Jere Vänttinen

Monikerroksisen perseptroniverkon opetusalgoritmin toteutus ja kokeellinen testaus

Diplomityö

Energia- ja
informaatiotekniikka
Diplomityö
Automaatio ja tietotekniikka

Vaasa 2025

VAASAN YLIOPISTO
Energia- ja informaatiotekniikka
Tekijä: Jere Vänttinen
Tutkielman nimi: Monikerroksisen perseptroniverkon opetusalgoritmin toteutus ja kokeellinen testaus: Diplomityö
Tutkinto: Diplomi-insinööri
Oppiaine: Automaatio ja tietotekniikka
Työn ohjaaja: Jouni Lampinen
Valmistumisvuosi: 2025 **Sivumäärä:** 115

TIIVISTELMÄ:
Tämän tutkimuksen tavoitteena on tutkia monitavoiteoptimointimenetelmän soveltuvuutta monikerroksisen perseptronineuroverkon koulutuksen tehostamiseen käyttämällä differentiaalievoluutioalgoritmia. Tutkimus vertaa perinteistä yksitavoiteoptimointia monitavoiteoptimointiin ja tarkastelee tästä saatuja tuloksia. Yksitavoiteoptimointi on altis neuronien permutaatio-ongelmalle, jossa kunkin piilokerroksen neuroneiden järjestelymahdollisuudet johtavat useisiin globaaleihin optimiratkaisuihin. Monitavoiteoptimoinnin avulla haetaan ratkaisua tähän ongelmaan ja ensisijaisesti verrataan monitavoiteoptimoinnin hyviä ja huonoja puolia perinteisempään yksitavoiteoptimointiin. Työn tutkimuskysymyksen voi muotoilla seuraavasti: “Miten monitavoiteoptimointi voi parantaa differentiaalievoluutioalgoritmin suorituskykyä monikerroksisen perseptroniverkon koulutuksessa?”.

Työ suoritettiin kouluttamalla monikerroksiselle perseptroniverkolle IRIS-datakokoelma. Vertailu suoritettiin yksitavoiteoptimoinnin ja monitavoiteoptimoinnin toteutusten välillä. Toteutus implementoitiin Python-ohjelmointikielellä ensin yksitavoiteoptimoinnille käyttämällä tavoitefunktiona luokittelutarkkuutta ja sitten monitavoiteoptimoinnille käyttämällä tavoitefunktioina luokittelutarkkuutta sekä toisen tavoitefunktion osalta kahta erilaista toteutusta. Tutkimuksessa implementoitiin kokeelliset ohjelmistot, joita voidaan kehittää edelleen tarpeen mukaan. Tulosten suorituskykymittareina käytettiin algoritmien tehokkuutta, luokittelutarkkuutta ja luotettavuutta. Näiden suorituskykymittareiden avulla vertailtiin yksi- ja monitavoiteoptimointimenetelmien suoriutumista toisiinsa nähden.

Tutkimuksen hypoteesi on, että monitavoiteoptimointi tehostaa differentiaalievoluutioalgoritmin suorituskykyä monikerroksisen perseptroniverkon koulutuksessa perinteiseen yksitavoiteoptimointiin verrattuna. Tutkimuksen tulokset osoittavat, että monitavoiteoptimointi ei juurikaan parantanut algoritmin tehokkuutta, vaan parannukset näkyivät etenkin algoritmin luokittelutarkkuudessa ja luotettavuudessa. Monitavoiteoptimointi oli suoritusaikansa ja vaadittavien resurssien puolesta heikompi yksitavoiteoptimointiin verrattuna, mutta monitavoiteoptimointi kykeni saavuttamaan parempia luokittelutarkkuuden optimiarvoja pienemmällä keskihajonnalla kuin yksitavoiteoptimointi. Monitavoiteoptimointi ei poistanut permutaatio-ongelman ilmenemistä, mutta antoi mahdollisia viitteitä permutaatio-ongelman vähentymisestä. Permutaatio-ongelman ilmeneminen saattoi kuitenkin johtua useasta eri tekijästä. Täten tutkimuksen tulokset tukevat jokseenkin ennen tutkimusta asetettua hypoteesia, joten monitavoiteoptimointia voidaan pitää lisätutkimuksen kannalta lupaavana menetelmänä monikerroksisten perseptroniverkkojen koulutuksessa differentiaalievoluutioalgoritmilla tulevaisuudessa. Tämän tutkimuksen tuloksia voidaan analysoida lisää tulevaisuudessa.

AVAINSANAT: Neuroverkko, Perseptroniverkko, Opetusalgoritmi, Differentiaalievoluutio, Yksitavoiteoptimointi, Monitavoiteoptimointi

