisäksi myös vakiotermi. Aktivaatiofunktiota merkitään kuviossa tunnuksella Φ.



Kuvio 4. Aktivaatiofunktion kuvaus

Aktivaatiofunktio valitaan tehtävän tarpeen mukaisesti. Yksinkertaisin aktivaatiofunktio on lineaarinen aktivaatio $\Phi(v) = v$. Yleisiä aktivaatiofunktioita ovat esimerkiksi etumerkkifunktio (*sign*), logistinen sigmoidifunktio (*sigmoid*) ja hyperbolinen tangenttifunktio (*tanh*). Näitä aktivaatiofunktioita käytettiin jo neuroverkkojen aikaisessa kehitysvaiheessa. Yhtälöluettelossa (3.2) on kuvattuna, kuinka aktivaatiofunktioksi $\Phi(v)$ voidaan valita jokin näistä funktioista (Aggarwal 2018, s. 12):

$$
\begin{aligned}
\Phi(v) &= \begin{cases} -1 & v < 0 \\ 1 & v \geq 0 \end{cases} && \textit{(sign)}, \\
\Phi(v) &= \frac{1}{1+e^{-v}} && \textit{(sigmoid)}, \\
\Phi(v) &= \frac{e^{2v}-1}{e^{2v}+1} && \textit{(tanh)}.
\end{aligned}
\tag{3.2}
$$

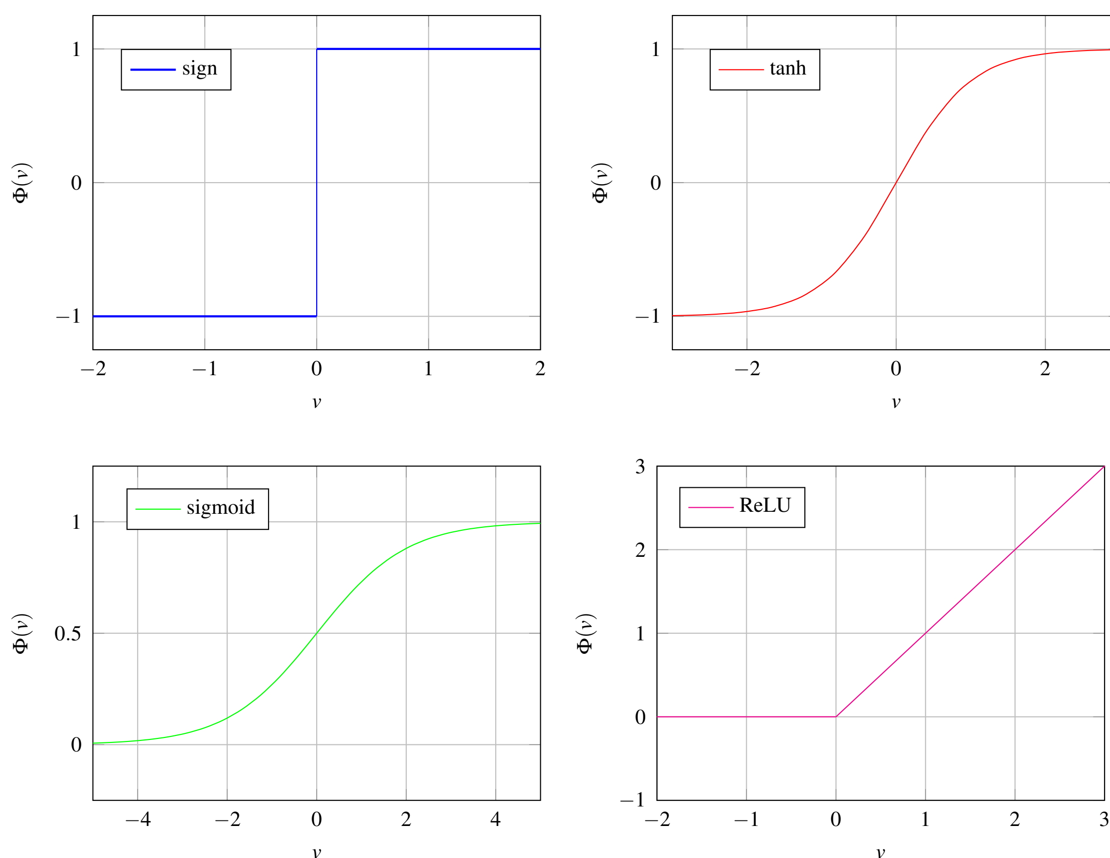
Etumerkkifunktio antaa arvoksi joko -1 tai 1, kun taas logistinen sigmoidifunktio antaa arvon väliltä $[0,1]$. Logistinen sigmoidifunktio on käytännöllinen esimerkiksi todennäköisyyteen liittyvien laskelmien tekemiseen. Hyperbolinen tangenttifunktio puolestaan antaa arvon

väliltä $[-1,+1]$. (Aggarwal 2018, s. 12)

Viime vuosina esimerkiksi ReLU-funktio (*Rectified Linear Unit*) on tullut suosituksi moderneissa neuroverkoissa (Aggarwal 2018, s. 133). Kun v:n arvo on positiivinen, ReLU-funktio antaa tulokseksi sisääntuloarvon v. Muussa tapauksessa $\Phi(v) = 0$.

$$\Phi(v) = \max(v, 0) \quad \textit{(ReLU-funktio)} \tag{3.3}$$

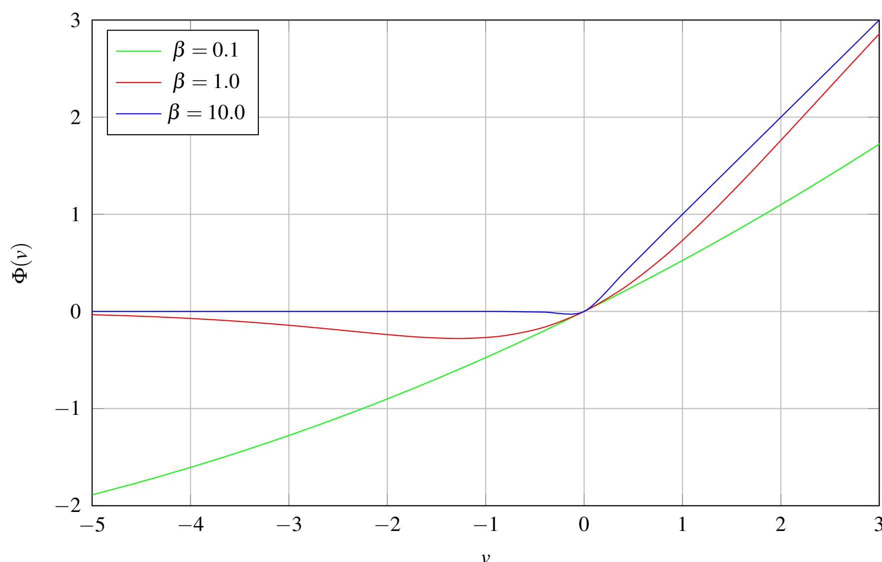
Kuviossa 5 on esiteltynä yllä mainittujen aktivaatiofunktioiden toiminta graafisessa muodossa, jossa v kertoo sisään tulevan arvon, ja $\Phi(v)$ kertoo funktion tulostaman arvon.



Kuvio 5. Erityyppisiä aktivaatiofunktioita

Swish-funktio sisältää sigmoid-funktion $(1+\exp(-v))^{-1}$, joka on kuvattu yhtälössä (3.4) merkinnällä σ:

$$\Phi(v) = v\sigma(\beta v) \quad \textit{(Swish-funktio)} \tag{3.4}$$



Kuvio 6. Swish-aktivaatiofunktio

Swish-funktion β on joko vakio tai koulutettava parametri. Mikäli $\beta = 0$, Swish-funktiosta muodostuu skaalattu lineaarinen funktio $\Phi(v) = \frac{v}{2}$. β:n lähestyessä ääretöntä, $\beta \rightarrow \infty$, sigmoidikomponentin tuottama tulos lähestyy yhtälön (3.1) mukaista porrasfunktiota, joka saa negatiivisilla luvuilla arvon 0 ja positiivisilla arvon 1. Tällöin Swish-funktio käyttäytyy kuten ReLu-funktio. Swish-funktiota voidaankin ajatella funktiona, joka interpoloi lineaarisen funktion ja ReLu-funktion välillä. Interpoloinnin astetta voidaan mallin kautta hallita, jos β on koulutettava parametri. Swish-funktion toiminta eri β:n arvoilla on havainnollistettu kuviossa 6. (Ramachandran, Zoph ja Le 2017)

3.1.2 Oppiminen neuroverkossa

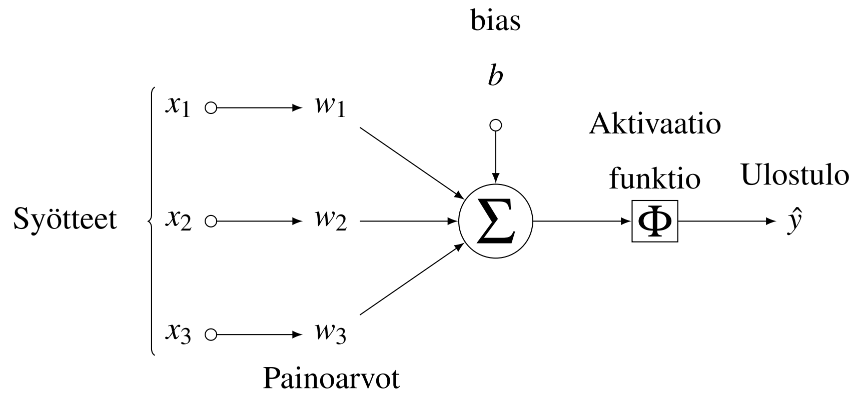
Aggarwal (2018, s. 5) esittää, että opetusdatainstanssi voidaan kuvata parina $(\overline{X}, y)$. Ensimmäinen elementti, $\overline{X} = [x_1 ... x_d] \in \mathbb{R}^d$, on vektori, joka sisältää d kappaletta komponentteja. Opetusdatan toinen komponentti, $y \in \{-1, +1\}$, sisältää havaitun arvon luokkamuuttujasta. Havaitulla arvolla tarkoitetaan tässä yhteydessä sitä, että se on annettu osana opetusdataa. Tehtävänä on ennustaa luokkamuuttuja tilanteissa, joissa sitä ei ole havaittu.

Perseptronin palauttama arvo voidaan kuvata yhtälöllä, jossa painokertoimien ja sisääntulo-

jen laskemisessa käytetään pistetuloa (Aggarwal 2018, s. 6):

$$\hat{y} = sign\{\overline{W} \cdot \overline{X} + b\} = sign\{\sum_{j=1}^{d} w_j x_j + b\},$$

jossa $\hat{y}$ kuvaa ulostulon ennustetta. Etumerkkifunktio, *sign*, muuntaa lasketun tuloksen arvoksi -1 tai 1, kuten yhtälöluettelossa (3.2) esiteltiin. Nyt perseptronin toiminta voidaan nähdä tarkemmin kuvion 7 mukaisesti, jossa on perseptronille kuvattuna kolme sisääntuloa:



Kuvio 7. Perseptroni

Virheen ennuste on täten $E(\overline{X}) = y - \hat{y}$. Neuroverkon painoarvoja päivitetään kun virhearvo $E(\overline{X})$ poikkeaa nollasta. Perseptronialgoritmin tavoitteena on minimoida ennustetun ulostulon $\hat{y}$ ja havaitun arvon y välinen ero. Tämän tyyppistä minimointitavoitefunktiota kutsutaan myös nimellä häviöfunktio (*engl. loss function*). Yksinkertainen häviöfunktio voidaan nyt esittää kaavalla (Aggarwal 2018, s. 6–7):

$$\text{arg min}_{\overline{W}} L = \sum_{(\overline{X},y)\in\mathcal{D}} (y - \hat{y})^2 = \sum_{(\overline{X},y)\in\mathcal{D}} (y - sign\{\overline{W} \cdot \overline{X} + b\})^2,$$

jossa $\mathcal{D}$ kuvaa opetusdataa eli $(\overline{X}, y) \in \mathcal{D}$. Yllä olevan häviöfunktion tarkoitus on havainnollistaa häviöfunktion toiminta. Kyseisessä häviöfunktiossa etumerkkifunktio ei ole kovinkaan käytännöllinen, sillä etumerkkifunktio tuottaa porrasmaisia hyppäyksiä datapisteiden välillä, jolloin se ei ole derivoituva. Tämän johdosta gradienttimenetelmää varten voidaan käyttää heuristisesti määriteltyä pehmennysfunktiota (Aggarwal 2018, s. 7):

$$\nabla L_{smooth} = \sum_{(\overline{X},y)\in\mathcal{D}} (y-\hat{y})\overline{X}$$

Yllä esitetyt tavoitefunktiot on kuvattu koko datan suhteen, mutta tarkemmin sanottuna opetusalgoritmi toimii niin, että se syöttää jokaisen syötedatainstanssin $\overline{X}$ verkkoon yksitellen tai pienissä sarjoissa tuottaakseen ennusteen $\hat{y}$. Sen jälkeen painoarvot päivitetään perustuen virhearvoon $E(\overline{X}) = y - \hat{y}$. Tarkemmin sanottuna kun datapiste $\overline{X}$ on syötetty verkkoon, painokertoimet $\overline{W}$ päivitetään seuraavanlaisesti (Aggarwal 2018, s. 7):

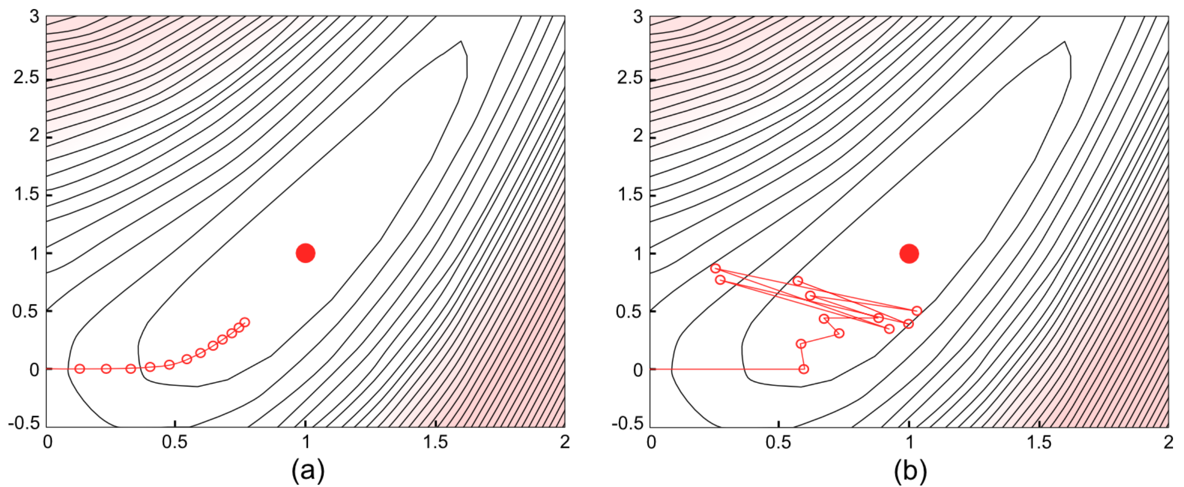
$$\overline{W} \leftarrow \overline{W} + \alpha E(\overline{X})\overline{X},$$

jossa α säätelee oppimisnopeutta (*engl. learning rate*).

Kuviossa 8 on havainnollistettu oppimisnopeuden periaate gradienttimenetelmässä yksinkertaisella funktiolla kuvattuna. Kuviossa aloitetaan kohdasta $(0,0)$ ja jatketaan 12 askelta käyttäen kiinteää oppimisnopeutta eli askelkokoa. Globaali minimi on kohdassa $(1,1)$ ja se on kuvattu punaisella pisteellä. Kuvion 8 a-kohdassa askelkoko on pieni ja b-kohdassa vastaavasti suurempi kuin a-kohdassa. Kuvion b-kohdassa on kuvattuna tilanne, jossa algoritmi ei toimi toivotulla tavalla globaalin minimin löytämiseen, sillä oppimisnopeus on tässä tapauksessa liian suuri eli gradienttia päivitetään liian suurella askelkoolla. Vastaavasti pieni oppimisnopeus tekee luonnollisesti lähestymisestä hidasta. Kiinteän oppimisnopeuden tapauksessa siis liian pieni oppimisnopeus tekee lähestymisestä hyvin hidasta, kun taas liian suurella oppimisnopeudella lähestyminen voi epäonnistua täysin (Murphy 2012, s. 247).

Perseptronialgoritmi käy jatkuvasti jaksoittain läpi opetusdataa satunnaisessa järjestyksessä ja iteratiivisesti säätää painoarvoja, kunnes lähentyminen on tapahtunut. Yksittäinen opetusdatapiste voi olla läpikäytynä useassa eri syklissä. Jokaista yksittäistä sykliä kutsutaan nimellä epokki (*engl. epoch*). (Aggarwal 2018, s. 7)

Neuroverkko tekee siis laskentaa sisään tulevien arvojen perusteella. Laskeminen tapahtuu siirtämällä sisään tulevien neuronien arvot seuraavan tason neuroneille käyttämällä painoarvoja välissä olevina parametreinä. Oppiminen tapahtuu muokkaamalla painoarvoja neuronien välissä. (Aggarwal 2018, s. 2)

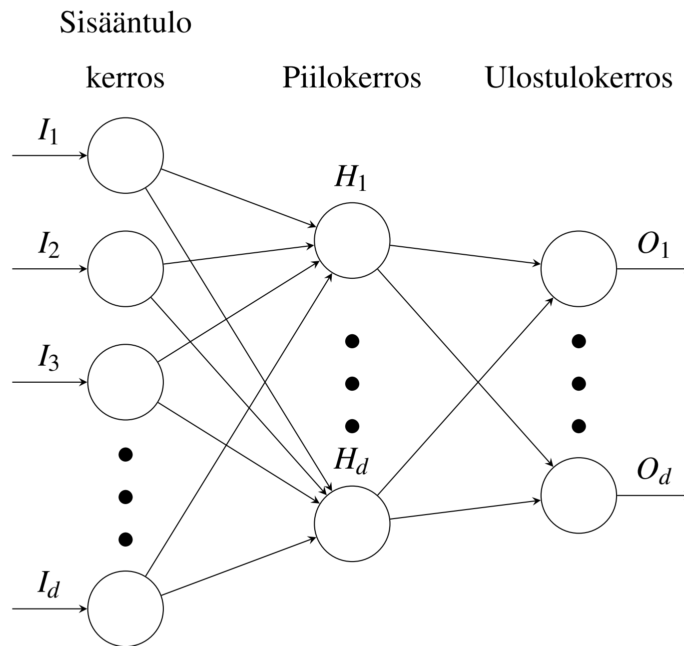


Kuvio 8. Oppimisnopeuden periaate gradienttimenetelmässä (mukaillen Murphy 2012, s. 247)

Tällaisessa verkossa ensimmäinen kerros tekee yksinkertaisia päätöksiä painotettujen sisääntulojen perusteella. Toinen kerros puolestaan tekee päätöksiä painottamalla ensimmäisen kerroksen tulokset. Tällä tavoin voimme muodostaa monitasoisen verkon, jota voidaan hyödyntää monimutkaisemmissa ja vaativammissa päätöksenteoissa. (Nielsen 2015)

Sisääntulo- ja ulostulokerroksen välisiä kerroksia kutsutaan piilokerroksiksi (*engl. hidden layer*). Verkkoa, jolla on yksi tai useampi piilokerros, kutsutaan monikerroksiseksi neuroverkoksi. Monikerroksisen neuroverkon yksinkertaistettu rakenne on kuvattu kuviossa 9, jossa I on sisääntulokerros, H on piilokerros ja O on ulostulokerros. Kuviossa ei ole kuvattuna vakiotermiä.

Neuroverkoilla on suuri määrä hyperparametreja, jotka toimivat säätiminä oppimisprosessille. Hyperparametreistä yhtenä esimerkkinä ovat piilokerroksien lukumäärä ja neuroneiden lukumäärä piilokerroksissa. Hyperparametrien avulla voidaan esimerkiksi säätää opettamisen nopeutta ja oppimisen tehokkuutta. Hyperparametrien valintamenetelmistä tunnetuin on ruudukkohakumenetelmä *(engl. grid search)*. Ruudukkohakumenetelmässä jokaiselle hyperparametrille valitaan etukäteen joukko arvoja, jolloin voidaan testata joukon kaikki mahdolliset kombinaatiot. Tämän läpikäynnin perusteella nähdään, millä kombinaatiolla löytyi paras opetustulos, jolloin hyperparametreiksi voidaan valita kyseiset arvot. Nopeuden ja tehokkuuden kannalta optimaalisten hyperparametrien etsiminen on kuitenkin haastavaa, sillä



Kuvio 9. Monikerroksisen neuroverkon rakenne

vaihtoehtojen määrä kasvaa hyperparametrien lukumäärään verrattuna eksponentiaalisesti. Lisäksi varsinkin hieman monimutkaisemmissa tapauksissa hyperparametrien arvioimisen kannalta riittävien opetustuloksien saaminen voi kestää tarpeettoman kauan. Esimerkiksi yksittäinen opetusajo voi tietyissä tapauksissa kestää prosessoinnin kannalta useita viikkoja, jolloin kaikkien kombinaatioiden etsiminen on hyvin epäkäytännöllistä. Monissa tapauksissa opettaminen voidaan kuitenkin lopettaa jo aikaisessa vaiheessa, mikäli nähdään, että opetustuloksista tulee muodostumaan heikkolaatuisia. (Aggarwal 2018, s. 125–126)

3.1.3 Vastavirta-algoritmi

Useimmiten keinotekoisten neuroverkkojen oppiminen perustuu ensisijaisesti gradienttimenetelmän käyttämiseen hyödyntäen aktivaatiofunktioiden tulosta (Aggarwal 2018, s. 16). Gradientti kertoo funktion suurimman muutosnopeuden ja sen suunnan. Rumelhart, Hinton ja Williams (1986) esittivät oppimismenetelmän, vastavirta-algoritmin, jossa toistuvasti säädetään verkon liitäntöjen painoarvoja minimoidakseen eron todellisen ulostulon ja toivotun ulostulon välillä.

