Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Sami Hurmerinta

Koneoppiminen vuoropohjaisessa pelissä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

24.5.2019

Tekijä Otsikko Sivumäärä Aika	Sami Hurmerinta Koneoppiminen vuoropohjaisessa pelissä 38 sivua 24.5.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaaja	Lehtori Antti Laiho

Insinöörityön päätarkoituksena oli tutkia koneopitun tekoälyn soveltuvuutta vuoropohjaisen pelin päätöksentekoon. Lisäksi tarkasteltiin, onko koneoppimisesta yleisesti realistista hyötyä pelikehityksessä nykypäivänä tai tulevaisuudessa. Työn tavoitteeksi asetettiin sellaisen peliprototyypin luominen tutkimuskäyttöön, joka olisi myöhemmin laajennettavissa osaksi isompaa peliprojektia.

Jotta koneoppimisen toimivuutta pystyttiin tutkimaan, sitä varten ohjelmoitiin testiympäristön prototyyppi käyttäen Unity-pelimoottoria. Koneoppimisympäristön luontiin käytettiin ML-Agents Unity -liitännäistä. Oppimistuloksia ja poikkeavuuksia oppimisessa tarkkailtiin TensorBoard-työkalulla. Lopputulosten oppimismalleja analysoitiin ja pohdittiin prosessin toimivuutta pelikehityksessä.

Pohdinnan tuloksena todettiin, että ML-agentit toimivat paremmin kuin oli odotettavissa, mutta luultavasti parantamisenkin varaa voisi olla. Kuitenkin Unity ML-Agents -työkalun tehokkuus indie-pelien kehityksessä ja pelitestauksessa osoittautui erinomaiseksi useisiin käyttötarkoituksiin.

Työssä saatiin aikaiseksi toimiva peli, joka kykenee hyödyntämään koneopetettua tekoälyä itsenäiseen oppimiseen ja pelaamaan itseään vastaan. Jatkossa peliä on tarkoitus laajentaa osaksi isompaa kokonaisuutta sekä parantaa koneopetutun mallin suorituskykyä.

Avainsanat	pelikehitys, koneoppiminen, Unity, vuoropohjainen

Author Title Number of Pages Date	Sami Hurmerinta Machine learning in a turn-based game 38 pages 24 May 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Antti Laiho, Senior Lecturer

The purpose of this final year project was to clarify whether machine learned artificial intelligence is suitable and easy enough to implement in turn-based games or not. The study was furthermore used to clarify if machine learning is realistically useful for indie games development. The objective was to develop a prototype of a game that can be used for this kind of research and is later extendable into a larger game project.

The game prototype was built with Unity game engine, and a toolkit called Unity ML-Agents was used to train the learning models. The results of the machine learning process were closely inspected in the TensorBoard data visualizer, and an analysis was made referring to these results.

The analysis found that the Unity ML-Agents toolkit's performance provided to be excellent in indie game development and application testing. ML-Agents worked better than expected even though there could be room for improvement as well.

The result was a working game that is capable of using a machine learning model as artificial intelligence and play against itself to self-learn. In the future, the game and the training model will be further developed.

Keywords	game development, machine learning, Unity, turn-based

Sisällys

1 Johdanto

Tekoäly on laaja-alainen teknologia ja tieteen laji, jota ei ole pystytty tarkasti määrittelemään. Alan asiantuntijat ovat kuvailleet tekoälyn olevan vain jotain, mitä ei ole vielä olemassa, tai jotain hienoa, mitä tietokoneet eivät osaa tehdä. Epätarkat kuvaukset osoittavat, miten vaikeaa tekoäly on määritellä. Tavallisessa merkityksessä tekoäly on mikä tahansa itsenäiseen toimintaan kykenevä asia.

Koneoppiminen on yksi tekoälyn osa-alueista, jossa tietokone toistuvasti havainnoi ja tulkitsee omia päätöksiään ilman ihmisen läsnäoloa. Perinteisten tekoälyn raja-arvoihin tukeutuvien mallien sijasta koneoppimisessa käytetään koneoppimisalgoritmeja. Yksi tällä hetkellä trendikkäimmistä algoritmisista lähestymistavoista on syväoppiminen, joka perustuu keinotekoisiin neuroverkkoihin.

Insinöörityön päätarkoituksena on tutkia, miten koneopittu tekoäly soveltuu pelisovelluksiin. Erityisesti tarkastellaan soveltuvuutta vuoropohjaiseen taisteluun, millaisia lähestymistapoja kannattaa käyttää ja onko koneopetetusta tekoälystä realistista hyötyä nyt tai lähitulevaisuudessa. Testiympäristöä varten luodaan toimiva prototyyppi pelistä. Prototyyppi tehdään Unity-pelimoottorilla, ja koneopetetun tekoälyn testaamisen simulointiin käytetään Unity ML-Agents -työkalupakkia. Kaikki tarvittava ohjelmointi tehdään C#-ohjelmointikielellä. TensorBoard-työkalun avulla visualisoidaan agenttien oppimistuloksia, ja etsitään mahdolliset virheet neuroverkoston kehityksessä.

Insinöörityön aiheen valinta selittyy tekijän omasta kiinnostuksesta pelisovellusten kehitykseen. Tekijä on aloittamassa pidempiaikaista pelinkehitysprojektia, joka vaatisi tekoälyltä strategisia päätöksiä ja ihmismäistä käyttäytymistä. Kyseessä on parhaillaan nopeasti kehittyvä ala, eikä selkeää vastausta tekijän esittämiin kysymyksiin ole vielä saatavilla. Lisäksi tavoitteena on pystyä osoittamaan indie-pelinkehitysyhteisölle, mitkä ovat koneoppimisen tämänhetkiset mahdollisuudet Unityn ML-agenteilla. Aiheen ajankohtaisuuden takia insinöörityössä käytetään kirjajulkaisujen lisäksi lähteinä useita verkkojulkaisuja.

2 Koneoppiminen

Koneoppiminen on tärkeä osa nykypäivän kehitystä. Yliopistot ja yritykset ovat kasvattaneet resursseja ja ajankäyttöä edistääkseen tietämystä alasta. Koneoppimisen keskeisimpänä tavoitteena on saada tietokone oppimaan ja toimimaan kuten ihminen, tekemällä havaintoja tosielämästä.

Vuonna 1943 neurofyysikko ja matemaatikko kirjottivat yhdessä, miten ihmisen aivojen neuronit voisivat toimia. Osoittaakseen toiminnan käytännössä, he rakensivat neuroverkon elektronisista piireistä. Vuonna 1952 kehitettiin IBM-tietokoneelle ohjelma, joka oppi pelaamaan tammipeliä sitä paremmin, mitä kauemmin tietokone pelasi sitä. Ohjelman kehittäjä Arthur Samuel kumosi tällä uskomuksen, etteivät tietokoneet pystyisi oppimaan kuten ihmiset. Samuel määritteli koneoppisen tutkimusalaksi, joka antaa tietokoneelle kyvyn ilman erillistä ohjelmointia. Muutamaa vuotta myöhemmin kehitettiin MADALINE, joka käytti neuroverkoilla toimivaa adaptiivista suodatinta poistaakseen puheluista kaiut. Se oli ensimmäinen neuroverkko, joka ratkaisi tosielämän ongelman. Suodatin on edelleen käytössä nykypäivän puheluissa. [1.]

Vuosikymmeniä myöhemmin, vuonna 1997, IBM:n kehittämä tietokone voitti ensimmäisen kerran shakin mestarin. Garry Kasparov vaati uusintaottelua, mutta IBM ei suostunut uusintaan. Vuonna 2012 Googlen luoma neuroverkko oppi luokittelemaan kissat ja ihmiset YouTube-videoiden esikatselukuvista. Simulaatioon syötettiin kolmen päivän aikana 10 miljoonaa satunnaista kuvaa, ja 20 000 kuvan jälkeen tietokone oppi itsenäisesti tunnistamaan kuvista kissat. [2; 3.] Seuraava peleihin liittyvä historiallinen hetki koettiin vuonna 2015, kun Googlen kehittämä AlphaGo voitti ensimmäisen kerran ammattilaispelaajan go-lautapelissä, jota pidetään yhtenä maailman vaikeimpana lautapelinä.

Pari vuotta myöhemmin Dota 2 -tietokonepelille luotiin botti, joka pystyi voittamaan maailman parhaat pelaajat yhden pelaajan välisessä taistelussa. Tämä oli suuri saavutus, sillä Dota 2:ta pidetään tietokonepelinäkin hyvin monimutkaisena, kuten go:ta tai shakkia lautapelien keskuudessa. OpenAI:n kehitystiimi jatkoi botin kehitystä soveltuvaksi myös viisi vastaan viisi -taistoihin, ja vuotta myöhemmin se voitti kokonaisen joukkueen parhaita pelaajia. OpenAI pystyi ennustamaan omat voittomahdollisuudet jo ennen pelin alkua ja pelin aikana. Se myös osasi suunnitella valitsemansa seuraavat toimenpiteet. [4.]

Vaikka koneoppimisen menetelmät ovat olleet olemassa jo vuosikausia, on nyt vasta päästy tilanteeseen, jossa koneoppimista pystytään helposti soveltamaan lähes mihin tahansa tosielämän ongelman ratkaisuun. Asiaa ovat edistäneet useat isojen yritysten kehittämät työkalut, ohjelmistokomponentti- ja matematiikkakirjasto, joissa on myös useasti vapaa lähdekoodi. Tämäntyyppiset kirjastot ja työkalut mahdollistavat helpomman ML-algoritmien implementoinnin pieneenkin projektiin, ilman syvällistä matematiikan ja teknologioiden osaamista. Viime aikoina koneoppimista on käytetty paljon muun muassa kuvien vaativaan ja automatisoituun käsittelyyn. Tavallisesta valokuvasta pystytään generoimaan esimerkiksi Picasson maalauksen näköinen tai muuntamaan videossa esiintyvän henkilön näyttämään toiselta henkilöltä, tai jopa generoimaan henkilöiden kasvokuvia, joita ei ole oikeasti olemassa. Nämä ovat tosin vielä aikaisessa kehitysvaiheessa.

Automaattinen valokuvaus on yksi Googlen kehittämä moderni sovellus, joka käytännössä ottaa jatkuvaa videokuvaa, mutta tallentaa parhaat hetket valokuvina, esimerkiksi sellaisen valokuvan, jossa kaikki kuvassa olevat henkilöt hymylevät tai että jotain mielenkiintoista tapahtuu. Toisin kuin aikaisemmin suunnitelluissa oppimisalgoritmeissa, joissa kuvasta on yritetty tunnistaa, onko objekti kissa vai ei, tässä yritetään tunnistaa kuvan laatua ja se, onko kuvan objekti mielenkiintoinen vai ei. [5.]

3 Algoritmit ja menetelmät

Algoritmi on vaiheittainen menetelmä, jolla päästään haluttuun lopputulokseen. Koneoppimisen algoritmeja käytetään tietotekniikassa datan louhintaan, jotta vältetään ihmisen tekemät virheet analyysissa. Prosessien optimoinnissa sillä voidaan korvata tietyntyyppistä koodia. Parhaan lopputuloksen saavuttamiseksi algoritmia voidaan toistaa useaan kertaan. Koneoppimisen algoritmit jaotellaan kolmeen eri kategoriaan, jotka ovat ohjattu oppiminen, ohjaamaton oppiminen ja vahvistusoppiminen. Käytännöllisyyden nimissä nämä kategoriat voidaan vielä ryhmitellä neljään luokkaan, jotka ovat klusterointi, regressio, luokittelu ja poikkeavuuksien havaitseminen. Tässä insinöörityössä ei kuitenkaan paneuduta syvällisesti kaikkiin menetelmiin, vaan annetaan lukijalle lähinnä pintapuolinen käsitys siitä, mitä näillä tarkoitetaan.

3.1 Koneoppimisen päämenetelmät

Mikäli algoritmille syötetään opetusdatana valmiita luokiteltuja esimerkkejä, joiden oikea vastaus on merkittynä jokaisen kohdalle, voidaan menetelmä kategorisoida ohjatuksi oppimiseksi. Menetelmä soveltuu erityisesti neuroverkkoihin, jossa käsitellään suuria määriä dataa. Ohjatuksi koneoppimiseksi voidaan kutsua myös regressio-ongelmaa, kun oikeaa vastausta ei ole luokiteltu, vaan se on lukuarvo. Esimerkki tällaisesta voisi olla keskitetyn mainonnan sovellus, joka perustuu hakukonetulosten klikkausten tunnistamineen verrattuna aikaisempaan verkkokäyttäytymiseen.

Puoliohjatuksi voidaan kutsua metodia, jossa algoritmille on syötetty vain muutamia luokiteltuja esimerkkejä arvoiksi, mutta tämän lisäksi käytetään suuria määriä tuntemattomia arvoja. Ohjatun ja ohjaamattoman koneoppimisen tekniikoita voidaan käyttää ristiin. [6.]