Sisällys

1 Johdanto 9

2 Neuroverkot 11

2.1 Neuroverkkojen perusteet 11

2.2 Monikerroksiset perseptroniverkot 12

2.3 Aktivaatio- ja virhefunktiot 14

2.4 Neuroverkon permutaatio-ongelma 16

3 Opetusalgoritmit 19

3.1 Yksitavoiteoptimointi 19

3.2 Differentiaalievoluutio 21

3.3 DE neuroverkon koulutuksessa 23

3.4 Monitavoiteoptimointi 24

4 Tutkimusmenetelmät 27

4.1 Aiempi tutkimus 27

4.2 Tutkimuksen suunnittelu 28

4.2.1 DE-algoritmin ja neuroverkon toteutus 29

4.2.2 Datakokoelma ja kokeiden toteutus 30

4.2.3 Suorituskykymittarit 31

4.3 Tavoitefunktiot 32

5 Tutkimuksen toteutus 33

5.1 Tutkimusympäristö 33

5.2 Ohjelmallinen toteutus 34

5.2.1 Pääohjelman toteutus 34

5.2.2 Neuroverkon toteutus 35

5.2.3 Differentiaalievoluutioalgoritmin toteutus 35

5.2.4 Yksi- ja monitavoiteoptimoinnin toteutus 36

5.3 Parametrien optimointi 36

5.4 Ohjelman toiminta ja havainnot toteutuksessa 42

6 Tulokset 46

6.1 Algoritmien suorituskyky 46
6.1.1 Tehokkuus 47
6.1.2 Luokittelutarkkuus 54
6.1.3 Luotettavuus 58
6.2 Optimointimenetelmien vertailu 61
6.3 Tulosten tulkinta 71
6.4 Monitavoiteoptimointi hakua ohjaavalla tavoitefunktiolla $\boldsymbol{f2}$ 73
6.5 Monitavoiteoptimointi hakua ohjaavalla funktiolla $\boldsymbol{f2}$ 500:lla sukupolvella 86
6.6 Tulosten syväanalyysi 87
7 Johtopäätökset 93
Lähteet 97
Liitteet 100
Liite 1. Yksitavoiteoptimoinnin toteutus 100
Liite 2. Monitavoiteoptimoinnin toteutus 107
Liite 3. Hakua ohjaavan funktion $\boldsymbol{f2}$ määrittely 115

Kuvat

Kuva 1. MLP-verkko (Lähde: Medium) 12
Kuva 2. Neuroverkon permutaatio-ongelma (Lähde: Researchgate, Yang, 2001) 18
Kuva 3. Differentiaalievoluutioalgoritmi (Lähde: WILEY) 23
Kuva 4. Populaation koon ja mutaatiovakion optimointi - Yksitavoiteoptimointi 39
Kuva 5. Risteytysvakion ja sukupolvien määrän optimointi - Yksitavoiteoptimointi 40
Kuva 6. Populaation koon ja mutaatiovakion optimointi - Monitavoiteoptimointi 41
Kuva 7. Risteytysvakion ja sukupolvien määrän optimointi - Monitavoiteoptimointi ... 42
Kuva 8. Yksitavoiteoptimoinnin tulostus 43
Kuva 9. Yksitavoiteoptimoinnin tulostus lopussa 43
Kuva 10. Monitavoiteoptimoinnin tulostus 44
Kuva 11. Monitavoiteoptimoinnin tulostus lopussa 44

Kuviot

Kuvio 1. Suoritusaika – Yksitavoiteoptimointi 48
Kuvio 2. Suoritusaika – Monitavoiteoptimointi 49
Kuvio 3. Luokittelutarkkuuden arvon kehittyminen – Yksitavoiteoptimointi 50
Kuvio 4. Luokittelutarkkuuden arvon kehittyminen – Monitavoiteoptimointi 51
Kuvio 5. Softmaxin arvon kehittyminen ja molempien funktioiden arvot– Monitavoiteoptimointi 52
Kuvio 6. Luokittelutarkkuuden ja evaluointien määrän suhde– Yksitavoiteoptimointi . 53
Kuvio 7. Luokittelutarkkuuden ja evaluointien määrän suhde– Monitavoiteoptimointi 54
Kuvio 8. Luokittelutarkkuuden loppuarvo ajojen yli – Yksitavoiteoptimointi 55
Kuvio 9. Luokittelutarkkuuden loppuarvo ajojen yli – Monitavoiteoptimointi 56
Kuvio 10. Datajoukkojen virhe – Yksitavoiteoptimointi 57
Kuvio 11. Datajoukkojen virhe – Monitavoiteoptimointi 58
Kuvio 12. Luokittelutarkkuuden keskihajonta – Yksi- ja monitavoiteoptimointi 59
Kuvio 13. Keskiarvo suhteessa kaikkiin tuloksiin 100 ajolta – Yksitavoiteoptimointi 60
Kuvio 14. Keskiarvo suhteessa kaikkiin tuloksiin ajoilta 100 ajolta – Monitavoiteoptimointi 61

Kuvio 15. Yksi- ja monitavoiteoptimointien suoritusajat 62
Kuvio 16. Yksi- ja monitavoiteoptimointien kokonaissuoritusajat 62
Kuvio 17. Yksi- ja monitavoiteoptimointien luokittelutarkkuus suhteessa sukupolviin . 63
Kuvio 18. Yksi- ja monitavoiteoptimointien luokittelutarkkuuden keskiarvo suhteessa evaluaatioihin 64
Kuvio 19. Yksi- ja monitavoiteoptimointien luokittelutarkkuuden loppuarvo 65
Kuvio 20. Yksi- ja monitavoiteoptimointien tulosten keskihajonta 66
Kuvio 21. Yksi- ja monitavoiteoptimointien lopputuloksen keskiarvo 67
Kuvio 22. Datajoukkojen tarkkuus – Yksitavoiteoptimointi 68
Kuvio 23. Datajoukkojen tarkkuus yksittäisellä ajokerralla - Monitavoiteoptimointi..... 68
Kuvio 24. Korkein ja pienin validointi- sekä testijoukon tarkkuus – Yksitavoiteoptimointi 69
Kuvio 25. Korkein ja pienin validointi- sekä testijoukon tarkkuus – Monitavoiteoptimointi 69
Kuvio 26. Suoritusaika – Monitavoiteoptimointi minimoivalla funktiolla 74
Kuvio 27. Luokittelutarkkuus suhteessa sukupolviin - Monitavoiteoptimointi minimoivalla funktiolla 75
Kuvio 28. Luokittelutarkkuus suhteessa evaluaatioihin - Monitavoiteoptimointi minimoivalla funktiolla 76
Kuvio 29. Luokittelutarkkuuden loppuarvo - Monitavoiteoptimointi minimoivalla funktiolla 78
Kuvio 30. Minimoivan funktion f2 loppuarvot 79
Kuvio 31. Luokittelutarkkuuden keskihajonta ja keskiarvo - Monitavoiteoptimointi minimoivalla funktiolla 81
Kuvio 32. Datajoukkojen tarkkuus - Monitavoiteoptimointi minimoivalla funktiolla 82
Kuvio 33. Datajoukkojen tarkkuus yksittäisellä ajokerralla - Monitavoiteoptimointi minimoivalla funktiolla 83
Kuvio 34. Korkein ja pienin validointi- sekä testijoukon tarkkuus - Monitavoiteoptimointi minimoivalla funktiolla 84