Yksikerroksisessa neuroverkossa oppimisprosessi tapahtuu suhteellisen suoraviivaisesti, sil-

lä virhe eli häviöfunktio voidaan laskea suoralla funktiolla painoarvoista, mikä mahdollistaa helpon gradientin laskemisen. Monikerroksisissa neuroverkoissa ongelmana on se, että häviö on monimutkainen yhdistetty funktio, johon vaikuttaa aikaisempien kerroksien painoarvot. Gradientti lasketaan tällöin käyttämällä vastavirta-algoritmia (*engl. backpropagation*). (Aggarwal 2018, s. 21)

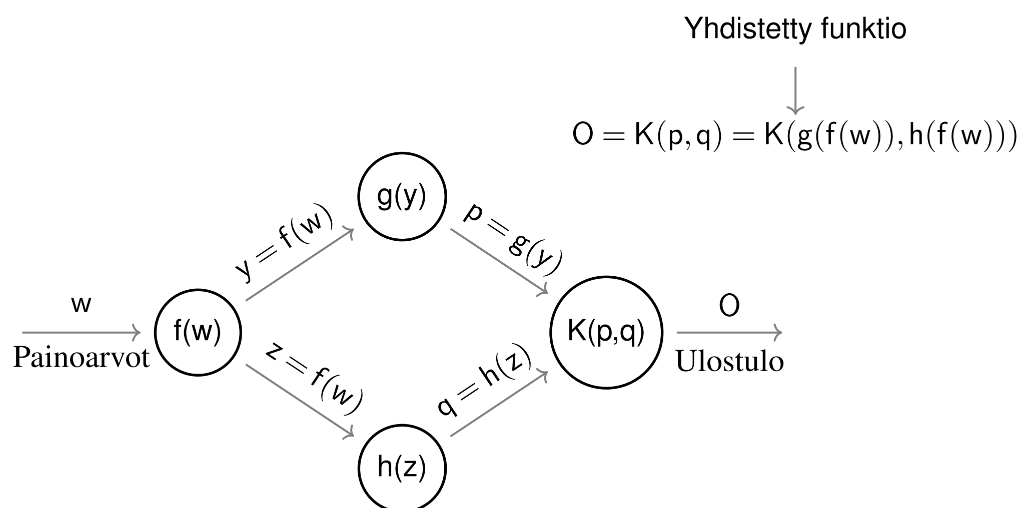
Vastavirta-algoritmien yhteydessä käytetään usein termejä esiaktivointiarvo (*pre-activation value*) ja jälkiaktivointiarvo (*post-activation value*). Esiaktivointiarvolla tarkoitetaan neuronin arvoa ennen kuin se on syötetty aktivaatiofunktiolle. Esiaktivointiarvoa voidaan kuvata notaatiolla $a_h = \overline{W} \cdot \overline{X} + b$. Jälkiaktivointiarvolla puolestaan tarkoitetaan neuronin tuottamaa tulosta aktivaatiofunktion jälkeen, josta käytetään notaatiota $h = \Phi(a_h)$. Tarkennuksena sanottakoon, että neuronin tuottama tulos on aina jälkiaktivointiarvo. Esiaktivointiarvoa käytetään kuitenkin vastavirta-algoritmin sisäisessä laskennassa. (Aggarwal 2018, s. 12)

Vastavirta-algoritmi hyödyntää differentiaalilaskennan ketjusääntöä, joka laskee virhegradientit summaamalla paikallisten gradienttien pistetulot eri poluista, jotka kulkevat aina sisääntulokerroksesta ulostuloon. Polulla tarkoitetaan tässä yhteydessä siis reittiä, joka kulkee eri neuronien kautta aina ulostuloon asti. Kuviossa 10 on havainnollistettu kaksi eri polkua. Vaikkakin gradienttien summaus sisältää eksponentiaalisen määrän polkuja, voidaan se laskea tehokkaasti dynaamisen ohjelmoinnin avulla. Vastavirta-algoritmi on suora sovellus dynaamisesta ohjelmoinnista. Vastavirta-algoritmi sisältää kaksi vaihetta, joita kutsutaan eteenpäin suuntautuvaksi ja taaksepäin suuntautuvaksi vaiheeksi. Eteenpäin suuntautuva vaihe vaaditaan laskemaan ulostuloarvot sekä paikalliset derivaatat eri solmuissa, kun taas taaksepäin suuntautuva vaihe tarvitaan kokoamaan pistetulot näistä paikallisista derivaatoista. Taaksepäin suuntautuvassa vaiheessa käydään läpi verkon kaikki eri polut tarkasteltavasta neuronista aina ulostuloon asti. (Aggarwal 2018, s. 21)

Eteenpäin suuntautuvassa vaiheessa opetukseen tarvittavat syötteet annetaan neuroverkolle. Tästä seuraa eteenpäin virtaava laskenta läpi kerroksien käyttämällä nykyisiä painoarvoja. Lopullista ennustettua ulostuloa voidaan verrata opetuksen syötteeseen, ja näin voidaan häviöfunktion derivaatta suhteessa ulostuloarvoon laskea. (Aggarwal 2018, s. 21)

Taaksepäin suuntautuvassa vaiheessa tulee nyt laskea häviön derivaatta suhteessa kaikkien

kerroksien painoarvoihin. Tarkemmin sanottuna taaksepäin suuntautuvassa vaiheessa tavoite on määrittää häviöfunktion gradientti ottaen huomioon eri painoarvot käyttämällä differentiaalilaskennan ketjusääntöä. Näitä gradientteja käytetään päivittämään painoarvoja. Tätä oppimisprosessia kutsutaan taaksepäin suuntautuvaksi vaiheeksi, koska gradientit lasketaan ulostuloneuronista aloittaen ja laskemista jatketaan siis taaksepäin suuntautuen. (Aggarwal 2018, s. 21)



Kuvio 10. Ketjusäännön periaate (mukaillen Aggarwal 2018, s. 22)

Yksinkertaistetussa kuviossa 10 on kuvattuna kaksi eri polkua, ja yhtälössä (3.5) on havainnollistettu kuvion mukainen ketjusäännön toiminta yhtälömuodossa. Osittaisderivaattojen pistetulot kootaan yhteen polkujen mukaisesti. Kokoaminen tapahtuu painoista w ulostuloon o. Tulokseksi siis syntyy ulostulon derivaatta, joka on suhteessa painoon w (Aggarwal 2018, s. 22):

$$\begin{aligned}\frac{\partial o}{\partial w} &= \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial w} \quad \text{[Monen muuttujan ketjusääntö]} \\ &= \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial w} \quad \text{[Yhden muuttujan ketjusääntö]} \\ &= \underbrace{\frac{\partial K(p,q)}{\partial p} \cdot g'(y) \cdot f'(w)}_{\text{1. polku}} + \underbrace{\frac{\partial K(p,q)}{\partial q} \cdot h'(z) \cdot f'(w)}_{\text{2. polku}}\end{aligned} \tag{3.5}$$

Oletetaan, että kerroksia on k kappaletta, ja neuroneita samalla kerroksella kuvataan merkin-

nöillä $h_1, h_2, \ldots, h_k$. Ulostuloa kuvataan merkinnällä o, jolle on laskettu häviö L. Lisäksi painoarvoja neuronilta h_r seuraavan kerroksen neuroniin h_{r+1} voidaan tällöin kuvata notaatiolla $w(h_r, h_{r+1})$ kun $r \in 1 \ldots k$. (Aggarwal 2018, s. 22)

Kun oletetaan, että vain yksi polku kulkee neuronista h_1 ulostuloon o, häviöfunktion gradientti voidaan derivoida käyttämällä ketjusääntöä (Aggarwal 2018, s. 22):

$$\frac{\partial L}{\partial w_{(h_{r-1}, h_r)}} = \frac{\partial L}{\partial o} \cdot \left[\frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}} \forall r \in 1 \ldots k$$

Oikeassa tilanteessa eri polkuja on eksponentiaalinen määrä, jolloin voidaan käyttää monen muuttujan ketjusääntöä (*engl. multivariable chain rule*). Tällöin käytetään yhdistettyä funktiota jokaista polkua kohden alkaen neuronista h_1 jatkaen aina ulostuloon o (Aggarwal 2018, s. 22):

$$\frac{\partial L}{\partial w_{(h_{r-1}, h_r)}} = \underbrace{\frac{\partial L}{\partial o} \cdot \left[\sum_{[h_r, h_{r+1}, \ldots h_k, o] \in \mathcal{P}} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\text{Vastavirta-algoritmi laskee } \Delta(h_r, o) = \frac{\partial L}{\partial h_r}} \frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}}, \tag{3.6}$$

jossa $\mathcal{P}$ kuvaa joukkoa kaikista olemassa olevista poluista neuronista h_r ulostuloon o asti. Lausekkeen oikea puoli, $\frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}}$, avataan yhtälössä (3.8). (Aggarwal 2018, s. 23)

Virheellä $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$ tarkoitetaan rekursiivista laskemista, jossa ensin lasketaan $\Delta(h_k, o)$ neuroneille h_k, jotka ovat siis lähimpänä ulostuloa o. Tämän jälkeen rekursiivisesti jatketaan laskemista aikaisempien kerroksien neuroneille. Lisäksi virhe $\Delta(o, o)$ jokaiselle ulostulolle alustetaan seuraavanlaisesti (Aggarwal 2018, s. 23):

$$\Delta(o, o) = \frac{\partial L}{\partial o}$$

Virhe $\Delta(h_r, o)$ voidaan derivoida käyttämällä monen muuttujan ketjusääntöä (Aggarwal 2018, s. 23):

$$\Delta(h_r, o) = \frac{\partial L}{\partial h_r} = \sum_{h: h_r \Rightarrow h} \frac{\partial L}{\partial h} \frac{\partial h}{\partial h_r} = \sum_{h: h_r \Rightarrow h} \frac{\partial h}{\partial h_r} \Delta(h, o) \tag{3.7}$$

Kun jokainen h on myöhäisemmällä kerroksella kuin h_r, virhe $\Delta(h,o)$ on jo laskettu kun $\Delta(h_r,o)$ on käsittelyssä. Tilanteessa, jossa h_r on kytketty neuroniin h painoarvot kuvataan $w(h_r,h)$. Tällöin yhden muuttujan ketjusäännöllä voidaan laskea yhtälön (3.7) $\frac{\partial h}{\partial h_r}$, joka toistetaan rekursiivisesti taaksepäin suuntautuen aloittaen ulostuloneuronista seuraavanlaisesti (Aggarwal 2018, s. 23):

$$\frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial a_h} \cdot \frac{\partial a_h}{\partial h_r} = \frac{\partial \Phi(a_h)}{\partial a_h} \cdot w_{(h_r,h)} = \Phi'(a_h) \cdot w_{(h_r,h)},$$

jonka vastaavat painoarvojen päivitykset on seuraavat (Aggarwal 2018, s. 23):

$$\Delta(h_r,o) = \sum_{h:h_r \Rightarrow h} \Phi'(a_h) \cdot w_{(h_r,h)} \cdot \Delta(h,o)$$

Näin gradientit ovat koottu yhteen taaksepäin suuntautuen, ja jokainen neuroni on käyty läpi. Lopuksi yhtälön (3.6) $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$ voidaan laskea (Aggarwal 2018, s. 23):

$$\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} = h_{r-1} \cdot \Phi'(a_{h_r}), \tag{3.8}$$

jolloin tuloksena saatu gradientti on derivaatta, jossa on huomioitu kaikkien kerroksien neuronit.

Kuten aiemmin mainittiin a_h on esiaktivointiarvo ennen kuin aktivaatiofunktio $\Phi(\cdot)$ on lisätty, jolloin $h = \Phi(a_h)$. Esiaktivointiarvo a_h määräytyy siis aikaisempien kerroksien neuroneista, joiden ulostulot ovat neuronin h syötteitä. Tällöin yhtälö (3.6) voidaan esittää muodossa (Aggarwal 2018, s. 24):

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial L}{\partial o} \cdot \Phi'(a_o) \cdot \left[\sum_{[h_r,h_{r+1},\dots h_k,o] \in \mathcal{P}} \frac{\partial a_o}{\partial a_{h_k}} \prod_{i=r}^{k-1} \frac{\partial a_{h_{i+1}}}{\partial a_{h_i}} \right]}_{\text{Vastavirta-algoritmi laskee } \delta(h_r,o) = \frac{\partial L}{\partial a_{h_r}}} \frac{\partial a_{h_r}}{\partial w_{(h_{r-1},h_r)}}$$

$\delta(h_r,o)$ kertoo rekursiivisen yhtälön. Virhe $\delta(o,o)$ voidaan laskea seuraavanlaisesti (Aggarwal 2018, s. 24):

$$\delta(o,o) = \frac{\partial L}{\partial a_o} = \Phi'(a_o) \cdot \frac{\partial L}{\partial o}$$

Monen muuttujan ketjusääntöä voidaan käyttää kuten yhtälössä (3.7) (Aggarwal 2018, s. 24):

$$\delta(h_r,o) = \frac{\partial L}{\partial a_{h_r}} = \sum_{h:h_r \Rightarrow h} \overbrace{\frac{\partial L}{\partial a_h}}^{\delta(h,o)} \underbrace{\frac{\partial a_h}{\partial a_{h_r}}}_{\Phi'(a_{h_r})w_{(h_r,h)}} = \Phi'(a_{h_r}) \sum_{h:h_r \Rightarrow h} w_{(h_r,h)} \cdot \delta(h,o),$$

joka on yleisempi kuvaus vastavirta-algoritmien yhteydessä. Häviön osittaisderivaatta painoarvot huomioiden on tällöin laskettavissa (Aggarwal 2018, s. 24):

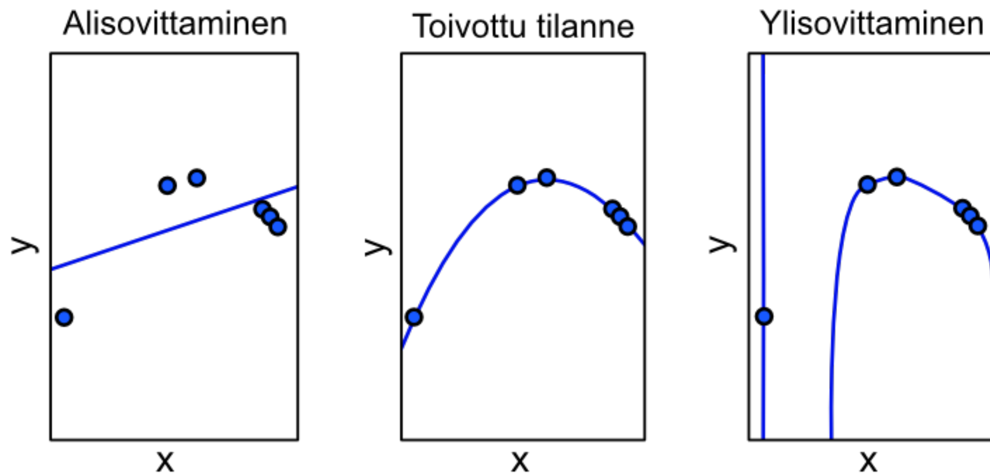
$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \delta(h_r,o) \cdot h_{r-1} \tag{3.9}$$

Neuronien painoarvojen päivittämisprosessia toistetaan jaksoittain (epokkeina) opetusdataa läpi käyden, kunnes virhe pienenee ja lähestyminen tapahtuu. Neuroverkko voi joskus vaatia tuhansia epokkeja painoarvojen päivittämisen yhteydessä. (Aggarwal 2018, s. 24)

3.1.4 Yli- ja alisovittaminen

Koneoppimisessa on kaksi keskeistä haastetta: ylisovittaminen ja alisovittaminen. Alisovittaminen tapahtuu, kun malli ei ole kykeneväinen saavuttamaan riittävän alhaista virhearvoa opetusdatasta eli se mukailee opetusdataa liian heikosti. Tällöin malli ei ole kykeneväinen ratkaisemaan monimutkaisia tehtäviä. Ylisovittaminen tapahtuu, kun malli mukailee opetusdataa liian tarkasti. Tällöin malli ei välttämättä pysty ratkaisemaan haluttuja tehtäviä. (Goodfellow, Bengio ja Courville 2016, s 110)

Ylisovittaminen voidaan välttää esimerkiksi, kun lopetetaan opettaminen tietyn tarkkuuden saavuttamisen jälkeen, mutta myös opetusdatan lisääminen voi olla ratkaisu, mikäli sen toteuttaminen tai hankkiminen ei tuota ongelmia (Nielsen 2015). Kuviossa 11 nähdään, kuinka alisovittunut malli ei pysty mukailemaan datapisteitä kovinkaan hyvin. Ylisovittunut malli toimii olemassa olevilla datapisteillä, mutta sen toiminta on heikkoa uudessa ympäristössä, sillä malli ei pysty toimimaan tehokkaasti erillisen datan suhteen.



Kuvio 11. Yli- ja alisovittaminen (mukaillen Goodfellow, Bengio ja Courville 2016, s. 111)

3.2 Vahvistusoppiminen

Vahvistusoppiminen tarkoittaa oppimista siitä, mitä tulee tehdä – miten toimia tietyssä tilanteessa, jotta saadaan maksimoitua numeerinen palkintosignaali (*engl. reward signal*). Agentille ei kerrota, mikä toiminto tulee tehdä, vaan agentin tulee havaita, mitkä toiminnot tuottavat eniten palkintoa kokeilemalla toimintoja. Haastavimmissa tehtävissä toiminnot eivät vaikuta välittömään palkintoon vaan myös myöhempiin palkintoihin. Nämä kaksi asiaa – kokeileminen ja erehtyminen sekä viivästyneet palkinnot ovat vahvistusoppimisen tärkeimpiä tunnuksenomaisia ominaisuuksia. (Sutton ja Barto 2018, s. 1)

Vahvistusoppiminen viittaa siis päämäärätietoiseen algoritmiin, joka pyrkii saavuttamaan tietyn tavoitteen. Tavoite voi olla esimerkiksi pelin voittaminen, joka vaatii useita eri liikkeitä. Vahvistusoppimisalgoritmi antaa palkinnon oikeasta liikkeestä, kun taas väärästä liikkeestä se antaa rangaistuksen eli negatiivisen palkinnon. (Campesato 2020, s. 175)

Agentin tulee siis tarkastella myös negatiivisia palkintoarvoja tuottavien toimintojen jälkeisiä tiloja ja niiden tuottamia palkintoarvoja. Agentin toimintoa (*engl. action*) kuvataan yleensä tunnuksella a, agentin tilaa (*engl. state*) tunnuksella s, agentille annettavaa palkintoa (*engl. reward*) tunnuksella r ja agentin aika-askelta (*engl. time step*) tunnuksella t.

Agentti kokeilee tilanteissa toimintoja. Kokeilemisen tuloksena agentti oppii yhä paremmin suotuisat toiminnot eri tilanteissa. Agentin tulee siis hyödyntää (*engl. exploit*) aikaisempia kokemuksia siitä, mitkä aikaisemmat toiminnot ovat johtaneet palkitsemiseen, mutta myös tutkia (*engl. explore*) mahdollisia uusia toimintoja, jotka johtavat parempaan lopputulokseen tulevaisuudessa (Sutton ja Barto 2018, s. 1-3).

Sutton ja Barto (2018, s. 6) esittävät, että vahvistusoppimisessa on neljä keskeistä elementtiä, jotka ovat tärkeitä esitellä: menettelytapa (*engl. policy*), palkintosignaali (*engl. reward signal*), arvofunktio (*engl. value function*) ja joissakin tapauksissa esiintyvä malli (*engl. model*) ympäristöstä. Menettelytapa π määrittelee agentin käyttäytymistavan tarkasteltavassa hetkessä. Karkeasti sanottuna menettelytapa on havaitun ympäristön tilan ja tilassa suoritettavan toiminnon välinen linkitys. (Sutton ja Barto 2018, s. 6)

Sutton ja Barto (2018, s. 6) käyttävät termiä palkintosignaali numeraalisesta palkinnosta, jonka ympäristö lähettää agentille aika-askeleella t. Palkintosignaali määrittää siis hyvän ja huonon tapahtuman agentin näkökulmasta. Agentin tehtävä on pitkällä tähtäimellä maksimoida kokonaispalkinnon määrä. Palkintosignaali on ensisijainen perusta menettelytavan muutokselle; jos menettelytavan valittu toiminto antaa huonon palkinnon, niin menettelytapaa voi olla tarvetta muuttaa johonkin toiseen toimintoon kyseisessä tilassa ollessa. Palkintosignaalista käytetään notaatiota $r_t \in \mathbb{R}$. Negatiivisesta palkintoarvosta puhuttaessa voidaan ajatella sen olevan rangaistus agentille ei-toivotun toiminnon suorittamisesta.

Palkintosignaalin määrittäessä mikä on hyvä toiminto välittömässä tilanteessa, arvofunktio (*engl. value function*) määrittää sen, mikä on hyvää pitkällä tähtäimellä. Karkeasti puhuen tilan odotusarvo on palkintojen kokonaismäärä, jonka agentti odottaa keräävänsä tulevaisuudessa tietyllä toimintaketjulla aloittaessaan etenemisen nykyisestä tilasta. Toiminnot valitaan arvofunktioiden perusteella. Arvofunktiot puolestaan perustuvat kokonaispalkintomäärään. Suurin haaste vahvistusoppimisessa on tulevaisuuden arvojen tehokas arviointi. (Sutton ja Barto 2018, s. 6)

Neljäs eli viimeinen elementti, jota joissakin vahvistusoppimisen järjestelmissä käytetään, on ympäristön malli. Ympäristön mallilla tarkoitetaan tilannetta, jossa pyritään matkimaan ympäristön käyttäytymistä. Yleisemmin sanottuna tavoitteena on mahdollistaa päätelmien teke-

misen siitä, kuinka ympäristö käyttäytyy. Esimerkiksi tietyssä tilassa ja toiminnossa malli voi ennustaa tuloksena seuraavan tilan ja seuraavan palkinnon. Malleja käytetään suunnitteluun, jolla tarkoitetaan tapaa päättää toimintatavasta huomioimalla tulevaisuuden tilanteet ennen kuin niitä on koettu. Vahvistusoppimisen menetelmät, jotka käyttävät malleja ja suunnittelua, ovat mallipohjaisia (*engl. model-based*) metodeja kun taas päinvastaisesti mallivapaat (*engl. model-free*) metodit perustuvat täysin oppimiseen kokeilemisen ja erehtymisen (*engl. trial-and-error*) kautta. (Sutton ja Barto 2018, s. 7)

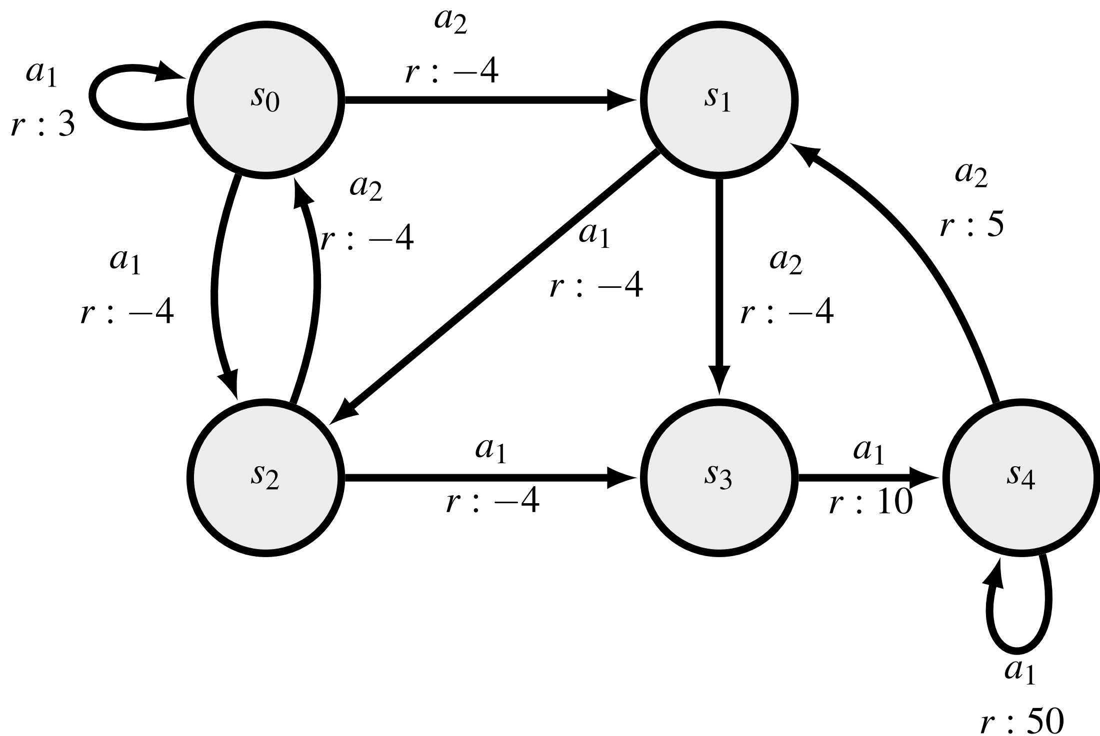
3.2.1 Markovin päätösprosessit

Markovin päätösprosessit (*engl. Markov Decision Process*) on matemaattisesti idealisoitu muoto eli kehys, johon vahvistusoppimisen teoreettiset ilmaisut voi perustaa. Markovin päätösprosessit ovat klassinen formalisointi päätöksenteon sarjalle, jossa toiminnot eivät ota huomioon vain välittömiä palkintoja, vaan myös toiminnon suorittamisen seurauksena syntyvät tilanteet. (Sutton ja Barto 2018, s. 47)

Markovin päätösprosessit muodostuvat tiloista ja toiminnoista sekä siirtymäsäännöistä tilasta seuraavaan. Tärkeä ominaisuus Markovin päätösprosessissa on se, että seuraavan toiminnon valintaprosessiin ei vaikuta aikaisemmin toteutuneet tapahtumat. (Campesato 2020, s. 179)

Kuviossa 12 on käytetty Markovin päätösprosessin kuvaamisessa vastaavaa notaatiota kuin Campesato (2020, s. 179). Merkinnät a_1 ja a_2 vastaavat toimintoja, joiden suorittamisen jälkeen vastaanotetaan palkinto r ja siirrytään seuraavaan tilaan. Ahne algoritmi ei päädy ikinä tilaan s_4, jossa palkintoarvo olisi suurin, sillä se valitsee aina suurimman palkinnon missä tahansa tilassa, joten tällöin kuviossa pysytään aina tilassa s_0. Vahvistusoppimisessa yhtenä ongelmana nousee esille siis se, että milloin agentin tulisi hyödyntää aikaisempaa tietoa ja milloin kokeilla uutta, mahdollisesti parempaan palkitsemiseen johtavaa toimintoa.