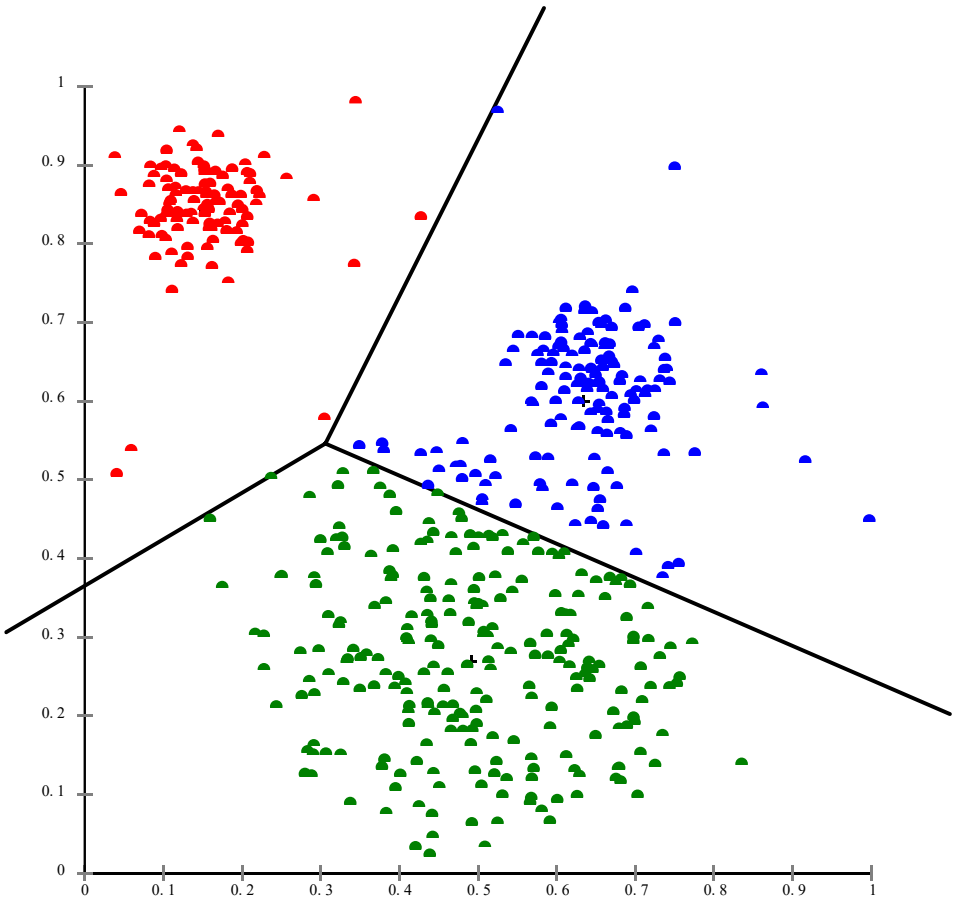
Ohjaamaton oppimismenetelmä on hyödyllinen, mikäli esimerkkidataa ei ole valmiiksi saatavilla, eli tässä prosessissa ei käytetä luokiteltuja esimerkkejä. Sen sijaan järjestelmälle voidaan asettaa tehtäväksi löytää kuvioita ja niiden korrelaatiota. Tietokoneen olisi tarkoitus itsenäisesti löytää opetusdatasta jonkinlainen rakenne, esimerkiksi ryhmittelemällä datapisteet. Ohjaamattomaksi oppimiseksi voidaan kutsua datan visualisointia, jossa visuaalisesti samankaltaiset symbolit ryhmitellään. Hyvä esimerkki on läheisten

ihmisten löytäminen sosiaalisen verkoston tietokannasta kasvonpiirteiden perusteella. [6.]

Ohjatun ja ohjaamattoman koneoppimisen lisäksi kolmas tyyppiesimerkki on vahvistusoppiminen. Tässä mallissa käytetään oppimisagenttia, jota kannustetaan palkitsemalla sitä aina, kun sen toiminta on tuottanut toivotun tuloksen. Oppimisen aikana agentti alkaa vähitellen suosia toimia, joita se on aikaisemmin yrittänyt ja todennut niiden olevan tehokkaimpia palkkion tuottamiseen. Jotta kannustava malli toimii oikein, on agentin käytettävä hyväkseen aikaisemmin todettuja tapoja saada palkinto, mutta samalla tutkittava parempia tapoja saavuttaa vielä tuottavampi lopputulos. Agentin on kokeiltavat uusia vaihtoehtoja, mutta suosittava asteittain niitä päätöksiä, jotka vaikuttavat olevan parhaita sillä hetkellä. Kannustavassa oppimisessa palkinto on agentin ainoa tavoite, näkemättä ympäristön kokonaiskuvaa. Tämä asettaa haasteita oppimisen onnistumiselle. Menestys saavutetaankin yleensä vain kokeilemalla ja oppimalla virheistä. Agentti voi olla passiivinen oppilas tai aktiivinen oppilas. Passiivinen oppilas katsoo vierestä esimerkiksi ihmisen pelaamaa peliä ja yrittää oppia vaiheet häneltä. Aktiivinen oppilas oppii omatoimisesti tutkimalla ympäristöä. [7, s. 600; 8.]

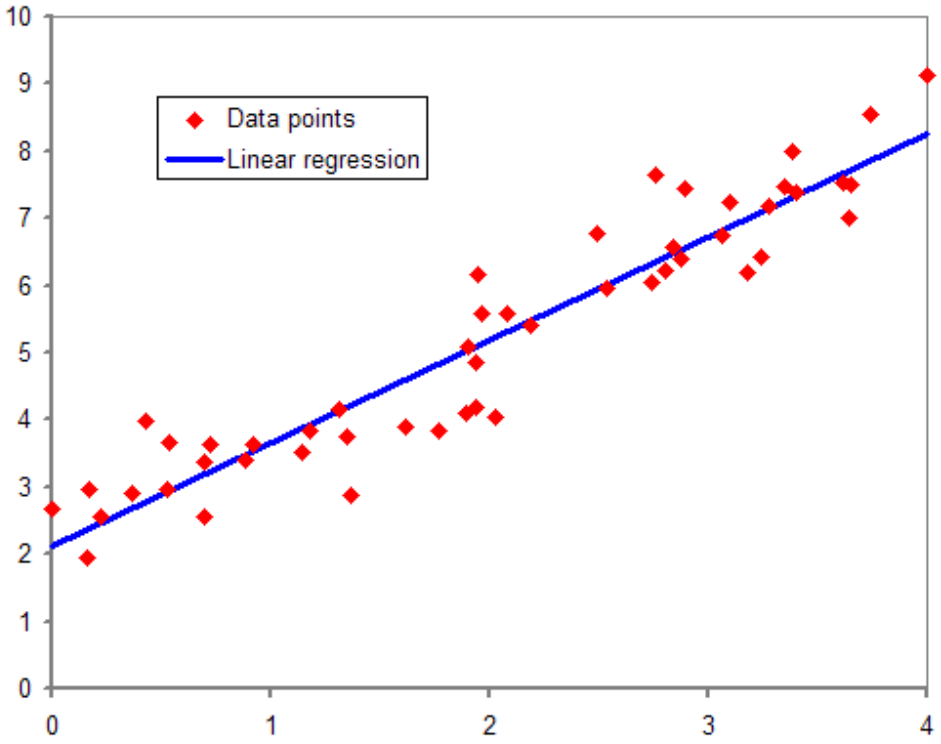
3.2 Klusterointi ja regressio

Klusteroinnilla tarkoitetaan aktiviteettia, jolla ryhmitellään aineiston näytteet jonkin yhteisen tekijän perusteella. Klusteroitavat näytteet voivat olla esimerkiksi kasvokuvia, joista halutaan tunnistaa samantyyppiset piirteet. Kuvassa 1 on muodostettu kolme klusteria, jotka on rajattu viivoilla. Viivat on saatu aikaiseksi Voronoin diagrammin avulla, joka piirtää klustereiden rajaviivat siten, että etäisyydet naapurialueiden ominaispisteistä ovat yhtäläiset. [9, s. 161]



Kuva 1. Esimerkki klusteroinnista [10].

Regressiot auttavat saavuttamaan vastauksen kysymyksiin ”Kuinka paljon?” tai ”Kuinka monta?” Lineaariregressio on yksi yleisimmistä regressiomalleista, jota käytetään koneoppimisessa, ja sitä on helppo tulkita. Vaikka algoritmi on helppo käsittää, on sillä silti saatu aikaiseksi hyödyllisiä tieteellisiä tuloksia. Lineaariregressiossa määrät ovat kertoimia tai painokertoimia, eli sillä käsitellään jatkuvia muuttujia. Kuvassa 2 nähdään, että lineaariregression tavoitteena on sijoittaa parhaiten sopiva viiva kaikille datapisteille, mikä mahdollistaa ennusteiden jäännösvariaatioiden marginaalin minimoimisen. [11, s. 208]



Kuva 2. Esimerkki lineaarisesta regressiosta, jossa viiva on asetettu datapisteiden keskelle [12].

Lineaariregression tuottamat tuloksen on mahdollista luokitella logistisella regressiolla. Toisin kuin lineaariregressiolla, logistista regressiolla pystytään käsittelemänään diskreettejä muuttujia, jotka ovat todennäköisyyksiä 0:n ja 1:n väliltä. Logistisessa regressiossa käytetään sigmoid-funktiota, joka on S-muotoinen käyrä, jolle voidaan syöttää mikä tahansa jatkuva arvo. [13, s. 142.]

3.3 Luokittelu

Luokittelua käytetään, kun pyritään arvioimaan kyllä tai ei ennustetta, kuten ”Onko kuvan sieni myrkyllinen?” Regressiolla ennustaminen ei sovellu luokitteluun, sillä yleensä halutaan arvioida 0:n tai 1:n välillä. Tämäntyyppiseen lopputulokseen päästään esimerkiksi seuraavalla logistisella funktiolla, joka on yleinen koneoppimisessa ja jota käytetään luokitteluun:

$$g(h) = \frac{1}{1 + e^{-2\beta h}}$$

k-NN- eli k-nearest neighbor -algoritmia käytetään luokittelussa ja regressio-ongelmien ratkaisuun, esimerkiksi ennusteanalyysissa pisteiden luokittelussa, joka perustuu sen naapureiden enemmistöäänestykseen. k-NN -algoritmi ei suorita koulutuspisteistä induktiivista päättelyä, mikä tarkoittaa, että koulutus on minimaalista. Tämä tarkoittaa myös, että sen koulutusvaihe on hyvin nopea. Algoritmi perustuu ominaisuuksien samankaltaisuuksien vertailuun. [8; 14.]

3.4 Poikkeavuuksien havaitseminen

Poikkeaminen on jotain, mikä erottaa jonkin asian normaalista tai odotetusta. Esimerkiksi pankit käyttävät tätä metodia tunnistamaan luottokortin epänormaalin käytön. Metodissa etsitään poikkeamia historiassa tehtyihin päätöksiin tai aktiviteetteihin. Poikkeamien havaitsemiseen voidaan käyttää monia erilaisia algoritmeja, mutta tavallisesti niiden implementointi online-tilaan on helppoa eikä niitä tarvitse valvoa. [6.]

3.5 Päätöspuut

Päätöspuu-päättelymuoto on yksi yksinkertaisimmista, mutta silti menestynein koneoppimisen algoritmeista. Päätöspuut ottavat syötteeksi arvon tai tilanteen, jota arvioidaan puun jokaisessa haarassa. Päätökset tehdään kyllä tai ei -periaatteella, eli päätöspuut edustavat Boolean-funktiota. Rajoitteena päätöspuun käytössä on se, että sillä pystytään käsittelemään vain yhtä päätöstä kerrallaan.

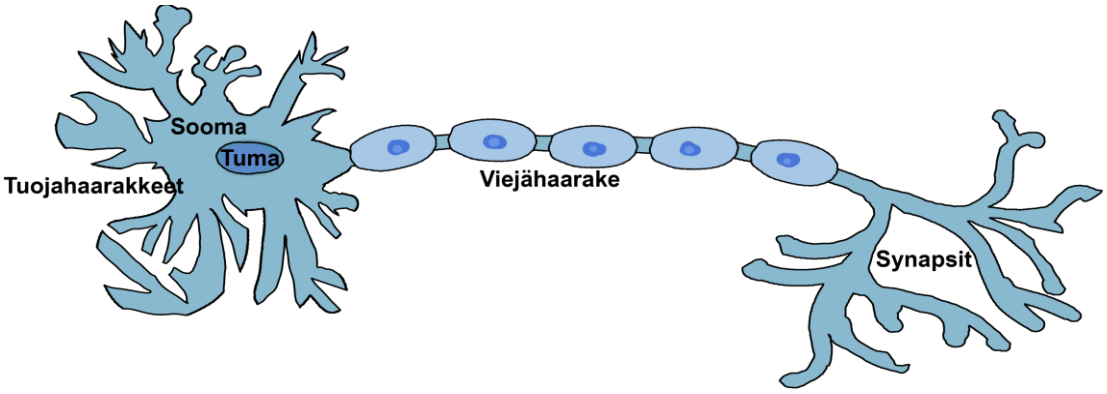
Päätöspuiden algoritmi tarjoaa yksinkertaisen esityksen sellaisista esiasteellisista tiedoista, joita voidaan käyttää jatkuvien päätösten tekemiseen ja objektien luokitteluun. Yksinkertaisuuden takia päätöspuulla ei voida ratkaista mielenkiintoisia tieteellisiä teorioita, mutta sitä käytetään useissa sovelluksissa, kuten tietokonepeleissä. [7, s. 538.]

Satunnainen metsä on päätöspuihin perustuva menetelmä, joka toimii rakentamalla lukuisia päätöspuita. Menetelmää voidaan käyttää luokittelussa, regressiossa ja muissa tehtävissä. Satunnaisuuden tarkoitus on välttyä opetuksen ylisovittamiselta.