Kuvio 35. Luokittelutarkkuus suhteessa evaluaatioihin - Monitavoiteoptimointi 500 sukupolvella 86
Kuvio 36. Luokittelutarkkuus 100 ajokerralla - Monitavoiteoptimointi 500 sukupolvella 87
Kuvio 37. PCA-analyysi painoarvoista - Monitavoiteoptimointi minimoivalla funktiolla 88
Kuvio 38. PCA-analyysi painoarvoista - Monitavoiteoptimointi 500 sukupolvella 89
Kuvio 39. PCA-analyysi painoarvoista - Yksitavoiteoptimointi 90
Kuvio 40. PCA-analyysi Hungarian-algoritmin jälkeen - Monitavoiteoptimointi minimoivalla funktiolla 91
Kuvio 41. Pareto-rintama - Monitavoiteoptimointi minimoivalla funktiolla 92

Taulukot

Taulukko 1. DE:n parametrit 31
Taulukko 2. Tavoitefunktiot 32
Taulukko 3. DE:n lähtöparametrit 37
Taulukko 4. Yksitavoiteoptimoinnin DE parametrit 37
Taulukko 5. Monitavoiteoptimoinnin DE parametrit 38
Taulukko 6. Evaluaatioiden määrä 80 % luokittelutarkkuuden saavuttamiseksi - Yksitavoiteoptimointi 70
Taulukko 7. Evaluaatioiden määrä 80 % luokittelutarkkuuden saavuttamiseksi - Monitavoiteoptimointi 70
Taulukko 8. Optimointimenetelmien suoriutuminen 71
Taulukko 9. Optimointimenetelmät suhteessa suorituskykymittareihin 72
Taulukko 10. Evaluointien määrä 80 % luokittelutarkkuuden saavuttamiseksi - Monitavoiteoptimointi minimoivalla funktiolla 85
Taulukko 11. Optimointimenetelmät suhteessa suorituskykymittareihin 94

Kaavat

Kaava 1. Sigmoid-funktio 15
Kaava 2. Softmax-funktio 15

Kaava 3. ReLU-funktio 16
Kaava 4. Keskineliövirhe 16
Kaava 5. DE:n mutanttivektori 21
Kaava 6. Monitavoiteoptimoinnin määrittely 26
Kaava 7. Pareto-optimaalisuuden määrittely 26
Kaava 8. Pareto-optimaalisuuden ehto 26
Kaava 9. Hakua ohjaava funktio $\boldsymbol{f2}$ 73

Lyhenteet

DE – Differential evolution (suom. differentiaalievoluutio)
MLP – Multi-Layer Perceptron (suom. monikerroksinen perseptroniverkko)
PCA – Principal Component Analysis (suom. pääkomponenttianalyysi)

1 Johdanto

Tekoäly ja koneoppiminen ovat viime vuosina mullistaneet teknologianalaa merkittävin edistyksin. Tekoälyn toiminta perustuu pitkälti optimointiin ja Vuontisjärvi (2023) toteaakin, että optimoinnin tarkoituksena on löytää yksi tai useampi ratkaisu, jotka ovat haluttujen tavoitteiden kannalta suotuisimpia. Tekoälyn toiminta perustuu osiltaan neuroverkkoihin, jotka muodostavat keskeisen teknologian ratkaistaessa monimutkaisia optimointiongelmia. Monikerroksiset perseptronineuroverkot (MLP) ovat yksi käytetyimmistä neuroverkkoarkkitehtuureista, ja niitä sovelletaan laajasti monimutkaisten epälineaaristen ongelmien ratkaisussa, joita voivat olla esimerkiksi kuvantunnistus ja datan analysoiminen. Neuroverkon onnistunut koulutus vaatii kuitenkin painokertoimien optimoinnin. Perinteisemmät menetelmät voivat kärsiä paikallisiin optimeihin jumittumisesta ja alhaisesta algoritmin tehokkuudesta. Optimointimenetelmistä differentiaalievoluutioalgoritmi vaikuttaa tarjoavan lupaavia mahdollisuuksia erityisesti laajan skaalan ongelmiin. Aiempaa tutkimusta on tehty etenkin yksitavoiteoptimointimenetelmän osalta, mutta monitavoiteoptimointimenetelmään ei ole tehty laajempaa tutkimusta ainakaan differentiaalievoluutioalgoritmin hyödyntämisen kannalta.