Epsilon-ahne menetelmä on eräs yksinkertaisimmista menetelmistä, jossa yhdistyy sekä aikaisemman informaation hyödyntäminen että uuden toiminnon kokeileminen. Epsilon-ahne menetelmän ajatus on se, että välillä satunnaisesti kokeillaan eri tiloihin siirtymistä. Näin voidaan samalla tutkia uusia ja mahdollisesti paremman lopputuloksen tuottavia siirtymiä. (Campesato 2020, s. 180)



Kuvio 12. Markovin päätösprosessi

Wiering ja Otterlo (2012, s. 10–12) esittävät, että Markovin päätösprosessi voidaan kuvata monikkona $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$, jossa

- $\mathcal{S}$ on äärellinen tila-avaruus, $s_1, \ldots, s_n$
- $\mathcal{A}$ on äärellinen joukko toimintoja, $a_1, \ldots, a_n$
- T on siirtymäfunktio, $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$
- R on palkintofunktio, $R : \mathcal{S} \to \mathbb{R}$.

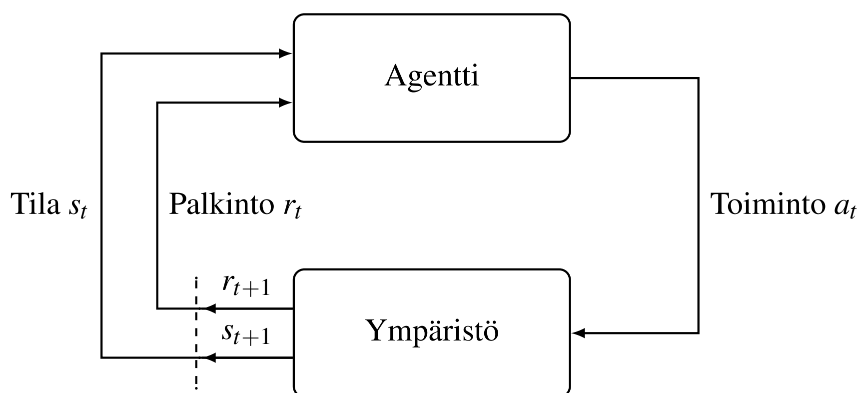
Markovin päätösprosessi koostuu siis tiloista, toiminnoista, tilojen välisistä siirtymistä ja palkintofunktion määritelmästä. Tilalla tarkoitetaan kuvausta päätöksentekoon vaikuttavasta ympäristöstä. Esimerkiksi shakissa pelinappuloiden muodostelma, eli niiden sen hetken sijainnit, muodostavat tilan. Tilan kuvaus aika-askeleella t esitetään notaatiolla s_t. Toimintoja voidaan käyttää ohjaamaan systeemin tilaa, mutta tietyissä järjestelmissä ei voida kaikkia toimintoja suorittaa kaikissa tiloissa. Joukkoa toimintoja, joita voidaan käyttää määritellyissä tiloissa $s \in \mathcal{S}$, kuvataan notaatiolla $\mathcal{A}(s) \subseteq \mathcal{A}$. (Wiering ja Otterlo 2012, s. 10–11)

Suorittamalla toiminnon $a \in \mathcal{A}$ tilassa $s \in \mathcal{S}$ järjestelmä toteuttaa siirtymän tilasta s uuteen tilaan $s' \in \mathcal{S}$. Siirtymäfunktio, T, voidaan kuvata yhtälöllä (Wiering ja Otterlo 2012, s. 11):

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots) = P(s_{t+1}|s_t, a_t) = T(s_t, a_t, s_{t+1}),$$

jossa P kuvaa todennäköisyyttä. Siirtymäfunktio määrittää todennäköisyysjakauman kaikille mahdollisille seuraaville tiloille eli se kuvaa todennäköisyyttä päätyä tilasta s tilaan s' toiminnon a toteuttamisen jälkeen. (Wiering ja Otterlo 2012, s. 11)

Palkintofunktio määrittää epäsuorasti oppimisen tavoitteen. Agentin tavoitteena on päätyä tiloihin, joista saa positiivisen palkinnon. Palkintofunktio määrittää palkinnot tilassa olemisen perusteella tai tilassa toteutetun toiminnon perusteella. Tila-palkintofunktio on kuvattu $R : \mathcal{S} \rightarrow \mathbb{R}$ ja se määrittää saadun palkinnon kyseisessä tilassa. Wiering ja Otterlo (2012, s. 11–12) kuitenkin tarkentavat, että on olemassa kaksi muutakin määritelmää: $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ ja $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$. Ensimmäinen määritelmä antaa palkinnon toiminnon toteuttamisesta agentin ollessa kyseisessä tilassa. Jälkimmäinen antaa palkinnon tilasiirtymän perusteella. Palkintofunktio antaa tulokseksi skalaariarvon, joka voi olla negatiivinen (rangaistus) tai positiivinen (palkinto). Erilaiset tarkoitusperät Markovin päätösprosessin kuvaamisessa alan kirjallisuudessa tuottavat lisähämmennystä palkintofunktion nimikkeen suhteen: palkintofunktiosta käytetään nimikettä häviöfunktio yleensä kun tavoitteena on minimoida tämä funktio. Palkintofunktiota kuvataan notaatiolla $R(s, a, s')$. (Wiering ja Otterlo 2012, s. 11–12)



Kuvio 13. Agentin ja ympäristön vuorovaikutus

Kuviossa 13 on kuvattu, kuinka jokaisen toiminnon jälkeen ympäristö tuottaa agentille seuraavan tilan ja palkintoarvon. Agentti ja ympäristö vuorovaikuttavat toisiinsa jokaisella aika-

askeleella (*engl. time step*). Agentti vastaanottaa kuvauksen ympäristön tilasta, $s_t \in \mathcal{S}$, jokaisella aika-askeleella t. Tilan kuvaukseen perustuen agentti valitsee toiminnon, $a_t \in \mathcal{A}(s)$. Toiminnon seurauksena, yhden aika-askelen jälkeen, agentti vastaanottaa numeerisen palkinnon r_{t+1} ja löytää itsensä uudesta tilasta s_{t+1}. (Sutton ja Barto 2018, s. 48)

Toimintojen suorittamisesta muodostuu historiatiedon sisältämä sekvenssi, jota kutsutaan trajektoriksi (*engl. trajectory*). Trajektorin sisältö voidaan kuvata alkamaan seuraavanlaisesti (Sutton ja Barto 2018, s. 48):

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots$$

3.2.2 Arvofunktio

Tulevaisuuden palkintosignaaleja laskiessa käytetään vähennyskerrointa (*engl.discount rate*), jota kuvataan tunnuksella γ (Sutton ja Barto 2018, s. 55):

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

jossa vähennyskertoimen arvo on välillä $[0,1]$. Vähennyskerroin määrittää aika-askeleella t, agentin tulevaisuuden palkintojen painoarvon. Esimerkiksi jos vähennyskerroin olisi 0, niin agentti yrittäisi vain maksimoida välitöntä palkintoa. Agentti ottaa huomioon tulevaisuuden palkinnot yhä voimakkaammin kun vähennyskerroin lähestyy arvoa 1. (Sutton ja Barto 2018, s. 55)

Arvofunktion avulla saadaan arvio siitä, kuinka optimaalista on toteuttaa annettu toiminta annetussa tilassa kun otetaan huomioon tulevaisuuden odotetut palkinnot toiminnon toteuttamisen jälkeen. Toisin sanoen arvofunktio $V^\pi(s)$ palauttaa odotusarvon kun tilasta s jatketaan menettelytavalla π (Sutton ja Barto 2018, s. 58). Tämä nähdään kaavassa:

$$V^\pi(s) = \mathbb{E}_\pi\{G_t | s_t = s\} = \mathbb{E}_\pi\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\}, \forall s \in \mathcal{S}, \tag{3.10}$$

jossa $\mathbb{E}_\pi[\cdot]$ kertoo odotusarvon menettelytavalla π (Wiering ja Otterlo 2012, s. 16). Samalla tavalla voidaan määrittää Q-funktio, joka kertoo toiminnon a odotusarvon tilassa s menette-

lytavalla π (Sutton ja Barto 2018, s. 58):

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\{G_t \big| s_t = s, a_t = a\} = \mathbb{E}_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a\right\}$$

Yhtälö (3.10) voidaan esittää rekursiivisesti. Tätä rekursiivista yhtälöä kutsutaan nimellä *Bellmanin yhtälö* (Sutton ja Barto 2018, s. 59; Wiering ja Otterlo 2012, s. 16) :

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}\{G_t \big| s_t = s\} \\ &= \mathbb{E}_{\pi}\{r_{t+1} + \gamma G_{t+1} | s_t = s\} \\ &= \sum_{s'} T(s, \pi(s), s') \Big(R(s,a,s') + \gamma V^{\pi}(s') \Big) \end{aligned}$$

Bellmanin rekursiivinen yhtälö kertoo tilan välittömän palkinnon odotusarvon lisäämällä siihen kaikkien seuraavien tilojen (s') palkintoarvot kun jatketaan menettelytavalla π (Sutton ja Barto 2018, s. 59).

Tavoitteena on yleensä löytää menettelytapa, joka tuottaa eniten palkintoa. Tämä tarkoittaa arvofunktion maksimoimista kaikilla tiloilla $s \in S$. Eniten palkintoa tuottavasta menettelytavasta käytetään nimitystä *optimaalinen menettelytapa*, jota kuvataan merkinnällä π^*. Optimaalinen ratkaisu $V^*(s) = V^{\pi^*}$ noudattaa yhtälöä (Wiering ja Otterlo 2012, s. 16) :

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s') \Big(R(s,a,s') + \gamma V^*(s') \Big)$$

Optimaalista tila-arvo funktiota V^* käyttämällä saadaan selville optimaalinen menettelytapa tilassa s (Wiering ja Otterlo 2012, s. 16):

$$\pi^*(s) = \arg\max_{a} \sum_{s' \in S} T(s,a,s') \Big(R(s,a,s') + \gamma V^*(s') \Big)$$

Kyseistä funktiota kutsutaan Bellmanin optimaalisuusyhtälöksi (*engl. Bellman Optimality equation*). $\pi^*(s)$ määrittää, että tilan arvo optimaalisen menettelytavan johdosta täytyy olla yhtä suuri kuin odotettu paluuarvo parhaalla mahdollisella menettelytavalla kyseisessä tilassa. Tämä on ahne menettely, sillä se etsii aina parhaimman menettelytavan käyttämällä arvofunktiota V. (Wiering ja Otterlo 2012, s. 16–17)

Vastaavasti optimaalisen tila-toimintofunktion (*engl. state-action function*) eli optimaalisen Q-funktion arvo saadaan (Wiering ja Otterlo 2012, s. 17):

$$Q^*(s,a) = \sum_{s'} T(s,a,s')\Big(R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\Big)$$

Q^*-funktiota käyttämällä optimaalisen menettelytavan valinta tulee helpommaksi. Q-funktiota käyttämällä saadaan selville, kuinka hyvä tietty toiminto määrätyssä tilassa on. Toisin sanoen Q-funktioiden avulla päästään suoraan kiinni tila-arvo pareihin. Q-funktiot ovat käytännöllisiä, sillä ne sisältävät painotetun summan eri vaihtoehdoista. Tällöin agentin ei tarvitse erikseen tehdä laskemista seuraavien askelien tiloista, sillä Q-funktio sisältää väliaikaistallennetun tiedon optimaalisesta tuloksesta myös seuraavien askelten suhteen. (Sutton ja Barto 2018, s. 65; Wiering ja Otterlo 2012, s. 17)

Q^* ja V^* välinen yhteys saadaan tietoon (Wiering ja Otterlo 2012, s. 17):

$$Q^*(s,a) = \sum_{s' \in S} T(s,a,s')\Big(R(s,a,s') + \gamma V^*(s')\Big)$$
$$V^*(s) = \max_a Q^*(s,a)$$

Nyt optimaalisen menettelytavan valinta voidaan yksinkertaistaa muotoon (Wiering ja Otterlo 2012, s. 17):

$$\pi^*(s) = \arg\max_a Q^*(s,a),$$

joka kertoo, että paras menettelytapa on se, jolla on suurin palkintojen odotusarvo. Kyseinen odotusarvo siis huomioi myös menettelytavan toteuttamisen jälkeisten tilojen tuottamien palkintojen odotusarvot (Wiering ja Otterlo 2012, s. 17). Seuraavassa luvussa käydään tarkemmin läpi yksi tapa, jolla menettelytapaa voidaan päivittää.

3.3 Proksimaalisen menettelytavan gradienttimetodi

Menettelytavan gradienttimetodin tavoitteena on optimoida menettelytapa maksimoimalla saavutettava kokonaispalkinto. Menettelytavan gradienttimetodi arvioi toimintojen todennäköisyyttä jokaisella askeleella maksimoidakseen kokonaispalkintoa. Jos menettelytapa on heikko, toiminto on todennäköisesti virheellinen ja toiminnosta saatava palkintokin on todennäköisesti heikko. Palkinnon vastaanottamisen perusteella päivitetään toiminnon toden-

näköisyyttä kyseisessä tilassa. Arvioinnin haasteena on gradienttimenetelmässä se, että toiminnon suorittamisesta seuraava palkinto ei ole usein välittömästi havaittavissa vaan se on sidottuna tulevaisuuden palkintoketjuihin. (Aggarwal 2018, s. 391–392)

Gradienttimenetelmässä lasketaan arvio menettelytavan gradientista. Arvion laskeminen voidaan esittää häviöfunktiolla (Schulman, Wolski ym. 2017):

$$L^{\mathrm{PG}}(\theta) = \hat{\mathbb{E}}_t[log\pi_\theta(a_t|s_t)\hat{A}_t],$$

jossa π_θ on menettelytapa. Notaatio $\hat{A}_t$ on hyötyfunktio (*engl. advantage function*), joka kertoo valitun toiminnon hyötyarvon eli arvion valitun toiminnon suhteellisesta arvosta nykyisessä tilassa (Schulman ym. 2018):

$$\hat{A}_t = A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

Toisin sanoen $\hat{A}_t$ laskee, onko agentin uusi toiminto parempi kuin oletettu toiminto vai oliko se huonompi kuin oletettu toiminto. Negatiivinen arvio vähentää valitun toiminnon valitsemisen todennäköisyyttä, kun taas positiivinen arvio parantaa toiminnon valitsemisen todennäköisyyttä. (Schulman, Wolski ym. 2017)

Mikäli tälle häviölle L^{PG} suoritetaan useita askeleita käyttämällä samaa trajektoria, se tuottaa yleensä suuren menettelytavan päivityskoon, mikä puolestaan johtaa menettelytavan huononemiseen (Schulman, Wolski ym. 2017). Menettelytavan päivittämisen yhteydessä tulee siis varmistaa, ettei uuden menettelytavan päivityskoko ole liian suuri, jotta kyseinen ongelma voidaan välttää.

TRPO (*Trust Region Policy Optimization*) on ratkaisu, jossa rajoitetaan menettelytavan päivityksen kokoa seuraavanlaisesti (Schulman, Wolski ym. 2017):

$$\max_\theta \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right]$$

$$\text{sikäli kun} \quad \hat{\mathbb{E}}_t\left[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right] \leq \delta$$

Tässä θ_{old} on vanha menettelytapa ennen päivitystä. Merkintä δ on parametri, joka toimii päivityskoon rajoittimena (Schulman, Levine ym. 2017). TRPO käyttää Kullback-Leibler divergenssiä (KL), joka on yksi tapa mitata kahden todennäköisyysjakauman eroavaisuus (Murphy 2012, s. 57). Kullback-Leibler divergenssi määritellään seuraavanlaisesti (Murphy 2012, s. 57):

$$KL(p||q) = \sum_{k=1}^{K} p_k log \frac{p_k}{q_k},$$

jossa p ja k ovat verrattavia jakaumia. TRPO hyödyntää Kullback-Leibler divergenssiä kertomalla sen kertoimella β, jonka tarkoitus on toimia rangaistuskertoimena: mitä enemmän uusi menettelytapa eroaa vanhasta, sitä suurempi rangaistus (Schulman, Wolski ym. 2017):

$$\max_{\theta} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t - KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right]$$

On kuitenkin hankala löytää yksittäistä arvoa β, joka suoriutuu hyvin eri ongelmien kohdalla – tai edes yksittäisen ongelman kohdalla, jossa ominaisuudet muuttuvat oppimisen aikana.

Schulman, Wolski ym. (2017) esittelivät uuden algoritmin nimeltään **proksimaalisen menettelytavan optimointi** (*Proximal Policy Optimization*) (PPO), joka on noussut erittäin käytetyksi gradienttimenetelmän algoritmiksi. Merkinnällä $r_t(\theta)$ kuvataan uuden ja vanhan menettelytavan todennäköisyyssuhdetta: $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, jolloin siis $r(\theta_{old}) = 1$. Tulokseksi $r_t(\theta)$ saadaan suurempi kuin 1, mikäli uusi menettelytapa on todennäköisempi kuin vanha. Vanha menettelytapa on puolestaan todennäköisempi tuloksen ollessa välillä $[0,1]$. TRPO pyrkii maksimoimaan yhtälöä (Schulman, Wolski ym. 2017):

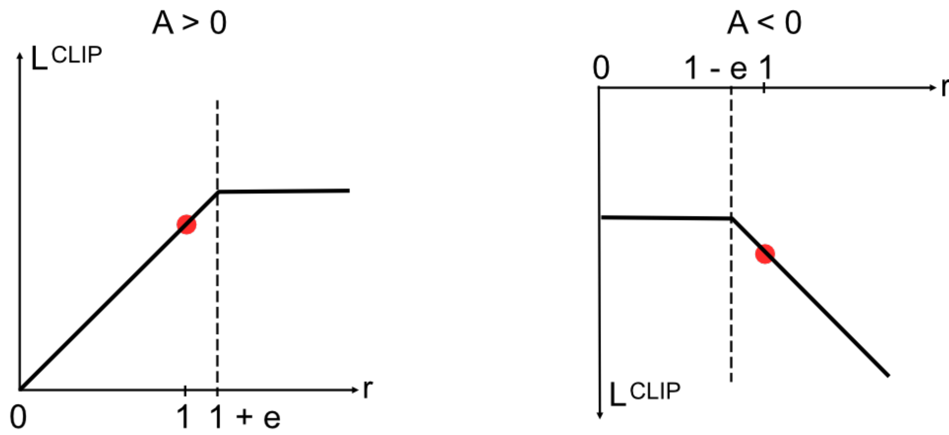
$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right], \tag{3.11}$$

jossa CPI viittaa menettelytavan iteraatioon. Ilman rajoitetta, L^{CPI} johtaa liian suureen menettelytavan päivityskokoon. PPO:ssa rangaistaan menettelytavan muutoksia, jotka liikuttavat $r_t(\theta)$ arvoa pois tuloksesta 1. Rangaistuksen tarkoitus on siis uuden ja vanhan menettelytavan muutoksen loiventaminen. PPO hyödyntää menettelytavan päivittämisen yhteydessä

yhtälöä (Schulman, Wolski ym. 2017):

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t) \right],$$

jossa epsilon, ε, on hyperparametri, joka voi olla esimerkiksi $\varepsilon = 0.2$. $\hat{\mathbb{E}}_t$ on odotusarvo PPO:n optimoimasta tavoitefunktiosta. Yhtälön sisällä oleva *min* kertoo, että leikattua ja leikkaamatonta funktiotulosta verrataan ja näistä kahdesta arvosta valitaan minimi. Ensimmäinen termi vastaa yhtälössä (3.11) esitettyä häviötä L^{CPI}. Toinen termi, $clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t$, leikkaa todennäköisyyssuhdetta, jolloin r_t pysyy intervallin $[1-\varepsilon, 1+\varepsilon]$ sisällä. (Schulman, Wolski ym. 2017)



Kuvio 14. Häviöfunktion L^{CLIP} todennäköisyyssuhteen leikkaus (mukaillen Schulman, Wolski ym. 2017)

Kuvio 14 näyttää leikkauksen toiminnan yhden aika-askeleen yhteydessä. Agentin toiminnan hyötysuhteen muutoksen ollessa liian suuri, oli se sitten positiivinen tai negatiivinen, leikataan muutosta, jottei päivitystä tehdä liikaa. Vasemmalla puolella nähdään agentin toiminnon odotusarvon olevan edellistä parempi. Tällöin liian suuri päivitys estetään, jottei lopulta päädyttäisiin huonoon lopputulokseen päivityksen suuruuden vuoksi. Oikealla puolella puolestaan agentin toiminnon odotusarvo on edellistä huonompi, jolloin myös liian suuri päivitys silti estetään, jottei yksittäinen arvio vaikuta menettelytavan teilaamiseen. Leikatun ja leikkaamattoman tavoitteen väliltä valitaan siis minimi. Mikäli valinnaksi muodostuu huonompi toiminto, se ei ole kuitenkaan ongelma, sillä lopulta kyseisestä toiminnosta tehdään

vähemmän todennäköinen: kun θ päivittyy niin myös θold päivittyy suhteessa siihen, kuinka paljon huonompi toiminto viime aika-askeleella suoritettiin. (Schulman, Wolski ym. 2017)

PPO voidaan nyt esittää iteratiivisella yhtälöllä (Schulman, Wolski ym. 2017):

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) - c_2 S[\pi_\theta](s_t) \right],$$

jossa c_1 ja c_2 ovat hyperparametrikertoimia, ja S kuvaa entropian bonusta, jonka avulla voidaan varmistaa, että agentin oppimisessa tapahtuu myös riittävästi uusien toimintojen etsintää (*engl. exploration*). L_t^{VF} puolestaan kuvaa arvofunktion keskimääräistä neliövirhettä: $L_t^{VF} = (V_\theta(s_t) - V_t^{targ})^2$. (Schulman, Wolski ym. 2017)

Algoritmi 1: PPO

```
1 for iteraatio = 1, 2, ... do
2     for toimija = 1, 2, ...,N do
3         Suorita menettelytapa π_θold ympäristössä T aika-askeleen verran
4         Laske hyötyennusteet Â_1, ..., Â_T
5     Optimoi häviötä L huomioimalla θ, K määrällä epokkeja ja minisarjakoolla (engl.
      minibatch size): M ≤ NT
6     θ_old ← θ
```

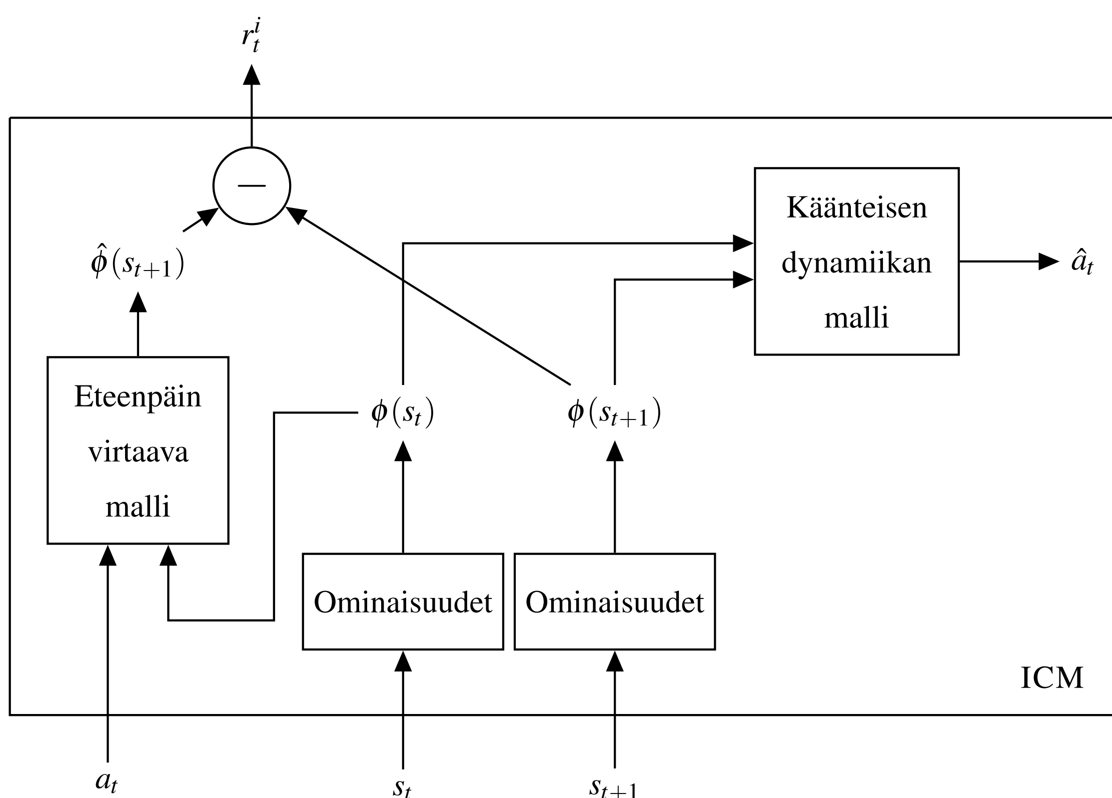
Algoritmista 1 (Schulman, Wolski ym. 2017) nähdään, että ensin sisimmäisessä silmukassa lasketaan arviot hyötysuhteista nykyisellä menettelytavalla. Sen jälkeen lasketaan kerätty jakso gradienttimenetelmää käyttäen. Kyseessä on siis vuorotteleva menetelmä.