3.6 Neuroverkot

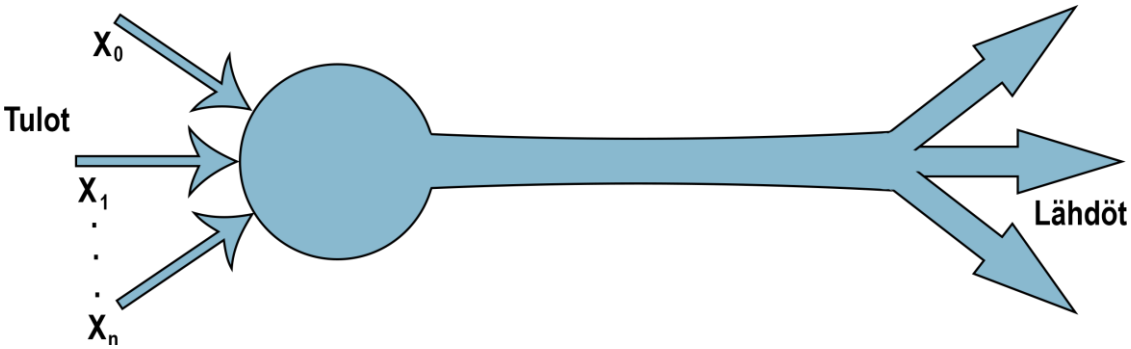
Keinotekoiset neuroverkot suunniteltiin alun perin jäljittelemään ihmisten aivoja, joten sen lisäksi, että ne on nimetty biologisen vastineensa perusteella, ne myös mallinnettiin ihmisaivojen neuronien mukaan. Keinotekoiset neuroverkot soveltuvat parhaiten oppimismalleihin, joissa käsitellään suuria määriä dataa. Neuroverkot koostuvat useista yksittäisistä neuroneista. Ne soveltuvat parhaiten kasvojen, puheen tai käsialan tunnistukseen. Niiden avulla pystytään myös opettamaan tietokonetta pelaamaan pelejä. Huonona ominaisuutena neuroverkot ovat alttiita ylisovittamiselle, joka esimerkiksi peleissä tarkoittaisi sitä, että tietokone on oppinut pelaamaan peliä täydellisesti, mutta se ei enää selviydy, mikäli peliä hieman muutetaan. Ongelmalta voidaan jossain määrin välttyä sillä, että peliä muutetaan satunnaisesti opetuksen aikana. Neuroverkot ovat kuitenkin yleensä niin joustavia, että ne saattavat mukautua mihin tahansa pelissä tapahtuvaan ilmiöön, joka aiheuttaa ylisovittamisen. [15.]

Biologisella neuronilla on tuojahaarakkeita, jotka vastaanottavat signaaleja, sooma, joka prosessoi sen, ja viejähaarake, joka lähettää signaalin seuraaville neuroneille. (Kuva 3.)



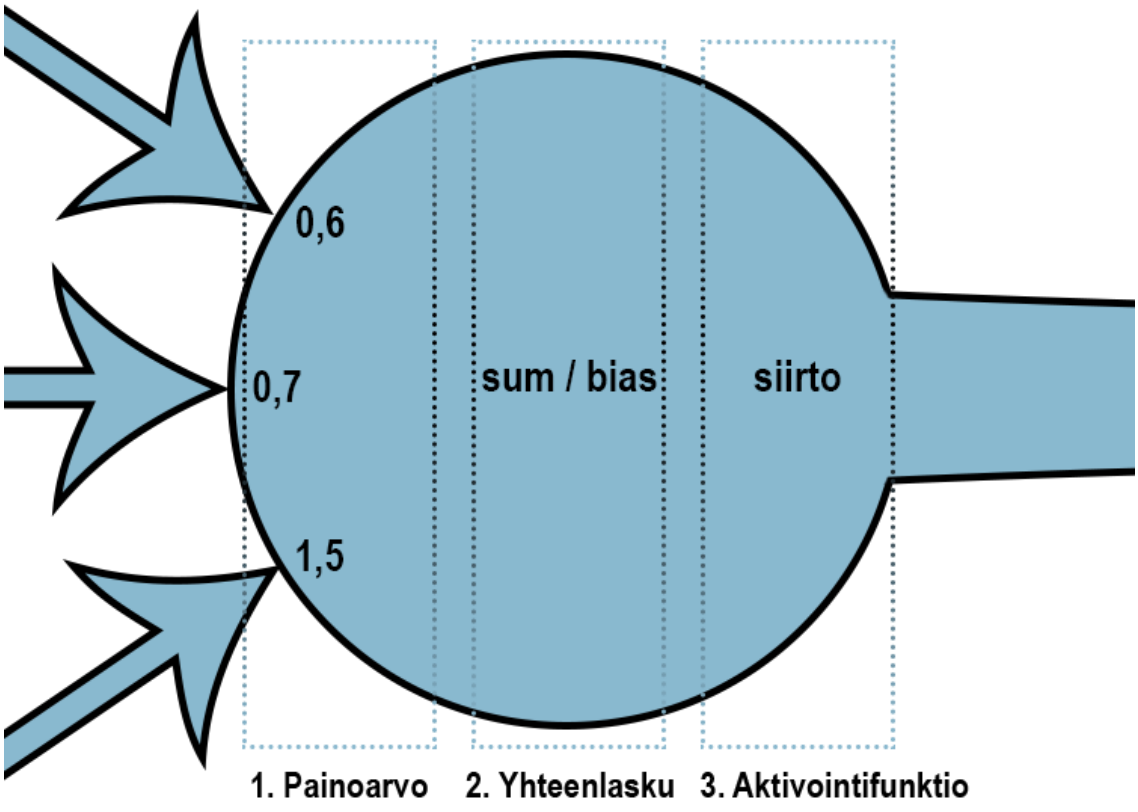
Kuva 3. Biologinen neuroni [16].

Vastaavasti keinotekoisella neuronilla (kuva 4.) on tulokanavat, prosessointivaihe ja yksi lähtö, joka voi välittää datan seuraaville neuroneille. Kun vertaillaan kuvia 3 ja 4, voidaan hahmottaa, että niiden toiminta on hyvin samantyyppinen.



Kuva 4. Keinotekoinen neuroni [16].

Kuvassa 5 voidaan tarkastella keinotekoisen neuronin prosessia tarkemmin, ja huomataan, että laskutoimitukset eivät ole kovinkaan monimutkaisia. Prosessi voidaan jakaa kolmeen vaiheeseen: painoarvo, yhteenlasku ja aktivointifunktio.



Kuva 5. Keinotekoinen neuroni lähempää tarkasteltuna [17].

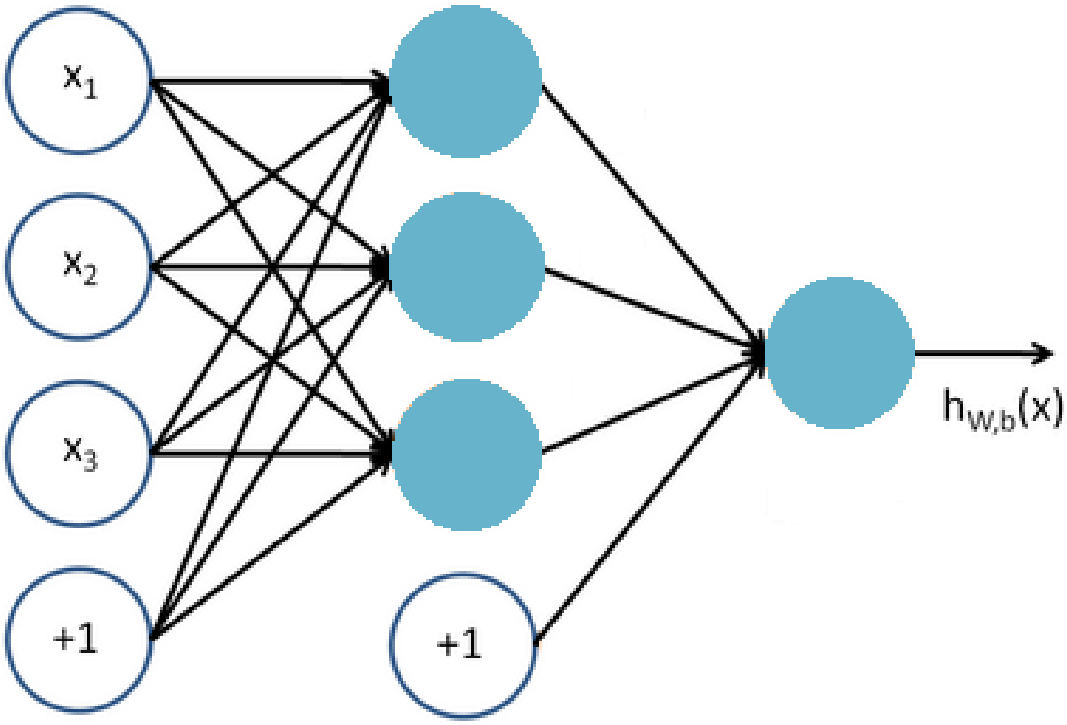
Tulo-signaalin saapuessa neuroniin, sen arvo kerrotaan painoarvolla, joka on määrätty kyseiselle tulolle. Oppimisvaiheessa neuroverkko voi korjata painoarvoja edellisen testituloksen virhearvon perusteella. Seuraavassa vaiheessa muokatut tulot lasketaan yhteen yhdeksi arvoksi ja summaan voidaan lisätä offset, jota kutsutaan bias-arvoksi. Neuroverkko korjaa bias-arvoa oppimisvaiheen aikana. Bias-arvo on ylimääräinen neuroni, joka on lisätty jokaiseen lähdekerrokseen, joka sisältää arvon ”+1”. Bias-arvot eivät ole yhteydessä mihinkään edellisen kerroksen neuroniin, joten ne eivät tässä mielessä edusta todellista aktivisuutta, mutta ne määrittävät, onko kyseinen neuroni aktivoitunut. [17.]

Painoarvot ovat siis lähtökohtaisesti satunnaisesti arvottuja arvoja, mutta jokaisen oppimisiteraation jälkeen arvot lähenevät haluttua lopputulosta. Viimeisessä vaiheessa neuronin laskutoimitus muutetaan lähtösignaaliksi seuraavaa siirtoa varten. Sigmoid-funktio, jota käytetään logistiseen regressioon, soveltuu tähän tarkoitukseen. Mutta myös esimerkiksi perceptroni on yksinkertainen binäärifunktio, jota voidaan käyttää neuronin aktivointifunktiona. Perceptroni on yksi käytetyimmistä neuroverkon osista, ja sitä käytetään binäärisessä luokittelussa. Vaikka myös lähin naapuri -menetelmä soveltuisi tähän tarkoitukseen, on perceptron-menetelmä vielä tehokkaampi. Perceptronin avulla pystytään tehostamaan luokittelua viivaamalla data kahteen osaan. Perceptroni palauttaa binääriarvon 1, mikäli arvo on positiivinen. Muussa tapauksessa palautuva arvo on aina 0. [18.]

$$f(x) = \begin{cases} 1 & if\ \omega \cdot x + b > 0 \\ 0 & else \end{cases}$$

Neuroverkko muodostuu, kun yhdistetään yksittäiset neuronit keskenään niin, että neuronin lähtö on toisen neuronin tulo.

Kuvan 6 esimerkissä on havainnollistettu verkkojen tulot. Vasemmanpuoleisessa osiossa on verkon kolme tuloa ja oikealla yksi lähtö. Ympyrät merkillä ”+1” ovat bias-arvoja. Keskimmäistä yhtymäkohtaa kutsutaan piilotetuksi kerrokseksi, koska sen arvot ovat tuntemattomia. [15.]



Kuva 6. Esimerkki neuroverkosta [17].

4 Testiympäristön komponentit

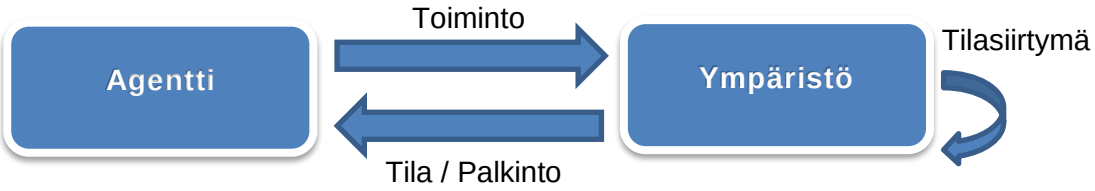
Ennen kuin siirrytään insinöörityössä tehdyn prototyypin toteutukseen, käydään läpi prototyypissä käytetyt komponentit. Tämä ympäristö toteutettiin Unityn versiolla 2018.3.8f ja Unity ML-Agents-työkalupakin versiolla 0.7. ML-Agents 0.8 julkaistiin työn aikana, mutta työkalunpakin komponentteihin ei ollut tullut muutoksia, joten päivittämiselle ei ollut tarvetta. ML-Agents-oppimisympäristön kokonaisuus muodostuu agenteista, aivoista ja akatemiasta, joka on liitettynä ulkoiseen Python-prosessiin.

4.1 Unity-pelimoottori

Unity on suosittu, harrastuskäyttöön ilmainen, monialustainen pelimoottori ja ohjelmistokehys, jolla pystyy tekemään 2D- ja 3D- pelejä sekä interaktiivisia kokemuksia. Unity tukee käytännössä ainoastaan C#-ohjelmointikieltä, sillä JavaScriptin tyyppisten UnityScript- ja Boo-ohjelmointikielien tuki on päättymässä. Pelien rakenne muodostetaan peliobjekteista, joiden sisällä on skriptattavia komponentteja. Vaihtoehtoisesti voidaan käyttää skriptattavia objekteja, joita ei tarvitse liittää pelialueelle. Skriptattavia objekteja käytetään yleensä datan tallennukseen. Unity ML-Agents-työkalupakki hyödyntää niitä aivojen helpompaan käyttöön. [19; 20.]

4.2 Agentti

Agentti on vahvistusopetuksessa yleisesti käytetty toimija, joka tutkii ympäristöään ja tekee päätökset havaintojen perusteella. Jokaiselle peliin asetettavalle agentille on annettava aivot, jotka kiteyttävät päätöksentekoprosessin. Agenttia palkitaan signaaliksi siitä, että se on tehnyt oikean valinnan. Tämän prosessin toimintaa voi hahmottaa kuvasta 7.



