Eero Särkiniemi

Itseoppivan tekoälyn kouluttaminen geneettisellä algoritmilla

Opinnäytetyö
Tieto- ja viestintätekniikka

2020

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Eero Särkiniemi	Insinööri (AMK)	Toukokuu 2020

Opinnäytetyön nimi	
Itseoppivan tekoälyn kouluttaminen geneettisellä algoritmilla.	31 sivua 0 liitesivua

Toimeksiantaja

Kaakkois-Suomen ammattikorkeakoulu, Gamelab

Ohjaaja

Lehtori Niina Mässeli

Tiivistelmä

Tämä opinnäytetyö käsittelee syväoppimista, joka on koneoppimisen osa-alue. Syväoppimista sovelletaan neuroverkkoihin. Neuroverkot ovat algoritmeja, jotka matkivat ihmisaivojen toimintaa. Neuroverkko muodostuu yksittäisistä neuroneista, jotka on asetettu tasoittain toistensa perään. Yksittäinen neuroni koostuu yhdestä tai useammasta sisääntulosta, vakiotermistä ja yhdestä ulostulosta. Sisääntuloilla ja vakiotermillä on painot, jotka toimivat näiden kertoimina. Neuronilla on myös aktivointifunktio, joka määrittää ulostulon arvon riippuen sisääntuloista.

Työhön toteutettiin yksinkertainen kaksiulotteinen peli. Neuroverkon päämäärä oli oppia pelaamaan kyseistä peliä. Neuroverkon koulutus tapahtui geneettisellä algoritmilla. Geneettinen algoritmi matkii Charles Darwinin evoluutioteorian perusteita.

Työssä perehdytään syväoppimisen, neuroverkkojen ja geneettisen algoritmin teoriaan ja toteutukseen sekä pelin toteutukseen. Työ on toteutettu Unity Engine -pelimoottoria käyttäen. Työn ohjelmointikielenä toimi C#-ohjelmointikieli.

Lopullisessa toteutuksessa tekoäly onnistui oppimaan pelin ja pelaamaan peliä onnistuneesti.

Asiasanat

tekoäly, peliohjelmointi, neuroverkot, koneoppiminen, syväoppiminen, geneettinen algoritmi

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Eero Särkiniemi	Bachelor of Engineering	May 2020

Thesis title

Training self-learning AI using genetic algorithm.

31 pages
0 pages of appendices

Commissioned by

South-Eastern Finland University of Applied Sciences, Gamelab

Supervisor

Niina Mässeli, Senior Lecturer

Abstract

This thesis covers deep learning, which is subclass of machine learning. Deep learning is used in neural nets. Neural nets are algorithms that mimic the way human brain operates. Neural nets consist of single neurons, that are set in layers one after another. Single neuron consists of one or more inputs, bias and one output. Inputs and bias have their own weights, that are used as multipliers. Neuron has also an activation function that sets neurons output depending from the inputs.

In this thesis a simple 2D game was created. Neural nets goal was to learn to play this simple game. In the training of the neural nets a genetic algorithm was used. Genetic algorithm mimics the premise of Charles Darwin's theory of evolution.

This thesis looks into theory and execution of deep learning, neural nets and genetic algorithm and into the execution of the simple game. Thesis was done using Unity Engine - game engine and programmed using C#-programming language.

In the final work AI successfully learned the game and was able to play it.

Keywords

artificial intelligence, game programming, neural nets, machine learning, deep learning, genetic algorithm

SISÄLLYS

1 JOHDANTO

Tässä opinnäytetyössä käsitellään syväoppimista, joka on yksi koneoppimisen osa-alueista. Syväoppimista sovelletaan neuroverkkoon, jota opetetaan pelaamaan yksinkertaista peliä geneettisen algoritmin avulla. Työn tarkoitus on esitellä syväoppimisen ja neuroverkkojen teoriaa, sekä pelin että neuroverkon suunnittelua ja toteutusta. Työ on toteutettu Kaakkois-Suomen ammattikorkeakoulun tieto- ja viestintätekniikan Gamelab-oppimisympäristölle.

Työssä toteutettiin yksinkertainen kaksiulotteinen peli Unity Engine -pelimoottorilla. Tämän jälkeen toteutettiin syväoppiva neuroverkko, sekä geneettinen algoritmi. Neuroverkkoa opetettiin pelaamaan toteutettua peliä. Opetus tapahtui geneettisen algoritmin avulla. Ohjelmointi tapahtui C#-ohjelmointikielellä.

Syväoppiminen ja neuroverkot eivät ole uusi asia. Vastaavanlaisia opinnäytetöitä on tehty aikaisemminkin. Esimerkiksi Teemu Ropilo käsittelee opinnäytetyössään (2019) ”Teaching a machine learning agent to survive in a 2D topdown environment” samankaltaisia asioita. Työssä on käytetty Unity Engine -pelimoottorin Machine Learning Agents toolkit -työkalua. Myös Riku Heino on käsitellyt samaa asiaa opinnäytetyössään (2019). Heino on soveltanut työssään matriisilaskentaa neuroverkkojen tasoihin.

2 TEKOÄLY

Tekoälyllä tarkoitetaan tietokoneohjelmia, jotka pystyvät suorittamaan niille annettuja tehtäviä älykkäästi. Tekoälyjä ei ole ohjelmoitu suorittamaan jotain tiettyä toimintoa, vaan ne voivat mukautua itsenäisesti, niille annettujen tehtävien perusteella. Konsepti tekoälystä on ollut olemassa jo satoja vuosia, mutta vasta viime aikoina näistä on tullut todellisuutta. (Mills 2018.)

Suurin osa nykyään käytössä olevista tekoälyistä on sovellettuja tekoälyjä (engl. *applied AI*). Nämä ovat johonkin tiettyyn tarkoitukseen kehitettyjä tekoälyjä. Esimerkkejä sovelletuista tekoälystä ovat itsenäisesti ajavat autot ja tekoälyllä ohjatut osakekauppajärjestelmät. (Mills 2018.)

Yleiset tekoälyt (engl. *generalized AI*) ovat harvinaisempia johtuen niiden luomisen vaikeudesta. Ideaalinen yleinen tekoäly olisi kykenevä kaikenlaisiin tehtäviin ihmisen tavoin. Ideaaliin yleiseen tekoälyyn pyrkiminen on johtanut koneoppisen kehittämiseen. (Mills 2018.)

2.1 Koneoppiminen

Koneoppiminen on yksi tekoälyn osa-alueista. Koneoppimisessa tekoäly oppii ja mukautuu itsenäisesti sille annetun datan perusteella. Koneoppimiseen johtanut läpimurto oli, kun huomattiin että on tehokkaampaa antaa tekoälyn oppia itsenäisesti kuin opettaa ja antaa sille tarvittava data. (Mills 2018.)

Koneoppiminen jaetaan yleensä kahteen alaluokkaan tämän oppimisen perusteella. Nämä luokat ovat ohjattu oppiminen ja ohjaamaton oppiminen. Se millainen koneoppimisen algoritmi valitaan, riippuu siitä mitä käsiteltävästä datasta halutaan ratkaista. (Rouse 2020.)

Yksi koneoppimisen tunnetuimpia käyttökohteita on sosiaalisen median suosittelujärjestelmät. Nämä pyrkivät näyttämään käyttäjälle kaikkein kiinnostavinta sisältöä. Muita sovelluksia koneoppimiselle on muun muassa virtuaaliset avustajat ja jo aikaisemmin mainitut, itsestään ajavat autot. (Rouse 2020.)

Koneoppimista hyödyntäviä viitekehyksiä on useita. Tällaisia ovat esimerkiksi Googlen kehittämä Tensorflow- ja Microsoftin Azure Machine Learning -viitekehykset (Tensorflow s.a.; Microsoft s.a.).