3.4 Uteliaisuusperusteinen tutkiminen

Pathak ym. (2017) esittelivät agentin uteliaisuusperusteisen tutkimisen (*engl. curiosity-driven exploration*), joka tarkoittaa agentin ympäristön tutkimiseen kannustavaa sisäistä mallia. Kyseessä on siis agentin sisäinen dynaaminen malli, jossa agentti oppii omien toimintojen seuraukset. Useimmissa skenaarioissa ulkoiset palkinnot eli ympäristön tuottamat palkinnot tapahtuvat harvoin tai ne esiintyvät yhdessä, jolloin hankala muodostaa hyvin muotoiltua palkintofunktiota. Tällaisissa tapauksissa uteliaisuus voi toimia sisäisenä palkintosignaalina, jo-

ka mahdollistaa agentin tutkimaan ympäristöä ja oppimaan taitoja, jotka voivat olla hyödyllisiä myöhemmin. Uteliaisuusperusteisessa tutkimisessa on kaksi alijärjestelmää: palkintogeneraattori, joka tuottaa uteliaisuusperusteisen sisäisen palkintosignaalin ja menettelytapa, joka tuottaa sarjan toimintoja maksimoidakseen kyseisen palkintosignaalin. (Pathak ym. 2017)

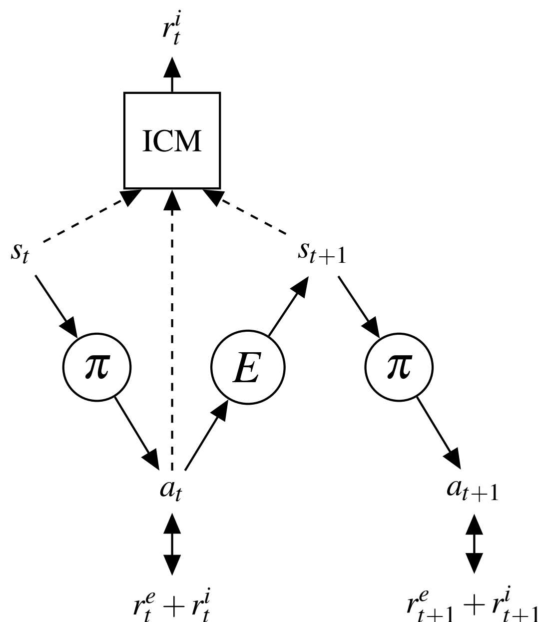
Sisäisten palkintojen lisäksi agentti voi vastaanottaa ulkoisia palkintoja vuorovaikuttavasta ympäristöstä (Pathak ym. 2017). Kuviossa 15 on kuvattu mallin sisäinen toimintalogiikka. Kuviossa 16 on puolestaan kuvattu mallin toiminta yhdistettynä ympäristöstä saataviin palkintoihin.



Kuvio 15. Sisäisen uteliaisuusmallin (ICM) toimintalogiikka (mukaillen Pathak ym. 2017)

Sisäistä uteliaisuuspalkintoa ajanhetkellä t kuvataan merkinnällä r_t^i. Ulkoista palkintoa puolestaan merkinnällä r_t^e. Menettelytavan alijärjestelmä on opetettu maksimoimaan näiden summa r_t. Ulkoisia palkintoja tapahtuu usein harvoin, jolloin myös niiden arvo on usein 0. Agentin ollessa nykyisessä tilassa s_t agentti vuorovaikuttaa ympäristön kanssa suorittamalla nykyisellä menettelytavalla toiminnon a_t päätyen tilaan s_{t+1}. Menettelytapa π on opetettu mak-

simoimaan tulos, jossa summataan ympäristön E tuottama ulkoinen palkinto r_t^e ja uteliaisuuteen perustuva sisäinen palkinto r_t^i. (Pathak ym. 2017)



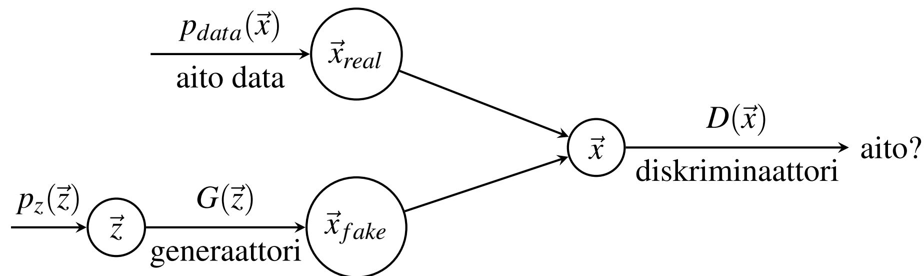
Kuvio 16. ICM yhdistettynä ulkoisiin palkintoihin (mukaillen Pathak ym. 2017)

Uteliaisuuspalkinnon tuottaa sisäinen uteliaisuusmoduuli (*engl. Intrinsic Curiosity Module*) (ICM), joka koostuu kahdesta osiosta: käänteisen dynamiikan mallista (*engl. inverse dynamics model*) ja eteenpäin virtaavasta mallista (*engl. forward model*). ICM muuntaa tilat s_t,s_{t+1} piirrevektoreiksi $\phi(s_t)$,$\phi(s_{t+1})$. Käänteisen dynamiikan malli arvioi toiminnon a_t, jolla agentti siirtyy tilasta s_t tilaan s_{t+1}. Tätä toiminnon ennustetta kuvataan merkinnällä $\hat{a}_t$. Eteenpäin virtaava malli vastaanottaa sisääntulot $\phi(s_t)$ ja a_t, jonka perusteella lasketaan tilan s_{t+1} piirrevektorin arvio $\hat{\phi}(s_{t+1})$. Käänteinen malli oppii piirreavaruuden (*engl. feature space*), joka muuntaa informaation relevantiksi ennustaakseen agentin toiminnot. Eteenpäin virtaava malli tekee ennustukset kyseissä piirreavaruudessa. Sisäisen uteliaisuuspalkinnon r_t^i palkintosignaali saadaan laskemalla ennustetun piirrevektorin $\hat{\phi}(s_{t+1})$ ja piirrevektorin $\phi(s_{t+1})$ erotus. (Pathak ym. 2017)

3.5 Generatiivinen kilpaileva verkko

Goodfellow ym. (2014) esittelivät GAN-menetelmän eli generatiivisen kilpailevan verkon. GAN-menetelmässä aluksi asetetaan opetusdata, jota menetelmässä verrataan neuroverkon tuottamaan synteettiseen dataan. Ideana on tuottaa synteettistä dataa, jota ei pysty erottamaan aidosta datasta. Kuviossa 17 nähdään generatiivisen kilpailevan verkon toimintaperiaate.

GAN-menetelmä voidaan jakaa kahteen ydinosaan: generaattoriin G ja diskriminaattoriin D. Generaattorin tarkoitus on luoda synteettistä dataa, kun taas diskriminaattorin tehtävä on päättää, onko generaattorin tuottama data aito vai keinotekoinen. Generaattori lähettää keinotekoisen datan diskriminaattorille analysoitavaksi. Jos diskriminaattori osaa päätellä, että data ei ole aito, niin generaattoria täytyy muokata, jotta tuotetun datan laatu paranee. Muokkaaminen tapahtuu käyttämällä vastavirta-algoritmia. (Campesato 2020, s. 153)



Kuvio 17. Generatiivisen kilpailevan verkon toimintaperiaate

Samanaikaisesti kun generaattoria muokataan, diskriminaattoria opetetaan maksimoimaan todennäköisyys generaattorin tuottaman datan aitouden tunnistamiseen. Toisin sanoen generaattori ja diskriminaattori pelaavat kahden pelaajan välistä minimax-peliä seuraavan funktion V(G,D) mukaisesti, jossa generaattori pyrkii minimoimaan funktion tuloksen, kun taas diskriminaattori pyrkii maksimoimaan sen. Tämä nähdään yhtälössä (Goodfellow ym. 2014):

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))],$$

jossa $p_z(z)$ kuvaa sisään tulevaa kohinadataa ja $P_{data}(x)$ alkuperäistä dataa. $D(x)$ on diskriminaattorin antama todennäköisyys sille, että onko data x oikeaa dataa. $D(G(z))$ on diskriminaattorin antama todennäköisyys sille, että onko generaattorin tuottama data oikeaa dataa.

$\mathbb{E}_x$ on odotusarvo alkuperäisestä datasta ja $\mathbb{E}_z$ on odotusarvo generaattorin tuottamasta synteettisestä datasta. (Goodfellow ym. 2014)

Diskriminaattori D pyrkii siis maksimoimaan todennäköisyyden sille, että se pystyy luokittelemaan oikein sekä alkuperäisen datan että generaattorin tuottaman datan. Generaattori G pyrkii minimoimaan funktion sisällä olevan lausekkeen $\log(1 - D(G(z))$ tuloksen. Alussa generaattori pärjää heikosti ja diskriminaattori pystyy helposti hylkäämään näytteet suurella luottamuksella, koska ne ovat selkeästi erilaisia verrattuna alkuperäiseen opetusdataan. Kun oppiminen etenee tarpeeksi diskriminaattori ei pysty enää tekemään eroa generaattorin tuottaman datan ja alkuperäisen datan välillä, jolloin $\mathbb{E}_{x \sim p_{data}(x)} = \mathbb{E}_{z \sim p_z(z)}$. (Goodfellow ym. 2014)

3.6 Imitaatio-oppiminen

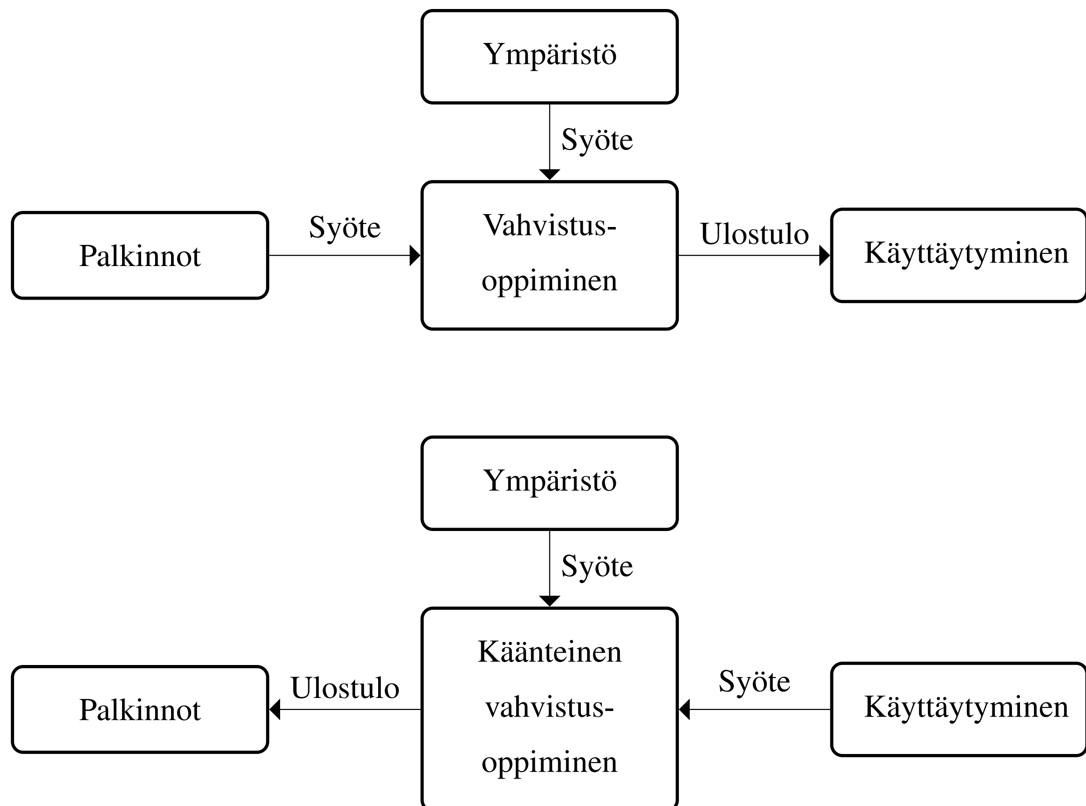
Imitaatio-oppiminen tarkoittaa oppimista annetuista asiantuntijan näytteistä eli trajektoreista. Imitaatio-oppimisessa on olemassa kaksi olennaista lähestymistapaa: käyttäytymisen kloonaaminen (*engl. behavior cloning*) ja käänteinen vahvistusoppiminen (*engl. inverse reinforcement learning*). Ho ja Ermon (2016) esittelivät uuden imitaatio-oppimisen menetelmän nimeltä GAIL, joka hyödyntää generatiivisten kilpailevien verkkojen perusperiaatetta.

Käyttäytymisen kloonaaminen on yleisnimike metodille, jossa ihmisen alikognitiiviset taidot voidaan kaapata ja toistaa uudelleen tietokoneohjelmassa. Ihmisen toiminnot nauhoitetaan ja nauhoitteiden logia käytetään syötedatana oppimisohjelmalle. Ohjelma tuottaa sarjan sääntöjä, jotka luovat uudelleen saman käyttäytymisen. Tätä menetelmää voidaan esimerkiksi käyttää rakentamaan automaattisia ohjausjärjestelmiä monimutkaisille tehtäville. (Sammut 2017)

3.6.1 Käänteinen vahvistusoppiminen

Käänteinen vahvistusoppiminen (*engl. inverse reinforcement learning*) tarkoittaa käyttäytymisen oppimista asiantuntijan (*engl. expert*) demonstraatiodatan pohjalta. Demonstraatiodata on siis syötedataa, johon pohjautuvaa käyttäytymistä agentti pyrkii imitoimaan mahdollisimman hyvin. Käänteisessä vahvistusoppimisessa on nimensä mukaisesti käänteinen

toimintalogiikka verrattuna vahvistusoppimiseen. Vahvistusoppimisen tarkoituksen ollessa käyttäytymisen tuottaminen niin, että maksimoidaan ennalta määrättyä palkintofunktion tuottamaa arvoa. Käänteinen vahvistusoppiminen pyrkii tuottamaan palkintofunktion tarkasteltavan käyttäytymisen perusteella. Tuotettua palkintofunktiota käytetään menettelytavan luomiseen. (Ng ja Russell 2000)



Kuvio 18. Vahvistusoppimisen ja käänteisen vahvistusoppimisen toimintalogiikan eroavaisuus

Käänteisen vahvistusoppimisen tapaukset voidaan jakaa pääasiasiallisesti kolmeen eri tyyppiin (Zhifei 2012):

- äärellinen Markovin päätösprosessi tunnetulla optimaalisella menettelytavalla
- ääretön Markovin päätösprosessi tunnetulla optimaalisella menettelytavalla
- ääretön Markovin päätösprosessi tuntemattomalla optimaalisella menettelytavalla, jossa demonstraatiot ovat annettu.

Näistä tapauksista viimeinen on lähimpänä käytännöllisempää ongelmaa, sillä yleensä vain

asiantuntijan demonstraatiodata on saatavilla, eikä optimaalista menettelytapaa tunneta. Käänteisessä vahvistusoppimisessa oletetaan, että demonstraatiodata on optimaalinen menettelytapa π^*. Demonstraatiodata sisältää seuraavat tiedot, joiden pohjalta agentille on tarkoitus löytää palkintofunktio R^*, joka selittää asiantuntijan käyttäytymisen (Zhifei 2012):

- äärellinen tila-avaruus $\mathcal{S}$
- sarja toimintoja $\mathcal{A} = a_1, a_2, ...a_k$
- siirtymän todennäköisyys $P^a_{ss'}$
- vähennyskerroin γ
- menettelytapa π.

3.6.2 Generatiivinen kilpaileva imitaatio-oppiminen

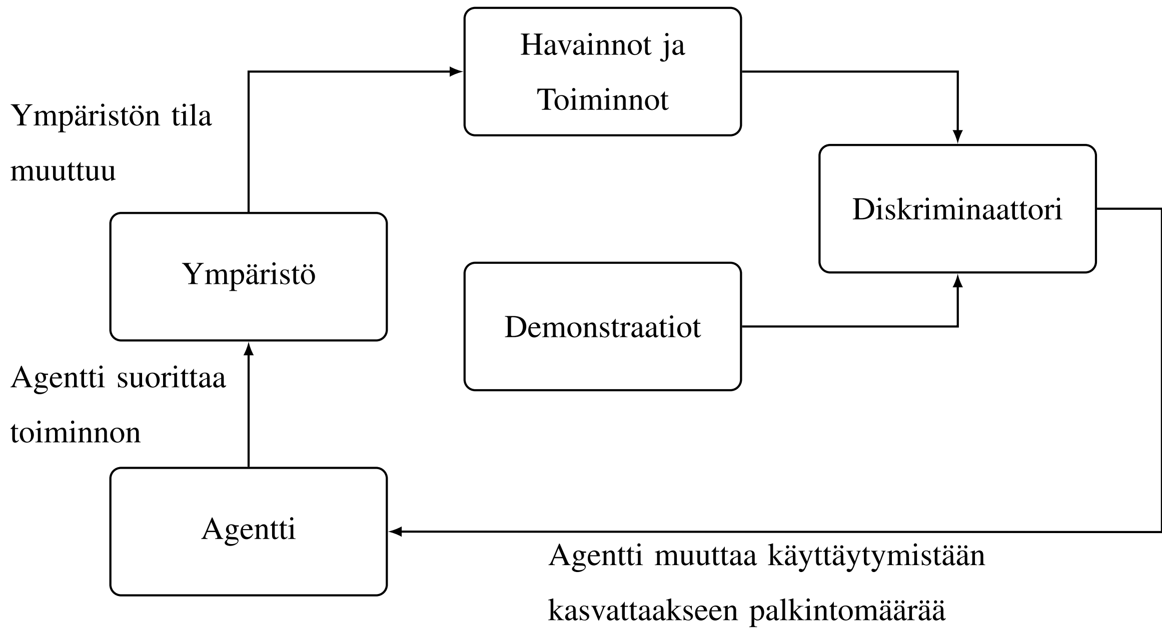
Käyttäytymisen kloonauksessa ja käänteisessä vahvistusoppimisessa on tiettyjä heikkouksia. Käyttäytymisen kloonaus onnistuu yleensä vain hyvin suurella määrällä dataa. Lisäksi se ei yleensä toimi muuttuvassa ympäristössä. Käänteinen vahvistusoppiminen oppii häviöfunktion, joka selittää asiantuntijan käyttäytymisen, mutta ei suoraan kerro oppijalle, kuinka tulee toimia. Käänteisen vahvistusoppimisalgoritmien ajaminen on erittäin hidasta, sillä se tarvitsee vahvistusoppimisen ajamisen sisäisessä silmukassa. (Ho ja Ermon 2016)

Ho ja Ermon (2016) esittivät uuden menetelmän, GAIL-menetelmän eli generatiivisen kilpailevan imitaatio-oppimisen (*engl. Generative Adversarial Imitation Learning*). GAIL yhdistää käänteisen vahvistusoppimisen ja generatiivisen kilpailevan verkon taustalla olevat ideat. GAIL-metodissa generaattorina toimii syväoppimisen metodi, joka generoi menettelytapoja, ja jossa diskriminaattori laskee generoidun menettelytavan ja demonstraation menettelytavan eron. (Zuo ym. 2020)

GAIL-menetelmässä, kuten GAN-menetelmässäkin, verrataan generaattorin tuottamaa havaintoa diskriminaattorin havaintoon. Generatiivinen malli on tämän tutkielman yhteydessä ympäristössä toimiva agentti. Diskriminaattorin työ on erotella generatiivisen mallin tuottama data oikeasta datasta. Kun Diskriminaattori ei pysty enää erottelemaan dataa, silloin generaattori on onnistuneesti jäljitellyt oikeaa dataa (Ho ja Ermon 2016).

Kuviossa 19 nähdään GAIL-menetelmän toiminta. Diskriminaattori vertailee demonstraa-

tiodatan havaintoja ja toimintoja agentin vastaaviin. Agentille annetaan palkinto sen perusteella kuinka hyvin sen toiminta vastaa demonstraatiodataa. GAIL-menetelmän algoritmikuvaus on nähtävissä algoritmissa 2 (Ho ja Ermon 2016).