Kuva 7. Vahvistusopetuksen oppimisjakso [21].

4.3 Aivot

Unityn ML-Agents-työkalupakin mukana tulee ominaisuus nimeltä aivot, joka on yksinkertaisuudessa agentin tekoälyn malli. Aivot ovat 0.7-versiossa skriptattavia objekteja, mikä mahdollistaa niiden helpomman käytön vaihtuvissa pelinäkymissä. Aivoille annetut parametrit vaikuttavat oppimisen nopeuteen ja laatuun, ja niiden tehtävänä on päättää niihin liitettyjen agenttien toiminnot. Tekoälynä pystytään käyttämään heuristisia tai itsenäisesti opetettuja aivoja. Heuristissa aivoissa päätöksenteon logiikka on käsin koodattu, ja suurin osa nykypäivän pelien tekoälyistä perustuukin heuristisiin algoritmeihin. Lisäksi on olemassa PlayerBrains-nimiset aivot, joiden avulla kehittäjä pystyy itse ohjaamaan ja testaamaan agenttien toimintoja.

Aivojen parametrien määritys on olennaisimmassa osassa oppimisen onnistumista, ja niille täytyykin kertoa, kuinka monta havainnointivektoria tarkkaillaan, sekä ovatko ne diskreettisiä vai jatkuvia. Havainnot kerrotaan aivoille AddVectorObs(); -funktiolla. Vektorihavaintojen tulisi sisältää kaikki muuttujat, jotka ovat olennaisia agenttien tietoon perustuvassa päätöksessä. Tapauksissa, jossa havaintoja täytyy muistaa, käytetään vektoreiden pinoarvoa, jolloin agentit pinoavat havaintoja menneisyydestä. [22.]

4.4 Akatemia

Akatemia on objekti, jonka lapsia ovat kaikki aivot peliympäristössä. Akatemia hallinnoi opetuksen kulkua, ja sille määritellään asioita, kuten renderöinnin ja ajan kehysnopeus. Renderöinnin kehysnopeudella on merkitystä, mikäli agenttien tai pelin toiminnot ovat riippuvaisia pelin ruudunpäivitysnopeudesta. Mikäli aika kulkee nopeammin kuin peli renderöi, saattavat esimerkiksi agenttien tekemät toiminnot jäädä suoriutumatta.

Jokaisella pelin agentilla voi olla vain yksi ja sama akatemia. Akatemia on vastuussa oppimisympäristön alustamisesta, nollaamisesta ja ympäristön muutoksista jokaisen simulaation jälkeen. [23.]

4.5 Tensorflow-tekoälymoottori ja PPO-vahvistusoppimisalgoritmi

Unityn ML-agentit tukevat PPO-vahvistusoppimisalgoritmia, joka käyttää neuroverkkoja agentin tekemien havaintojen kartoittamiseen ja parhaan vaihtoehdon valinnan tekemiseen. PPO-algoritmi on implementoitu TensorFlow-tekoälymoottorilla, joka toimii erillisessä Python-prosessissa. PPO on gradienttiin perustuva algoritmi, jota pidetään vastaaviin algoritmeihin nähden yksinkertaisena. Aikaisemmat kokeet ovat osoittaneet, että PPO soveltuu erityisesti liikkumiskykyä vaativiin Atari-peleihin. [24.]

Vahvistusoppimismallin menestykäs koulutus edellyttää harjoituksen hyperparametrien asettamista pelille sopivalle tasolle, sillä oletusparametrit eivät aina anna toivottua suorituskykyä. PPO-hyperparametreista on olemassa parhaita käytäntöjä, joita voidaan soveltaa parhaan suorituskyvyn saavuttamiseen. Perushyperparametreja ovat

- Gamma, jolla määritellään, kuinka pitkälle tulevaisuuteen agentin tulisi huolehtia mahdollisista palkkioista. Arvo on tyypillisesti väliltä 0,8–0,995. Tapauksissa, joissa palkkio on välitön, sen arvo voi olla pienempi. [25.]

- Lambda, jolla määritellään, kuinka paljon agentti luottaa sen nykyisen etuarvioinnin estimaattiin laskettaessa päivitettyä estimaattia. Alhainen lambda-arvo tarkoittaa käytännössä sitä, että agentti luottaa enemmän nykyiseen arvioon. Korkeampi arvo taas tarkoittaa, että agentti luottaa enemmän ympäristöstä saatuihin palkintoihin. Tyypillinen arvo on väliltä 0,9–0,95. [25.]

- Buffer_size eli puskurin koko, joka vastaa sitä, kuinka monta kokemusta agentin täytyy kerätä, ennen kuin oppimismallia päivitetään. Tyypillisesti suurempi buffer_size johtaa vakaampaan oppimisprosessiin. [25.]

- Batch_size eli erän koko, joka on niiden kokemusten määrä, joita käytetään gradienttimenetelmän iteraatiossa. Tyypillisesti sen pitää olla murto-osa buffer_size-arvosta. [25.]

- Num_epoch, joka on kokemuspuskurin läpi kulkevien läpäisyjen lukumäärä silloin, kun gradienttia lasketaan. Mitä suurempi batch_size, sitä suurempi tämän

arvo voi olla. Arvon laskeminen takaa vakaammat päivitykset hitaamman oppimisen kustannuksella. [25.]

- Learning_rate, joka vastaa jokaisen gradientin kaltevuuden vahvuutta. Tämän arvoa tulisi laskea, mikäli koulutus on epävakaa. [25.]

- Time_horizon, joka vastaa sitä, kuinka monta kokemusta yhden agentin täytyy kerätä, ennen kuin ne lisätään kokemuspuskuriin. Jos tämä raja saavutetaan ennen episodin päättymistä, käytetään arviota ennustamaan agentin nykyisestä tilasta odotettua kokonaistuottoa. Tämän arvon pitäisi olla riittävän suuri, jotta se voi sisältää kaiken tärkeän käyttäytymisen agentin toimintojen sarjassa. Tyypillinen arvo on väliltä 32–2048. [25.]

- Max_steps, jonka arvolla tarkoitetaan, kuinka monta simulaation vaihetta suoritetaan. Tätä arvoa suurennetaan monimutkaisiin ongelmiin. [25.]

- Beta, jolla määritetään entropian vahvuus. Arvon suurentaminen varmistaa, että päätökset ovat enemmän satunnaisia. Tämä arvo tulisi olla säädetty niin, että entropia ei pienene liian hitaasti tai nopeasti. [25.]

- Normalize, jolla voidaan määritellä vektorien havainnointien normalisointi. Normalisointi voi olla hyödyllinen tapauksissa, joissa on monimutkaisia jatkuvan ohjauksen ongelmia, mutta sen käyttö voi olla haitallista, jos käytettään diskreettiä ohjausta. [25.]

- Num_layers, joka vastaa sitä, kuinka monta piilotettua kerrosta on havainnointitulojen jälkeen. Pienempi määrä kerroksia nopeuttaa oppimista, mutta kerroksia saattaa tarvita vaikeampien ongelmien ratkaisuun. [25.]

- Hidden_units, jolla tarkoitetaan sitä, kuinka monta yksikköä on neuroverkon jokaisessa kerroksessa. Yksinkertaisissa ongelmissa tämän arvon pitäisi olla pieni, mutta suuri sellaisissa tilanteissa, joissa toiminta on hyvin monimutkaista päätös- ja havainto-muuttujien välillä. Arvo on tyypillisesti väliltä 32–512. [25.]

Tensorflow on alun perin Googlen Brain-kehitystiimin koneoppimista varten kehittämä avoimen lähdekoodin ohjelmistokirjasto. TensorFlow'n avulla Unity pystyy luomaan neuroverkon nopeasti ja tallentamaan aivoista TensorFlow-mallin, joka voidaan myöhemmin ladata peliin erilliseksi tekoälyksi. Malli on .nn-tiedosto, joka sisältää yksien aivojen neuroverkon. Tämän insinöörityön ML-Agents-versiossa Unity käyttää mallin luomiseen kevyttä neuroverkkokirjastoa nimeltä Barracuda, joka tukee prosessoria tai näytönohjainta oppimisen prosessointiin. Barracudan kirjaston kehitys on vasta alkuvaiheessa. [25.]

Oppimisen kehitystä tarkastellaan TensorBoard-työkalulla, joka mahdollistaa TensorFlow-mallin visualisoinnin. TensorBoard-prosessi käynnistetään Python-komennolla, jolloin siihen pääsee käsiksi verkkoselaimella TCP/IP 6006-portista.

5 Testiympäristön toteutus

Insinöörityön toteutusvaiheessa vietiin testiympäristön komponentit käytännön muotoon käyttäen luvuissa 3 ja 4 esiteltyjä asioita. Aluksi piti suunnitella peli, joka on tarpeeksi monimutkainen, että pelaaminen vaatii jonkinlaista strategiaa. Koska tarkoituksena oli tutkia tekoälyn soveltuvuutta, ei peliä tarvinnut tässä vaiheessa suunnitella erityisen viihdyttäväksi. Prototyyppiä varten oli väliaikainen 2D-grafiikka jo valmiina, sillä pelin ulkoasua oli jo aikaisemmin suunniteltu sivuprojektina. Testauksessa oli tarkoituksena hyödyntää Unityn kehittämää avoimen lähdekoodin koneoppimisen liitännäistä. Unityn ML-agenttien PPO-algoritmia ei ole juurikaan käytetty vastaavanlaisiin peleihin, sillä sitä pidetään parhaiten soveltuvana jatkuvaa liikkumista vaativiin peleihin. On ainakin yksi samatyyppinen strategiapeliprototyyppi nimeltä Metal Warfare, joka sijoittui toiseksi Unityn järjestämässä ML-agentit-haasteessa. [26.]

Sen lisäksi, että peli oli suunniteltava lähtökohtaisesti kaksinpeliksi, oli kehityksen aikana pidettävä mielessä se, että tietokone näkee pelin eri tavalla kuin ihmisen näköaisti. ML-agenttien opetuksessa olisi mahdollista asettaa agentille kamera, jolla se pystyisi visualisoimaan pelin oppimisen aikana ihmismäiseen tapaan. Kuitenkin lähtökohtaisesti pyrittiin siihen, että kameraa ei tarvitsisi käyttää, sillä se vaatii huomattavasti enemmän tehoa tietokoneelta. Tämä tarkoittaa sitä, että agentti pystyisi havainnoimaan koko pelin tapahtumat vain numeeristen arvojen perusteella eli vektoreina. Oppimisen kannalta pelin piti olla myös täysin toimiva jo ennen itsenäisen oppimisen alkua. Hiemankin rikkinäinen koodi saattaa häiritä päätöksentekoa tai aiheuttaa pelin kaatumisen jossakin vaiheessa opetusta.

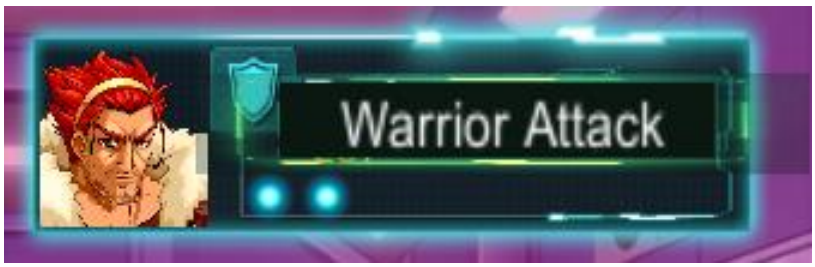
5.1 Ulkoasu ja käyttöliittymä

Pelin visuaalisuudella ei ollut tekoälyn oppimisen kannalta merkitystä, mutta se auttaa hahmottamaan pelin tapahtumia huomattavasti, esimerkiksi tekikö agentti järkeviltä tuntuvia päätöksiä tai pysähtyikö peli virheeseen. Oli myös suotavaa, että peli näyttäisi ja tuntuisi valmiilta mahdollisimman aikaisessa vaiheessa, jotta kehitystyö ei tunnu ylivoimaisen hitaalta. Agentit eivät käytä kameraa pelin visuaalisointiin, mutta niille silti kerrottiin vektoreina kaikki silmämääräisesti pelistä löytyvät havainnot. Tässä pelissä agenttien saamien havaintovektorien määrä oli 280.

Vuorossa oleva hahmo korostuu pohjavärillä vuoron alussa. Myös pelaajan tekemä valinta korostuu samanlaisella pohjavärillä. Kuvassa 8 näkyy vuorossa oleva hahmo, ja peli odottaa parhaillaan pelaajan toimintoa.

Kuva 8. Ensimmäinen täysin toimiva versio pelistä.

Kuvassa 9 voi havaita, että kohdetta valittaessa pohjaväri korostuu ja tässä tapauksessa ”Warrior Attack”-hyökkäysvaihtoehto olisi seuraavaksi valittavissa. Käytettävän kyvyn pystyy valitsemaan joko painamalla hiiren oikeaa nappia tai painamalla näppäimistöstä ennalta määrättyä nappia. Agentin päätökset ovat sidottuna näppäimistön painalluksiin, joten agentin kaikki toiminnot näkyvät kuten ihmispelaajalla.



Kuva 9. Valittavissa oleva kyky.

Kuvassa 10 esitellään yläreunan paneeli, josta näkyy hahmojen vuorojärjestys. Siitä pelaaja pystyy tarkastelemaan, minkä hahmon vuoro on seuraavaksi, mutta toistaiseksi ei ole kuitenkaan mahdollista tietää, kumman pelaajan vuoro on seuraavaksi. Myös pelaajien jäljellä olevien vuorojen määrä näkyy samasta paikasta. Keskelle asettuu se hahmo, jonka päätöstä odotetaan parhaillaan.