Tämän tutkimuksen tutkimuskysymys voidaan muotoilla seuraavasti: “Miten monitavoiteoptimointi voi parantaa differentiaalievoluutioalgoritmin suorituskykyä monikerroksisen perseptroniverkon koulutuksessa?”. Differentiaalievoluutioalgoritmin käyttö neuroverkkojen koulutuksessa tuo esiin tiettyjä haasteita. Yksi keskeisimmistä ongelmista on neuronien permutaatio-ongelma, joka liittyy piilokerrosten neuroneiden järjestelymahdollisuuksiin. Tämä ongelma synnyttää useita yhtä hyviä globaaleja optimiratkaisuja, mikä hidastaa optimointiprosessia ja vähentää algoritmin luotettavuutta. Yksitavoiteoptimoinnin osalta algoritmi jää myös usein jumiin paikalliisiin optimiratkaisuihin, eikä näin ollen löydä laajassa skaalassa olevaa globaalia optimiratkaisua. Algoritmi olisi tärkeää saada keskittymään yhteen globaaliin optimiratkaisuun. Tässä tutkimuksessa pyritään ratkaisemaan tämä ongelma monitavoiteoptimoinnin avulla, jossa lisätään vähintään toinen tavoitefunktio

ohjaamaan hakua kohti yhtä suosikkijärjestystä neuroneille. Lisäksi neuroverkon koulutuksen tehoa ja laatua on syytä parantaa, jotta verkko oppii luokittelemaan dataa nopeammin paremmalla tarkkuudella. Tutkimuksen tärkeimpänä tavoitteena on kuitenkin selvittää, voiko monitavoiteoptimointi tehostaa differentiaalievoluutioalgoritmin käyttöä monikerroksisen perseptroniverkon koulutuksessa perinteiseen yksitavoiteoptimointiin verrattuna. Tutkimus siis rajataan vertaamaan yksi- ja monitavoiteoptimointimenetelmiä toisiinsa.

Tutkimuksen hypoteesi on, että monitavoiteoptimointi parantaa differentiaalievoluutioalgoritmin suoriutumista monikerroksisen perseptroniverkon koulutuksessa. Tutkimus toteutetaan opettamalla monikerroksiselle perseptroniverkolle IRIS-datakokoelma. Tutkimuksen suorittamiseksi on toteutettava implementaatio differentiaalievoluutioalgoritmille ja monikerroksiselle perseptroniverkolle. Näiden kahden ilmentymän kontrolloimiseksi luodaan myös erillinen pääohjelma. Sekä yksi- että monitavoiteoptimoinnille tehdään kummallekin oma toteutuksensa Python-ohjelmointikielellä. Kokeet suoritetaan kahdella eri tavalla: ensin perinteisellä yksitavoiteoptimoinnilla, jossa optimoidaan vain verkon luokittelutarkkuutta, ja sen jälkeen monitavoiteoptimointimenetelmällä, jossa lisätään toinen tavoitefunktio luokittelutarkkuuden rinnalle vähentämään neuronien järjestyksen virheitä. Tuloksia analysoidaan tiettyjen suorituskykymittareiden näkökulmista, jotka ovat tehokkuus, luokittelutarkkuus ja luotettavuus. Lopuksi tulokset analysoidaan ja ne visualisoidaan erilaisilla kuvaajilla sekä taulukoilla. Tuloksista tehdään erilliset johtopäätelmät ja pohditaan mahdollista tulevaisuuden tutkimusta.

2 Neuroverkot

Tässä luvussa käsitellään neuroverkkojen perustoimintaa ja perehdytään monikerroksisiin perseptroniverkkoihin sekä selvitetään, mikä on aktivaatio- ja virhefunktioiden tehtävä neuroverkoissa. Luku koostuu kolmesta alaluvusta, jotka käyvät nämä edellä mainitut asiat läpi. Luvun tavoite on käsitellä neuroverkot riittävällä tarkkuudella, jotta lukija saa perusidean neuroverkkojen toiminnasta.

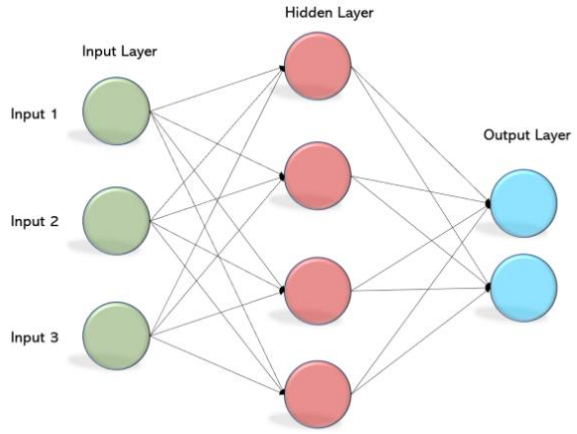
2.1 Neuroverkkojen perusteet

Juvonen (2020) toteaa, että neuroverkot on alun perin kehitetty mallintamaan aivojen toimintaa. Neuroverkkojen kehityksen tavoitteena on ollut kaapata aivojen laskennallinen toiminta, joka eroaa perinteisten tietokoneiden laskentatavasta. Neuronimalli on yksinkertaistettu kuvaus tästä toiminnasta, jossa neuroni toimii laskentayksikkönä prosessoiden saamaansa informaatiota. Yksinkertaisimmillaan neuroverkko sisältää vain yhden neuronin, jolloin käytetään käsitettä perseptroni.