Kuvio 19. GAIL-menetelmän toimintaperiaate [1]

GAIL-metodissa generaattoria kuvataan merkinnällä G_θ ja diskriminaattoria merkinnällä D_ω, joissa θ ja ω ovat parametreja. GAIL päivittää parametria ω gradienttiaskeleella, ja puolestaan parametria θ TRPO-askeleella (Ho ja Ermon 2016). Generaattorin tehtävänä on tuottaa menettelytapa $\pi_\theta(a|s)$. Tuotettu menettelytapa perustuu agentin nykyiseen tilaan s. Menettelytavan perusteella tuotetaan toiminto a_t, jonka agentti suorittaa. Diskriminaattori määrittää onko toiminto a_t nykyisessä tilassa s_t demonstraatiodatan toiminto vai generaattorin tuottama toiminto. Tuloksena diskriminaattori tuottaa todennäköisyyden sille, kuinka hyvin a_t vastaa demonstraation toimintoa. (Zuo ym. 2020)

Optimointitavoite diskriminaattorilla D_ω on minimoida:

$$\hat{\mathbb{E}}_{\tau_E}[\nabla_\omega \log(D_\omega(s,a))] + \hat{\mathbb{E}}_{\tau_D}[\nabla_\omega \log(1 - D_\omega(s,a))],$$

[1] Unity Blog: https://blogs.unity3d.com/2019/11/11/training-your-agents-7-times-faster-with-ml-agents

jossa $\tau_E \sim \pi_{\theta_i}$ on generaattorin trajektoreita, ja $\tau_D \sim \pi_D$ on demonstraatiodatan trajektoreita (Zuo ym. 2020). Päivityskertojen edetessä diskriminaattori osaa yhä paremmin erottaa generoidun toiminnon demonstraation toiminnosta. Optimointitavoite generaattorilla G_θ on maksimoida lauseke:

$$\hat{\mathbb{E}}_{\tau_E}[\nabla_\theta \log(D_{\omega_{i+1}}(s,a))]$$

Generaattorin päivittäminen tuottaa menettelytavan, joka kaventaa generatiivisen menettelytavan ja demonstraation menettelytavan etäisyyttä, jotta se pystyisi lopulta huijaamaan diskriminaattoria. Generaattorin ja diskriminaattorin opettamisen seurauksena saavutetaan lopulta tila, jossa kumpikaan ei voi enää saavuttaa parempaa lopputulosta. Tällöin generaattorin toiminto on lähellä demonstraation toimintoa ja imitaatio-oppimisen tavoite agentilla on saavutettu. (Zuo ym. 2020)

Algoritmi 2: Generatiivinen kilpaileva imitaatio-oppiminen

Input: Asiantuntijan trajektorit $\tau_E \sim \pi_E$, alustava menettelytapa ja diskriminaattorin parametrit θ_0, ω_0

1 **for** *i = 0, 1, 2, ...* **do**

2 Näytetrajektorit $\tau_i \sim \pi_{\theta_i}$

3 Päivitä diskriminaattorin parametriä ω_i gradientilla, jolloin päivityksen jälkeen parametri on ω_{i+1}

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_\omega \log(D_\omega(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_\omega \log(1 - D_\omega(s,a))]$$

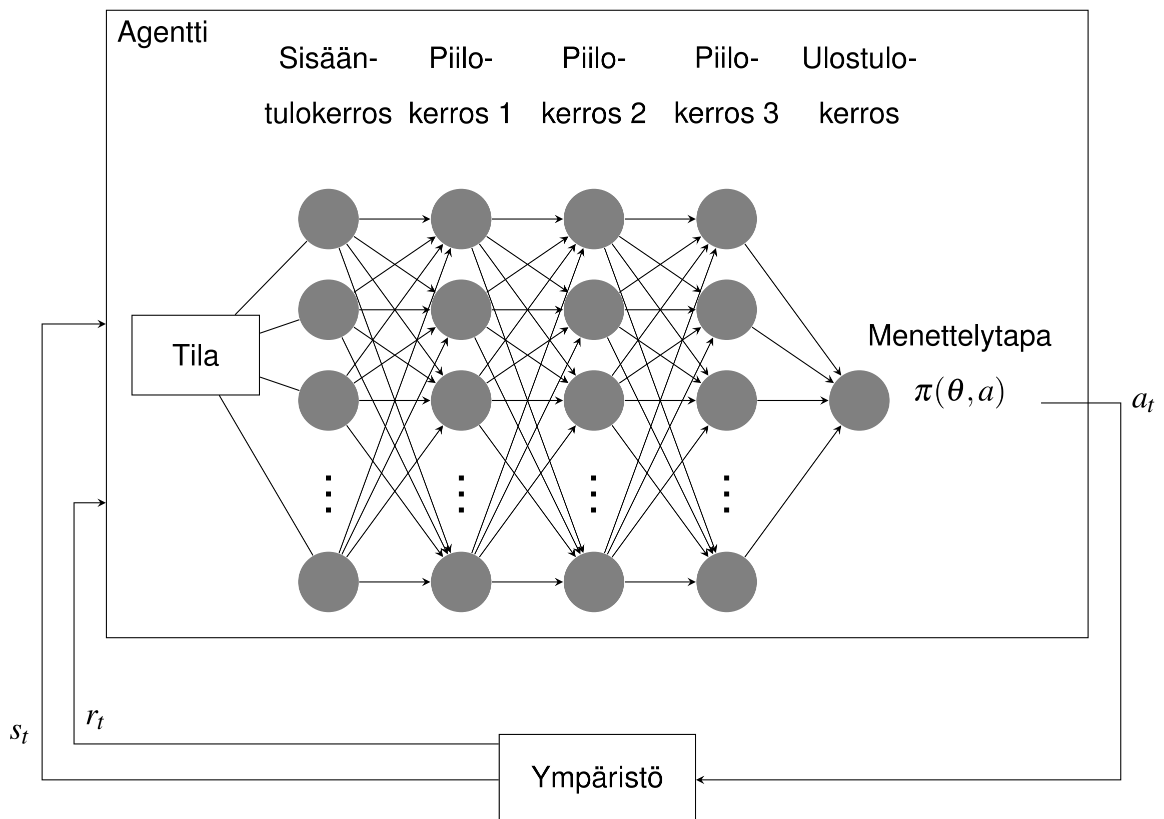
4 Päivitä menettelytapaa askeleesta θ_i seuraavan askeleeseen θ_{i+1}, käyttämällä TRPO-sääntöä häviöfunktiolla $\log(D_{\omega_{i+1}}(s,a))$. Tarkemmin sanottuna ota gradienttiaskel, jossa on käytetty KL-divergenssiä:

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_\theta \log \pi_\theta(a|s) Q(s,a)] - \lambda \nabla_\theta H(\pi_\theta),$$

$$\text{jossa } Q(\bar{s},\bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{\omega_{i+1}}(s,a)) | s_0 = \bar{s}, a_0 = \bar{a}]$$

3.7 Syväoppiminen

Campesato (2020, s. 99) määrittelee syväoppimisen niin, että syväoppiminen sisältää vähintään kaksi piilokerrosta neuroverkossa. Syväoppimiseen liittyy usein suuri opetusdata-aineisto ja syväoppimisen arkkitehtuurit nojaavat perseptroneihin neuroverkkojen perustana (Campesato 2020, s. 19). Syvä vahvistusoppiminen (*engl. deep reinforcement learning*) yhdistää syväoppimisen ja vahvistusoppimisen. Campesato (2020, s. 19) esittää, että syvän vahvistusoppimisen ero perinteiseen vahvistusoppimiseen voidaan yksinkertaistettuna kuvata niin, että agentin opettamisessa käytetään syvää neuroverkkoa. Verrattuna perinteiseen vahvistusoppimiseen, syväoppimisen avulla agenttia voidaan opettaa hyvin moniulotteisella datalla, mikä mahdollistaa haastavampien tehtävien suorittamisen ja pitemmän aikavälin tarkastelun, jotta agentti voi oppia tehokkaammin aikaisemmista kokemuksistaan (Mnih ym. 2015). Kuviossa 20 on havainnollistettu kolmella piilokerroksella syvän vahvistusoppimisen toiminta. Agentti vuorovaikuttaa ympäristön kanssa ja sisääntulokerros saa syötteenä agentin sen hetken tila-arvot. Menettelytapaa päivitetään neuroverkon avulla.



Kuvio 20. Syvän vahvistusoppimisen perusperiaate

Syväoppimisen avulla on saavutettu erittäin hyviä tutkimustuloksia haasteellisissa ympäristöissä. Syväoppimisen avulla opetettu agentti on oppinut muun muassa pelaamaan Atari 2600 -pelikonsolien pelejä (Mnih ym. 2013), Go-peliä (Silver ym. 2016), jossa agentti päihitti Euroopan mestaruuksiakin voittaneen ammattilaispelaajan sekä Dota 2 -peliä (Berner ym. 2019), jossa tekoälyjärjestelmä päihitti pelin maailmanmestarit. Näistä tutkimustuloksista kerrotaan tarkemmin luvussa 3.9.

3.8 Koneoppiminen Unity-pelimoottorissa

Viime vuosina on tapahtunut merkittävää edistymistä tutkimuksessa ja algoritmien suunnittelussa syväoppimisen saralla. Simulaatioympäristöt ja -alustat ovat auttaneet motivoimaan tutkijoita tehokkaampien algoritmien pariin. Simulaatioympäristöllä tarkoitetaan tutkimusalustaa, jossa vahvistusoppimisen yhteisö testaa ideoita ja algoritmeja. Moni nykyisistä tutkimusalustoista pohjautuu suosittuihin videopeleihin tai pelimoottoreihin. Tämä on osa pitempiaikaista trendiä, jossa pelit ovat palvelleet alustoina tekoälyn tutkimusta. Tämä trendi voidaan jäljittää ihan aikaisimpiin tekoälyn saralla tehtyihin töihin, joissa tekoälyratkaisuja sovelletaan pelien pelaamiseen. Tekoälyn toteuttamisen haasteet tarvittavassa etsinnässä, päätöksenteossa ja suunnittelussa peleihin liittyen, ovat myös samoja haasteita, jotka kiinnostavat tekoälyn tutkijoita. Tämä on motivoinut monenlaisia tutkimuksia videopelien ja tekoälyratkaisujen yhdistämiseen, kuten pelien pelaamiseen, pelaajan mallintamiseen ja sisällön luomiseen liittyviä tekoälytutkimuksia. (Juliani ym. 2020, s. 2)

Nykyaikaiset pelimoottorit ovat tehokkaita työkaluja simuloimaan visuaalisesti realistisia maailmoja pitkälle kehitetyillä fysiikoilla ja monimutkaisilla vuorovaikutuksilla agenttien ja muuttuvan ympäristön välillä. Lisäksi pelimoottorit tarjoavat käyttöliittymän, joka on kehitetty olemaan intuitiivinen, helppokäyttöinen ja saatavilla monelle alustalle. Viime vuosina on kehitetty useita simulaatioalustoja, joiden tarkoitus on tuottaa haasteita ja vertailuanalyysejä (*engl. benchmarks*) syväoppimisen algoritmeille. Moni näistä alustoista pohjautuu jo olemassa oleviin peleihin tai pelimoottoreihin, sisältäen niiden tiettyjä vahvuuksia ja heikkouksia. Toisinkuin moni tutkimusalusta, Unity-pelimoottori tarjoaa yleisen alustan, joka ei ole sidottuna vain tietyn tyyppisen pelattavuuden tai simulaation toteutukseen. (Juliani ym. 2020, s. 8)

Unityn ML-Agents [2] on avoimen lähdekoodin projekti, joka mahdollistaa tutkijoille ja kehittäjille koneoppimiseen suunnattujen simuloitujen ympäristöjen luonnin Unity editorissa ja vuorovaikuttamisen Python-ohjelmointirajapintaa käyttäen. ML-Agents sisältää kaiken toiminnallisuuden, jolla voidaan määritellä oppimisprosessi Unityn editorin sisällä käyttäen C#-ohjelmointikieltä. ML-Agents mahdollistaa demonstraatioiden nauhoittamisen opettamista varten. Kehittäjä voi aloittaa simulaation ja kontrolloida yhtä tai useampaa agenttia generoiden asiantuntija demonstraatioita, joita käyttää opettamiseen ja imitaatio-oppimisen algoritmien arviointiin. Saatavilla oleva Python-ohjelmistopaketti sisältää rajapinnan Unityn ympäristöön. Python toteutus mahdollistaa myös esimerkiksi olemassa olevien vahvistusoppimisalgoritmien liittämisen ML-Agents ympäristöön rajapinnan avulla. (Juliani ym. 2020, s. 10-13)

ML-Agents ohjelmistopaketti sisältää agenttityypin luokan, joka mahdollistaa peliobjektin määrittämisen agentiksi. Agentti-peliobjekti kerää havainnot, tekee toiminnot ja vastaanottaa palkinnot. Agentti voi kerätä havaintoja käyttämällä erilaisia sensoreja, jotka keräävät informaatiota eri tavoin. Sensorit voivat kerätä tietoa muun muassa renderöidystä kuvasta, säteensuuntaustuloksista tai mielivaltaisilla pituusvektoreilla. (Juliani ym. 2020, s. 12)

Agentit voivat myös päivittää samaa menettelytapaa. Tällöin siis agentit suorittavat samaa menettelytapaa ja jakavat kokemusdataa oppimisen aikana. Tämä mahdollistaa rinnakkaisen opetuksen usealla agentilla. Lisäksi skenessä [3] voi olla useita agentteja, jotka suorittavat eri menettelytapoja, mahdollistaen näin moniagenttiskenaarion, jossa esimerkiksi agentin oppimiseen voi vaikuttaa toisen agentin toiminnot. (Juliani ym. 2020, s. 12)

Unityssä fysiikan simulointi ja kuvan renderöinti tapahtuu asynkronisesti. Näin ollen on mahdollista nopeuttaa simuloinnin nopeutta ilman tarvetta nopeuttaa kuvan renderöintiprosessia. On myös mahdollista ajaa simulaatioita ilman renderöintiä, mikäli siihen ei ole tarvetta. Skenaarioissa, joissa renderöinti on toivottavaa, on myös mahdollista kontrolloida ruudunpäivitysnopeutta (*engl. frame rate*) ja pelilogiikan nopeutta. (Juliani ym. 2020, s. 10)

[2] ML-Agents GitHub sivusto: https://github.com/Unity-Technologies/ml-agents

[3] Unityn skenen kuvaus ja dokumentaatio: https://docs.unity3d.com/Manual/CreatingScenes.html

3.9 Aihealueen tutkimukset

Vaikeusasteiden ollessa epäsuhdassa pelisuunnittelijan odotuksiin nähden voidaan tätä epäsuhtaisuutta kutsua myös pelin balanssiin liittyväksi bugiksi. Zheng ym. (2019) esittävät, että pelien eri bugit voidaan kategorisoida viiteen eri luokkaan:

- pelin kaatumiseen johtavat bugit
- pelin jumiutumiseen johtavat bugit
- loogiset bugit
- pelin balanssiin liittyvät bugit
- käyttäjäkokemukseen liittyvät bugit.

Loogiset bugit eivät yleensä riko peliä, mutta johtavat odottamattomiin tuloksiin, kuten virheisiin pistelaskussa. Tämän tyyppiset bugit johtuvat yleensä pelilogiikan virheellisestä toteutuksesta. Pelin balanssiin liittyvät bugit tarkoittavat epäsuhtaa pelin balanssissa suunnittelijan odotuksiin nähden. Käyttäjäkokemukseen liittyvät bugit alentavat käyttäjäkokemusta. Tällaisia bugeja voivat olla esimerkiksi puuttuvat ääniefektit, puutteelliset ohjetekstit tai kirjoitusvirheet. (Zheng ym. 2019)

Pelin dynamiikka on liian kompleksinen, jotta sen voisi täysin formalisoida. Edes pelisuunnittelijat eivät pysty suunnittelemaansa peliä täysin formalisoimaan, sillä usein pelit haarautuvat valtavasti ja niissä on satunnaisuutta. Näin ollen on tarpeellista rakentaa tekoälyyn pohjautuva agentti tutkimaan tätä dynamiikkaa. (Aponte, Levieux ja Natkin 2009)

Roohi ym. (2019) esittivät uuden simulaatiomallin, jonka avulla voidaan ennustaa ne kentät, joissa pelaajat todennäköisesti luovuttavat koko pelin pelaamisen suhteen. Heidän tutkimuksensa koski *Angry Birds Dream Blast* -peliä, jonka pelaajadataan heillä oli pääsy. Tutkimuksessa he tutkivat syväoppimismenetelmällä kehitetyn agentin hyödyntämistä yhdistettynä pelaajadataan, jossa käy ilmi, kuinka pelaajamäärä kehittyy pelin eri kentissä. Kehitetyn mallin perusteella he saivat tulokseksi, että opetettua agenttia voidaan käyttää ennustamaan pelaajien todennäköisimmin luovuttamiseen johtavat kentät, esimerkiksi kentän liiallisen haasteellisuuden tai päinvastoin kyllästymisen vuoksi, mikä voi johtua kenttien liiallisesta helppoudesta. Vaikeustason mittaamiseen he käyttivät kentän läpäisyprosenttia, joka lasketaan onnistumisen ja kaikkien yritysten lukumäärien suhteesta.

Myös Kristensen ja Burelli (2020) käyttävät tutkimuksessaan kentän läpäisyn prosenttiosuutta agentin validointiin. He vertaavat sitä ihmispelaajien vastaavaan tulokseen. Heidän tutkimuksensa koski vahvistusoppimisen hyödyntämistä pelitestauksessa Unityn ML-Agents ympäristöä käyttäen. Menettelytavan optimointiin he käyttivät proksimaalista menettelytavan optimointialgoritmia. He sovelsivat vahvistusoppimista *Lily's Garden* -puzzlepeliin. Tutkimuksen yhtenä tarkoituksena oli selvittää, voisiko agentin käyttäminen kyseisen genren pelitestauksessa olla hyödyllistä tuotannossa kenttäsuunnittelijoiden kannalta. Tutkimuksessa he opettivat agenttia valikoiduissa kentissä. He nostavat esille, että kenttäsuunnittelijan tulisi luottaa koneopetetun agentin toimintaan, jotta sen käyttämisestä on konkreettista hyötyä. Tuloksista kävi ilmi, että agentin läpäisyprosentti on huonompi niissä kentissä, jotka eivät ole agentille tuttuja opetuksen kautta, joten ennakkoon tuntemattomien kenttien kohdalla agentin käyttäminen on tältä osin ongelmallista kenttäsuunnittelijalle.

Myös Gudmundsson ym. (2018) toteuttivat ennustemallin, jonka avulla koneopetetun agentin avulla voidaan ennustaa ihmispelaajien onnistumisprosentti *Candy Crush Saga* -pelissä. Tutkimus suuntautui pelin genren mukaisesti *Match-3* tyyppisiin puzzlepeleihin. Heillä oli käytössä pelaajadataa, jonka pohjalta he toteuttivat mallin. Mallissa he mittaavat vaikeutta kentän läpäisyprosentilla. Ennustemallin tuloksien perusteella he esittävät, että mallin avulla voidaan ennustaa uuden kentän vaikeusaste alle minuutissa, kun taas perinteisen ihmispelaajilla toteutettavan pelitestauksen he kertovat kestävän seitsemän päivää.

Bergdahl ym. (2020) tutkivat syväoppimisen hyödyntämistä yleisemmin automatisoituun pelitestaukseen. Heidän työssään agentti osoittautui hyödylliseksi esimerkiksi bugien löytämisen suhteen. Agentin avulla myös pystyi huomaamaan bugit, joita hyödyntämällä agentti pystyi saavuttamaan paremman lopputuloksen. He loivat myös lämpökartan (*engl. heat map*), jonka avulla voidaan nähdä agentin navigointi pelikentässä. Tämä voi olla hyvin hyödyllinen visualisoitu tieto kentän balansoinnin kannalta. Vaikeustason määrittämisen suhteen he puolestaan esittivät, että aikaa, joka agentilla kuluu tehtävän hallitsemiseen, voidaan pitää indikaattorina siitä, kuinka vaikea peli voisi olla ihmispelaajalle. He esittävät johtopäätöksessään, että vahvistusoppimista voidaan hyödyntää pelitestaukseen hyvin, kun sen integrointi on modulaarista. Tällöin agentin opettaminen on huomattavasti helpompaa ja nopeampaa, kun se opetetaan suoriutumaan yksittäisistä tehtävistä, jotka on eriytetty kokonaisen pelin

laajemmasta logiikasta.

De Mesentier Silva ym. (2017) toteuttivat eri pelityylin agentteja, joiden käyttäytymistä he testasivat *Ticket To Ride* -pelissä. *Ticket To Ride* on suosittu lautapeli, josta on julkaistu myös digitaalinen versio. He osoittivat, että heidän menetelmällään on mahdollista havainnoida pelimaailman osien keskinäistä balanssia, sekä nähdä eri pelistrategioiden suhteellisia vahvuuksia eri pelaajamäärillä ja havainnoida eri maailmojen eroja voittavan pelistrategian osalta. Agentin avulla voidaan myös helposti nähdä pelin logiikkaan tehtyjen muutosten vaikutus. Tutkimuksen ohessa he myös huomasivat, että agentti löysi pelitiloja, joita ei ollut katettu pelisäännöillä.

Isaksen, Gopstein ja Nealen (2015) osoittivat menetelmässään, että käyttämällä tekoälyn ohjastamaa pelaajaa voidaan tekoälyn suoriutumisen perusteella muokata haasteparametreja halutun vaikeusasteen saavuttamiseksi. He myös esittävät, että tekoäly voisi tehdä tarvittavat parametrien muutokset, jolloin pelisuunnittelu nopeutuisi entisestään. He arvioivat agentin toimintaa elinaika-analyysimenetelmällä ja vertasivat agentin pistekertymää ihmispelaajien tuloksiin käyttäjätestien perusteella. Heidän kokeensa kohdistui tunnettuun *Flappy Bird* -peliin, joka on pelimekaniikoiltaan minimaalinen pistekertymäpohjainen peli. Tutkimuksessa heillä oli saatavilla parametrisoitu versio alkuperäisestä pelistä.

Tekoälytutkimuksiin erikoistunut yritys nimeltään **OpenAI** puolestaan kehitti tekoälyjärjestelmän *OpenAI Five* [4], joka muun muassa päihitti *Dota 2* -pelin maailmanmestarit. OpenAI Five perustuu syväoppimiseen, jonka avulla he opettivat joukkueellisen agentteja pelaamaan Dota 2 -peliä. Dota 2 on tiimipohjainen peli, jossa kaksi viiden henkilön joukkuetta pelaavat vastakkain. Tekoälyjärjestelmän kehitykseen heillä meni kaiken kaikkiaan aikaa 10 kuukautta. Kehityksen aikana he tekivät kuitenkin muutoksia muun muassa opetukseen liittyen, sillä kehityksen aikana opetusprosessissa huomattiin parannettavaa, kuten esimerkiksi ylisovittumista, kun tietty opetusinformaatio toistui liian useasti. Resurssien ja pitkän opetukseen kuluvan ajan vuoksi opetuksen aloittaminen alusta ei olisi ollut mahdollista jokaisen muutoksen jälkeen. Myös itse peliin tuli kehityksen aikana erilaisia pelattavuuteen liittyviä päivityksiä, jotka lisättiin kehitysvaiheessa mukaan tekoälylle opetettavaksi. Tutkimuksen lopuksi toteutettiin kuitenkin vielä uusi opetusajo lopullisessa opetusympäristössä. Tämän ajon kesto oli

[4] OpenAI Five -projektin verkkosivusto: https://openai.com/projects/five/

2 kuukautta. (Berner ym. 2019)

OpenAI Five:n yksi tärkeimmistä menestykseen johtavista tekijöistä on muun muassa sen kyky oppia pitkältä aikajaksolta sekä kompleksisista tilojen ja toimintojen kombinaatiovaihtoehdoista. Dota 2 -peli asetti hyvin pitkän aikariippuvuuden ongelman: kun moni vahvistusoppimisen jakso kestää tuhansia aika-askelia, Dota 2 -pelisessiot voivat kestää kymmeniä tuhansia aika-askelia. OpenAI Five käyttää proksimaalista menettelytavan optimointia. (Berner ym. 2019)

Muutaman viime vuoden aikana koneoppimisen hyödyntämistä on siis tutkittu pelialan eri osa-alueisiin lupaavin tuloksin. Lopputulosta arvioidessa agentin toimintaa verrataan usein luonnollisesti pelaajadataan. Kuten OpenAI:n tutkimuksesta käy kuitenkin ilmi, kompleksisen ja haastavan pelin opettamiseen liittyvä kehitystyö voi sisältää useita iteraatioita, ja näin ollen kehitysaika voi kasvaa suhteellisen suureksi. Seuraavassa luvussa kuvataan tämän tutkimuksen kehitystyön eri askeleet ja havainnot, joiden perusteella toteutettiin lopullinen opetusympäristö parametreineen.

4 Tutkimus

Tutkielmassa suoritettiin kvantitatiivinen tutkimus, joka sisälsi vertailevan tutkimuksen ja kokeellisen tutkimuksen. Tutkielmassa verrattiin koneopetetun agentin suoriutumista suhteessa käyttäjädataan. Käyttäjädata kerättiin käyttäjätestien muodossa. Koneoppimisen menetelmänä käytettiin syväoppimisen ja imitaatio-oppimisen yhdistelmää niin, että menetelmien tuottamia palkintosignaaleja painotettiin määritellyillä kertoimilla. Simulaatioympäristönä käytettiin Unity-pelimoottoria ja agentin opettamisen välineenä toimi ML-agents, joka on Unitylle toteutettu koneoppimisen avoimen lähdekoodin ohjelmistopaketti. Menettelytavan optimointialgoritmina käytettiin proksimaalista menettelytavan optimointia eli PPO:ta. Aktivaatiofunktiona käytettiin Swish-funktiota, joka on myös ML-Agents ohjelmistopaketin oletusfunktio [1].