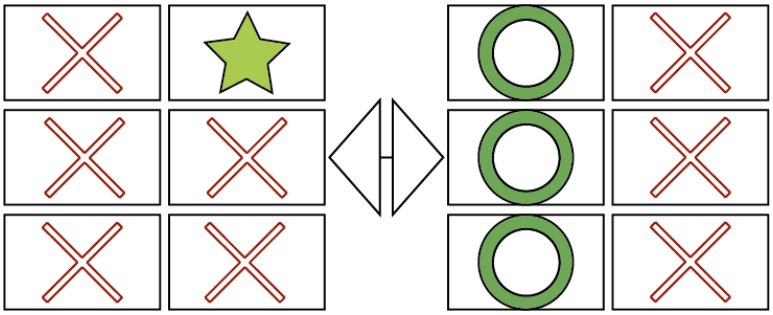
Kuva 10. Vuoropaneeli.

5.2 Pelimekaniikka

Tässä pelissä hahmot pysyvät pääasiassa paikallaan pelialueen sisällä, mutta hahmoilla on eri kykyjä, joista jokainen kyky on rajoitettu käytettäväksi vain tietylle osalle pelilautaa. Prototyypin erilaiset kyvyt ovat lähihyökkäys, kaukohyökkäys, selkään puukotus, itsensä parannus ja ryhmäparannus. Hahmot voivat liikkua vaihtamalla paikkaa samalla horisontaalisella rivillä olevan tiimiläisen kanssa. Jonkun pitää olla aina eturivissä, eli hahmot eivät voi liikkua eturivistä mihinkään, mikäli takarivissä olevat hahmot eivät ole elossa.

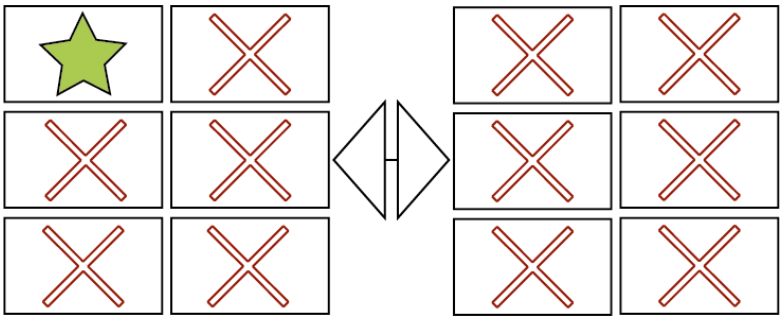
Kummassakin joukkueessa on yhteensä vähintään 2 ja enintään 12 hahmoa. Voittaja on se joukkue, joka on eliminoinut vastustajan ensin kokonaan tai jolla on kaikkien vuorojen loppuessa enemmän hahmoja hengissä.

Seuraavissa kuvaajissa tähti kuvastaa vuorossa olevaa hahmoa ja ympyrät vuorossa olevan hahmon mahdollisia valintoja. Vuorossa oleva hahmo pystyy myös aina valitsemaan itsensä. Kuvassa 11 vuorossa oleva hahmo on eturivissä, ja se pystyy hyökkäämään lähitaistelukyvyllä vastustajan eturivissä oleviin hahmoihin, mutta ei takarivissä oleviin.



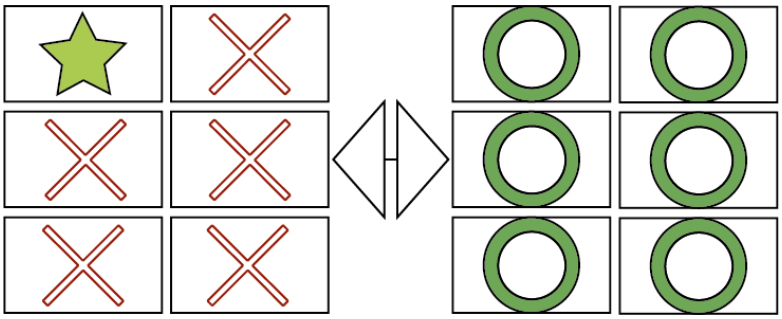
Kuva 11. Kuvaaja lähihyökkääjän valinnoista eturivissä.

Mikäli hahmo onkin takarivissä, kuten kuvassa 12, se ei pysty käyttämään lähitaisteluky-kyä yhteenkään hahmoon.



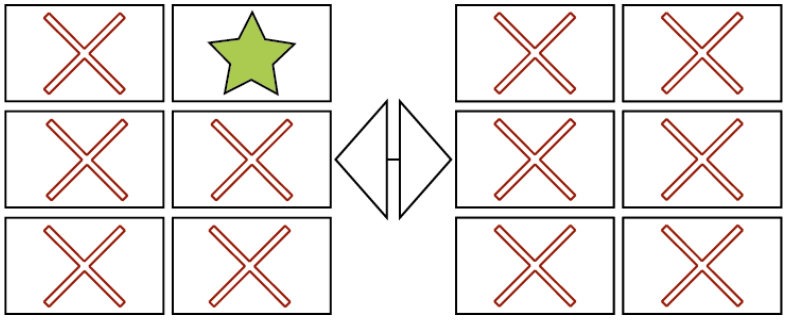
Kuva 12. Kuvaaja lähihyökkääjän valinnoista takarivissä.

Jos vuorossa oleva hahmo on turvallisesti takarivissä, kuten kuvassa 13, se pystyykin käyttämään kaukohyökkäystä mihin tahansa vastustajaan.



Kuva 13. Kuvaaja kaukohyökkääjän valinnoista takarivissä.

Kuten kuvassa 14 voi hahmottaa, vastaavaa kuvan 13 kaukohyökkäystä ei kuitenkaan voi käyttää eturivissä ollenkaan.



Kuva 14. Kuvaaja kaukohyökkääjän valinnoista eturivissä

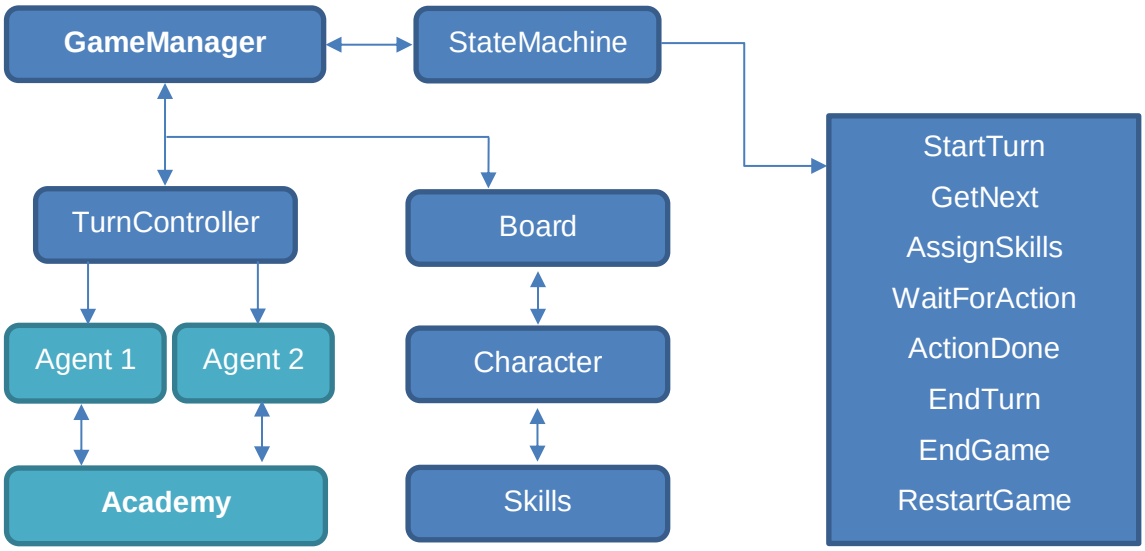
Samalla idealla selkään puukotus on mahdollista vain vastustajan taaimmaisiin elossa oleviin vastustajiin, ja parannuskyvyt toimivat vain omaan joukkueeseen. Jos kaikki eturivissä olevista hahmoista ovat kuolleita, takarivissä olevat hahmot siirtyvät automaattisesti eturiviin.

Pelaajien aloitusvuorojärjestys määräytyy hahmojen nopeusattribuutin mukaan: mitä nopeampi hahmo, sitä aikaisempi vuoro. Jokaisella kyvyllä on myös minimiaika, jonka joutuu odottamaan käytön jälkeen. Ajasta vähennetään hahmon nopeus, jolloin nopeammat hahmot ohittavat jonossa hitaat. Hahmoilla on myös hyökkäys ja puolustusattribuutit, jotka määrittävät tehdyn vahingon määrän. Vahingon määrä on myös suhteessa hyökkäävän hahmon osumapisteiden määrään. Mitä paremmassa kunnossa hyökkäävä hahmo on verrattuna puolustavaan hahmoon, sitä enemmän puolustava hahmo menettää osumapisteitä. Lähitaistelussa kumpikin menettää osumapisteitä, joten heikossa kunnossa olevalla hahmolla ei välttämättä kannata hyökätä paremmassa kunnossa olevaan.

Yksi erikoiskyvyistä on kilpi, jolla pystyy suojelemaan samassa rivissä olevia hahmoja, mutta tällöin suojeleva hahmo saa osumat. Toisella erikoiskyvyllä pystyy murhaamaan minkä tahansa vastustajan hahmoista, mutta kyvyn onnistumismahdollisuus riippuu vastustajan osumapisteiden suhteesta hyökkäävään hahmoon. Lisäksi murhaa yrittävä hahmo menettää kaikki loput toimintapisteet, oli hyökkäys onnistunut tai ei.

5.3 Ohjelmistorakenne

Pelin ydinsilmukkaa pitivät yllä GameManager- ja StateMachine-luokat. Nimiensä mukaisesti StateMachine on vastuussa pelin tilojen vaihtumisesta, kun taas GameManager on vastuussa koko pelisilmukan ylläpidosta. Kuvassa 15 näkyy, että TurnController saa GameManagerin kautta tiedon vuorojen vaihtumisesta, jolloin se suorittaa kaikki vuoron vaihtumiseen liittyvät toiminnot.



Kuva 15. Prototyypin komponentit.

Samaan tapaan GameManager antaa luvan toimintojen suorittamiseen pelilaudalla vuorossa olevalle agentille. TurnController poistaa käytöstä kummankin agentin, kun vuoron lopettava toiminto on suoritettu. Seuraavan vuoron alussa TurnController ottaa sen agentin käyttöön, jonka vuoro on. Board, eli pelilautaluokka välittää GameManagerille pelilaudan tapahtumat ja sisältää pelilaudan säännöt.

5.4 Agenttien määritykset

Pelissä on kaksi agenttia, joista toinen on käytännössä ykköspelaaja ja toinen kakkospelaaja. Kummallakin agentilla on myös omat aivot. Vaihtoehtoisesti voisi tehdä jokaisesta hahmosta oman agentin, jotka olisivat liitettynä kaksiin eri aivoihin. Tästä olisi

kuitenkin todennäköisesti enemmän haittaa kuin hyötyä, sillä oppimisen kulkua olisi sitä hankalampi seurata, mitä useampi agentti on käytössä.

WaitForAction on yksi tilakoneen tiloista, ja siinä seuraava hahmo on jo valittuna ja agentin pitäisi tehdä jokin päätös. Agentti pystyy tekemään päätöksen vain silloin, kun ohjelma sitä pyytää. Pyyntökäsky lähetään vuorossa olevalle agentille joka 0,5 sekunnin välein, niin kauan, kunnes agentti onnistuu tekemään hyväksyttävän toiminnon. Tilakoneen siirtyessä seuraavaksi ActionDone-tilaan, estetään agentin toiminnot ja odotetaan vuoron päättymiseen asti.

WaitForAction-tilassa päätöstä vaaditaan vuorossa olevalta agentilta esimerkkikoodin 1 avulla. While-loopin jälkeen on pieni viive, että päätöstä ei vaadita turhaan uudestaan, ennen kuin agentti on saanut yrittää tehdä jotain. Kun agentti on saanut aikaiseksi järkevän päätöksen, while loop lopetetaan.

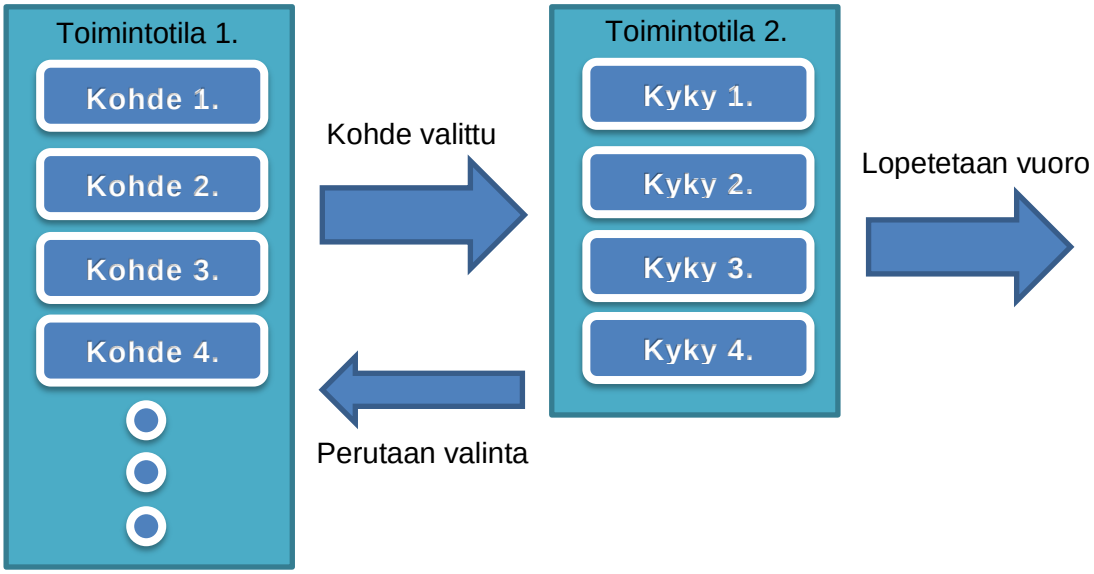
```
IEnumerator RequestDecisionDelay()


{
    while (State.WaitForAction == state)
    {
        if (currentPlayerTurn == 1)
        {
            agent1.RequestDecision();
        }
        else if (currentPlayerTurn == 2)
        {
            agent2.RequestDecision();
        }

        yield return new WaitForSeconds(0.5f);
        yield return new WaitForEndOfFrame();
    }

}
```

Esimerkkikoodi 1. Funktio, joka pyytää agenteilta päätöstä niiden omalla vuorolla.