Juvosen (2020) mukaan neuroverkot eivät kuitenkaan kykene täysin esittämään tai selittämään biologisten hermoverkkojen kompleksisuutta. Myöskään neuroverkkojen laskennalliset ominaisuudet eivät aina ole linjassa biologisten hermoverkkojen toiminnan kanssa. Näistä syistä neuroverkkojen tutkimus voidaan jakaa kahteen suuntaan: laskennalliseen neurotieteeseen ja neuroverkkotieteeseen. Laskennallinen neurotiede hyödyntää neuroverkkoja biologisten hermoverkkojen tutkimiseen, kun taas neuroverkkotiede keskittyy kehittämään järjestelmiä, jotka kykenevät oppimaan monimutkaisia toimintoja riippumatta niiden yhteydestä biologisiin hermoverkkoihin.

Neuroverkkojen tehokkuus perustuu Juvosen (2020) mukaan niiden rakenteeseen, oppimiskykyyn ja yleistämisominaisuuksiin. Rakenteellisesti neuroverkot jaetaan kolmeen pääluokkaan: yksikerroksisiin, monikerroksisiin ja takaisinkytkettyihin verkkoihin. Kaikille näille tyypeille yhteistä on informaation eteenpäin syöttäminen syötekerroksesta tuloskerrokseen (engl. feed forward). Yksikerroksisissa

neuroverkoissa on vain syöte- ja tuloskerros, kun taas monikerroksisissa neuroverkoissa on lisäksi yksi tai useampi piilokerros. Monikerroksisiin neuroverkkoihin perehdytään tarkemmin luvussa 2.2. Useiden piilokerrosten sisältäviä verkkoja kutsutaan syväoppimisverkoiksi (engl. deep learning), koska niiden oppiminen tapahtuu näissä piilokerroksissa. Kuvassa 1 on mallinnettu monikerroksinen neuroverkko, jonka käyttöön myös tässä työssä keskitytään. Kuvasta voidaan huomata vihreä syötekerros (engl. Input layer), punainen piilokerros (engl. Hidden Layer) ja sininen tuloskerros (engl. Output Layer).



Kuva 1. MLP-verkko (Lähde: Medium)

Juvonen (2020) kertoo myös, että neuroverkot voivat olla täysin kytkettyjä, jolloin jokainen solmu on yhteydessä edellisen ja seuraavan kerroksen kaikkiin solmuihin. Tämä rakenne mahdollistaa informaation tehokkaan kulun ja laskennan verkon eri osissa. Täysin kytketyt verkot ovat hyödyllisiä monimutkaisissa epälineaarisissa ongelmissa topologiansa ansiosta.

2.2 Monikerroksiset perseptroniverkot

Monikerroksinen perseptroniverkko (MLP) on yksi yleisimmin käytetyistä neuroverkkoarkkitehtuureista. Kairamon (2022) mukaan MLP on tehokas menetelmä monimutkaisten epälineaaristen suhteiden mallintamiseen ja soveltuu erityisesti

luokittelutehtäviin ja ennustamiseen. MLP:n perusidea on laajentaa yksinkertaisemman perseptronin mallia niin, että siinä on useampia kerroksia, kuten syötekerros, piilokerros ja tuloskerros. Syötekerros vastaanottaa datan syötteet, ja sen jälkeen tieto kulkee piilokerroksista tuloskerrokseen. MLP:n rakenne mahdollistaa sen, että verkko oppii monimutkaisempia suhteita ja tekee päätöksiä näistä suhteista.

Syötekerroksen neuroneilla ei ole laskennallista roolia, vaan ne vain vastaanottavat tiedon ja välittävät sen seuraaviin kerroksiin. Piilokerroksissa puolestaan tieto prosessoidaan ja aktivoidaan, yleensä käyttämällä aktivaatiofunktiota, josta esimerkkejä ovat Sigmoid, Softmax, Tanh tai ReLU. Tässä tutkimuksessa keskitytään alustavasti Sigmoid- ja Softmax-funktioihin. Aktivaatiofunktiot ovat keskeisiä, koska ne tuovat verkkoon epälineaarisuuden, joka on välttämätöntä monimutkaisessa oppimisessa. Piilokerroksissa voidaan käsitellä myös laajempia ja monimutkaisempia piirteitä kuin syötekerroksessa, ja näin verkko oppii havaitsemaan piilotettuja rakenteita datassa. Kairamo (2022) korostaa, että piilokerrosten määrä ja niiden neuroneiden määrä vaikuttavat suoraan verkon kykyyn oppia ja ratkaista ongelmia sekä lisäksi siihen, kuinka laskennallisesti raskas verkko on.

Tuloskerros tuottaa lopullisen päätöksen, joka voi olla esimerkiksi luokittelutulos. Jos verkkoa käytetään luokittelutehtävissä, tuloskerros käyttää yleensä Softmax-aktivaatiofunktiota, joka muuttaa piilokerroksesta saadut arvot todennäköisyyksiksi, jotka summautuvat yhteen. Näin ollen luokat ennustetaan todennäköisyyksinä, mikä tekee verkosta erittäin soveltuvan moniluokkaluokitteluun. Toivanen (2013) käyttää Softmax-aktivaatiofunktiota, koska hän kokee sen olevan hyödyllinen MLP-verkon opettamisessa.

MLP-verkko oppii painokertoimia ja bias-arvoja virhefunktion perusteella, joka arvioi verkon ennusteen ja todellisten arvojen välistä eroa. Kairamo (2022) kertoo laskevansa virheen keskineliövirheellä, joskin luokittelutehtävissä käytetään usein myös muita virheen mittareita. Virheen minimointi tapahtuu algoritmeilla, jossa verkon virhe jaetaan

kunkin painon mukaan, ja painot päivitetään gradientin suuntaan, jotta virhe pienenee. Toivanen (2013) toteaa, että tämä prosessi mahdollistaa sen, että MLP oppii parhaiten mallintamaan datan piirteitä ja parantamaan ennusteitaan.