2.2 Syväoppiminen

Syväoppiminen on koneoppimisen osa-alue, eli voidaan sanoa, että syväoppiminen on koneoppimista. Syväoppimisessa hyödynnetään neuroverkkoja, joissa on useita tasoja. Nämä jäljittelevät karkeasti ihmisaivojen tapaa toimia ja oppia. (Hamilo 2013.)

Syväoppimisen sovellukset hyötyvät suuresta määrästä dataa, jota on viime vuosina tullut paremmin saataville. Saatavilla olevan datan määrä lisäksi sy-

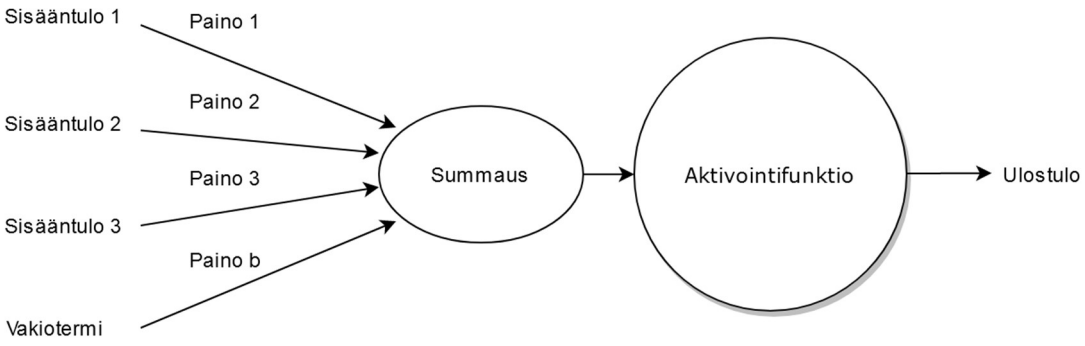
väoppimisen sovellukset ovat hyötyneet tietokoneiden kasvaneesta laskentatehosta. Syväoppimista hyödynnetäänkin tänä päivänä kaikkein kehittyneimmissä tekoälyn sovelluksissa (Rouse 2020). Syväoppimisella pystytään löytämään ratkaisuja monimutkaisiin ongelmiin. Käytännön sovelluksia, joista on tullut syväoppimisen ansiosta todellisuutta, ovat esimerkiksi kasvojentunnistus ja monipuolisemmat kielenkääntäjät. (Marr 2018.)

2.3 Neuroni

Neuroni on neuroverkon perusosa. Neuroverkot koostuvat yksittäisistä neuroneista. Yksinkertaisin neuroverkko koostuu yhdestä neuronista. Tällaisen neuroverkon kehitti Frank Rosenblatt vuonna 1957. Hän nimitti yhden neuronin verkkoa perceptroniksi. Neuroneja kutsutaankin joskus perceptroneiksi. (Shiffman 2012, 448.)

Neuroni koostuu yhdestä tai useammasta sisääntulosta, sekä yhdestä ulostulosta. Nämä ovat vain eri suuruisia lukuja. Jokaisella sisääntulolla on oma paino, jolla tämä kerrotaan. Sisääntulojen lisäksi neuronilla on vakiotermi (engl. *bias*). Tämä on kuin sisääntulo, mutta aina arvolla yksi (Brownlee 2016). Kuvassa 1 on kuvattu neuroni, jolla on kolme sisääntuloa.

Neuronin sisällä on prosessori tai aktivointifunktio. Tämä on matemaattinen funktio, joka määrittää ulostulon arvon, joka riippuu sisääntulojen ja näiden painojen tulojen summasta. (Shiffman 2012, 448.)



Kuva 1. Esimerkki neuronista, jolla on kolme sisääntuloa

2.3.1 Painot

Painot ovat lukuja, jotka toimivat sisääntulojen kertoimina. Ennen kuin sisääntulot menevät aktivointifunktioon ne kerrotaan omalla painollaan. Painojen avulla eri sisääntulojen vaikutusta ulostuloon voidaan kasvattaa tai pienentää. Yleensä painot alustetaan pieniksi sattumanvaraisiksi numeroiksi. (Brownlee 2016.)

Neuronin koulutus on pohjimmiltaan vain oikeiden painojen etsimistä. Koulutuksessa painojen arvoja muutetaan pikkuhiljaa ajan kuluessa, kunnes neuroni toimii halutulla tavalla. (Leon 2017.)

2.3.2 Aktivointifunktio

Aktivointifunktio on matemaattinen funktio neuronin sisällä. Se saa sisäänsä summan, joka muodostuu, kun sisääntulot on kerrottu omilla painoillaan ja summattu tämän jälkeen yhteen. Summaan lisätään vielä yksi kerrottuna vakiotermin painolla. (Brownlee 2016.)

$$f(i_1 \cdot w_1 + i_2 \cdot w_2 + \cdots i_n \cdot w_n + 1 \cdot w_b)$$

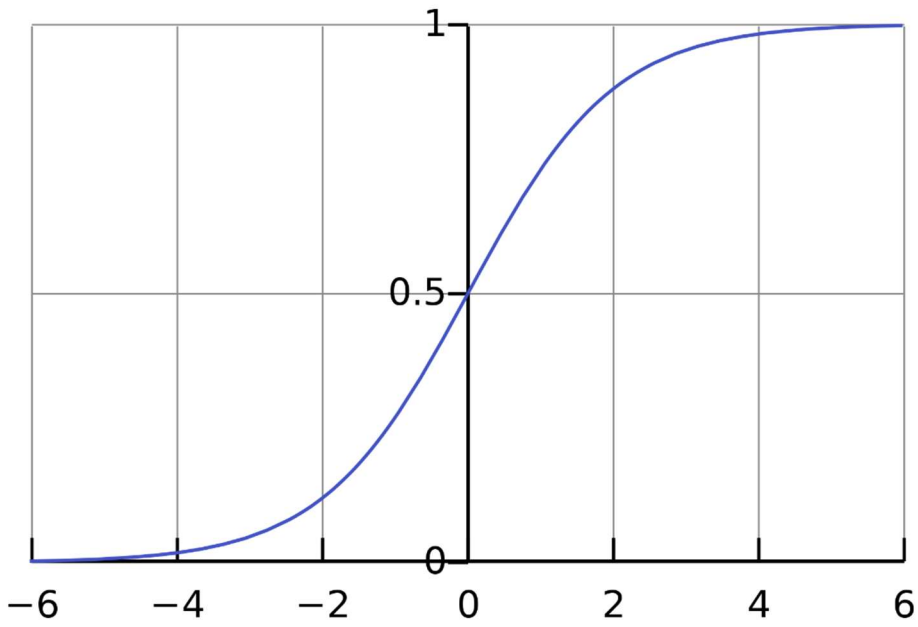
jossa	i_n	neuronin sisääntulo	[-]
	w_n	sisääntulon paino	[-]
	w_b	vakiotermin paino	[-]

Yksi yleisimmistä aktivointifunktioista neuroverkoissa on logistinen funktio (Brownlee 2016).

$$f(x) = \frac{1}{1 + e^{-x}}$$

jossa	e	Neperin luku	[-]

Tällä rajataan neuronin ulostulon arvo nollan ja yhden väliin. Funktio lähestyy yhtä positiivisilla arvoilla ja nollaa negatiivisilla arvoilla, näitä kuitenkaan saavuttamatta (kuva 2). Nollalla funktio saa arvon 0,5.