Tutkimusta varten toteutettiin prototyyppi, jonka katsottiin vastaavan bullet hell -genreä. Tarkoitus oli selvittää, mukaileeko opetetun agentin suoriutuminen eri kentissä niiden vaikeustasoa. Tuloksien perusteella arvioitiin, voidaanko koneopetetun agentin avulla löytää bullet hell -genren vaikeustasojen erot eri kenttien välillä eli onko agenttia mahdollista käyttää peli- ja kenttäsuunnittelun apuvälineenä vaikeustasojen määrittämisessä. Toisin sanoen arvioitiin, pystytäänkö kehittämään tarpeeksi luotettava koneopetettu agentti, jonka avulla voidaan saada tietoon kenttien väliset vaikeustasot ja sen pohjalta tehdä kenttiin tarvittavia muutoksia. Konkreettinen esimerkki tällaisen agentin hyödyntämiselle on tilanne, jossa halutaan pelin toteuttavan flow-teoria mukaista tasaisesti nousevaa haastavuutta.

Bullet hell -genressä, kuten myös muissa suuren haasteen tuottamiseen pyrkivissä genreissä, kenttien vaikeustasoerot voivat olla hankalasti hahmotettavia, joten parhaassa tapauksessa automatisoitu testaus voi olla ratkaisu tähän ongelmaan. Agentin opettamismenetelmiin liittyen tehtiin alustavaa tutkimusta, jossa selvitettiin, miten agentti oppii parhaiten. Esitutkimuksessa testattiin eri hyperparametriarvoja, agentille annettavien havainnoitavien asioiden vaikutusta oppimiseen sekä sitä, minkälainen oppimisympäristö mahdollistaa parhaimmat oppimistulokset.

[1] ML-Agents lähdekoodi: https://github.com/Unity-Technologies/ml-agents/blob/release_5/ml-agents/mlagents/trainers/models.py#L92-L95

4.1 Pelisuunnittelu

Prototyyppi kehitettiin itsenäisesti grafiikkoja myöten, lukuun ottamatta ääniä, joina käytettiin ulkopuolisten äänikirjastojen musiikkitiedostoja. Kehittämällä prototyypin eri osa-alueet itsenäisesti, tarkoitus oli myös pyrkiä kehitystyöhön, joka mukailee käytännön tilannetta indie-kehittäjän asemassa. Prototyyppiä kehitettiin iteratiivisesti käyttäen yhtä genren koehenkilöä pelitestaajana.

Lopullinen versio laitettiin palvelimelle, josta käyttäjä pystyi sen lataamaan omalle koneelle. Käyttäjätestejä varten prototyyppiin toteutettiin myös pelitietojen automaattinen tallentaminen tietokantaan. Tietokantana käytettiin Googlen *Firebase* -tietokantaa. Kerätyn datan avulla pelikerrat voitiin yksilöidä niin, että pelaajan anonymiteetti säilyy. Pelaajan ja agentin suorituksesta tallennettiin tietokantaan kenttien läpäisyprosentti, joista muodostettiin tutkimustulokset. Pelitiedoista tallennettiin kentän läpäisyprosentin lisäksi myös tiedot siitä, missä vihollisaallossa pelihahmo on hävinnyt kentän, jolloin saadaan myös kentän sisäistä dataa edistymisestä.

4.1.1 Prototyypin ydinmekaniikat

Toteutettavassa prototyypissä täytyy olla genren ydinmekaniikat toteutettuna, jotta vaikeustason balansointia voidaan tutkia nimenomaisesti bullet hell -genreen liittyen. Tämän tutkielman kannalta toissijaiset mekaniikat voivat tehdä tuloksien arvioinnista hankalaa. Tutkimusta varten toteutettiin pelkistetty prototyyppi pelistä, jonka katsotaan olevan selkeästi bullet hell -genren peli. Vaikka valitun genren ydinmekaniikat ovat pysyneet vuosien saatossa samoina, niin esimerkkipeliksi pyrittiin valitsemaan myös peli, jonka katsotaan olevan suosittu nykyhetkellä, jotta prototyypin suunnittelussa otetaan huomioon myös nykyhetken tila.

Bullet hell -genrellä tarkoitetaan nimensä mukaisesti haastavaa ampumispeliä, jossa pelaajahahmo kohtaa useita vihollisia ja joutuu luotisateeseen useita kertoja. Bullet hell on genrenä ammuskelupelien alakategoria, mutta genrellä ei ole täysin tarkkaa määritelmää. Ammuskelupelien genret eivät myöskään ole toisiaan poissulkevia. Esimerkiksi *Twin-stick shooter*-genre tarkoittaa käytännössä vain ammuskelupelin syötelaitteen ohjausmenetelmää, jossa hahmon liikkuminen tapahtuu peliohjaimen toista ohjaussauvaa käyttäen ja puolestaan

tähtääminen tapahtuu toisella ohjaussauvalla.

Enter The Gungeon valittiin esimerkkipeliksi, josta pelkistetty prototyyppi tehtiin, sillä se on yksi viime vuosien menestyneimmistä indie-peleistä bullet hell -genren sisällä. Steam -jakelualustan käyttäjäarvio [2] on kirjoitushetkellä 94.38%, jota voidaan pitää erittäin myönteisenä palautteena. Enter The Gungeon on pelinä tutkielman tekijälle jo entuudestaan tuttu, mutta tutkielmaa varten toteutettiin myös pienimuotoista pelitutkimusta itse peliä pelaamalla, jotta pelkistettyä prototyyppiä tehdessä otetaan huomioon tarkasti alkuperäisen pelin oleellisimmat pelimekaniikat.

Enter the Gungeon -pelin perusmekaniikat ovat liikkuminen, syöksyväistö (*engl. dodge roll*) ja tietysti ampuminen. Syöksyväistön suorittamisen aikana pelaajaan ei osu luodit. Ampumisessa ja syöksyväistössä on aina suorituksen jälkeen pieni viive toiminnon uudelleen aktivoimiseen, jottei toimintoa voi käyttää yhtäjaksoisesti. Pelin ympäristö generoidaan proseduraalisesti eri huoneista niin, että huoneiden muoto on etukäteen määritetty. Huoneiden sisältö, kuten huoneessa olevat viholliset, luodaan myös ajonaikaisesti, jolloin pelaaja ei voi etukäteen tietää kohdattavaa vihollista. Karkeasti sanottuna, pelaajan tehtävänä on päihittää huoneen kaikki viholliset, jonka jälkeen pelaaja voi jatkaa seuraavaan huoneeseen. Pelissä on kuitenkin esimerkiksi hyvin paljon eri aseita ja vihollistyyppejä sekä pelaajahahmokohtaista progressiota pelikerran edetessä, joten näiltä osin prototyypissä ei pyritä jäljittelemään alkuperäistä peliä.

Pelkistetyn prototyypin osalta toteutettavat kentät voidaan katsoa olevan Enter The Gungeon -peliin verrattavia huoneita. Opetettua agenttia voisi käyttää myös proseduraalisesti luoduissa kentissä, jotka koostuvat näistä huoneista. Vihollistyyppejä Enter The Gungeonissa on suuri määrä ja niiden käyttäytymisessä on myös suhteellisen paljon eroavaisuuksia. Toimintaperiaatteeltaan perusvihollistyypit voidaan karkeasti kategorisoida vihollisiin, jotka ampuvat ja vihollisiin, jotka liikkuvat pelaajaa päin ja antavat vahinkoa räjähtämällä. Vihollistyyppien osalta prototyyppiin toteutettiin kolme perusvihollistyyppiä, jotka ovat kuvattu luvussa 4.1.2. Enter The Gungeon -peliin olennaisesti kuuluvat pomotaistelut rajataan tämän tutkimuksen ulkopuolelle.

[2] Enter the Gungeon - Steam Database: https://steamdb.info/app/311690/

Kuvio 21. Kuvankaappaus toteutetusta prototyypistä

Enter The Gungeonissa huoneissa on myös vihollisaaltoja ja vihollisten ilmestyminen aaltoina toteutettiin myös prototyyppiin. Vihollisaaltojen ilmestyessä, pelaajalle annetaan visuaalinen tieto siitä, mihin kohtaan viholliset ilmestyvät animoidun tekstuuriefektin avulla, joka toimii pelaajalle merkitsijänä (*engl. signifier*). Merkitsijä kertoo käyttäjälle, mihin tietyn toiminnon voi kohdistaa (Norman 2013, s .13). Ilman merkitsijöiden käyttöä pelaaja ei ehdi ennakoida vihollisaaltojen ilmestymistä, jolloin peli voi tuntua epäreilulta pelaajaa kohtaan. Merkitsijän värillä annetaan myös pelaajalle tieto siitä, mikä vihollistyyppi ilmestyy kyseiselle paikalle. Toteutetussa prototyypissä sekä viholliset että pelaaja tuhoutuvat yhdestä luodista. Pelaaja voi kuitenkin aloittaa saman kentän yhä uudelleen.

4.1.2 Prototyypin vihollistyypit

Vihollistyyppien 1 ja 2 toiminnan eroavaisuutena on ammuttavien ammusten määrä ja ammuksen nopeus. Vihollistyyppi 1 ampuu nopeammin etenevän ammuksen. Vihollistyyppi 2 puolestaan tekee tarpeelliseksi syöksyväistön käyttämisen pelaajan kannalta. Vihollistyypit on kuvattu taulukossa 1.

Taulukko 1. Prototyypin vihollistyyppien ominaisuudet

Vihollistyyppi	Pelaajalle tuotettava vahinko	Liikkuminen
Vihollistyyppi 1	Yksittäiset ammukset	Seuraa pelaajaa, mutta pyrkii pitämään välimatkan pelaajaan
Vihollistyyppi 2	Ampuu 10 ammusta kerralla hajaumamuodostelmassa	Seuraa pelaajaa, mutta pyrkii pitämään välimatkan pelaajaan
Vihollistyyppi 3	Räjähdys kohdatessaan pelaajan	Liikkuu pelaajaa kohti

4.1.3 Prototyypin kentät

Prototyypin kenttien suunnittelussa lähtökohtana oli tutkimuskysymys, mutta suunnittelussa täytyi ottaa huomioon myös käyttäjätestit. Tutkimuskysymystä ajatellen oli tärkeää, että prototyypissä on bullet hell -genren mukaisesti paljon haastetta. Käyttäjätestejä ajatellen oli myös huomioitava, että prototyypin läpäisy on mahdollista käyttäjän kannalta inhimillisessä ajassa, joten prototyyppi ei voinut olla kovin pitkäkestoinen. Tästä johtuen prototyypin kahden viimeisimmän kentän vaikeustaso nousee huomattavasti ja näin ollen vaikeustason nousu ei noudata optimaalisen flow-käyrän mukaista nousua.

Yksittäisten käyttäjätestien ongelmana on se, että pelaajat oppivat pelin pelimekaniikat käyttäjätestin edetessä yhä paremmin, jolloin testauskerran alkupään haasteet voivat näyttäytyä käyttäjädatan mukaan vaikeammilta suhteessa viimeisimpiin kenttiin. Tässä tutkielmassa ongelmaa pyritään lieventämään toteuttamalla tutoriaalikenttä, jonka tarkoituksena on opettaa pelaajalle pelimekaniikat ja vihollistyypit. Tutoriaalikentän tuloksia ei kuitenkaan sisällytetä tutkimustuloksiin.

Kenttien määrässä päädyttiin neljään kenttään. Lisäksi toteutettiin yksinkertainen tutoriaalikenttä, jotta testihenkilöillä on hieman kokemusta pelimekaniikoista ennen varsinaista tuloksiin raportoitavaa kenttää. Koehenkilön käyttäminen pelitestaajana kehitysvaiheessa osoittautui erittäin hyödylliseksi, sillä vaikeustaso paljastui ensimmäisissä testeissä aivan liian haastavaksi. Liiallinen haastavuus olisi johtanut siihen, etteivät edes genren pelaajat olisi kyenneet läpäisemään prototyyppiä kovinkaan hyvin.

Bullet hell -genre on lähtökohtaisesti haastava, jolloin genreen tottumaton ei todennäköisesti läpäise prototyypin kenttiä kovin hyvällä prosentilla. Tutkielman kannalta agentin toimintaa on mielekkäintä verrata kohderyhmän suoriutumiseen. Tämän vuoksi käyttäjätestin kohderyhmäksi valittiin lähtökohtaisesti pelaajia, joilla on kokemusta bullet hell -genrestä. Agenttia arvioidaan vertaamalla sen läpäisyprosenttia prototyypin eri kentissä käyttäjädatan läpäisyprosentteihin. Läpäisyprosenttia on myös käytetty mittaamaan vaikeustasoa useassa aihealueeseen liittyvässä viimeaikaisessa tutkimuksessa, joita on kuvattu luvussa 3.9. Koneopetettua agenttia simuloidaan eri kentissä 300 kertaa, jonka perusteella saadaan läpäisyprosentti ohjelmallisesti laskettua. Taulukossa 2 on kuvattu prototyypin eri kenttien sisältö.

Taulukko 2. Prototyypin kentät

Kenttä	Vihollismäärä	Selite
0	4	Tutoriaalikenttä, jonka tarkoitus on opettaa pelimekaniikat ja esitellä vihollistyypit.
1	8	Yksi vihollisaalto. Sisältää vihollistyypit 1 ja 2.
2	9	Yksi vihollisaalto. Sisältää vihollistyypit 1 ja 2.
3	16	Kaksi vihollisaaltoa. Sisältää kaikki vihollistyypit.
4	23	Kolme vihollisaaltoa. Sisältää kaikki vihollistyypit.

4.2 Agentin opettamiseen liittyvä kehitystyö

Opettamisen tehokkuutta vertailtiin eri hyperparametriarvoja kokeilemalla, mutta myös agentin oppimisympäristön muutoksilla sekä agentille annettavien havainnoitavien muuttujien suhteen. Oppimisprosessiin liittyvien asioiden, kuten agentille määritettävien havainnoitavien asioiden vaikutus oppimisen tehokkuuteen, oli etusijalla suhteessa hyperparametrien hienosäätöön, sillä niiden vaikutus oppimistuloksiin oli hyvin suurta. Agentin opettamisen yhteydessä *TensorBoard* -visualisointityökalu oli hyvin tärkeä datan visualisointityökalu, sillä se mahdollistaa muun muassa oppimistulosten vertailun eri opetuskertojen kesken. Opetuskerta lopetettiin, mikäli oli selkeästi havaittavissa, ettei oppimistulos tule parantumaan valitulla hyperparametriarvolla tai muulla agentin oppimiseen liittyvällä muutoksella.

Tutkielman teknisen osion kehityksessä käytettiin seuraavia ohjelmistopaketteja ja niiden

versioita:

- Unity, versio 2020.1.15f1.
- com.unity.ml-agents (C#), versio 1.6.0.
- ML-Agents (Python), versio 0.22.0.
- Communicator (C#-Python), versio 1.2.0.
- PyTorch, versio 1.7.0.

4.2.1 Agentin toiminnot

Agentin opettamiseen valittiin vain haasteen kannalta oleellisimmat pelimekaniikat eli osumien väistäminen ja ampuminen. Prototyypin eri kenttien huoneissa sijaitsee myös seuraavaan kenttään pääsyn mahdollistava ovi, joka aukenee, kun viholliset ovat tuhottu. Ovelle liikkuminen on kuitenkin vihollisten tuhoamisen jälkeen toissijaista vaikeustason todentamisen kannalta, joten opettamisen tehostamiseksi se rajattiin opetusprosessista pois. Agentin tapauksessa kenttä katsotaan läpäistyksi vihollisten tuhouduttua. Taulukossa 3 nähdään agentille määritetyt toiminnot.

Taulukko 3. Agentin toiminnot

Indeksi	Arvo	Selite
0	Vasemman ohjaussauvan x-arvo	Liikkumisen x-arvo
1	Vasemman ohjaussauvan y-arvo	Liikkumisen y-arvo
2	Oikean ohjaussauvan x-arvo	Tähtäämisen x-arvo
3	Oikean ohjaussauvan y-arvo	Tähtäämisen y-arvo
4	Vasemman liipaisinnapin arvo	Syöksymisen arvo
5	Oikean liipaisinnapin arvo	Ampumisen arvo

Pelihahmon liikkuminen toteutuu peliohjaimen kahdella ohjaussauvalla. Prototyyppi noudattaa siis *Twin-stick shooter*-ohjaustyyliä. Kaikki toiminnot ovat float-tyyppisiä, sillä Unityn ML-Agents vaatii float-tyyppisen taulukon, joka asetetaan koodissa agentin suoritettaviksi toiminnoiksi. Tässä tutkielmassa liikkumisen ja tähtäämisen x ja y-arvot ovat väliltä $[-1,1]$, kun taas syöksymisen ja ampumisen arvot ovat puolestaan $\{0,1\}$. Oppimisen aikana agentti kokeilee eri toiminta-arvoja, minkä seurauksena agentti esimerkiksi pystyy liikkumaan.

Imitaatio-oppiminen mahdollistaa sen, että agentti saa jo alussa tietoon demonstraatiodataa, jota se pyrkii jäljittelemään. Oppimisen edetessä agentti osaa yhä paremmin käyttää toimintoja palkintomäärän saavuttaakseen.

4.2.2 Agentille määritetyt havainnot

Agentille annettavien havaintojen osalta todettiin myös oppimisen olevan tehokkainta, kun havainnot ovat oleellisia tehtävän suorittamisen kannalta ja kun ne ovat mahdollisimman yksiselitteisiä. Oppimisen kannalta tehokkuus laskee, kun agentille asetetaan liian suuri määrä eri muuttujia havainnoitavaksi, mitkä eivät tehtävän suorittamisen kannalta ole välttämättömiä. Tehtävän suorittamisen kannalta myös liian suuri määrä havainnoitavaksi annettavia muuttujia tuo agentin oppimisprosessiin lisää kompleksisuutta. Tällöin oppiminen ymmärrettävästi hidastuu, kun agentin täytyy oppia eri muuttujien vaikutusta palkintosignaalin maksimoimiseen.

Agentti saa tiedon vihollisista säteensuuntausta (*engl. raycast*) käyttämällä eli agentista ammutaan säteitä, jotka osuvat pelimaailman objekteihin. Kehitysvaiheessa agentille annettiin esimerkiksi normaalin säteensuuntauksen lisäksi tieto vihollisten sijainnista erillisen vektoritaulukon muodossa. Vektoridataa testattiin niin normalisoituneilla vektoreilla kuin etäisyyden sisältävillä vektoreilla. Tätä testattiin siksi, että agentista lähtevät säteet osuvat usein vihollisten luoteihin, jolloin agentti ei tiedä luodin takana olevasta vihollisesta mitään. Opetustulokset ja agentin suoriutuminen eivät kuitenkaan parantuneet vektorimuotoisen vihollisdatan lisäämisellä, vaan päinvastoin opetustulokset heikkenivät. Tämän selittänee se, että agentin tulee ensisijaisesti väistää luotia, jottei pelikerta loppuisi kyseiseen osumaan. Mainittu vektorimuotoinen vihollisdata vaikutti siis olevan ylimääräistä informaatiota, jolla itseasiassa oli tuloksia heikentävä vaikutus. Agentille siis kentässä onnistumisen kannalta prioriteettina on väistää ensin kohtisuoraan lähestyvä luoti, jonka jälkeen voidaan toimia vihollisen päihittämiseksi. Säteensuuntauksen käyttäminen yksinään tässä tapauksessa siis paitsi yksinkertaisti opettamista, niin myös nopeutti agentin oppimista.

Agentin havainnoitavaksi on määritelty seuraavat peliobjektien tägit (*engl. tag*) säteensuuntausta käyttäen:

- Viholliset, jotka on eritelty vihollistyypittäin
- Vihollisten ammukset vihollistyypittäin
- Kentän seinät eli pelialueen reunat
- Vihollisen ilmestymistä pelikenttään indikoivan merkitsijäobjektin tägi.

Tägien avulla agentti pystyy erottamaan vihollis- ja ammustyypit toisistaan. Vihollisen tuhouduttua ohjelmakoodi muuttaa vihollisen tägin, jolloin agentti ei enää havainnoi kyseistä vihollista, vaan pelkästään aktiivisena olevat viholliset. Lisäksi skriptin kautta agentille annetaan havainnoitavaksi taulukossa 4 näkyvät numeraaliset tiedot omasta toiminnastaan.

Taulukko 4. Agentille havainnoitavaksi annetut muuttujat

Tieto	Tyyppi
Agentin aseobjektin suuntavektori	Vector2
Agentin liikkumisnopeus	float
Ampuminen sallittu	boolean
Syöksyminen sallittu	boolean

Ampumisen ja syöksymisen boolean-arvot ilmaisevat pystyykö agentti käyttämään kyseistä toimintoa. Näiden kohdalla pelissä on kummankin toiminnon toteuttamisen jälkeen pieni viive toiminnon uudelleen aktivoimiseen, jottei toimintoa voi käyttää yhtäjaksoisesti koko ajan. Mikäli näitä tietoja ei välitetä agentille havainnointia varten, se ei myöskään ole silloin tietoinen siitä, että toiminto ei aktivoidu.

4.2.3 Ulkoiset palkinnot

Agentin opettamisen yhteydessä agentille annetaan ympäristössä toimimisen suhteen palkintoja vahvistusoppimisen avulla. Nämä palkinnot ohjaavat agenttia haluttuun toimintaan opettamisen aikana. Annettavat palkinnot ovat seuraavanlaisia:

- Agentin kuoleminen: -2
- Vihollisen tuhoaminen: +1
- Kentän läpäisy: +1
- Ampumiskomennon tapahtuessa ja säteen osuttaessa vihollisobjektiin: +0.5

- Jokaisella aika-askeleella annettava rangaistus: -0.0001.

Jokaisella aika-askeleella tapahtuva rangaistus pakottaa agentin kokeilemaan nopeammin eri toimintoja palkintomäärän kasvattamiseksi, jolloin oppiminenkin nopeutuu. Myös alustavissa testeissä todettiin, että oppimisessa tapahtuu huomattavaa nopeutumista näin tehden. Prototyypissä vihollisen eliminointi tapahtuu ammuttavan ammusobjektin törmätessä siihen. Agentti ei saa välitöntä palautetta ampumisen onnistumisesta vaan palkinnon saaminen tapahtuu viiveellä. Tästä johtuen agentin oppiminen tähtäämisen suhteen oli hidasta, sillä agentin tulee tarkastella palkintomäärää usean aika-askeleen päähän. Tähtäämisen oppimista tehostettiin antamalla ampumiskomennon tapahtuessa palkinto, mikäli agentista tähtäyssuuntaan lähtevä säde osuu vihollisobjektiin.

4.2.4 Hyperparametrit ja palkintosignaalit

Hyperparametriarvoja lähdettiin hienosäätämään ML-Agents dokumentaation suositusarvojen pohjalta [3]. Valitut hyperparametriarvot valikoituivat siis iteratiivisen kehitystyön tuloksena, jossa testattiin lukuisia eri parametriarvoja. Oleellisimmat agentin opettamiseen käytetyt yleiset asetukset ja hyperparametrit on kuvattu taulukossa 5. Opetusalgoritmina käytettiin PPO:ta. Oleellisimpana huomiona verrattuna ML-Agent dokumentaation suositusarvoihin, mainittakoon se, että agentin muistiin ja kokemuspuskurin (*engl. experience buffer*) kokoon liittyviä arvoparametreja asetettiin vastaamaan pidempää aikaväliä.