Kairamo (2022) huomauttaa, että MLP:n tehokkuus ei perustu ainoastaan sen rakenteeseen, vaan myös optimointiprosessiin ja sen parametreihin. Parametreja ovat esimerkiksi oppimisnopeuden ja piilokerrosten neuronien määrä, joiden oikeanlainen säätö on olennaista verkon suorituskyvyn kannalta. Liian suuri määrä piiloneuroneita voi johtaa ylisovittamiseen (engl. overfitting), jossa malli oppii liian hyvin opetusdatan erityispiirteet, mutta ei kykene yleistämään uusiin, tuntemattomiin syötteisiin. Vastaavasti liian yksinkertainen verkko voi jäädä alisovitukseen (engl. underfitting), jolloin se ei pysty oppimaan tarpeeksi datan rakenteita. Tämä tekee MLP:n suunnittelusta ja optimoinnista erityisen tärkeää, jotta verkon suorituskyky saadaan optimaaliseksi.

Kairamo (2022) ja Toivanen (2013) korostavat yhdessä, että MLP:n tehokkuus ja soveltuvuus riippuvat sen kyvystä oppia epälineaarisia suhteita ja yleisesti käsitellä suuria määriä dataa. Kaksikerroksiset ja monikerroksiset verkot omaavat enemmän kapasiteettia ja joustavuutta oppimiseen, mutta niiden laskennallinen monimutkaisuus ja koulutusvaatimukset voivat kasvaa merkittävästi verkon koon kasvaessa. Tämä tuo haasteita esimerkiksi laskentatehon puolesta. Tämän vuoksi verkon rakenteen ja parametrien optimointi on tärkeä vaihe MLP:n koulutuksessa, jotta saadaan muodostettua tasapaino tehokkuuden ja laskennallisen monimutkaisuuden välillä.

2.3 Aktivaatio- ja virhefunktiot

Aktivaatiofunktio on neuroverkon keskeinen komponentti, joka määrittää, kuinka neuronin laskema arvo välitetään seuraavaan kerrokseen. Aktivaatiofunktio mahdollistaa epälineaarisuuden lisäämisen verkkoon, mikä on keskeistä monimutkaisempien ongelmien ratkaisemiseksi. Ilman aktivaatiofunktiota neuroverkko pystyisi vain suorittamaan lineaarisia muunnoksia, mikä rajoittaisi sen

soveltamismahdollisuuksia. Aktivaatiofunktio määrittää, kuinka syöte z muunnetaan neuronin aktivoinniksi. Aktivaatiofunktio voi olla esimerkiksi Sigmoid, Softmax, ReLU tai lineaarinen funktio (Haasiomäki, 2019). Alla olevissa kaavoissa esitelty Sigmoid, Softmax ja ReLU.

Sigmoid-funktio muuntaa syötteen arvoon, joka vaihtelee välillä 0 ja 1. Tämä ominaisuus tekee siitä erityisen hyödyllisen luokittelutehtävissä, joissa tulokset voidaan tulkita todennäköisyyksinä. Sigmoid-funktion kaava (kaava 1) on seuraava:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Kaava 1. Sigmoid-funktio

Softmax-funktio on erityisesti tuloskerroksessa käytettävä aktivaatiofunktio, joka muuntaa verkon tuottamat arvot todennäköisyyksiksi. Tämä on hyödyllistä moniluokkaluokittelutehtävissä, joissa halutaan laskea todennäköisyydet sille, mihin luokkaan syöte todennäköisimmin kuuluu. Softmax-funktion kaava (kaava 2) on:

$$y_k = \frac{\exp\,(z_k)}{\sum_{j=1}^{m} \exp\,(z_j)}$$

Kaava 2. Softmax-funktio

missä y_k on luokan *k* todennäköisyys, z_k on piilokerroksesta tuleva arvo, ja *m* on luokkien määrä.

ReLU on tehokas syväoppimisessa. ReLU toimii niin, että se asettaa kaikki negatiiviset arvot nollaksi ja jättää positiiviset arvot ennalleen. ReLU:n kaava (kaava 3) on:

$$ReLU(z) = \max(0, z)$$

Kaava 3. ReLU-funktio

Virhefunktio puolestaan mittaa verkon ennusteen ja todellisten arvojen välistä eroa ja arvioi, kuinka hyvin verkko suoriutuu tehtävästään. Virhefunktio on keskeinen osa ohjattua oppimista, sillä sen avulla optimoidaan verkon painokertoimia ja pyritään pienentämään virhettä. Virhefunktion on oltava derivoituva, jotta verkko voi hyödyntää gradienttipohjaisia optimointialgoritmeja oppimisprosessissa. Virhefunktion valinta riippuu usein tehtävän luonteesta ja datan tyypistä, sillä eri virhefunktiot soveltuvat parhaiten tietynlaisiin ongelmiin (Haasiomäki, 2019).

Yksi esimerkki yleisestä virhefunktiosta on keskineliövirhe (MSE), jota hyödynnetään etenkin regressiotehtävissä. Keskineliövirhe mittaa ennusteen ja todellisen arvon eroa. Keskineliövirheen kaava (kaava 4) on:

$$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y}_i)^2$$

Kaava 4. Keskineliövirhe

Missä N on datapisteiden määrä, y_i on todellinen arvo ja $\bar{y}_i$ on verkon ennuste.