Kuva 2. Logistisen funktion kuvaaja välillä –6, +6

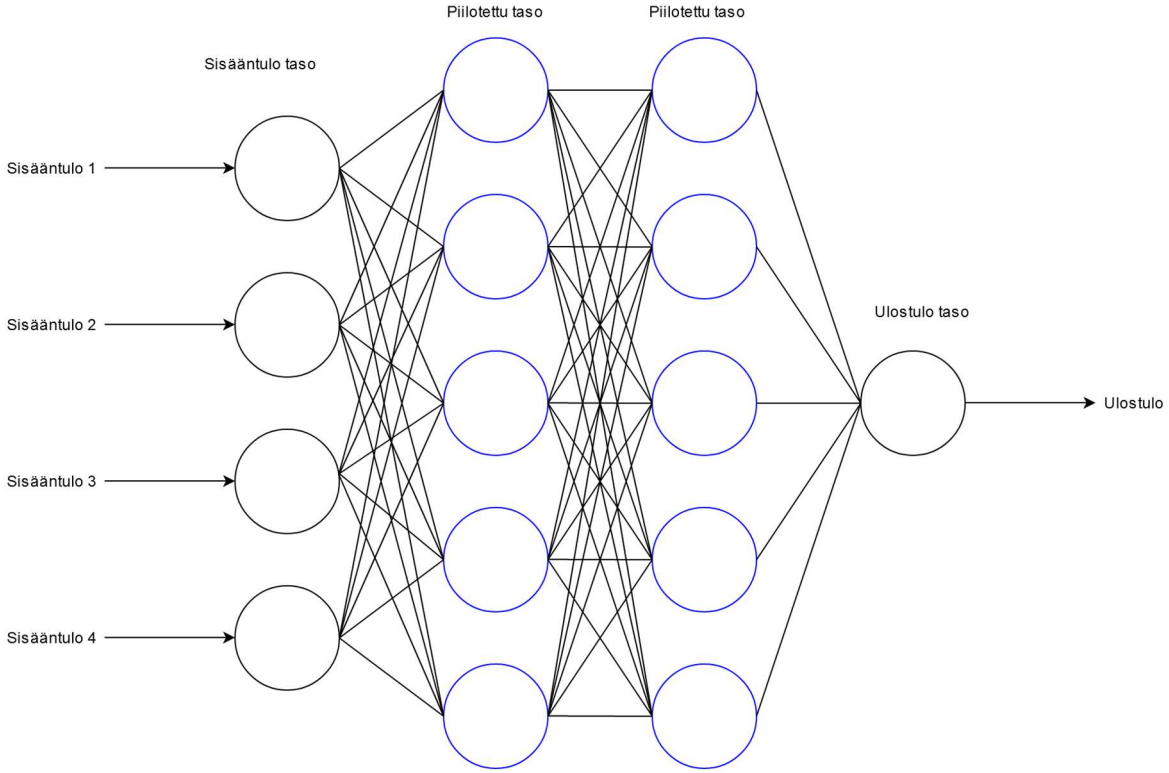
2.4 Neuroverkko

Neuroverkko matkii ihmisaivojen toimintatapaa. Se pystyy sopeutumaan muuttuvaan sisääntulodataan ja ulosantaa parhaan mahdollisen ulostulon ilman, että ulostulon kriteereitä tarvitsisi muokata. (Chen 2019.)

Bernard Widrow ja Marcian Hoff kehittivät vuonna 1959 neuroverkon, joka toimi mukautuvana suodattimena ja jolla poistettiin kaikuja puhelinlinjoista. Tämä oli ensimmäinen käytännön ongelmaan sovellettu neuroverkko ja se on käytössä vielä nykypäivänä. (Strachnyi 2019.)

Neuroverkko muodostuu, kun useita neuroneita asetetaan tasoittain toistensa perään. Ensimmäinen taso saa sisääntuloikseen jotain dataa, mistä halutaan ratkaista jotain. Ensimmäisen tason jälkeen jokainen neuroni saa sisääntuloikseen edellisen tason neuronien ulostulot. Yleensä ensimmäisen tason neuronit eivät suorita minkäänlaisia laskutoimituksia, vaan ne syöttävät sisään tulevan datan seuraavalle tasolle. Viimeisellä tasolla on verkon ulostulot, joita voi

olla yksi tai enemmän. Ensimmäistä tasoa nimitetään sisääntulotasoksi ja viimeistä tasoa ulostulotasoksi. Näiden välissä olevia tasoja nimitetään piilotetuiksi tasoiksi, koska nämä eivät ole suoraan esillä verkon ulkopuolelle. Kuvassa 3 on kuvattu neuroverkko, jossa tasoja on yhteensä neljä. Näistä kaksi on piilotettuja tasoja. (Brownlee 2016.)



Kuva 3. Esimerkki neuroverkosta

2.5 Geneettinen algoritmi

Geneettisillä algoritmeilla tarkoitetaan algoritmeja, jotka mallintavat Charles Darwinin evoluutioteoriaa. Algoritmien ei ole tarkoitus simuloida luontoa, vaan ottaa mallia siitä. Algoritmien tarkoitus onkin matkia luonnonvalintaa ja kelpoisimman eloonjäämistä (engl. *survival of the fittest*) tietokoneohjelmassa. (Shiffman 2012, 391.)

Geneettisessä algoritmissa täytyy olla populaatio, jotain mitä halutaan kehittää. Lisäksi tämä tarvitsee toimiakseen kolme periaatetta: perinnöllisyyden, variaation ja valinnan. Perinnöllisyydellä tarkoitetaan, että osa yksilön piirteistä siirtyy tämän jälkeläisille seuraavaan sukupolveen. Variaatiolla tarkoitetaan, että populaatiossa on oltava mahdollisimman paljon erilaisia piirteitä. Jos näin

ei ole, populaatio ei pysty kehittymään tarpeeksi. Valinnalla tarkoitetaan sitä, että algoritmilla on jokin keino määrittää, mitkä yksilöt lisääntyvät. Yleensä tämä toteutetaan kelpoisimman eloonjäämisellä. Tämä tarvitsee tavan määrittää yksilöille jonkinlaisen kelpoisuusarvon. (Shiffman 2012, luku 9.)

2.5.1 Kelpoisuusfunktio

Kelpoisuusfunktio on funktio, joka määrittää geneettisen algoritmin populaation yksilöille jonkinlaisen kelpoisuusarvon. Arvo voi esimerkiksi kasvaa sitä mukaa, kuinka kauan yksilö on hengissä. Mitä suurempi kelpoisuusarvo yksilöllä on, sitä suuremmalla todennäköisyydellä yksilö lisääntyy ja tämän perimä jää populaatioon. (Shiffman 2012, 397.)

Jokaisella ongelmalla täytyy olla omanlainen kelpoisuusfunktio. Vaikein osuus funktion muodostamisessa on keksiä sopiva funktio ratkaistavalle ongelmalle. Kelpoisuusfunktioiden muodostamiseen ei ole mitään määrättyjä sääntöjä, mutta tietyntyyppiset ratkaisut ovat yleisiä tietyntyppisissä ongelmissa. Esimerkiksi luokittelutehtävissä, jossa käytössä on ohjattu oppiminen, virhettä mitataan tyypillisesti euklidisella metriikalla tai taksigeometrialla ja tätä käytetään kelpoisuusfunktiona. (Mallawaarachchi 2017.)

2.5.2 Mutaatiot

Mutaatioilla geneettisessä algoritmissa tarkoitetaan uusia piirteitä, joita yksilöt voivat saada populaation ulkopuolelta. Nämä uudet piirteet tuodaan pienelle määrälle yksilöitä sattumanvaraisesti niiden luomisen yhteydessä. Mutaatioilla lisätään populaation piirteiden variaatioita. Tarkoitus on yrittää löytää uusia piirteitä, jotka hyödyttävät populaation kehitystä. Näin ei kuitenkaan aina ole, mutta haitalliset mutaatiot poistuvat kelpoisimman eloonjäämisessä. (Shiffman 2012, 401.)

Mutaatioiden määrä populaatiossa täytyy pitää pienenä, jottei alkuperäinen perimä mutatoidu kokonaan pois. Mutaatioiden määrää ei voida kuitenkaan pitää niin pienenä, ettei hyödyllisiä mutaatioita löydettäisi. (Shiffman 2012, 401.)