Palkintosignaalien osalta käytettiin kolmen palkintosignaalin yhdistelmää eri painokertoimella: ulkoisia palkintoja, uteliaisuuspalkintoja ja imitaatio-oppimisen palkintoja. Erityisesti imitaatio-oppimisen painotusarvoa testattiin eri arvoilla. Esitutkimuksessa kävi ilmi, että agentti oppii alussa nopeammin, kun imitaatio-oppimisen painotusarvo on suurempi. Pitemmällä aikavälillä vahvistusoppimisen suurempi osuus painotuksessa mahdollistaa kuitenkin yhä paremman palkintomäärän saavuttamisen, joten imitaatio-oppimisen painoarvoa pienennettiin niin, että imitaatio-oppiminen toimii painoarvoltaan vain pienenä ohjaavana tekijänä. Demonstraatiodatan laadulla oli myös hyvin suuri vaikutus. Tutkimuksessa pyrittiinkin tuottamaan hyvää pelitulosdataa demonstraatioita nauhoittaessa. Lopulliset palkintosignaalien

[3] ML-Agents konfiguraatiotiedoston tarkemmat määritelmät ja selitteet: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md

Taulukko 5. Ote lopullisista opettamiseen käytetyistä asetuksista ja hyperparametriarvoista

Tunniste	Arvo	Selite
time_horizon	512	Askelten lukumäärä, jonka jälkeen agentin kokemus lisätään kokemuspuskuriin.
learning_rate	$3e^{-4}$	Alustettu oppimisnopeus gradienttimenetelmälle.
batch_size	2048	Kokemusten lukumäärä jokaisella gradienttimenetelmän iteraatiolla.
buffer_size	20480	Kokemusten lukumäärä, joka kerätään ennen menettelytavan päivittämistä.
beta	0.04	Entropian säännönmukaisuuden vahvuus. Kertoo, kuinka paljon menettelytapaa satunnaistetaan eli kuinka paljon agentti kokeilee uusia toimintoja eri tiloissa.
epsilon	0.2	Vastaa hyväksyttävää raja-arvoa vanhan ja uuden menettelytavan välillä gradienttimenetelmän päivityksen yhteydessä.
lambd	0.95	Kertoo, kuinka paljon agentti luottaa tulevaisuuden palkintoarvioon (arvion päivityksen yhteydessä).
num_epoch	3	Kertoo, kuinka monta kertaa kokemuspuskuri käydään läpi, kun suoritetaan gradienttimenetelmän optimitointi.
num_layers	2	Piilokerrosten lukumäärä neroverkossa.
hidden_units	512	Neuroneiden lukumäärä piilokerroksissa.
memory_size	512	Agentin muistin koko.
sequence_length	128	Määrittää, kuinka pitkä kokemusjakson tulee olla opetuksen aikana.

määritykset on kuvattu taulukossa 6. Esitutkimuksessa kokeiltiin myös vaihtoehtoisten palkintosignaalien asetusten muuttamista, mutta ne eivät tuottaneet parempia oppimistuloksia, joten niiden osalta pysyttiin oletusarvoissa. Nämä vaihtoehtoiset asetukset ovat kuvattuna tarkemmin ML-Agents dokumentaatiossa [3].

Taulukko 6. Palkintosignaalien konfiguraatiot

Tunniste	Arvo	Selite
reward_signals	extrinsic	Ulkoiset palkinnot.
strength gamma	1.0 0.99	Painokerroin. Tulevaisuuden palkintoarvojen vähennyskerroin.
reward_signals	curiosity	Sisäinen uteliaisuusmoduuli.
strength gamma	0.02 0.99	Painokerroin. Tulevaisuuden palkintoarvojen vähennyskerroin.
reward_signals	gail	Imitaatio-oppiminen.
strength gamma demo_path	0.02 0.99 ../demo01.demo	Painokerroin. Tulevaisuuden palkintoarvojen vähennyskerroin. Demonstraatiodatan polku.

4.2.5 Agentin opetusympäristö

Opetuksen aikana agentin oppimista arvioitiin pääasiallisesti suhteessa agentin keräämään ulkoisten palkintojen kumulatiiviseen palkintomäärään. Myös pelkkä visuaalinen tarkastelu agentin toimimisesta opetuksen aikana auttoi huomaamaan opetukseen ja oppimiseen liittyviä ongelmia. Visuaalisen tarkastelun avulla huomattiin esimerkiksi bugeja oppimisympäristössä sekä tiettyjä oppimisvaikeuksia, kuten tähtäämisen vaikeus, jonka oppimista vahvistettiin ulkoisten palkintojen avulla. Visuaalisen tarkastelun perusteella pystyi myös huomaamaan mallissa tapahtuvan ylisovittamisen: kun tietty kenttä toistui opettamisvaiheessa liian usein, pystyi agentin liikkumisesta huomaamaan jo opettamisen aikana, että agentin liikkuminen muovautuu hyvin vahvasti vain kyseiseen kenttään sopivaksi.

Demonstraatiodataa nauhoitettiin prototyypin eri kentissä yhteensä 15 minuuttia, koska lyhyempi nauhoituksen pituus heikensi opetustuloksia. Nauhoitetun datan laatu vaikutti myös huomattavasti agentin oppimistuloksiin. On ymmärrettävää, että mitä heikompi pelitulos demonstraatiodatassa on, sitä heikommin agentti oppii.

Oppimisympäristön suhteen kokeiltiin myös peliympäristöä, jonka eri kentissä on hyvin paljon satunnaisuutta. Tällä pyrittiin testaamaan, kuinka hyvin agentti toimii jo olemassa olevissa kentissä, kun oppiminen on tapahtunut dynaamisessa ympäristössä. Tässä peliympäristössä myös kenttien muoto oli erilainen kenttien välillä. Agentin opetustulokset kuitenkin huononivat niin paljon, ettei agentin opettamisen katsottu olevan käytännöllistä tällaisessa ympäristössä. Tästä syystä lopullinen opettaminen tapahtuu kentissä, jotka vastaavat myös prototyypin kenttiä. Tätä valintaa tuki myös tutkijoiden Kristensen ja Burelli (2020) tutkimustulos, jossa agentin toiminta oli huomattavasti heikompi tuntemattomien kenttien kohdalla, vaikka pelimekaniikat pysyivätkin samoina.

Seuraavaksi tutkittiin, voisiko agenttia opettaa kurssioppimisen avulla eli niin, että haastetta kasvatetaan aina kun agentti on saavuttanut määritetyn palkintorajan oppimisen aikana. Kurssioppimista hyödynnettiin niin, että kentät vaihtuvat haasteellisemmiksi aina kun agentti saavuttaa tietyn palkintomäärän. Tuli kuitenkin selkeästi ilmi, että tällä tavoin tehden agentti alkaa unohtamaan aikaisemmassa kentässä toimivaksi todetun käyttäytymisen eli malli ylisovittuu aina viimeisimmän kentän suhteen. Hyperparametreja, jotka mahdollistavat aikaisempien kokemusten huomioimisen, pyrittiin asettamaan pidemmälle aikavälille, mutta tässä tapauksessa aikaväli oli liian suuri, sillä opetuksessa olisi pitänyt tarkastella useiden miljoonien takaisten aika-askelten kokemuksia. Yksi keino olisi opettaa agentille uudelleen myös aikaisempia vaiheita, mutta tämä todettiin käytännön toteutuksen kannalta haasteelliseksi, sillä se olisi vaatinut useita tarkasteluajoja opettamisen yhteydessä. Tästä syystä kurssioppimisen hyödyntäminen tällä tavoin ei enää vaikuttanut kovinkaan käytännölliseltä tai järkevältä kehitystyön kannalta.

Kokonaisen prototyypin oppimisen kannalta parhaimmaksi opetustavaksi todettiin opetusympäristö, jossa agentti kohtaa prototyypin jokaisen vihollisaallon haasteita. Tutkimuksessa vihollisaaltojen ilmestyminen satunnaistettiin, mutta vihollisaallot vastasivat prototyypin kenttien vihollisaaltoja. Agentin sijainti satunnaistettiin jokaisen kentän alustuksen yhteydessä,

koska useita vihollisaaltoja sisältävän kentän läpäisy vaatii sen, että agentti osaa toimia uuden vihollisaallon aktivoituessa myös sen hetkisessä sijainnissaan. Kentän alustaminen tapahtuu aina, kun kentän viholliset on päihitetty tai vastaavasti agentin hävitessä kentän.

Oppimisen nopeuttamiseksi skenessä opetetaan samanaikaisesti kuutta eri agenttia, jotka päivittävät samaa menettelytapaa. Agentin opettamisen lopettamiseksi oli tarkoituksena määritellä raja, jolloin agentti katsotaan oppineeksi. Osoittautui kuitenkin, että tällaista rajaa on hyvin hankala, ellei mahdoton määritellä. Esimerkiksi agentin keräämä kumulatiivinen palkintomäärä voi oppimisen aikana heikentyä pitkänkin tarkasteltavan aikavälin aikana, jonka jälkeen jälleen parantua. Lisäksi liian lyhyt opetusaika ei mahdollistanut kovinkaan hyviä lopputuloksia prototyypin haastavuuden vuoksi.

ML-Agents mahdollistaa opetuksen keskeyttämisen niin, että opetusta voidaan myöhemmin jatkaa, kunhan neuroverkon ja agentin sisäisen mallin määritykset ovat entisellään. Hyperparametrien sekä agentille määritettävien toimintojen ja havainnoitavien asioiden osalta ei siis ollut mahdollista tehdä opetuksen aikaisia muutoksia. Kehitysvaiheessa kuitenkin opetuskerrat lopetettiin jo aikaisessa vaiheessa, mikäli pystyttiin selvästi huomaamaan, että agentin opetustulokset tulevat olemaan heikkoja.

Pitkäkestoisen opetusprosessin vuoksi agentin opetukseen tehtiin kuitenkin tiettyjä muutoksia ja kokeiluja niin, ettei opetusta aloitettu uudelleen jokaisen muutoksen jälkeen. Tällaisia muutoksia olivat esimerkiksi opetuksen aikana huomatut oppimisympäristön bugit ja opetuksen tehostamista varten toteutetut muutokset, kuten agentille annettavien palkintojen muuttaminen ja agentin sijainnin satunnaistaminen opetusjaksojen alustuksen yhteydessä. Tämä menettely mahdollisti huomattavasti nopeamman kehitystyön, sillä muutoksien vaikutuksen oppimiseen pystyi havainnoimaan paljon nopeammin. Lopullinen opetusajo toteutettiin kuitenkin puhtaasti, eli ilman kesken opetuksen toteutettavia muutoksia. Tämä lopullinen opetusajo, jonka aikana muutoksia ei siis tehty, paransi myös lopulta oppimistuloksia jonkin verran. Lopullista käyttäjädataan verrattavaa agenttia opetettiin 75 miljoonan aika-askeleen verran, mikä vastasi suunnilleen neljää viikkoa yhtäjaksoista opetusta.

Tutkielman toteutushetkellä Unityn ML-Agents ei esimerkiksi mahdollistanut ainakaan dokumentoitua tapaa laskennan ja opetusympäristön siirtämisestä ulkopuoliselle pilvipalvelul-

le, jolloin opettaminen olisi ollut nopeampaa lisälaskentatehon avulla. Tämän tutkielman puitteissa ei ollut mahdollista hankkia pääsyä tehokkaalle palvelimelle opettamista varten, ja tarkempaa tutkimusta ei toteutettu siitä, voisiko ympäristön siirtäminen ulkopuolisiin pilvipalveluihin olla tältä osin mahdollista. Opettaminen vaatii siis myös peliympäristön, jonka kanssa agentti vuorovaikuttaa. ML-Agents on vielä uusi ohjelmistopaketti ja se on jatkuvan kehitystyön alla. Agentin opettamiseen dedikoitu pilvipalvelu on kuitenkin tällä hetkellä tekeillä [4]. Agentin opetuksessa jouduttiin tyytymään kahteen tietokoneeseen, joilla kehitystyötä tehtiin rinnakkain. Kahden koneen käyttäminen rinnakkain nopeutti kuitenkin omalta osaltaan eri opetusmenetelmien ja hyperparametriarvojen testaamista.

4.3 Tulokset

Agenttia opetettiin ensin kokonaisen prototyypin kaikissa kentissä. Luvussa 4.3.1 käydään läpi kyseisen agentin tulokset prototyypin kaikissa kentissä, joita verrataan käyttäjädataan. Kokonaisen prototyypin opettamisen jälkeen suoritettiin vielä kenttäkohtainen opetus, jonka tarkoituksena oli tarkastella, voisiko agentin palkintokertymädatan perusteella havaita kenttien vaikeustasoerot. Kenttäkohtaisen opettamisen tuloksia käsitellään luvussa 4.3.2.

4.3.1 Kokonaisen prototyypin opettaminen

Käyttäjätesteihin valittiin 9 pelaajaa, jotka pitivät itseään bullet hell -genren pelaajina. Käyttäjätestit haluttiin kohdentaa bullet hell -genren pelaajatyyppiin, jotta käyttäjätesteihin osallistujien taito on oletettavasti hyvällä tasolla. Käyttäjän oli suoriuduttava vähintään viimeiseen kenttään saakka, jotta käyttäjän tulosdata sisällytetään lopullisiin käyttäjätestien tuloksiin. Kaikkiin kenttiin vaadittiin siis tulos, jotta saadaan vertailukelpoista dataa kenttien välisistä vaikeustasoeroista. Tästä syystä kahden pelaajan tulokset katsottiin olevan tutkimuksen kannalta vajaat, koska kaikista kentistä ei ollut dataa. Lopullisiin käyttäjätestien tuloksiin huomioitiin siis seitsemän käyttäjän pelitulosdata.

Pelaajat saivat pelata pelin useita kertoja läpi ja pelikerroista johdettiin pelaajakohtainen keskiarvo läpäisykertoimen tulokseksi. Kentän läpäisykerroin käyttäjädatasta lasketaan kaaval-

[4] Unity Machine Learning Agents Cloud: https://create.unity.com/ML-Agents-cloud-get-in-touch

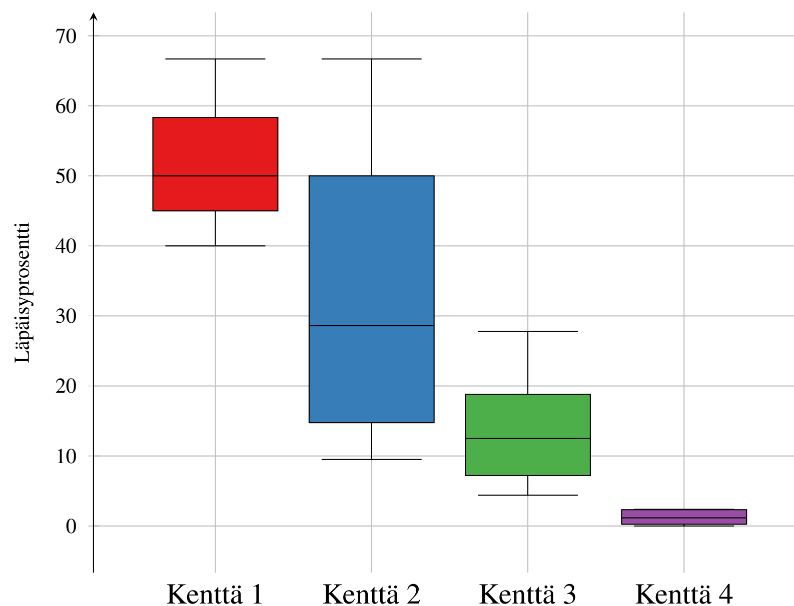
la:

$$\text{Läpäisykerroin}_{pelaaja} = \frac{\text{Läpäisykerrat}}{\text{Yritysten määrä}}$$

$$\text{Läpäisykerroin}_{summa} = \sum_{n=1}^{\text{pelaajamäärä}} \text{Läpäisykerroin}_{pelaaja_n}$$

$$\text{Läpäisykerroin} = \frac{\text{Läpäisykerroin}_{summa}}{\text{pelaajamäärä}} * 100$$

Pelaajien läpäisykertoimet summataan yhteen, minkä jälkeen jaetaan käyttäjien määrällä ja kerrotaan sadalla, jotta saadaan lopullinen kenttäkohtainen läpäisyprosentti koko käyttäjädatalla. Agentin tapauksessa läpäisykerroin lasketaan samalla tavalla, mutta luonnollisesti pelaajamäärä on läpäisykertoimen laskennassa 1, joten summaaminen voidaan ohittaa. Agenttia simuloitiin 300 kertaa prototyypin jokaisessa kentässä ja agentin suoriutumisesta tallennettiin kenttien läpäisyprosentit tulosdataksi.

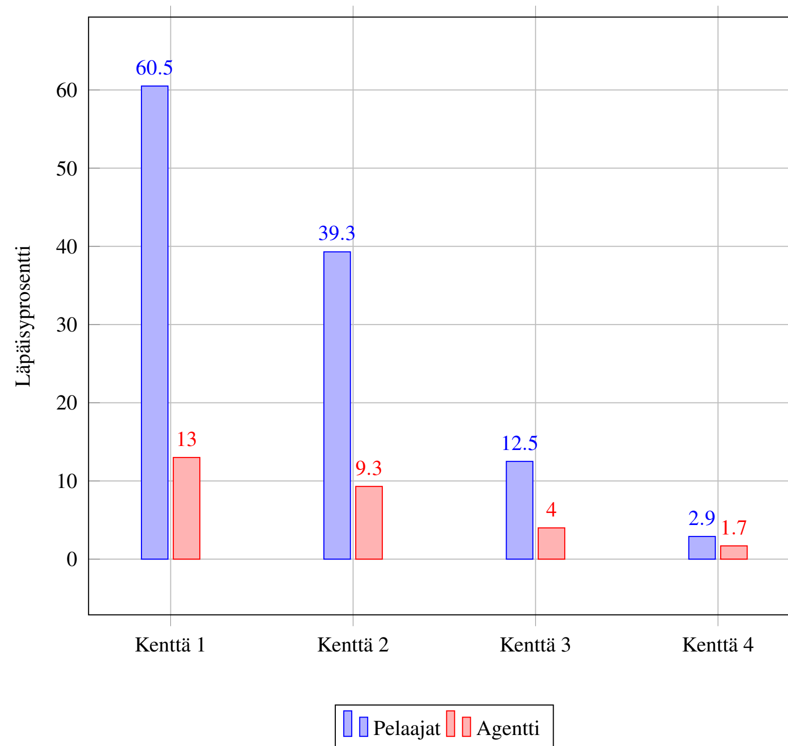


Kuvio 22. Käyttäjätestien läpäisyprosenttien hajonta

Kuviossa 22 nähdään käyttäjätestien läpäisyprosenttien hajonta prototyypin eri kentissä perinteisenä laatikko-janakuviona. Kuviossa palkin keskiviiva kertoo mediaanin. Väritetyn laatikon ylä- ja alareunat kertovat yläkvartiilin ja alakvartaalin kentän läpäisytuloksista. Tarkemmin sanottuna väritetyn alueen sisällä on 50% kentän tuloksista. Janan ääriviivat kertovat pienimmät ja suurimmat arvot, joita ei lasketa poikkeamiksi. Tarkempi kuvaus laatikko-

janakuvion merkinnöistä on nähtävillä alaviitteen lähteestä [5].

Käyttäjätestien perusteella voidaan todeta, että viimeinen kenttä oli hyvin haasteellinen pelaajille, kun taas kaksi ensimmäistä kenttää olivat suhteellisen helppoja. Näin ollen kenttien vaikeustasot vastasivat hyvin sitä, mihin prototyypin vaikeuden määrittämisessä pyrittiinkin.



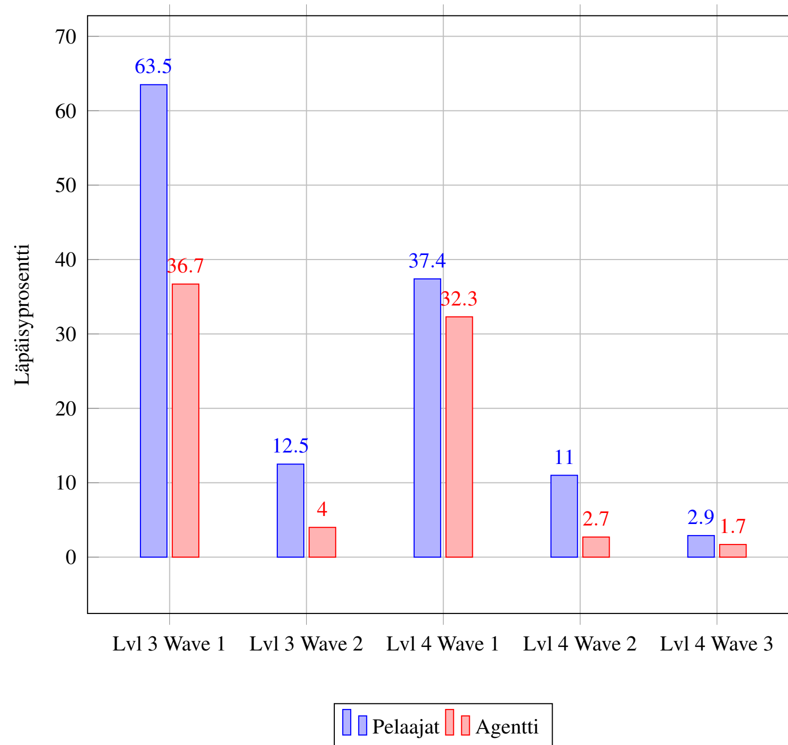
Kuvio 23. Koetulokset

Kuviossa 23 on raportoitu käyttäjätestien ja agentin läpäisyprosentit prototyypin eri kentissä. Kentät 3 ja 4 sisälsivät useamman kuin yhden vihollisaallon. Kuviossa 24 on esitelty kenttien 3 ja 4 vihollisaaltokohtaiset läpäisyprosentit, jotta voidaan nähdä tarkemmin, kuinka agentti suoriutuu näiden kenttien sisällä. Vihollisaallon läpäisy vaatii siis edellisen aallon läpäisyn eli kentän viimeisen aallon läpäisyprosentti vastaa myös kentän lopullista läpäisyprosenttia.

Agentin tulokset olivat heikompia kuin pelaajien keskimääräinen tulos. Opetetun agentin läpäisykertoimien ero suhteessa pelaajadataan kuitenkin pienenee kenttien vaikeutuessa. Viimeisessä kentässä agentin läpäisyprosentti on hyvin lähellä pelaajadatan keskiarvotulosta.

[5] Towards Data Science - Understanding Boxplots: https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51

Lopullista agenttia opetettiin siis 75 miljoonan aika-askeleen verran.



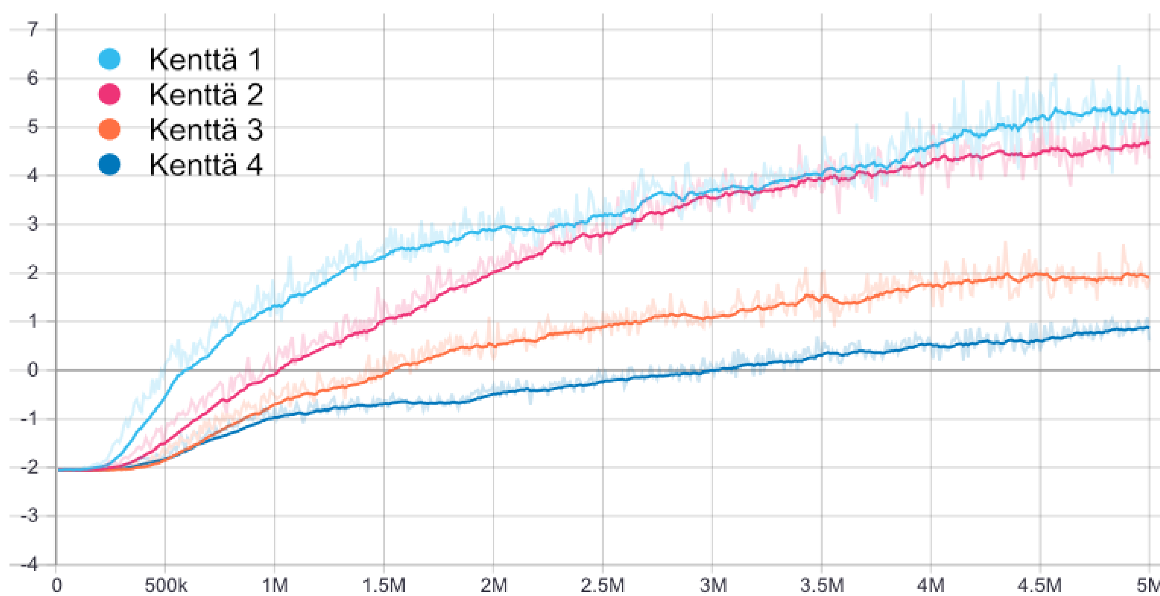
Kuvio 24. Kahden viimeisen kentän vihollisaaltokohtaiset tulokset

4.3.2 Kenttäkohtainen opettaminen

Kenttäkohtaisella opettamisella haluttiin varmistaa, voisiko kenttäkohtainen opetus olla varteenotettava ratkaisu, sillä näin voidaan estää mallin ylisovittaminen toiseen kenttään. Agenttia opetettiin jokaisessa kentässä erillisinä opetuskerroilla 5 miljoonaa aika-askelta. Kenttäkohtaisessa opetuksessa haluttiin tarkastella agentin palkintokertymän kehitystä eri kentissä opetuksen aikana, jotta nähdään, voisiko palkintokertymästä tulkita kenttien välisiä vaikeustasoeroja jo aikaisessa opetusvaiheessa.