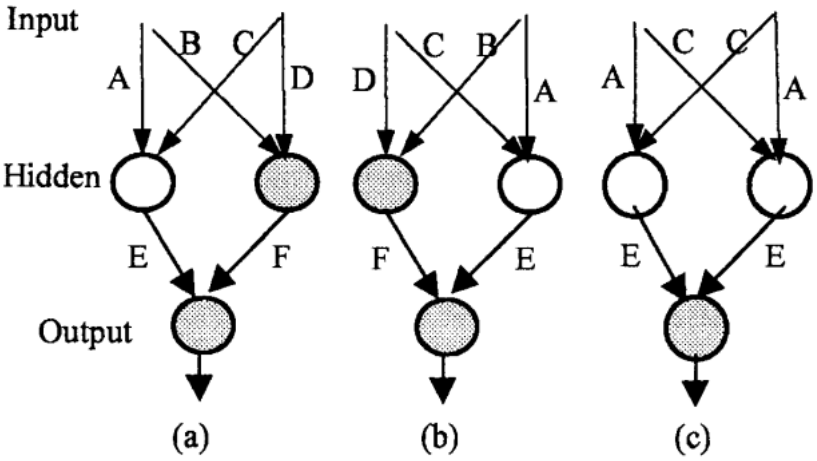
2.4 Neuroverkon permutaatio-ongelma

Neuroverkon permutaatio-ongelma on yksi keskeisimmistä haasteista neuroverkkojen koulutuksessa. Ongelma muodostuu siitä, että saman toiminnallisen tehtävän toteuttamiseksi voi olla useita rakenteellisesti erilaisia neuroverkkoja. Tämä tarkoittaa sitä, että jokaisen rakenteellisesti erilaisen verkon ulostulo voi olla sama. Täten ratkaisujen osalta saatetaan päätyä useisiin yhtä hyviin globaaleihin optimiratkaisuihin. Tämä hidastaa optimointiprosessia ja vähentää algoritmin luotettavuutta. Ongelma

korostuu etenkin geneettisten algoritmien yhteydessä, koska esimerkiksi risteytysoperaattori ei välttämättä toimi odotetusti (Zankinski, 2017).

Spronck (1996) korostaa, että permutaatio ongelma ilmenee pääasiassa kahdella eri tavalla. Ensimmäinen näistä on piilotettujen neuronien redundanssi, jossa yksittäisen neuronin painojen ja bias-arvojen merkit voidaan kääntää ilman vaikutusta verkon lopulliseen toimintaan. Toinen ilmenemismuoto on piilokerroksen redundanssi, jossa kahden piilokerroksen neuronin paikkaa vaihtamalla saadaan erilainen verkon rakenne ulostulon pysyessä samana. Näiden kahden ilmiön yhdistelmänä verkolle, jossa on n piilokerroksen neuronia, voi olla $2^n n!$ rakenteellisesti erilaista, mutta toiminallisesti identtistä esitystä.

Verkon permutaatioiden määrä riippuu piilokerrosten määrästä sekä piilokerroksen neuronien määrästä. Neuroverkon permutaatioiden lukumäärä yhden piilokerroksen verkossa saadaan neuronien määrän kertomana $k!$, jossa k on neuronien määrä piilokerroksella. Useamman piilokerroksen verkossa permutaatioiden lukumäärä saadaan neuronien määrän kertoman tulolla $k_1! \times k_2! \times \dots \times k_n!$, missä n on piilokerrosten lukumäärä. Esimerkiksi kahden piilokerroksen ja 5 piilokerroksen neuronin verkossa on $5! \times 5! = 14400$ permutaatiota. Toisena esimerkkinä 15 piilokerroksen ja 5 neuronin verkossa on $5! \times 5! \dots 5! = 120^{15}$ permutaatiota. Permutaatioiden määrä siis kasvaa valtavasti aina, kun piilokerroksia tulee lisää. Tämän tutkimuksen tapauksessa käytetään yhtä piilokerrosta viidellä neuronilla, joten erilaisia permutaatioita on yhteensä 14400. Tämän takia paikalliseen optimiratkaisuun jumiin jääminen on haitallista ja algoritmin olisikin hyvä pyrkiä yhteen globaaliin optimiratkaisuun. Tärkeää on siis saada algoritmi kiinnostumaan yhdestä tietystä globaalista optimiratkaisusta kaikkien ratkaisujen sijaan. Alla olevassa kuvassa 2 on havainnollistettu neuroverkon permutaatio-ongelma. Lähtöyksilöt (a) ja (b) suorittavat saman tehtävän ja niillä on sama kelpoisuusarvo (engl. fitness value). Tämä yhdistelmä tuottaa jälkeläisen (c), jossa on kaksi piilokerroksen neuronia ja joka suorittaa lähes saman tehtävän.



Kuva 2. Neuroverkon permutaatio-ongelma (Lähde: Researchgate, Yang, 2001)

Spronck (1996) ehdottaa ratkaisuksi permutaatio-ongelmaan seuraavia alla olevia vaihtoehtoja:

- Jätetään ongelma huomiotta
- Rajoitetaan risteytyksen käyttöä
- Järjestellään piilokerroksen neuronit uudestaan ennen risteytystä
- Geneettinen kaskadioppiminen (Genetic cascade learning)
- Uuden mekanismin käyttöönotto, joka pakottaa tietynlaisen toteutuksen mallinnukselle

3 Opetusalgoritmit

Tässä luvussa perehdytään opetusalgoritmeihin ja niiden soveltamiseen differentiaalievoluutiossa. Ensin käsitellään perinteisiä algoritmeja kuten yksitavoiteoptimointia ja tämän jälkeen syvennytään itse differentiaalievoluutioon ja sen käyttöön neuroverkkojen koulutuksessa. Lopuksi katsotaan, mitä on monitavoiteoptimointi ja mitä uutta se tuo perinteisiin menetelmiin verrattuna.