3 TYÖN SUUNNITTELU JA TAVOITTEET

Alkuun oli selvitettävä mahdolliset kirjastot ja työkalut, joita työssä voitaisiin käyttää. Yksi tälläinen oli Unity engine -pelimoottorin Machine learning agents toolkit -työkalu. Tämä kirjasto mahdollistaa erilaisten koneoppimiskirjastoiden, esimerkiksi Googlen Tensorflow-viitekehyksen, käyttämisen pelimoottorissa. (Juliani 2017.)

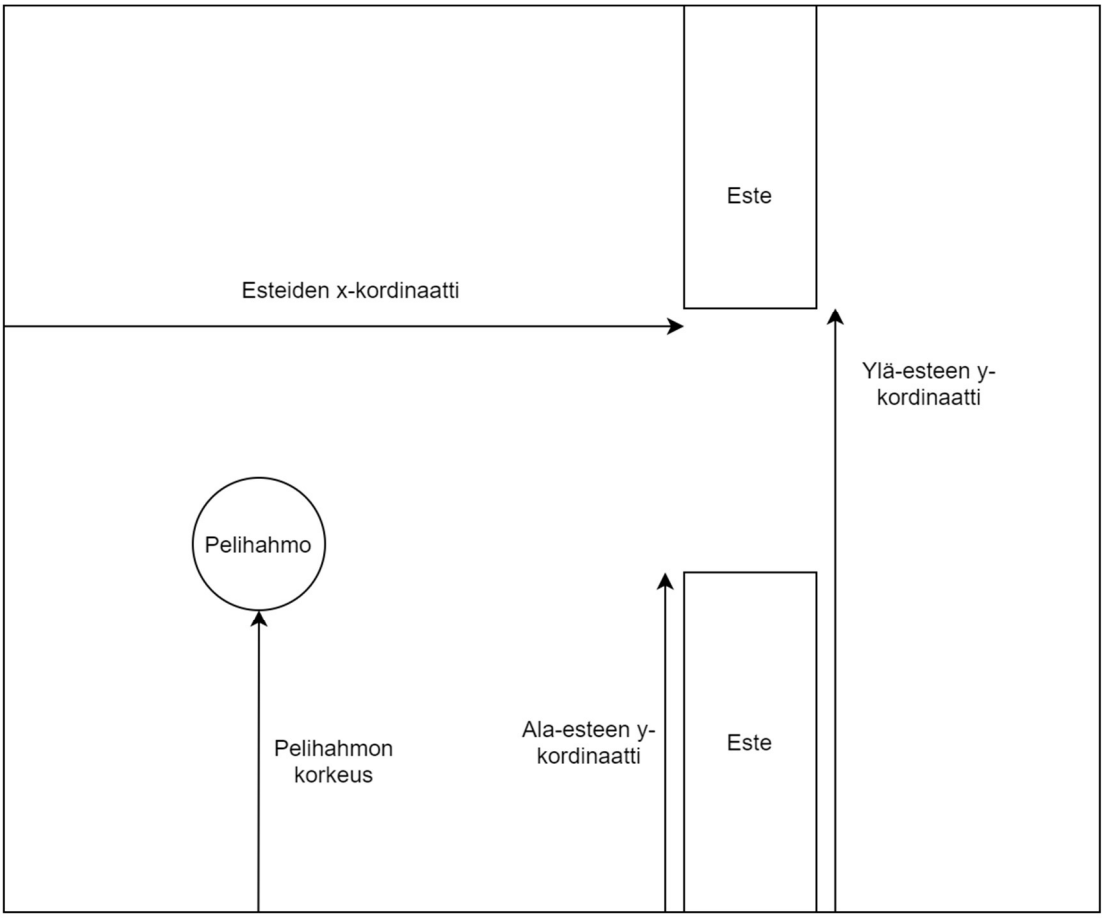
Koska työn tavoite oli ymmärtää neuroverkkojen toimintaperiaate, eikä luoda täydellisesti toimivaa ohjelmaa, päätettiin olla käyttämättä ulkoisia kirjastoja ja pyrittiin kirjoittamaan kaikki koodi itse. Neuroverkon koulutukseen valittiin geneettinen algoritmi, koska esimerkiksi vastavirta-algoritmi (engl. *backpropagation*) olisi vaatinut virheen laskemista. Vastavirta-algoritmi on yksi yleisimmistä tavoista kouluttaa monikerroksisia neuroverkkoja. (Nicholson 2019.)

Työn tavoitteiksi jäivät toimivan yksinkertaisen pelin ja neuroverkon luominen, sekä neuroverkon kouluttaminen geneettistä algoritmia käyttäen.

3.1 Pelin suunnittelu

Varsinainen työn suunnittelu alkoi pelin ideoinnilla. Pelin piti olla nopea ja yksinkertainen toteuttaa, jotta muille työn osioille jäisi enemmän aikaa. Pelin tuli olla myös yksinkertainen pelata, eli pelissä piti olla mahdollisimman vähän kontrolleja. Loppujen lopuksi peliksi ideoitiin yksinkertaisen Flappy bird -pelin klooni. Pelissä pelihahmo putoaa vähitellen ja pelaajan on pidettävä tämä ilmassa hyppimällä. Lisäksi pelaajan on väistettävä oikealta tulevia esteitä. Lopullisessa työssä tekoälyn päätettäväksi jää, milloin sen pitää painaa hyppyä.

Ennen kuin valmis neuroverkko pystyy toimimaan, tämä tarvitsee jonkinlaista dataa sisääntuloihinsa. Neuroverkko sai viisi sisääntuloa, jotka saivat datakseen pelihahmon korkeuden, sekä yläesteen vasemman alakulman ja alaesteen vasemman yläkulman koordinaatit (kuva 4). Myöhemmin huomattiin, että esteiden x koordinaatti on aina sama, joten neuroverkko sai loppujen lopuksi vain neljä sisääntuloa.



Kuva 4. Hahmotelma pelin suunnitelmasta

3.2 Neuroverkon suunnittelu

Neuroverkko täytyi suunnitella dynaamiseksi, jotta sen kokoa pystytään muuttamaan helposti ilman, että koodiin täytyy tehdä muutoksia. Kokeilematta on mahdoton tietää, montako ja minkä kokoista tasoa siinä täytyy olla (Tanwar 2019). Peli luo käynnistyessään ennalta määrätyn kokoisen neuroverkon.

Neuroverkko käsittelee sisääntulonsa jokaisella ruudun päivityksellä. Tämä tapahtuu oletuksena 50 kertaa sekunnissa eli 20 millisekunnin välein (Unity 2020b). Joka kerta kun ruutua päivitetään, neuroverkko päättää, hyppääkö se vai ei.

3.3 Geneettisen algoritmin suunnittelu

Geneettinen algoritmi pyrkii poimimaan pisimmille selvinneet yksilöt sen populaatiosta. Tämä tapahtuu laskemalla ensin koko populaation kokonaiskelpoisuus. Kun tämä on laskettu, voidaan jokaiselle yksilön kelpoisuudelle laskea

prosentuaalinen osuus kokonaiskelpoisuudesta. Tämän jälkeen arvot kerrotaan sadalla ja pyöristetään kokonaisluvuiksi. Tämä on yksilön lisääntymismahdollisuus.

Kun lisääntymisarvot on laskettu, luodaan lista mihin jokainen yksilö lisätään tämän lisääntymisarvon verran. Esimerkiksi jos populaation kokonaiskelpoisuus on arvoltaan 5 000 ja yksilön kelpoisuusarvo on 500, yksilön lisääntymisarvoksi tulee 10 ja tämä lisätään listaan 10 kertaa. Tämän ansiosta lopullisessa listassa on n. 100 yksilöä. Huomioitavaa on, että pyöristysten takia lopullisen listan koko ei ole vakio, sillä jos yksilön lisääntymisarvo pyöristetään nollaan sitä ei lisätä listaan ollenkaan.