Jotta agentti oppisi aikaisemmin kuvatut pelimekaniikan säännöt, on sen vähintään tiedettävä, mitä kykyjä hahmolla on ja mitkä osat pelilaudasta ovat milloinkin valittavissa. Aktiivisella hahmolla on korkeintaan neljä eri kykyä käytettävissä, ja valittavissa olevat kyvyt syötetään agentille havainnointivaiheessa ominaisuusvektorina, joka koostuu joukoista liukulukuja. Samalla tavalla agentille syötetään xy-koordinaatisto vektoreina, joissa on jokaisen hahmon sijainti pelilaudalla, sekä vuorossa olevan hahmon sijainti. Arvot syötetään aina samassa järjestyksessä, jolloin aivoilla on mahdollisuus vähitellen oppia syötettyjen lukujen vaikutus ympäristössä.

Agenttien aivoille täytyy asettaa vektorien pinoarvo korkeammaksi kuin 1, jotta ne muistavat osumapisteet menneisyydestä ja näin ollen pystyvät havainnoimaan, mitkä toiminnot aiheuttavat vahinkoa. Yhdessä vektorissa agentti saa 280 havaintovektoria, ja kun vektorien pinoarvoa nostetaan, havaintojen määrä moninkertaistuu. Tässä testiympäristössä käytetään 8:aa pinoa eli 280 x 8 = 2240 vektoria, jotta agentit muistaisivat pidemmälle ja voidaan odottaa mielenkiintoisempia tuloksia. Luultavasti kuitenkin jo 2 tai 3 pinoa riittäisi hyvin samantyyppiseen tulokseen. Agenteille syötetään siis aikamoinen määrä havaintoja, joista osa on sellaisia, joita ihmispelaajakaan ei näe, esimerkiksi vastustajan hyökkäys-, puolustus-, nopeus- ja älykkyyspisteet. Myös jokaiseen hahmoon kohdistuva kyky syötetään numeerisena arvona, jolloin agenttien pitäisi oppia, mihin hahmoon pystyvät mitäkin kykyä käyttämään.

Kuvasta 16 nähdään, että agentilla on kaksi diskreettiä toimintotilaa, joista ensimmäisessä on enintään kaksitoista haaraa ja seuraavassa neljä. Haarat kuvastavat agentin mahdollisia toimintoja kussakin tilassa. Ensimmäisessä toimintotilassa on valittavissa kokonaisluku väliltä 1–12. Päätös vaikuttaa siihen, mikä osa pelialueesta tulee valituksi. Valinnan tehtyään agentti vastaanottaa havaintoja, jotka ovat valittuun kohteeseen käytettävissä olevia kykyjä. Toisessa vaiheessa agentilla on valittavissaan kokonaisluku väliltä 1–4. Päätös suorittaa valitun kyvyn toiminnot ja lopettaa pelaajan vuoron. Vaihtoehtoisesti agentti voi palata ensimmäiseen toimintotilaan eli vaihtaa valittua kohdetta.



Kuva 16. Diskreettisten toimintotilojen kuvaaja.

Unity ML-Agents mahdollistaa joko diskreetin tai jatkuvan toimintotilan, mutta koska tässä pelissä agentti suorittaa vain yhden toiminnon kerrallaan, on diskreetti toimintotila parempi. Jatkuva toimintotila sopii peleihin, joissa toiminnot ovat jatkuvia ja saatetaan painaa montaa nappia samaan aikaan, kuten peleissä, joissa liikutaan jatkuvasti eri suuntiin.

Seuraavaksi tarkastellaan visuaalisesti, mitä toimintotilat tarkoittavat konkreettisesti. Kuvassa 17 näkyy, että korostettuna olevan hahmon vuoro on alkanut, mutta se ei ole vielä tehnyt mitään. Tässä vaiheessa vuorossa olevaa agenttia pyydetään valitsemaan kohde, jolloin toimintotila 2 aktivoituu. Mikä valittu kohde ei ole valittavissa, vuorossa olevaa agenttia rangaistaan pienellä negatiivisella palkinnolla. Näin tapahtuu esimerkiksi silloin, kun hahmolla on vain hyökkäyskykyjä, mutta agentti yrittää silti valita kohteeksi oman tiimin hahmon.

Kuva 17. Vuoro alkanut ja toimintotila 1 on käynnissä. Odottaa kohteen valintaa.

Kuvassa 18 agentti on valinnut takarivissä olevan vastustajan hahmon, jolloin peli on siirtynyt toimintotilaan 2, koska kohteeseen on kaksi kykyä valittavissa. Valintoja on joko ”Backstab” tai ”Assassinate”.

Kuva 18. Toimintotila 2 käynnissä. Odottaa kyvyn valintaa.

Kuvassa 19 agentti valitsi hyökkäyskyvyn ”Assassinate”, jolla oli tässä tapauksessa 40 prosentin mahdollisuus onnistua. Valinnan jälkeen vuoro loppui, ja valittu hyökkäys osoittautui onnistuneeksi.

Kuva 19. Lopetetaan vuoro.

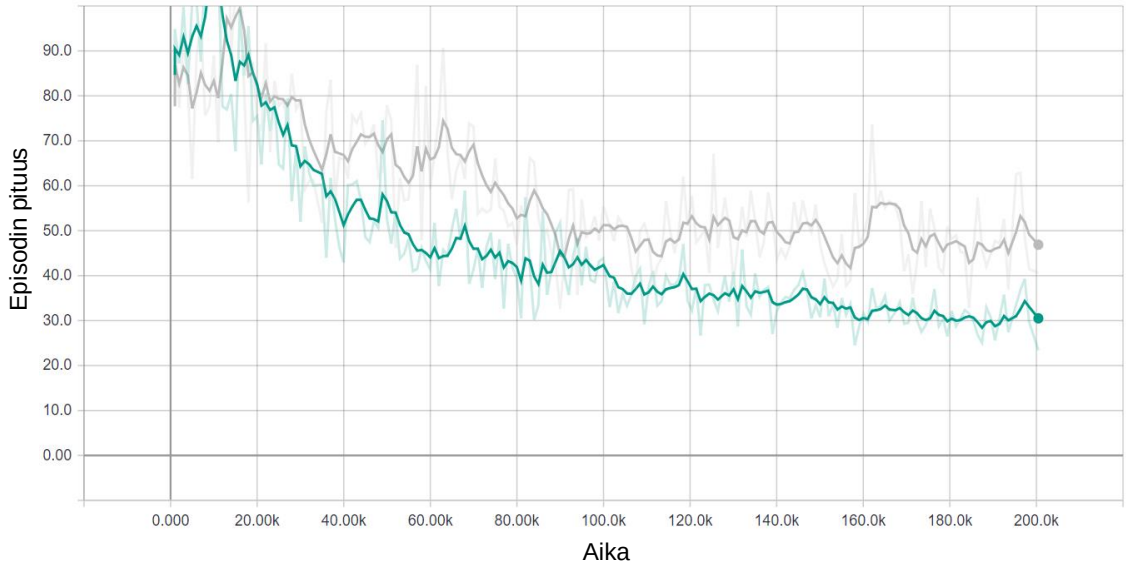
Agentin AddReward()-palkitsemisfunktiota kutsutaan aina silloin, kun se on onnistuneesti valinnut kyvyn, tehnyt vahinkoa vastustajalle, tuhonnut vastustajan tai voittanut pelin. Ainoastaan pelin häviämisestä ja oman hahmon kuolemisesta agentti saa rangaistuksen. Palkkioiden määrässä joutuu olemaan tarkkana, ettei agenttia rangaista negatiivisesti liikaa eikä positiivinen palaute ole liian suuri. Huonot palkkioiden määritykset saattavat johtaa heikkoon oppimiseen, jossa agentti välttelee esimerkiksi negatiivisia toimintoja, vaikka lopputulos olisi ollut positiivinen. Palkkioiden määritys tavallisesti onnistuu parhaiten vain kokeilemalla.

6 Koneoppimisen testaus

Ensimmäisessä testissä asetettiin ykköspelaajalle ja kakkospelaajalle erisuuruiset palkkiot, jotta pystyttiin arvioimaan tulosten perusteella sopivat raja-arvot seuraavaan testiin. Ykköspelaajaa palkittiin mm. pelin voittamisesta vain +0,1:n arvolla, kun taas kakkospelaaja sai samasta suorituksesta +1,1. Myös pelin häviämisestä ykköspelaajaa rangaistiin -0,1:n arvolla ja kakkospelaajaa -1,0:lla. Oletuksena oli, ettei kummankaan agentin opetus tulisi kunnolla onnistumaan, vaan tässä vaiheessa kokeillaan hyvien käytäntöjen rajoja. Hyviin käytänteisiin kuuluu, ettei agentteja yleensä palkita suuremmalla arvolla kuin +1,0, jotta oppiminen olisi mahdollisimman vakaa. Positiivista palkitsemista pitäisi suosia enemmän kuin negatiivista, ja negatiivista palautetta tulisi antaa vain, jos haluaa agentin suoriutuvan tehtävästä nopeammin.

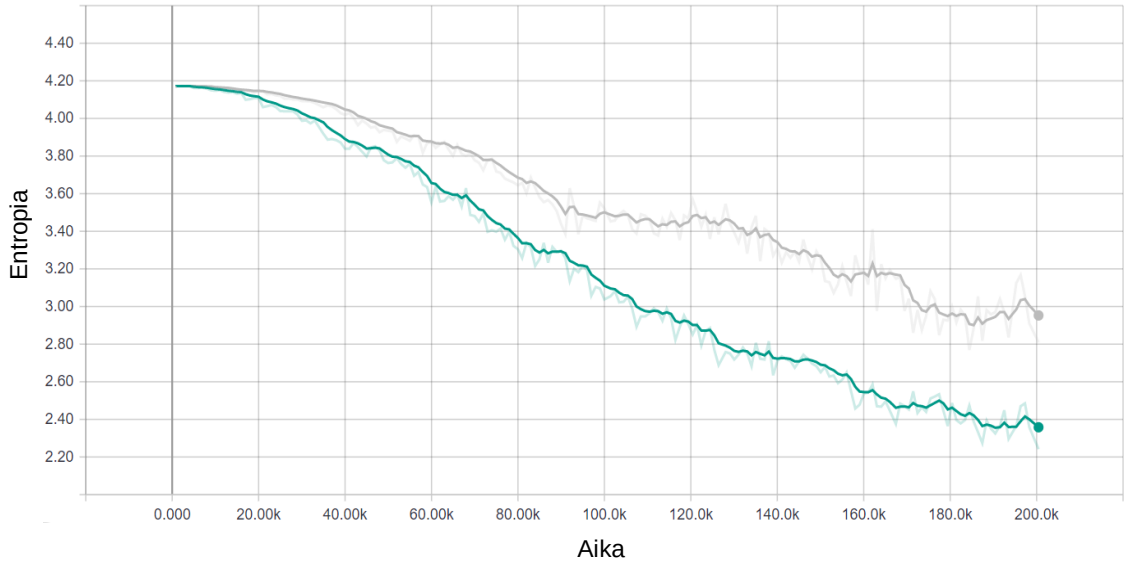
Useissa peleissä agentti oppii pelaamaan peliä tavallisesti 200 000 episodin paikkeilla, joten opetus päätettiin pysäyttää siinä vaiheessa ja tarkastella tuloksia. Episodi on ohjelmoitu päättymään viimeistään silloin, kun erä loppuu ja peli alkaa alusta. Tiedetään siis, että 1 000 episodin pituus korreloi sen kanssa, miten nopeasti agentti on oppinut tekemään päätöksiä. Episodin pituus on se aika, jonka kukin agenteista on käyttänyt päätöksen etsimiseen, sillä tässä pelissä agentin episodi pysäytetään manuaalisesti kutsumalla agentin Done()-funktiota, kun agentti on lopettanut toimintansa.

Kuvan 20 viivadiagrammia tutkiessa nähdään, että episodin pituuden arvo on ollut aluksi 80, mutta 200 000 episodin kohdalla toinen agenteista on päässyt 30:een, kun taas toinen jumittunut 45:n paikkeilla. Kakkospelaaja on diagrammissa harmaa viiva. Lähes koko opetusjakson ajan toinen agenteista on siis ollut kolmanneksen hitaampi päätöksenteossa. Episodien pituus on pysynyt samalla tasolla jo pitkän aikaan, mikä viittaa ongelmaan kummankin agentin oppimisessa. Se todennäköisesti tarkoittaakin, että agentit ovat lopettaneet harkittujen päätösten tekemisen.



Kuva 20. Episodin pituus oppimisen aikana.