Agentti ei pystynyt läpäisemään prototyypin vaikeimpia kenttiä valitulla opetusajalla. Tulos oli kuitenkin odotettavissa luvun 4.3.1 kokeen perusteella, sillä prototyypin viimeinen kenttä oli syy, jonka vuoksi agenttia opetettiin lopulta 75 miljoonan aika-askeleen verran. Jokaisen kentän opettamista kentän läpäisyyn asti ei katsottu käytännölliseksi ratkaisuksi, sillä prototyyppiä useamman kentän kohdalla pelin kehitysaika pitenisi huomattavasti.

Eri opetuskertojen agentin kumulatiivinen palkintokertymä on nähtävissä kuviossa 25. Kuviossa pystyakseli kertoo agentin numeraalisen palkintokertymän, joka siis kehittyy opetuksen edetessä taulukossa 6 annettujen palkintosignaalien perusteella. Kuvion palkintokertymäkäyrät ovat tasoitettuja, mutta raakadata on nähtävissä taustalla himmeämmällä värityksellä.



Kuvio 25. Agentin palkintomäärän kehitys opetuksen aikana

Eri kenttien vaikeustasoerot ovat noin 1.5 miljoonan aika-askeleen jälkeen havaittavissa, mutta varsinkin kenttien 1 ja 2 välillä nähdään, että opetuksen edetessä ero kapenee agentin oppiessa paremmaksi. Tämä johtuu siitä, että kentässä 2 agentilla on mahdollisuus kerätä suurempi vahvistusoppimisen palkintomäärä kentän useamman vihollisen vuoksi. Palkintokertymien perusteella voidaan suuntaa antavasti nähdä vaikeustasoerot verrattaessa luvun 4.3.1 käyttäjätesteihin, mutta suoraa johtopäätöstä kenttien vaikeustasoista ei palkintokertymästä ilman tulkintaa voida tehdä.

4.4 Pohdinta

Tutkimuksessa selvitettiin kolmen tutkimuskysymyksen avulla koneopetetun agentin soveltuvuutta pelitestaukseen bullet hell-genren pelissä. Tutkimuksen suorittamisen jälkeen on haastavaa antaa yksiselitteistä vastausta tutkimuskysymyksiin, sillä kysymysten vastauksia

määrittää suuresti pelin vaikeus ja pelissä olevien opetettavien toimintojen laajuus. Seuraavissa aliluvuissa käydään läpi tutkimuskysymyksien vastauksia sekä niihin olennaisesti liittyviä seikkoja, jotka nousivat esiin tutkimuksen aikana.

4.4.1 Tutkimuskysymys 1

Ensimmäinen tutkimuskysymys oli muodossa: "Pystytäänkö koneopetetun agentin avulla päättelemään bullet hell -genren kentän suhteellinen vaikeustaso luotettavasti?". Ensimmäisellä tutkimuskysymyksellä tarkoitettiin siis sitä, että ovatko agentin pelitulokset kenttien vaikeustasojen mukaisia. Tutkimuksessa kenttien vaikeustasot todennettiin käyttäjätesteillä. Agentin tulokset mukailevat käyttäjätestien tuloksia ja kenttien väliset vaikeustasoerot ovat selkeästi nähtävissä. Näin ollen tutkimuskysymykseen voidaan vastata myöntävästi.

4.4.2 Tutkimuskysymys 2

Toisessa tutkimuskysymyksessä kysyttiin: "Onko opetetun agentin suoriutuminen bullet hell -genren pelaajien tasolla?". Tuloksista nähdään, ettei opetetun agentin suoriutuminen vastaa pelaajadataa kovin hyvin. Agentin suoriutuminen sinällään kyllä mukailee kenttien vaikeusasteita verrattaessa tuloksia pelaajadataan, mutta varsinkin prototyypin kahden ensimmäisen kentän osalta agentin läpäisykertoimessa on suuri ero suhteessa pelaajadataan. Tämä johtui mallin ylisovittumisesta viimeisiin kenttiin. Opetusvaiheessa pyrittiin opettamaan erityyppisiä vihollisaaltoja niin, ettei ylisovittumista tapahtuisi, mutta ylisovittamisen estäminen osoittautui haasteelliseksi. Kenttien 1 ja 2 vihollisaaltojen muodostelmat olivat lähellä toisiaan, jolloin aikaisemmilla opetuskerroilla tapahtui ylisovittumista kyseisiin kenttiin. Tuolloin agentin läpäisyprosentit olivat huomattavasti paremmat kentissä 1 ja 2, mutta myöhempiä kenttiä agentti ei pystynyt läpäisemään lainkaan. Tämän vuoksi päädyttiin lisäämään muiden kenttien ja niiden vihollisaaltojen ilmaantuvuutta opetuksen yhteydessä. Tällainen kenttäkohtainen ylisovittuminen on haasteellinen asia ratkaistavaksi, sillä kehittäjän voi olla hyvin vaikea päätellä, mikä olisi mahdollisimman tasapainoinen opetussuhde eri kenttien osalta.

Useiden bullet hell -kenttien tapauksessa agentin oppiminen riippuu siis paljolti kenttien

eroavaisuuksista. Mikäli kenttien välillä on suuria eroavaisuuksia, on agentilla haasteellista oppia sellainen menettelytapa, joka toimii jokaisessa kentässä. Tämän tutkielman puitteissa kehitetyssä prototyypissä kenttien rakenne oli sama ja muuttuvina ominaisuuksina olivat vihollisaallot. Silti ylisovittumista tapahtui kehitysvaiheen testeissä, kun tietty kenttä toistui opetusvaiheessa liian usein tai tiettyjen kenttien vihollisaaltojen samankaltaisuuden vuoksi. Kokonaisen pelin opettamisen kannalta on hyvin tärkeää, että agentille opetetaan tasaisesti kaikkia kenttiä. Mikäli kuitenkin opetusvaiheessa toistuu liian usein esimerkiksi kaksi samankaltaista kenttää, niin ongelmaksi voi muodostua agentin pyrkimys oppia menettelytapa, joka toimii parhaiten ensisijaisesti näissä kahdessa kentässä.

4.4.3 Tutkimuskysymys 3

Kolmas tutkimuskysymys kuului: "Miten ja millä menetelmillä koneopetettua agenttia voidaan hyödyntää bullet hell -genren pelitestauksessa, jotta sen hyödyntäminen on kehitysajan kannalta käytännöllistä?". Pelikehityksen kannalta kokonaisen bullet hell -pelin balanssin testaaminen tuntuu työläältä prosessilta, mikäli samalla pitäisi kehittää myös itse peliä. Yksittäisen tehtävän suorittamisen suhteen agentin oppimistuloksia voidaan arvioida muutamien tuntien jälkeen. Tässä tutkielmassa agenttia kuitenkin opetettiin haasteellisen prototyypin kaikkien kenttien läpäisyyn, jolloin agentin tuloksia pystyi tarkemmin arvioimaan tuntien sijasta vasta useiden vuorokausien jälkeen. Tässä tapauksessa pienten muutosten testaaminen agentin oppimiseen vie suhteettomasti aikaa. Tämän osalta täytyy siis kuitenkin muistaa, että tarkasteltavana genrenä oli haasteellinen bullet hell -genre. Vaikeustasoltaan helpommat genret ovat kehittäjän työmäärän kannalta huomattavasti kevyempiä koneopetetun agentin hyödyntämiseen. Työmäärää arvioitaessa kyse on tietenkin myös resursseista. Ainakin indie-kehittäjälle näin laajamittainen toteutus tuottanee lisääntyvän kehitysajan vuoksi lisähaasteensa. Vastaavasti taas suuremmassa pelifirmassa resurssit eivät välttämättä ole niin suuri ongelma.

Bergdahl ym. (2020) esittivät, että opettamisen aikaa, joka agentilla kuluu kentän hallitsemiseen voidaan pitää indikaattorina sille, kuinka vaikea kenttä on ihmispelaajalle. Tässä tutkielmassa suoritettiin myös kenttäkohtainen opettaminen, jonka perusteella voidaan todeta, että haasteellisessa bullet hell -genren pelissä jokaisen kentän opettaminen olisi kehittäjälle

hyvin aikaavievä prosessi. Tutkimusta varten kehitetyssä prototyypissä oli vain neljä kenttää, joten kokonaisen pelin kaikkien kenttien opettaminen erikseen ei olisi kovin mielekästä.

Koneopetettu agentti osoittautui kuitenkin jo prototyypin kehitysvaiheessa hyödylliseksi, sillä sen avulla löytyi niin loogisia bugeja kuin vihollisen toimintalogiikasta johtuneita kenttäkohtaisia balansointibugeja. Tällaisia bugeja kehittäjä ei välttämättä kovinkaan helposti pysty huomaamaan. Prototyypin osalta yksinkertaisina esimerkkeinä tällaisista bugeista mainittakoon collider-komponenttien välissä oleva hyvin pieni aukko, josta agentti oppi pakenemaan pelikentältä käyttäen syöksyväistöä oikea-aikaisesti. Lisäksi kehitysvaiheessa vihollisen ampumisetäisyys oli lyhyempi kuin kentän koko, jolloin agentti pyrki pysymään riittävän kaukana vihollisista välttääkseen osumat ja ampumaan vihollisia vain tämän etäisyyden päästä. Nämä ovat kenttäsuunnittelua ajatellen tärkeitä tietoja, jotta voidaan välttää tällaisten ongelmien päätyminen lopulliseen versioon. Varsinkin visuaalisesti näkymättömien bugien kohdalla agentin hyödyllisyys suhteessa ihmistestaajiin luonnollisesti korostuu.

Prototyypissä pelaajahahmolla oli vain yksi asetyyppi, eikä pelaajahahmoon liittyvää pelin sisäistä progressiota ollut. Tällaiset pelin aikana muuttuvat ominaisuudet voivat tuottaa agentin kannalta hyvin kompleksisen logiikan, mikä hidastaa agentin oppimista huomattavasti. Laskentatehon lisääminen ja kehittyvät koneoppimismenetelmät toki mahdollistavat yhä kompleksisempien ympäristöjen oppimisen.

Kuten Aponte, Levieux ja Natkin (2009) mainitsivat, agentin käyttäytymisen ollessa parhaimmillaan stabiilimpi kuin ihmispelaajan, se voi ainakin teoriassa antaa ihmispelaajaa luotettavampia tuloksia pelin suorittamisesta. Pelisuunnittelijan tulee kuitenkin huomioda pelin vaikeuskäyrää suunnitellessa, että ihmispelaaja oppii pelin edetessä pelimekaniikkojen käytön yhä paremmin. Tämän vuoksi agentin käyttäminen vaikeustason määrittämiseen vaatii silti pelisuunnittelijan omaa arviointia ja näkemystä. Agentin puutteena voidaan myös pitää sitä, ettei sen käyttäytymisestä pystytä päättelemään esimerkiksi ohjauslaitteen kontrollien haastavuutta. Lisäksi pelissä voi olla esimerkiksi erilaisia visuaalisia efektejä, jotka vaikuttavat haasteeseen: kameran tärähdys räjähdyksen tapahtuessa voi vaikuttaa pelaajan pelaamiseen, kun taas pelimaailmassa toimivaan agenttiin sillä ei ole vaikutusta.

5 Yhteenveto

Tutkimuksessa selvitettiin koneoppimisen keinoin koneopetetun agentin soveltuvuutta pelitestauskäyttöön. Tutkimuksen kohteena oli erityisesti vaikeusasteen balansointi koneopetetun agentin avulla. Agentin soveltuvuutta arvioitiin niin käytännöllisyyden kuin luotettavuuden osalta. Luotettavuutta selvitettiin vertaamalla agentin suoritustuloksia käyttäjädataan. Tutkimustuloksista kävi ilmi, ettei agentin taitotaso ihan yltänyt bullet hell -genren pelaajan taitotasolle, kun tarkastellaan agentin suoriutumista prototyypin kaikkien kenttien suhteen. Tulos ei ollut kuitenkaan yllätys, sillä genressä vaaditaan hyvin nopeaa reagointia ja päätöksien tekemistä refleksinomaisesti. Agentin tulisi oppia menettelytapa, joka on hyvin tehokas virheiden välttämiseen. Agentin tulokset kuitenkin mukailevat käyttäjädatan tuloksia niin, että kenttien vaikeustasoerot voidaan nähdä.

Agenttia opetettiin läpäisemään prototyypin kaikki kentät. Kehitysvaiheessa kuitenkin tuli selkeästi ilmi, että on hyvin haastavaa opettaa agenttia niin, ettei ylisovittumista tapahtuisi tiettyä kenttää kohtaan. Kun opetusvaiheessa toistuu tietty kenttä tai joidenkin kenttien tietty ominaisuus, agentti oppii nämä kentät tai ominaisuudet paremmin verrattuna toisiin. Tämä osoittautui suurimmaksi epäluotettavuustekijäksi.

Jatkotutkimuksena olisi hyödyllistä tutkia ympäristön siirtämistä tehokkaalle palvelimelle tai pilvipalveluun lisälaskentatehon vuoksi. Lisälaskentatehon avulla agenttia voitaisiin opettaa tehokkaammin oppimaan bullet hell -genren tai jonkin muun haasteellisen genren pelimekaniikat yhä paremmin. Tämä mahdollistaisi näin ollen myös ajankäytöllisesti tehokkaamman tutkimisen ylisovittamisen ongelmaan liittyen.

Yleisesti ottaen koneopetettu agentti osoittautui kuitenkin hyödylliseksi pelitestauksen apuvälineeksi. Ihmistestaajiin verrattuna agentin hyötynä voidaan pitää sen tehokkuutta löytää loogisia bugeja, joita on hyvin hankala huomata. Liika kompleksisuus pelilogiikassa tuo kuitenkin haastetta agentin tehokkaan oppimisen suhteen. Kokonaisen pelin ja sen kaikkien kenttien oppiminen voi viedä suhteettomasti kehitysaikaa, ellei sitten pelin logiikka ole hyvin yksinkertainen ja sisällä vain vähän dynaamista sisältöä.

Lähteet

Adams, Ernest, ja Joris Dormans. 2012. *Game Mechanics : Advanced Game Design.* Voices That Matter. New Riders. ISBN: 9780321820273.

Aggarwal, Charu C. 2018. *Neural Networks and Deep Learning: A Textbook.* Springer. ISBN: 978-3-319-94462-3. https://doi.org/10.1007/978-3-319-94463-0.

Aleem, Saiqa, Luiz Capretz ja Faheem Ahmed. 2016. "Critical Success Factors to Improve the Game Development Process from a Developer's Perspective". *Journal of Computer Science and Technology* 31 (syyskuu): 925–950. https://doi.org/10.1007/s11390-016-1673-z.

Aponte, Maria-Virginia, Guillaume Levieux ja Stéphane Natkin. 2009. "Measuring the level of difficulty in single player video games". *Entertainment Computing* 2 (tammikuu). https://doi.org/10.1016/j.entcom.2011.04.001.

Bergdahl, Joakim, Camilo Gordillo, Konrad Tollmar ja Linus Gisslén. 2020. "Augmenting Automated Game Testing with Deep Reinforcement Learning". Teoksessa *2020 IEEE Conference on Games (CoG),* 600–603. https://doi.org/10.1109/CoG47356.2020.9231552.

Berner, Christopher, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi ym. 2019. "Dota 2 with Large Scale Deep Reinforcement Learning". arXiv: 1912.06680 [cs.LG].

Campesato, Oswald. 2020. *Artifical Intelligence, Machine Learning, And Deep Learning.* Mercury Learning / Information LLC.

Csikszentmihalyi, Mihaly. 1991. *Flow: The Psychology of Optimal Experience.* New York, NY: Harper Perennial, maaliskuu. ISBN: 0060920432.

De Mesentier Silva, Fernando, Scott Lee, Julian Togelius ja Andy Nealen. 2017. "AI-Based Playtesting of Contemporary Board Games". Teoksessa *Proceedings of the 12th International Conference on the Foundations of Digital Games.* FDG '17. Association for Computing Machinery. ISBN: 9781450353199. https://doi.org/10.1145/3102071.3102105.

Goodfellow, Ian, Yoshua Bengio ja Aaron Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville ja Yoshua Bengio. 2014. "Generative Adversarial Networks". arXiv: 1406.2661 [stat.ML].

Gudmundsson, Stefan Freyr, Philipp Eisen, Erik Poromaa, Alex Nodet, Sami Purmonen, Bartlomiej Kozakowski, Richard Meurling ja Lele Cao. 2018. "Human-Like Playtesting with Deep Learning", 1–8. Elokuu. https://doi.org/10.1109/CIG.2018.8490442.

Ho, Jonathan, ja Stefano Ermon. 2016. "Generative Adversarial Imitation Learning". Teoksessa *Advances in Neural Information Processing Systems 29,* 4565–4573. Curran Associates, Inc. http://papers.nips.cc/paper/6391-generative-adversarial-imitation-learning.pdf.

Isaksen, Aaron, Dan Gopstein ja Andy Nealen. 2015. "Exploring Game Space Using Survival Analysis". Teoksessa *FDG 2015.* http://www.fdg2015.org/papers/fdg2015_paper_02.pdf.

Juliani, Arthur, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy ym. 2020. "Unity: A General Platform for Intelligent Agents". arXiv: 1809.02627v2.

Juul, Jesper. 2003. "The game, the player, the world: looking for a heart of gameness". Teoksessa *DiGRA Conference 2003 - Proceedings of the 2003 DiGRA International Conference: Level Up.* ISBN: ISSN 2342-9666. http://www.digra.org/wp-content/uploads/digital-library/05163.50560.pdf.

Koster, Raph. 2004. *Theory of Fun for Game Design.* Paraglyph Press. ISBN: 9781306810487. http://ebookcentral.proquest.com/lib/jyvaskyla-ebooks/detail.action?docID=3384727.

Kristensen, Jeppe Theiss, ja Paolo Burelli. 2020. "Strategies for Using Proximal Policy Optimization in Mobile Puzzle Games". *International Conference on the Foundations of Digital Games* (syyskuu). https://doi.org/10.1145/3402942.3402944.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra ja Martin Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning". abs/1312.5602. arXiv: 1312.5602.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves ym. 2015. "Human-level control through deep reinforcement learning". *Nature* 518 (helmikuu): 529–33. https://doi.org/10.1038/nature14236.

Murphy, Kevin P. 2012. *Machine Learning: A Probabilistic Perspective.* The MIT Press. ISBN: 0262018020.

Ng, Andrew, ja Stuart Russell. 2000. "Algorithms for Inverse Reinforcement Learning". *ICML '00 Proceedings of the Seventeenth International Conference on Machine Learning* (toukokuu). https://ai.stanford.edu/~ang/papers/icml00-irl.pdf.

Nielsen, Michael A. 2015. *Neural Networks and Deep Learning.* Determination Press. http://neuralnetworksanddeeplearning.com.

Norman, Donald A. 2013. *The Design of Everyday Things: Revised and Expanded Edition.* Kindle edition. New York: Basic Books. ISBN: 978-0-465-07299-6.

Pathak, Deepak, Pulkit Agrawal, Alexei A. Efros ja Trevor Darrell. 2017. "Curiosity-Driven Exploration by Self-Supervised Prediction". Teoksessa *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW),* 488–489. https://doi.org/10.1109/CVPRW.2017.70.

Ramachandran, Prajit, Barret Zoph ja Quoc V. Le. 2017. "Searching for Activation Functions". arXiv: 1710.05941v2 [cs.NE].

Roohi, Shaghayegh, Asko Relas, Jari Takatalo, Henri Heiskanen ja Perttu Hämäläinen. 2019. "Predicting Game Difficulty and Churn Without Players". *34th IEEE/ACM International Conference on Automated Software Engineering (ASE),* https://doi.org/10.1145/3410404.3414235.

Rosenblatt, Frank. 1958. “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review* 65 6:386–408. https://doi.org/10.1037/H0042519. http://web.stanford.edu/class/psych209a/ReadingsByDate/01_30/Rosenblatt58Perceptron.pdf.

Rumelhart, David E., Geoffrey E. Hinton ja Ronald J. Williams. 1986. “Learning representations by back-propagating errors”. *Nature* 323:533–536. https://doi.org/10.1038/323533a0.

Salen, Katie, ja Eric Zimmerman. 2003. *Rules of Play: Game Design Fundamentals.* The MIT Press. ISBN: 0262240459.

Sammut, Claude. 2017. “Behavioral Cloning”. Teoksessa *Encyclopedia of Machine Learning and Data Mining,* toimittanut Claude Sammut ja Geoffrey I. Webb, 120–124. Boston, MA: Springer US. ISBN: 978-1-4899-7687-1. https://doi.org/10.1007/978-1-4899-7687-1_69.

Schell, Jesse. 2020. *The Art of Game Design: A Book of Lenses, 3rd Edition.* CRC Press, Taylor / Francis Group. ISBN: 9781138632059. https://doi-org.ezproxy.jyu.fi/10.1201/b22101.

Schulman, John, Sergey Levine, Philipp Moritz, Michael Jordan ja Pieter Abbeel. 2017. “Trust Region Policy Optimization”. arXiv: 1502.05477v5 [cs.LG].

Schulman, John, Philipp Moritz, Sergey Levine, Michael I. Jordan ja Pieter Abbeel. 2018. “High-Dimensional Continuous Control Using Generalized Advantage Estimation”. arXiv: 1506.02438 [cs.LG].

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford ja Oleg Klimov. 2017. “Proximal Policy Optimization Algorithms”. arXiv: 1707.06347 [cs.LG].

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser ym. 2016. “Mastering the game of Go with deep neural networks and tree search”. *Nature* 529, numero 7587 (tammikuu): 484–489. ISSN: 1476-4687. https://doi.org/10.1038/nature16961.

Sutton, Richard S., ja Andrew G. Barto. 2018. *Reinforcement Learning.* MIT Press. http://incompleteideas.net/sutton/book/RLbook2018.pdf.

Sweetser, Penelope, ja Peta Wyeth. 2005. “GameFlow: A Model for Evaluating Player Enjoyment in Games”. (New York, NY, USA) 3, numero 3 (heinäkuu): 3. https://doi.org/10.1145/1077246.1077253.

Wiering, Marco, ja Martijn van Otterlo. 2012. *Reinforcement Learning: State-of-the-Art.* Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-27645-3.

Yasoubi, Ali, Reza Hojabr ja Mehdi Modarressi. 2016. “Power-Efficient Accelerator Design for Neural Networks Using Computation Reuse”. *IEEE Computer Architecture Letters* 16 (tammikuu): 1–1. https://doi.org/10.1109/LCA.2016.2521654.

Zheng, Yan, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen ja Changjie Fan. 2019. “Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning”. *34th IEEE/ACM International Conference on Automated Software Engineering (ASE),* 772–784. https://doi.org/10.1109/ASE.2019.00077.

Zhifei, Shao. 2012. “A review of inverse reinforcement learning theory and recent advances”, 1–8. Kesäkuu. ISBN: 978-1-4673-1510-4. https://doi.org/10.1109/CEC.2012.6256507.

Zuo, Guoyu, Kexin Chen, Jiahao Lu ja Xiangsheng Huang. 2020. “Deterministic generative adversarial imitation learning”. *Neurocomputing* 388:60–69. ISSN: 0925-2312. https://doi.org/10.1016/j.neucom.2020.01.016.