Kun uutta populaatiota luodaan, otetaan listasta sattumanvaraisesti yksilö, josta uusi yksilö periytetään. Tämä tarkoittaa käytännössä sitä, että uuteen yksilöön kopioidaan vanhan painot, ja tämän jälkeen osaa painoista säädetään sattumanvaraisesti. Säädöt ei eivät saa olla liian suuria, ettei alkuperäisen yksilön perintö katoa kokonaan. Näin yhtäkään yksilöä ei poisteta suoraan populaatiosta, vaan paremmin pärjäävillä yksilöillä on vain suurempi todennäköisyys jatkaa perintöään.

3.4 Kelpoisuusfunktion suunnittelu

Kelpoisuusarvojen täytyy kasvaa eksponentiaalisesti, jotta paremmin pärjäävät yksilöt erottuvat paremmin geneettisessä algoritmissa. Myös yksilöiden pelissä saamien pisteiden täytyy vaikuttaa huomattavasti näiden kelpoisuuteen.

Tämä toteutettiin seuraavasti: Neuroverkoilla on laskuri, joka laskee, kuinka monta kertaa tämä on prosessoinut itsensä. Jos yksilö on hengissä, yksilön kelpoisuusarvoon lisätään laskurin arvo. Kelpoisuuteen lisätään myös yksilön pistemäärä kerrottuna sadalla ja tämän jälkeen nostettuna toiseen potenssiin. Lisäksi jos yksilö on hengissä kun 90 % populaatiosta on kuollut, tämän kelpoisuuteen lisätään vielä 500. Kelpoisuusarvoa päivitetään joka ruudunpäivityksessä.

4 TYÖN TOTEUTUS

Työn toteutus alkoi pelimoottorin valinnalla. Pelimoottoriksi työn kehitykseen valittiin Unity engine -pelimoottorin version 2019.3.0f6, koska se oli työn aloitushetkellä pelimoottorin uusin versio. Toinen vaihtoehto olisi ollut Unreal Engine -pelimoottori. Koodieditoriksi valittiin Microsoftin Visual studio 2019 -ohjelmisto. Ohjelmointikieleksi valikoitui C#-ohjelmointikieli.

Työn toteutus alkoi pelin toteuttamisella. Tämän jälkeen toteutettiin neuroverkko, sekä siihen liittyvä geneettinen algoritmi.

4.1 Pelin toteutus

Pelin toteuttamisessa täytyi ottaa huomioon geneettisen algoritmin populaatio. Pelin pelaajamäärää piti pystyä muuttamaan dynaamisesti. Peliin toteutettiin myös totuusarvomuuttuja, jonka avulla käyttäjä voi valita pelaako peliä ihminen vai neuroverkko.

Pelin grafiikat ovat kaksiulotteiset alkuperäistä Flappy bird -peliä jäljitellen. Itse pelihahmo on avaruusolio, joka lentää avaruusaluksellaan avaruuden läpi. Pelin taustaksi valittiin Hubble-avaruusteleskoopin kuvia, joilla saatiin peliin avaruudellinen tunnelma (kuva 5).

Kuva 5. Kuvankaappaus valmiista pelistä

4.1.1 Gamemanager-luokka

Gamemanager-luokka on luokka, joka hallitsee peliä. Tämä pitää huolen pelaajahahmojen luomisesta, pitää kirjaa pisteistä sekä lopettaa pelin, kun pelaajat ovat kuolleet. Gamemanager-luokan kautta hallitaan myös neuroverkon kokoa sekä pelaajien määrää. Luokka sisältää listan, missä pelin kaikki pelihahmot ovat. Myös käyttöliittymän päivittäminen tapahtuu Gamemanager-luokassa johtuen käyttöliittymän yksinkertaisuudesta.

Kun peli loppuu, Gamemanager-luokka palauttaa pelin sen alkutilaan. Näin mitään ei tarvitse ladata uudestaan muistiin. Kun viimeinen pelaaja kuolee, Gamemanager-luokka odottaa lyhyen ajan ennen alkutilan palauttamista (kuva 6).

```
1 reference
IEnumerator WaitForReset(float time)          //Reset the game after x seconds
{
    resetting = true;
    yield return new WaitForSeconds(time);
    ResetGame();
    resetting = false;
}
```

Kuva 6. Gamemanager-luokka odottaa hetken ennen pelin alkutilan palauttamista

Kun peli palautetaan alkutilaan, Gamemanager-luokka palauttaa tiettyjen muuttujien arvot siihen mitä nämä ovat olleet pelin alussa. Tällaisia ovat esimerkiksi pelaajan pistemäärä sekä esteiden nopeus. Gamemanager-luokka myös kutsuu muiden luokkien tarvittavia funktioita. Jos neuroverkot pelaavat peliä, kutsuu Gamemanager-luokka myös funktiota, jossa geneettinen algoritmi on toteutettu (kuva 7).

```
1 reference
public void ResetGame()
{
    Debug.Log("GAME OVER! Score: " + score);
    uiManager.ScoreText.text = "Score: 0";
    pipeHandler.ResetPipeHandler();
    SpaceBackground.ResetBackground();
    wallSpeed = 0.05f;
    score = 0;
    gameOver = false;
    firstPipe = true;

    foreach(GameObject player in playerList)                    //Loop the playerList
    {
        player.transform.position = playerSpawnLocation.transform.position;     //Place all the players into spawnposition
        PlayerMovement pm = player.GetComponentInChildren<PlayerMovement>();    //Find the playermovement component from gameobjects
        pm.ResetPlayer();                                       //Reset players
    }

    if(!humanPlayer)                                            //If neuralnet played
    {
        EvolveNeuralNets();                                     //Evolve the neuralnets
        generation++;
        uiManager.GenText.text = "Generation: " + generation;   //Print generation to UI
    }
}
```

Kuva 7. Gamemanager-luokan ResetGame-funktio

4.1.2 Pelihahmo

Pelaaja pystyy antamaan pelihahmolleen vain hyppykomentoja. Pelihahmo ei liiku sivuttaissuunnassa ollenkaan, mutta liikkuvalla taustalla luodaan illuusio avaruuden läpi lentävästä pelihahmosta.

Painovoima vetää pelihahmoa jatkuvasti ruudun alareunaan. Kun pelaaja antaa pelihahmolle hyppykomennon, pelihahmon sen hetkinen nopeus nollataan ja tälle annetaan pystysuunnassa voimaa. Annettava voima on muuttuja, jolloin sitä voidaan muuttaa pelimoottorissa (kuva 8).

```
2 references
public void Jump()
{
    rb2d.velocity = Vector2.zero;               // Zero out the velocity
    rb2d.AddForce(new Vector2(0, upForce));     //Add upforce to velocity
}
```

Kuva 8. Pelihahmon Jump-funktio

Esteiden keskellä on näkymätön alue, johon osuessaan pelaaja kasvattaa pisteitään, sekä merkkaa esteen läpäistyksi (kuva 9). Tätä tietoa tarvitaan uusien esteiden luomisessa.

```
0 references
void OnTriggerEnter2D(Collider2D col)
{
    if (col.gameObject.CompareTag("Gate"))
    {
        col.gameObject.GetComponent<LaserPipe>().hasBeenPassed = true; // Mark which pipes has been passed.
        score++;
    }
}
```

Kuva 9. Pelihahmon OnTriggerEnter2D-funktio

Kun pelihahmo osuu seinään, nollataan tämän nopeus, merkataan pelihahmo kuolleeksi, sekä vaihdetaan pelihahmon grafiikka (kuva 10).

```
0 references
void OnCollisionEnter2D(Collision2D col)
{
    if (col.gameObject.CompareTag("Wall"))
    {
        rb2d.velocity = Vector2.zero;       // Zero out the velocity
        isDead = true;
        sr.sprite = sprites[1];             //Switch the sprite
    }
}
```

Kuva 10. Pelihahmon OnCollisionEnter2D-funktio

Kun pelihahmo palautetaan alkutilaan, nollataan sen nopeus ja pisteet, sekä merkataan pelihahmo eläväksi (kuva 11). Pelihahmon sijainti palautetaan alkupisteeseen Gamemanager-luokassa. Myös pelihahmon grafiikka vaihdetaan takaisin.

```
1 reference
public void ResetPlayer()
{
    isDead = false;
    score = 0;
    sr.sprite = sprites[0];
    rb2d.velocity = Vector2.zero;
}
```

Kuva 11. Pelihahmon ResetPlayer-funktio

4.1.3 Esteet

Pelin esteet liikkuvat ruudulla oikealta vasemmalle. Esteet ovat pelissä putkia, tämän takia niihin viitataan koodissa englannin kielen sanalla ”pipe”. Esteiden

vauhtia kasvatetaan, kun pelaaja saa pisteen. Kun esteet menevät ruudun ulkopuolelle, ne tuhotaan. Näin pidetään pelin muistin käyttö kurissa.