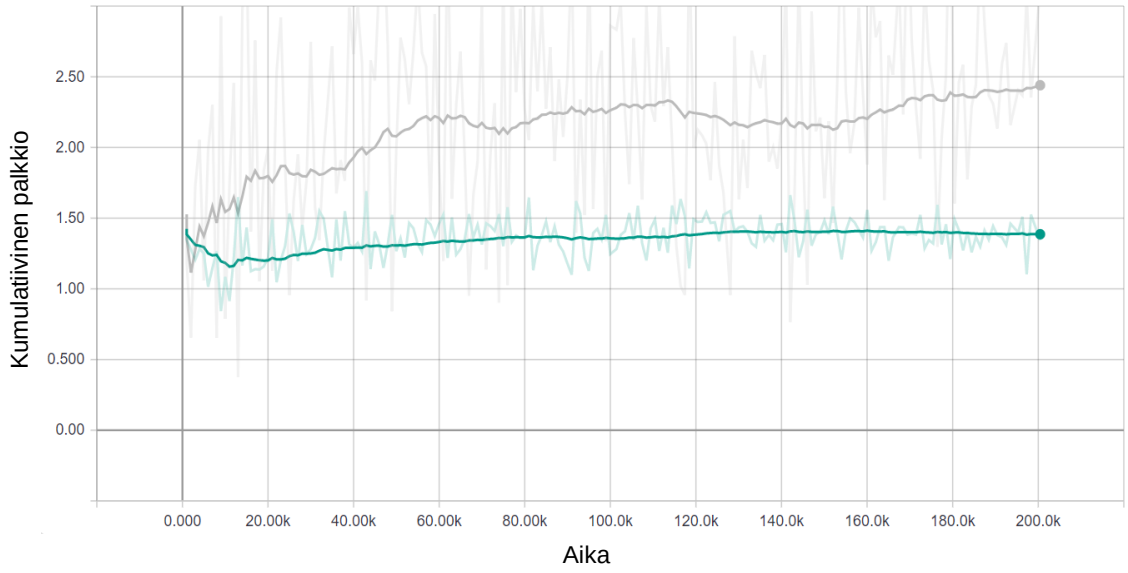
Seuraavaksi tutkitaan entropiaa, joka esitetään kuvassa 21. Entropia käytännössä ilmaisee päätösten satunnaisuutta. Entropian viivadiagrammi osoittaakin, että agenttien ero satunnaisissa päätöksissä näyttäisi kasvavan, mitä pidemmälle oppiminen etenee. Tässä vaiheessa näyttää siltä, että kakkospelaajan agentilla on huomattavia vaikeuksia tehdä päätöksiä.



Kuva 21. Päätösten entropian pitäisi laskea oppimisen aikana.

Entropia ei saisi laskea liian nopeasti eikä liian hitaasti. Kuvasta näkyy, että entropia näyttäisi laskevan sopivaa vauhtia, mikä osoittaa, että PPO:n hyperparametrien beeta-arvon pitäisi ainakin olla sopivalla tasolla.

Viimeisessä viivadiagrammissa, joka nähdään kuvassa 22, tarkastellaan agenttien kumulatiivista palkkiota. Kakkospelaaja on ansainnut enemmän palkkiota, niin kuin pitääkin, mutta huomioitava asia tässä on se, että ykköspelaajan palkkio ei ole kasvanut kumulatiivisesti, ja se on jopa lähtenyt laskuun. Laskuun on tosin voinut vaikuttaa se, että kakkospelaaja on voittanut enemmän pelejä jakson lopussa. Ykköspelaajalle se kuitenkin tarkoittaa sitä, että oppiminen häiriintyy.



Kuva 22. Kumulatiivinen palkkion pitäisi nousta oppimisen aikana.

Seuraavaksi kokeiltiin pelata diagrammeissa harmaata agenttia vastaan. Testissä oli havaittavissa, että agentti teki silloin tällöin omituisia päätöksiä, esimerkiksi lopettamalla vuoron tekemättä mitään. Suurin osa päätöksistä oli kuitenkin järkeviä, mutta kokonaisuudessa pelityyli ei vaikuttanut älykkäältä. Diagrammien tarkastelu ja testaus antoi viitteitä siitä, että kakkospelaajan agentilla on vielä opittavaa. Oppimisprosessia jatkettiin vielä 500 000 episodiin asti, jotta voidaan tarkastella, jatkaako ykköspelaajan agentti heikkoa suoritustaan. Sama trendi jatkuikin siihen asti, mutta 500 000 jälkeen kakkospelaajan oppiminen näytti lähes pysähtyneen, mikä saattaa olla merkki ylisovittamisesta.

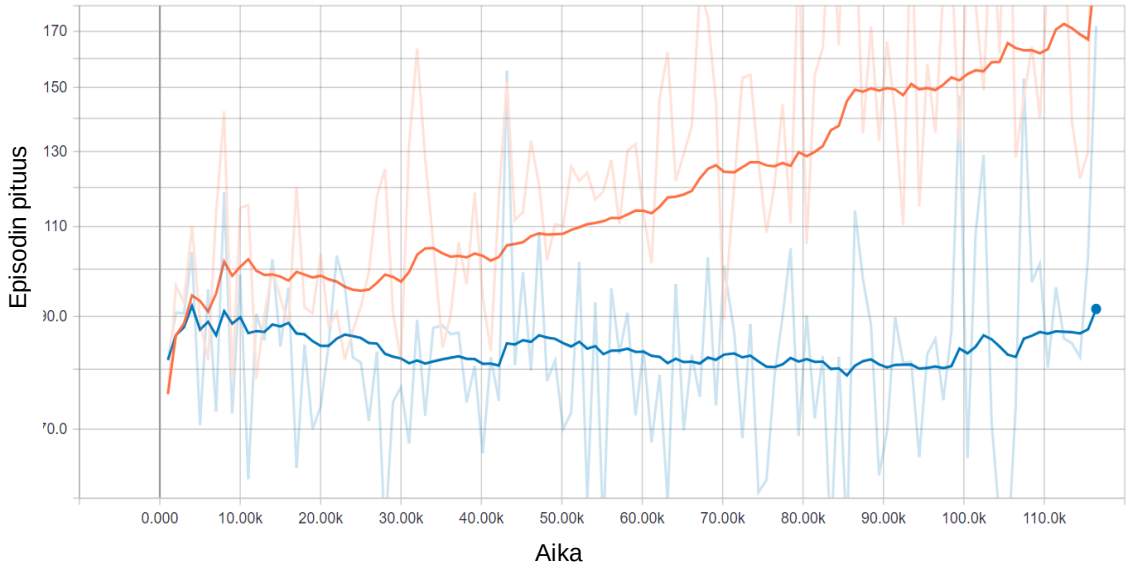
Opetus päätettiin pysäyttää ja kokeilla pelaamista kakkosagenttia vastaan. Havaittavissa ei ollut muuta eroa, kuin että päätöksenteko oli hieman nopeampaa. Tästä ensimmäisestä testistä opittiin, että pelin häviämisestä tai voittamisesta saatu palkkio voi olla suuri, mutta välissä annettavien kannustimien täytyy olla jotain kummankin agentin väliltä.

Seuraavassa vaiheessa opetus aloitettiin alusta, mutta palkkioita säädettiin maltillisemmiksi. Vuoron ohittamisesta agenteille annetiin nyt vain 0,001:n palkkio, jotta agentti ei lopeta vuoroa tekemättä mitään, ellei oikeasti pysty tekemään mitään muutakaan. Vahingon tuottamisesta saatu suurempi palkkio vaikutti myös tekevän agentista turhan aggressiivisen hyökkäämään.

Palkkioiden osalta arvioin seuraavat määritykset lopulta sopiviksi:

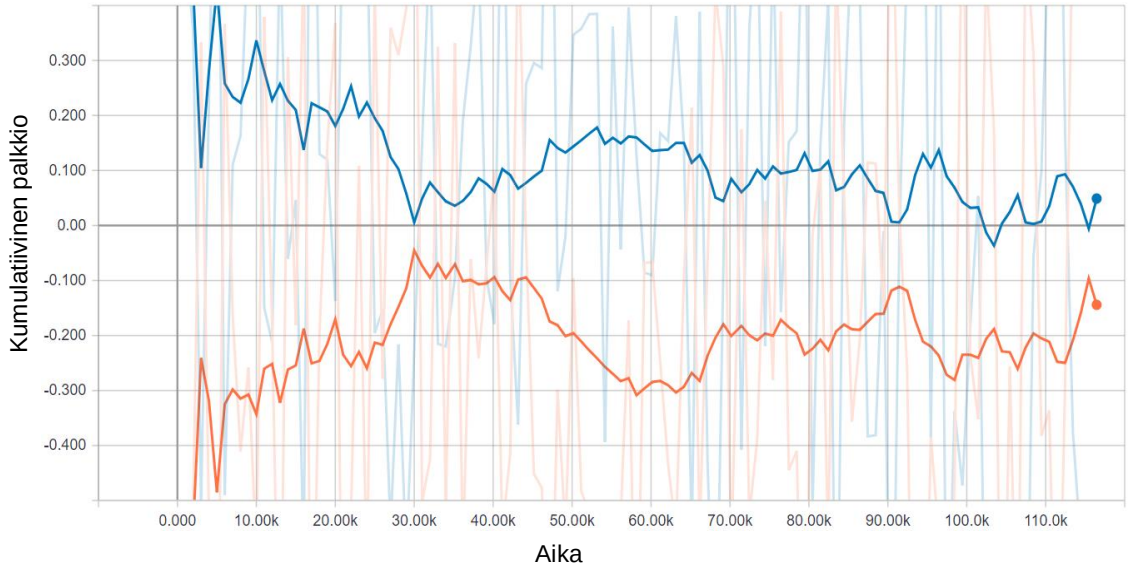
- pelin häviäminen: -1,0
- hahmon menetys: -0,05
- pelin voittaminen: +1,0
- hahmon tappo: +0,05
- vahingon tuottaminen: + 0,002
- kyvyn valinta: +0,01.

Testiä suoritettiin yli 100 000 kierrosta, ja jälkipuoliskolla oli silmämääräisestikin havaittavissa, että toisen agentin päätöksenteko alkoi jälleen kärsiä. Ongelman pystyi parhaiten näkemään, kun vertasi kuvan 23 episodin pituutta ja kuvan 24 kumulatiivista palkkiota.



Kuva 23. Episodin pituus toisessa testissä.

Alakynnessä oleva agentti saa liikaa negatiivista palautetta, jolloin agentti ns. luulee pelaavansa peliä väärin.



Kuva 24. Kumulatiivinen palkkio toisessa testissä.

Tilastojen mukaan toinen agenteista voitti pelin 1,4 kertaa useammin, mutta silti palkkio oli jäänyt lähelle nollaa. Tämä oli selvästi epäonnistunut yritys, mutta tästä pystyi arvioimaan, että palkitsemista pitää säätää ainakin agentin voittokertoimen verran positiivisemmaksi.

Kolmanteen testiin valmistauduttiin muuttamalla opetusta positiivisempaan suuntaan, jotta pelin häviämisestä tulisi hieman vähemmän negatiivista palautetta. Tavoitteena oli, että kumulatiivinen palkkio pysyisi kummallakin agentilla positiivisena.

Agenteille asetettiin seuraavat palkkioiden parametrit:

- pelin häviäminen: -0,5
- hahmon menetys: -0,09
- pelin voittaminen: +0,9
- hahmon tappo: +0,1
- vahingon tuottaminen: +0,025
- kyvyn valinta: +0,01.

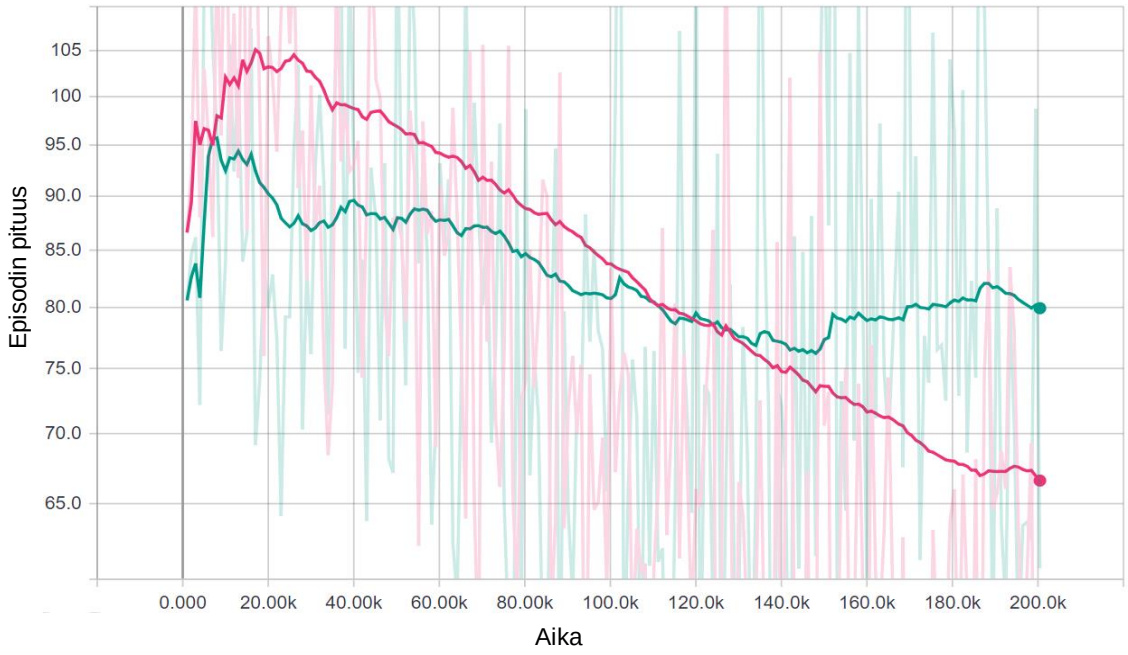
Nämä parametrit osoittautuivat tässä vaiheessa optimaalisiksi, sillä nyt agentit olivat tasavertaisia ja kumpikin kehittyi pelaamalla toisiaan vastaan. Oppiminen vaikutti olevan hitaampaa, mutta teoriassa oppimisen tason pitäisi olla laadukkaampaa, kun kumpikin agentti parantaa tulosta asteittain. Pieni ongelman alku oli havaittavissa kolmannessakin testissä, kun opetussetti oli toiminut 200 000 episodin verran. Ongelman alku johtui jälleen siitä, että toinen agenteista oli voittanut pelin useamman kerran peräkkäin, jolloin vastapuolen agentti oli alkanut epäröidä. Oppimisprosessi päätettiin lopettaa siihen ja kokeilla pelata agentteja vastaan. Agentit pelasivat nyt hyvin älykkäästi eivätkä yrittäneet vain hyökätä. Tekoäly osasi pitää tärkeät hahmot elossa ja suojella heikompiaan.

Oli hämmentävää huomata agenttien löytäneen ohjelmointivirheen, joka teki mahdolliseksi ampua kahta hahmoa samaan aikaan yhden sijasta. Tämä antoi myös hyvän idean tehdä siitä oikean ominaisuuden myöhempään versioon. Ohjelmistovirheen löytämistä edesauttoi se, että agentit kokeilevat tehdä kaikkea mahdollista löytääkseen parhaan tuloksen. Jos agentti vaihtoi kohdetta monta kertaa ennen hyökkäyskyvyn valintaa, saattoi valittuja hahmoja olla aktiivisena yhden sijasta kaksi samaan aikaan. Tämä

vaikutti hieman myös oppimisen laatuun, siten että agentit kokeilivat varmuuden hahmojen vaihtamista ennen lopullista valintaa. Pääasia on kuitenkin se, että agentit oppivat pelaamaan peliä, vaikka peli ei toiminut täysin niin kuin se oli suunniteltu.

Lopputulokset

Tässä pelissä episodin pituuden tarkastelu näytti antavan parhaan viitteen oppimisen kehityksestä. Parhaaseen lopputulokseen pääsi, kun keskimääräinen episodin pituus laski tasaista vauhtia, mutta ei kuitenkaan liian nopeasti. Opetus tuli lopettaa, kun suurempi poikkeama oli havaittavissa, kuten kuvassa 25 näkyy.



Kuva 25. Episodin pituus kolmannessa testissä.

Alhainen episodin pituus ei välttämättä tarkoita parasta tulosta. Esimerkiksi ensimmäisessä testissä episodin pituus oli huomattavasti alhaisempi, mutta pelitestissä oli havaittavissa, että agentit enemmänkin arvailivat, kuin tekivät harkittuja päätöksiä.

Testissä käytettiin seuraavia oletuksesta eroavia hyperparametreja, vaikka oletusparametritkin olisivat varmasti myös hyvin käyttökelpoiset. Kyseessä on vain pieni hienosäätö, jonka tarpeellisuudesta ei ole varmuutta.

- gamma: 0,91
- buffer_size: 2000
- max_steps: 2,0e5
- time_horizon: 200
- curiosity_strength: 0,005

Loppuanalyysi

ML-agentit toimivat pelissä hieman paremmin kuin oli odotettavissa, mutta luultavasti parantamisenkin varaa voisi olla. Varsinkin kahden agentin välinen opetus on hyvin hankalaa, koska toisen agentin epätavallinen käyttäytyminen vaikuttaa myös vastapuolen agenttiin. Kun agenteille ja aivoille lopulta löytyi pelille sopivat parametrit, oppimisprosessi kesti senkin jälkeen useita tunteja. Kun peliin lisätään uusia ominaisuuksia, saattaa oppiminen hankaloitua uudestaan. Vaikuttaisi kuitenkin siltä, että koneopetetun agentin tekoäly soveltuu mainiosti pelien geneerisiin tapahtumiin, jossa ympäristö pysyy suhteellisen samanlaisena. Pelien testauksessa tästä on myös todella paljon hyötyä, varsinkin jos halutaan testata suorituskykyä tai etsiä joitakin erikoisia ohjelmistovirheitä. Unityn ML-agenttien kehityskin on tosin vielä beetavaiheessa, joten kannattaa jäädä odottamaan tulevia versioita ennen lopullista päätöstä Unityn ML-agenttien käytöstä. Tässä vaiheessa voi kuitenkin olla tyytyväinen tuloksiin ja keskittyä pelin muihin osiin, kuten erilaisten kykyjen lisäykseen ja yksinpelikampanjan kehittämiseen. Kampanjanmoodissa todennäköisesti myös voidaan hyödyntää koneopittua tekoälyä taisteluiden simulointiin ja voittomahdollisuuksien laskemiseen.

Insinöörityö antoi näkemyksen siitä, että kynnys koneoppimisen implementointiin on nyt sellaisessa vaiheessa, että sitä kannattaa hyödyntää pelikehityksessä aina, kun mahdollista. Sillä ei kutenkaan kannata korvata kaikkia perinteisiä tekoälyjä, koska peli saattaa menettää ennalta-arvattavuuden, tekemällä pelistä vähemmän hauskan. Esimerkiksi parhaat perinteiset pomotaistelut edellyttävät, että pelaaja tunnistaa ja opettelee vihollisen käyttäytymismallin ulkoa. Tällaisissa tilanteissa tekoäly on se, joka tekee pelistä

mielenkiintoisen. Vaikka tähän voisi olla ratkaisuna perinteisen käyttäytymismallin ja koneoppimisen sekoittaminen, se ei välttämättä ole kaiken vaivan arvoista, sillä yksinkertaisen käyttäytymismallin ohjelmoinnin pitäisi olla huomattavasti helpompaa. Koneopetettu malli soveltuu peleissä luultavasti parhaiten rutiininomaisiin pelisilmukoihin, joissa satunnaisuus on hyvä asia. Parhaiten se voisi sopia esimerkiksi roguelike-tyylilajin peleihin, jotka sisältävät proseduraalisia elementtejä muutenkin. Erityisesti indie-kehittäjät hyötyisivät vastaavanlaisista koneoppimisen työkaluista, jottei monimutkaisen tekoälyn ohjelmointiin tarvitse käyttää aikaa eikä ihmisresursseja tarvitse käyttää ohjelmiston testaukseen.

7 Yhteenveto

Koneoppiminen on ollut vuosikymmeniä olemassa, mutta se ei ole aikaisemmin ollut yhtä helposti käytettävissä ilman syvällistä alan tietämystä. Erityisesti vahvistusoppimismenetelmää hyödyntäville neuroverkko- ja päätöspuu-malleille on olemassa useita valmiita avoimen lähdekoodin ohjelmistokirjastoja, työkaluja ja algoritmeja, joita pystytään implementoimaan osaksi tietokonepeliä.

Insinöörityön tarkoituksena oli selvittää koneopetuksen hyödyllisyyttä pelisovelluksiin ja luoda peliprototyyppi alustaksi, jolla toimivuutta pystyy tutkimaan. Projektin aihe mahdollisti todella laajan ulottuvuuden koneoppimisen eri osa-alueisiin, mutta insinöörityö pyrittiin pitämään maltillisena, jotta resurssien puitteissa saataisiin edes pieniä hyödyllisiä tuloksia aikaiseksi. Näin ollen tulokseksi saatiin pelisovellus, joka on laajennettavissa osaksi isompaa kokonaisuutta, ja saatiin osoitettua, että koneopetettu malli kykenee täyttämään kyseisen vuoropohjaisen pelin minimivaatimukset.

Koneopetuksesta syntyneessä mallissa voi olla vielä huomattavaa parantamisen varaa, mutta peliin täytyisi lisätä ominaisuuksia joka tapauksessa, joten mallin kehitykseen ei kannata tässä vaiheessa käyttää aikaa. Tällä hetkellä näyttäisi myös siltä, että ominaisuuksien lisäys saattaisi hankaloittaa opetusprosessia, mutta samaan aikaan Unity tekee jatkuvaa kehitystä ML-Agents-työkalupakkiin, jotta prosessi olisi mahdollisimman yksinkertainen ja nopea. Tämän insinöörityön aikana testauksessa ei kuitenkaan yritetty käyttää visuaalista havainnointia, joka havaintojen suuren määrän vuoksi saattaisi olla varteen otettava asia.

On jo havaittavissa, että tulevaisuutta ajatellen koneoppimiseen käytettävät työkalut tuovat lisäarvoa pelikehitystyölle merkittävästi. Toisarvoinen tavoite olikin osoittaa koneopetuksen hyödyllisyys indie-pelien kehitykseen. Insinöörityö antoi tekijälle näkemyksen siitä, että ainakin Unity-pelimoottorille on olemassa tarpeeksi hyvä työkalu, että sen soveltuvuutta pelin tekoälyksi tai pelitestauksen automatisointiin kannattaisi harkita. Työkalua testattiin vain yhden tyylilajin pelillä, joten syvälliseen johtopäätökseen ei ole realistista päästä.

Lähteet

1 A history of machine learning. Verkkoaineisto. Google. <https://cloud.with-google.com/build/data-analytics/explore-history-machine-learning/> Luettu 23.3.2019.

2 Oremus, Will. 2012. In Artificial Intelligence Breakthrough, Google Computers Teach Themselves To Spot Cats on YouTube. Verkkoaineisto. Slate. <https://slate.com/technology/2012/06/google-computers-learn-to-identify-cats-on-youtube-in-artificial-intelligence-study.html> Päivitetty 27.6.2012. Luettu 3.4.2019.

3 Clark, Liat. 2012. Google's Artificial Brain Learns to Find Cat Videos. Verkkoaineisto. Wired. <https://www.wired.com/2012/06/google-x-neural-network/> Päivitetty 26.6.2012. Luettu 3.4.2019.

4 Our team of five neural networks, OpenAI Five, has started to defeat amateur human teams at Dota 2. 2018. Verkkoaineisto. OpenAI. <https://openai.com/blog/openai-five/> Päivitetty 25.6.2018. Luettu 4.4.2019.

5 Agarwala, Aseem. 2018. Automatic Photography with Google Clips. Verkkoaineisto. Google. <https://ai.googleblog.com/2018/05/automatic-photography-with-google-clips.html> Luettu 20.4.2019.

6 Kashyap, Patanjali. 2017. Machine Learning for Decision Makers. Apress.

7 Russell, Stuart & Norvig, Peter. 2016. Artificial Intelligence: A Modern Approach. Third Edition. Pearson Education.

8 Bronshtein, Adi. A Quick Introduction to K-Nearest Neighbors Algorithm. Verkkoaineisto. Noteworthy. <https://blog.usejournal.com/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7> Luettu 13.4.2019.

9 Blostein, Dorothea. 2001. Graphics Recognition Algorithms and Applications. Springer Verlag Berlin And Heidelberg GmbH Co KG.

10 Chire. Verkkoaineisto. Wikimedia commons. <https://commons.wikimedia.org/wiki/File:KMeans-Gaussian-data.svg> Luettu 1.4.2019.

11 Maglogiannis, Ilias G. 2007. Artificial Intelligence Applications in Computer Engineering. IOS Press.

12 Anachronist. Verkkoaineisto. Wikimedia commons. <https://fi.m.wikipedia.org/wiki/Tiedosto:Normdist_regression.png> Luettu 1.4.2019.

13 Panesar, Arjun. 2019. Machine Learning and AI for Healthcare: Big Data for Improved Health Outcomes. Apress.

14 McCrea, Nick. An Introduction to Machine Learning Theory and Its Applications Toptal. Verkkoaineisto <https://www.toptal.com/machine-learning/machine-learning-theory-an-introductory-primer> Luettu 20.3.2019.

15 Kim, Tai-Hoon. 2010. Use of Artificial Neural Network in Pattern Recognition Vol. 4, No.2. Verkkoaineisto. <https://pdfs.semanticscholar.org/0c2b/61e35b876462d73e2454f62769ecba905dab.pdf> Luettu 18.4.2019.

16 Nagyfi, Richárd. 2018. The differences between Artificial and Biological Neural Networks. Verkkoaineisto. <https://towardsdatascience.com/the-differences-between-artificial-and-biological-neural-networks-a8b46db828b7> Päivitetty 4.9.2018. Luettu 14.4.2019.

17 Kapur, Rohan. 2016. What is bias in artificial neural network. Verkkoaineisto. <https://www.quora.com/What-is-bias-in-artificial-neural-network> Päivitetty 2.4.2016. Luettu 19.4.2019.

18 Berger, Christoph. Perceptrons - the most basic form of a neural network. Verkkoaineisto. Applied Go. <https://appliedgo.net/perceptron/> Luettu 2.3.2019.

19 Editor. Verkkoaineisto. Unity-Technologies. <https://unity3d.com/unity/editor> Luettu 30.3.2019.

20 Multiplatform. Verkkoaineisto. Unity-Technologies. <http://unity3d.com/unity/multiplatform/> Luettu 30.3.2019.

21 Introducing Unity Machine Learning Agents. Verkkoaineisto. Unity-Technologies. <https://blogs.unity3d.com/pt/2017/09/19/introducing-unity-machine-learning-agents/> Luettu 4.4.2019.

22 Brains. Verkkoaineisto. Unity-Technologies. <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Brains.md> Luettu 27.4.2019.

23 Creating an Academy. Verkkoaineisto. Unity-Technologies. <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Academy.md> Luettu 27.4.2019.

24 Shulman, John. Proximal Policy Optimization Algorithms. Verkkoaineisto. Cornell University. <https://arxiv.org/abs/1707.06347> Luettu 23.3.2019.

25 Training with Proximal Policy Optimization. Verkkoaineisto. Unity-Technologies. <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md> Luettu 19.4.2019.

26 ML-Agents Challenge I. Verkkoaineisto. Unity-Technologies. <https://connect.unity.com/challenges/ml-agents-1> Luettu 30.3.2019.