Esteillä on käsittelijä, joka pitää huolen esteiden luomisesta ja tuhoamisesta. Käsittelijä sisältää listan, mihin esteet lisätään niiden luomisen yhteydessä (kuva 12).

```
3 references
void SpawnPipe()
{
    GameObject pipe = Instantiate(pipePrefab, spawnlocation.transform.position, Quaternion.identity);
    PipeList.Add(pipe);
}
```

Kuva 12. Esteidenkäsittelijän SpawnPipe-funktio

Kun esteet luodaan, nämä muuttavat korkeuttaan sattumanvaraisesti. Näin esteiden korkeuksiin saadaan variaatiota. Pelin ensimmäiselle esteelle on asetettu tietyt arvot. Esteen luomisen yhteydessä tarkistetaan, onko este ensimmäinen. Jos näin on, este asetetaan ennalta määrättyyn sijaintiin (kuva 13).

```
0 references
void Start()
{
    gm = GameObject.FindObjectOfType<GameManager>();


    topPipeOffset = Random.Range(minOffset, maxOffset);
    botPipeOffset = Random.Range(-1 * minOffset, -1 * maxOffset);

    if (gm.firstPipe)
    {
        topPipeOffset = 5.0f;
        botPipeOffset = -5.0f;
        gm.firstPipe = false;
    }

    topPipe.transform.position = new Vector2(transform.position.x, transform.position.y + topPipeOffset);
    botPipe.transform.position = new Vector2(transform.position.x, transform.position.y + botPipeOffset);
}
```

Kuva 13. Esteidenkäsittelijän Start-funktio

Kun peli palautetaan alkutilaan, käsittelijä tuhoaa esteet listasta ja tyhjentää listan. Tämän jälkeen käsittelijä luo uuden esteen (kuva 14).

```
1 reference
public void ResetPipeHandler()
{
    foreach (GameObject pipe in PipeList)
    {
        Destroy(pipe);
    }

    PipeList.Clear();
    SpawnPipe();
}
```

Kuva 14. Esteiden käsittelijän ResetPipeHandler-funktio

4.2 Neuroverkon toteutus

Neuroverkko on pelissä liitetty pelaajaobjektiin. Itse neuroverkko on vain kaksiulotteinen neuronitaulukko. Ensimmäisellä tasolla on neljä neuronia ja viimeisellä tasolla on yksi. Piilotettujen tasojen määrä ja koko voivat vaihdella, joten nämä luodaan koodissa dynaamisesti (kuva 15).

```
neuralnet = new Neuron[layers][];                    //Setting up how many layers we have in the neural net.

for(int i = 0; i < layers; i++)
{
    if (i == 0)                                      //Input layer is the first layer.
    {
        neuralnet[i] = new Neuron[4];                //There is 4 inputs so the input layer size is always 4.
    }

    else if (i == layers - 1)                        //Output layer is the last layer
    {
        neuralnet[i] = new Neuron[1];                //There is only 1 output.
    }

    else                                             //Layers between
    {
        neuralnet[i] = new Neuron[layerSize];        //Layers between get their size from variable.
    }
}
```

Kuva 15. Neuroverkon tasojen alustus

Ensimmäisellä tason neuroneilla on yksi sisääntulo. Koska ensimmäisellä tasolla neuronien määrä on aina neljä, täytyy toisen tason neuroneilla olla neljä sisääntuloa (kuva 16).

```
for(int i = 0; i < neuralnet.Length; i++)                           //Setting the inputs.
{
    for (int j = 0; j < neuralnet[i].Length; j++)
    {
        if (i == 0)                                                 //First layer is the input layer.
        {
            neuralnet[i][j].inputs = new float[1];                  //On the first layer every neuron has only one input.
        }

        else if (i == 1)
        {
            neuralnet[i][j].inputs = new float[4];                  //On the second layer every neuron has four input.
        }                                                           //Four is the number of input neurons.

        else
        {
            neuralnet[i][j].inputs = new float[layerSize];          //Neurons on every other layer have layerSize amount of inputs.
        }

        neuralnet[i][j].weights = new float[neuralnet[i][j].inputs.Length];
    }
}
```

Kuva 16. Neuroverkon neuronien sisääntulojen alustus

Kun verkko prosessoi itsensä, ensimmäisen tason sisääntuloihin tuodaan neuroverkkoon liitetyn pelaajan korkeus, sekä esteiden koordinaatit Gamemanager-luokasta. Jos neuroverkkoon liitetty pelaaja on kuollut, verkko ei käsittele itseään (kuva 17).

```
1 reference
public void Prosess()
{
    if(!playerMovement.isDead)
    {
        tickCounter++;

        for(int i = 0; i < neuralnet.Length; i++)
        {
            for(int j = 0; j < neuralnet[i].Length; j++)
            {
                if(i == 0)                                                      //Input layer
                {
                    switch (j)
                    {
                        case 0:
                            neuralnet[i][j].inputs[0] = transform.position.y;   //Players position.y value
                            break;

                        case 1:
                            neuralnet[i][j].inputs[0] = gameManager.pipeHandler.topPipeLocation.x;      //Top and bot pipe position.x value
                            break;

                        case 2:
                            neuralnet[i][j].inputs[0] = gameManager.pipeHandler.topPipeLocation.y;      //Top pipe position.y value
                            break;

                        case 3:
                            neuralnet[i][j].inputs[0] = gameManager.pipeHandler.botPipeLocation.y;      //Bottom pipe position.y value
                            break;

                        default:
                            break;
                    }

                }
```

Kuva 17. Neuroverkon prosessointi ensimmäisellä tasolla

Ensimmäisen tason jälkeen seuraavien tasojen sisääntuloille tuodaan edellisen tason ulostulot (kuva 18).

```
else                                                        //Layers between.
{
    for(int k = 0; k < neuralnet[i][j].inputs.Length; k++)
    {
        neuralnet[i][j].inputs[k] = neuralnet[i - 1][k].output;  //Get outputs from previous layer.
    }
}
```

Kuva 18. Neuroverkon prosessointi piilotetuilla tasoilla

Viimeisen tason ulostuloa verrataan neuroverkossa olevaan kynnysarvoon. Jos tämä on pienempi kuin neuroverkon viimeisen tason ulostulo, neuroverkkoon liitetty pelihahmo hyppää (kuva 19).

```
if(i == neuralnet.Length - 1)               //Last layer
{

    output = neuralnet[i][0].output;

    if (output > JumpThreshold)             //If the final output is larger than the threshold jump.
    {
        playerMovement.Jump();
    }
}
```

Kuva 19. Neuroverkon prosessointi viimeisellä tasolla

Neuroverkko laskee kelpoisuutensa aina prosessoinnin jälkeen. Kelpoisuusarvo lasketaan kuten tämän työn luvussa kolme on kuvattu (kuva 20).

```
1 reference
void CalculateFitness()                                                 //Neuralnets fitness is calculated here.
{
    fitness += (playerMovement.score * 100) * (playerMovement.score * 100);     //Score*100^2

    fitness += tickCounter;                                             //Amount of times the neural net processed itself.

    if(gameManager.deadPlayers > gameManager.playerCount * 0.9f)        //Last 10% of players get + 200 into their fitness.
    {
        fitness += 500;
    }
}
```

Kuva 20. Neuroverkon kelpoisuusfunktio

4.3 Neuronin toteutus

Neuroni sisältää taulukot sisääntuloista ja painoista, sekä arvot ulostulolle ja vakiotermin painolle (kuva 21). Neuronin sisääntulotaulukon koko asetetaan neuroverkon alustuksessa ja sisääntulojen arvot asetetaan, kun neuroverkko prosessoi itsensä.

```
public float[] inputs;
public float[] weights;
public float output;
public float biasWeight;
```

Kuva 21. Neuronin muuttujat

Kun neuroni prosessoi itsensä, se summaa sisääntulot kerrottuina painoilla. Tämän jälkeen summaan lisätään vakiotermin painon arvo. Lopuksi summa viedään aktivointifunktiolle (kuva 22).

```
1 reference
public void HandleWeights()
{
    float sum = 0;                          //Reset the sum.

    for(int i = 0; i < inputs.Length; i++)
    {
        sum += (inputs[i] * weights[i]);    //Add to the sum each input times its weight.
    }

    sum += biasWeight;                      //Add bias to the sum.

    Activate(sum);                          //Pass the calculated sum to the activation function.
}
```

Kuva 22. Neuronin HandleWeights-funktio

Aktivointifunktio vie saamansa arvon logistiselle funktiolle. Arvon, jonka logistinen funktio palauttaa viedään laukaisufunktiolle (kuva 23).

```
1 reference
public void Activate(float value)
{
    Fire(Sigmoid(value));                   //Calculating sigmoid and passing it to Fire().
}
```

Kuva 23. Neuronin aktivointifunktio

Logistinen funktio palauttaa sille annetun arvon tämän työn luvun kaksi mukaisesti. Unity Engine -pelimoottorin Mathf-kirjasto on yleisimmistä matemaattisista funktioista koostuva kirjasto. Sieltä löytyvä Exp-funktio palauttaa arvon, joka on neperin luku korotettuna potenssiin sille annetulla arvolla (kuva 24). (Unity 2020a.)

```
1 reference
private float Sigmoid(float x)
{
    return 1 / (1 + Mathf.Exp(-x));
}
```

Kuva 24. Logistinen funktio

Laukaisufunktio asettaa sille annetun arvon neuronin ulostuloon (kuva 25).

```
1 reference
void Fire(float value)
{
    output = value;                                    //Set new value to output.
}
```

Kuva 25. Neuronin laukaisufunktio

4.4 Geneettisen algoritmin toteutus

Geneettinen algoritmi on toteutettu Gamemanager-luokan sisään. Tämä siksi, että Gamemanager-luokka tietää pelin loppumisesta ja käynnistää sen uudelleen.

Aluksi luodaan desimaalilukumuuttuja, jonka arvoksi asetetaan nolla. Tähän muuttujaan lisätään jokaisen pelihahmon kelpoisuusarvo. Näin saadaan laskettua koko populaation kokonaiskelpoisuus (kuva 26).

```
1 reference
void EvolveNeuralNets()
{
    float totalFitness = 0;

    foreach (GameObject player in playerList)                    //Loop the playerList
    {
        NeuralNet nn = player.GetComponentInChildren<NeuralNet>();    //Find the neuralnet component from gameobjects
        totalFitness += nn.fitness;                                   //Adding all the fitness values together.

    }
```

Kuva 26. Populaation kokonaiskelpoisuuden laskeminen

Kun populaation kokonaiskelpoisuus on laskettu, voidaan jokaiselle neuroverkolle lasketa tämän lisääntymismahdollisuusarvo. Tämä tapahtuu jakamalla neuroverkon kelpoisuus populaation kokonaiskelpoisuudella. Tämä arvo tallennetaan neuroverkon muuttujaan (kuva 27).

```
foreach (GameObject player in playerList)                    //Loop the playerList again once the totalFitness is calculated
{
    NeuralNet nn = player.GetComponentInChildren<NeuralNet>();    //Find the neuralnet component from gameobjects

    nn.chanceToMultiply = nn.fitness / totalFitness;          //Neuralnets chance to multiply is its fitness precent from totalFitness.
                                                              //Higher the fitness the higher chance its has to multiply.
}
```

Kuva 27. Neuroverkkojen lisääntymismahdollisuusarvojen laskeminen

Kun jokaiselle neuroverkolle on laskettu lisääntymismahdollisuusarvo, tämä arvo kerrotaan sadalla ja pyöristetään lähimpään kokonaislukuun. Näin saadaan jokin kokonaisluku nollan ja sadan väliltä. Tämän jälkeen neuroverkko lisätään evoluutiovarantolistaan tämän kokonaisluvun verran. Lopuksi saadaan lista, jonka pituus on noin 100. Pyöristysten takia evoluutiovarantolistan koko ei ole vakio (kuva 28).

```
foreach (GameObject player in playerList)                    //Loop the playerList third time once every neuralnet
{                                                            //has its chanceToMultiply calculated from totalfitness.

    NeuralNet nn = player.GetComponentInChildren<NeuralNet>();    //Find the neuralnet component from gameobjects.

    nn.chanceToMultiply *= 100;                               //Multiply the chanceToMultiply by 100.
    nn.chanceToMultiply = Mathf.Round(nn.chanceToMultiply);   //Rounding the chance to nearest integer.
    int n = (int)(nn.chanceToMultiply);                       //n is equal to chanceToMultiply * 100.
                                                              //This gives an int between 0 and 100.
    for (int i= 0; i < n; i++)
    {
        Evolutionpool.Add(nn);                                //Add neuralnet to Evolutionpool n times.
    }                                                         //The Evolutionpool has each neuralnet times their chanceToMultiply percent.
}                                                             //For example if a neuralnets chanceToMultiply is 5%
                                                              //(or 0.05f) its now 5 times in mating pool.
                                                              //The count of Evolutionpool is about 100.
                                                              //It fluctuates because decimals are rounded.
```

Kuva 28. Neuroverkkojen lisääminen evoluutiovarantoon

Tämän jälkeen Gamemanager-luokan pelaajalistan objektit tuhotaan ja luodaan uudestaan. Kun pelaajalista on alustettu, arvotaan jokaiselle pelaajalle satunnainen kokonaisluku. Tämän luvun perusteella valitaan evoluutiovarannosta neuroverkko, jonka painot kopioidaan pelaajalistan neuroverkkoon. Kun painot on kopioitu, kutsutaan 90 % todennäköisyydellä neuroverkon kehittymisfunktiota (kuva 29). Tämä siksi, että osa edellisen sukupolven perimää halutaan jättää vähän seuraavaan sukupolveen.

```
foreach (GameObject player in playerList)
{
    NeuralNet nn = player.GetComponentInChildren<NeuralNet>();      //Find the neuralnet component from the gameobject

    int randomInt = Random.Range(0, (Evolutionpool.Count - 1));     //Take a random neuralnet from evolutionpool

    nn.CopyWeightsFrom(Evolutionpool[randomInt]);                   //Copy weights from random net at evolutionpool

    float randomFloat = Random.Range(0.0f, 1.0f);

    if (0.9f > randomFloat)                                         //Do this 90% of the time
        nn.Evolve();                                                //Evolve the copy

}
```

Kuva 29. Neuroverkkojen painojen kopioiminen

Kun neuroverkko kehittyy, se kutsuu jokaisen neuroninsa kehittymisfunktiota (kuva 30).

```
1 reference
public void Evolve()
{
    for (int i = 0; i < neuralnet.Length; i++)
    {
        for (int j = 0; j < neuralnet[i].Length; j++)
        {
            neuralnet[i][j].Evolve();                    //Evolve every neuron.
        }
    }
}
```

Kuva 30. Neuroverkon Evolve-funktio

Neuronin kehittymisfunktiossa sen painoihin lisätään tai vähennetään sattumanvaraisia lukuja (kuva 31). Muutokset halutaan pitää pieninä, koska liian rajut muutokset painoihin voivat pilata neuronin oppimisen (Brownlee 2016). Tästä syystä myöskään jokaista painoa ei muuteta. Myös vakiotermin painon arvoa muutetaan.

```
1 reference
public void Evolve()
{
    for(int i = 0; i < weights.Length; i++)
    {
        float randomFloat = Random.Range(0.0f, 1.0f);

        if(randomFloat > 0.5f)
            weights[i] += Random.Range(-0.025f, 0.025f);

    }

    biasWeight += Random.Range(-0.025f, 0.025f);
}
```

Kuva 31. Neuronin Evolve-funktio

5 YHTEENVETO

Lopullisessa toteutuksessa neuroverkot oppivat välillä pysyttelemään ruudun keskellä. Tällä tavoin ne pysyivät hengissä loputtomasti ja pystyivät jatkamaan pelaamista. Pelin esteissä ei ollut tarpeeksi variaatiota.

Itse pelin toteuttaminen onnistui nopeasti ja ilman ongelmia. Pääosa työn toteutukseen käytetystä ajasta kului neuroverkon ja geneettisen algoritmin toteutukseen.

Neuroverkon sisääntulot olisivat voineet olla erilaiset. Vaihtoehtoja neuroverkon sisääntuloille olisi voinut olla esimerkiksi pelihahmon pystysuuntainen nopeus. Tällä olisi saattanut olla vaikutusta neuroverkon oppimiseen.

Geneettisessä algoritmissa muutokset neuronien painoihin olivat liian suuria. Tämä johti usein siihen, että sukupolvi ei pärjännyt yhtä hyvin kuin sitä edeltänyt sukupolvi. Myös mutaatiot aiheuttivat ongelmia populaatioiden kehittymiseen. Tästä syystä lopullisessa toteutuksessa mutaatiot jätettiin kokonaan pois geneettisestä algoritmista. Algoritmin populaatio ei myöskään voinut olla pyöristysten takia liian iso, sillä jos kaikkien populaation yksilöiden kelpoisuusarvot olivat alle puoli prosenttia, evoluutiovarantolista jäi tyhjäksi. Tämä johti algoritmin kaatumiseen. Geneettisen algoritmin olisi voinut toteuttaa omaan luokkaansa, ettei Gamemanager-luokka olisi paisunut niin isoksi. Jos työtä jatkettaisiin eteenpäin tämä olisi hyvä tehdä koodin selkeyden vuoksi.

Virheiden löytäminen työssä osoittautui vaikeaksi. Rajallinen kokemus aihetta kohtaan johti usein siihen, että virheiden paikantamiseen kului kauan aikaa. Oli vaikea selvittää, johtuivatko virheet neuronien, neuroverkon vai geneettisen algoritmin toteutuksesta.

Työn tekeminen opetti paljon neuroverkoista, sekä niiden toteuttamisesta. Tämä oli myös työn tavoite. Tulevaisuudessa samanlaisen työn voisi luoda jonkin valmiin koneoppimiskirjaston avulla. Esimerkiksi Unity engine -pelimoottorin Machine learning agents toolkit -työkalu, jota Teemu Ropilo käytti opinnäytetyössään (2019). Toinen vaihtoehto olisi toteuttaa neuroverkoilla jokin muunlainen projekti, jossa koulutuksessa voitaisiin käyttää vastavirta-algoritmia.

LÄHTEET

Brownlee, J. 2016. Crash Course On Multi-Layer Perceptron Neural Networks. Blogi. Päivitetty 19.8.2019. Saatavissa: https://machinelearningmastery.com/neural-networks-crash-course/ [viitattu: 26.3.2020].

Chen, J. 2019. Neural Network. Saatavissa: https://www.investopedia.com/terms/n/neuralnetwork.asp [viitattu: 26.3.2020].

Hamilo, M. 2013. Hermoverkkoja matkivat tietokoneet oppivat tunnistamaan esineitä. Artikkeli. Saatavissa: https://suomenkuvalehti.fi/jutut/ulkomaat/hermoverkkoja-matkivat-tietokoneet-oppivat-tunnistamaan-esineita/ [viitattu 26.3.2020].

Heino, R. 2019. Itseoppiva tekoäly pelissä. Kaakkois-suomen ammattikorkeakoulu. Tieto- ja viestintätekniikan koulutusohjelma. Opinnäytetyö. PDF-dokumentti. Saatavissa: http://urn.fi/URN:NBN:fi:amk-2019121126000 [viitattu 29.4.2020].

Juliani, A. 2017. Introducing: Unity Machine Learning Agents Toolkit. Blogi. Saatavissa: https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/ [viitattu: 2.4.2020].

Leon, K. 2017. Making a Simple Neural Network. Artikkeli. Saatavissa: https://becominghuman.ai/making-a-simple-neural-network-2ea1de81ec20 [viitattu 28.4.2020].

Mallawaarachchi, V. 2017. How to define a Fitness Function in a Genetic Algorithm? Artikkeli. Saatavissa: https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4 [viitattu: 28.4.2020].

Marr, B. 2018. What Is Deeplearning AI? A Simple Guide With 8 Practical Examples. Verkkolehti. Saatavissa: https://www.forbes.com/sites/bernardmarr/2018/10/01/what-is-deep-learning-ai-a-simple-guide-with-8-practical-examples/#633f7bc08d4b [viitattu 29.4.2020].

Microsoft. s.a. Azure Machine Learning. WWW-dokumentti. Saatavissa: https://azure.microsoft.com/en-us/services/machine-learning/ [viitattu 28.4.2020].

Mills, T. 2018. Machine Learning Vs. Artificial Intelligence: How Are They Different? Verkkolehti. Saatavissa: https://www.forbes.com/sites/forbestechcouncil/2018/07/11/machine-learning-vs-artificial-intelligence-how-are-they-different/#7ef74e103521 [viitattu: 27.3.2020].

Nicholson, C. 2019. A Beginner's Guide to Backpropagation in Neural Networks. Artikkeli. Saatavissa: https://pathmind.com/wiki/backpropagation [viitattu 24.4.2020].

Ropilo, T. 2019. Teaching a machine learning agent to survive in a top-down environment. Kaakkois-suomen ammattikorkeakoulu. Tieto- ja viestintätekniikan koulutusohjelma. Opinnäytetyö. PDF-dokumentti. Saatavissa: http://urn.fi/URN:NBN:fi:amk-2019120925566 [viitattu 29.4.2020].

Rouse, M. 2020. machine learning (ML). Artikkeli. Saatavissa: https://searchenterpriseai.techtarget.com/definition/machine-learning-ML [viitattu 28.4.2020]

Shiffman, D. 2012. The Nature of Code. PDF-dokumentti. Saatavissa: https://natureofcode.com/ [viitattu 6.5.2020].

Strachnyi, K. 2019. Brief History of Neural Networks. Artikkeli. Saatavissa: https://medium.com/analytics-vidhya/brief-history-of-neural-networks-44c2bf72eec [viitattu 28.4.2020].

Tanwar, S. 2019. Introduction to neural networks. Artikkeli. Saatavissa: https://medium.com/@sanchittanwar75/introduction-to-neural-networks-660f6909fba9 [viitattu 24.4.2020].

Tensorflow. s.a. Why TensorFlow. WWW-dokumentti. Saatavissa: https://www.tensorflow.org/about [viitattu 28.4.2020].

Unity. 2020a. Mathf struct in UnityEngine. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2019.3/Documentation/ScriptReference/Mathf.html [viitattu 22.4.2020].

Unity. 2020b. MonoBehaviour.FixedUpdate(). WWW-dokumentti. Saatavissa: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html [viitattu 24.4.2020].

KUVALUETTELO