Opetusalgoritmit ovat keskeinen osa neuroverkkojen toimintaa, sillä niiden avulla verkko oppii optimoimaan painokertoimiaan annetun datan perusteella. Ne määrittävät, kuinka neuroverkko pystyy oppimaan monimutkaisia suhteita ja tekemään ennusteita. Ilman tehokkaita opetusalgoritmeja verkot eivät voisi oppia tai yleistää tietoa uusiin tapauksiin, mikä rajoittaisi niiden soveltuvuutta monimutkaisiin ongelmiin.

3.1 Yksitavoiteoptimointi

Yksitavoiteoptimointi on prosessi, jossa pyritään optimoimaan yksi selkeästi määritelty tavoitefunktio, joka voi olla esimerkiksi suorituskyvyn maksimointi tai virheen pienentäminen. Savic (2002) kuvailee yksitavoiteoptimointia suoraviivaisena menetelmänä, joka sopii erityisesti tilanteisiin, joissa ongelman ratkaisu voidaan kuvata yksittäisellä mittarilla. Tämä tekee siitä erittäin käyttökelpoisen teknisissä sovelluksissa, joissa päätöksenteon kriteerit ovat selkeästi määriteltävissä.

Neuroverkon koulutuksessa yksitavoiteoptimointi on usein ensimmäisenä tarkasteltava opetusalgoritmi. Yleensä optimoitava tavoite on virhefunktio, joka mittaa verkon ennusteen ja todellisen arvon välistä eroa. Yksi esimerkki yleisesti käytetyistä virhefunktioista on keskineliövirhe, jota käytetään muun muassa regressiotehtävissä. Keskineliövirhe on esitelty edellisessä luvussa 2.3. Kyseisen funktion avulla painokertoimia säädetään niin, että virhe minimoituu, mikä johtaa parempaan mallin suorituskykyyn.

Yksitavoiteoptimoinnin vahvuuksia ovat sen yksinkertaisuus ja tehokkuus. Optimoimalla vain yhtä tavoitetta voidaan välttää tarpeettoman monimutkaisia päätöksentekoprosesseja ja keskittyä ongelman ytimeen. Tämä on kätevää tilanteissa, joissa ratkaisu voidaan saavuttaa keskittymällä yhteen mittariin. Tämä yksi mittari voi olla esimerkiksi ennustetarkkuus. Savic (2002) painottaa, että tämä lähestymistapa sopii hyvin sovelluksiin, joissa ongelma voidaan yksinkertaistaa ja mallintaa selkeän tavoitefunktion avulla.

Yksitavoiteoptimoinnilla on kuitenkin myös rajoituksia. Monimutkaisemmissa ongelmissa, joissa useat tavoitteet kilpailevat keskenään, yksitavoiteoptimointi voi olla riittämätön. Esimerkiksi neuroverkoissa voi ilmetä ristiriitaisia tavoitteita, kuten tarkkuuden maksimointi ja laskennallisen tehokkuuden säilyttäminen, joita yksitavoiteoptimointi ei pysty käsittelemään samanaikaisesti. Savicin (2002) mukaan tällaisissa tapauksissa yksitavoiteoptimointi voi johtaa ratkaisuihin, jotka suosivat yhtä tavoitetta muiden kustannuksella. Neuroverkoissa yksitavoiteoptimointi voi kohdata myös muita haasteita. Yksi näistä on paikallisiin optimaalisuuspisteisiin juuttuminen, mikä voi estää verkkoa saavuttamasta globaalia optimaalisuutta. Tämä korostuu erityisesti syvissä verkoissa, joissa hakutila on laaja ja monimutkainen. Lisäksi yksitavoiteoptimointi ei pysty ratkaisemaan neuronien permutaatio-ongelmaa, joka syntyy monikerroksisissa perseptroniverkoissa ja se voi hidastaa oppimisprosessia.

Savic (2002) toteaa, että yksitavoiteoptimointi on perusta monimutkaisemmille optimointimenetelmille, jotka kykenevät käsittelemään useita samanaikaisia tavoitteita. Näin ollen yksitavoiteoptimointi toimii perustavanlaatuisena lähtökohtana neuroverkkojen optimoinnissa. Yksitavoiteoptimointi mahdollistaa selkeän ja tehokkaan tavan käsitellä ongelmia, joissa päätavoite on yksiselitteinen ja selvästi määritelty. Tämä ei kuitenkaan aina riitä, koska optimointiongelmat eivät aina ole yksiselitteisiä. Hyvä esimerkki monimutkaisemmasta optimointimenetelmästä on monitavoiteoptimointi, joka pyrkii ratkaisemaan yksitavoiteoptimointiin liittyviä ongelmia erilaisella lähestymistavalla. Monitavoiteoptimointiin keskitytään tarkemmin luvussa 3.3.

3.2 Differentiaalievoluutio

Differentiaalievoluutio (DE, engl. differential evolution) on populaatiopohjainen optimointimenetelmä, joka on suunniteltu erityisesti epälineaaristen, monimutkaisten ja ei-differentioituvien funktioiden optimointiin. Differentiaalievoluutioalgoritmi kuuluu evoluutiolaskennan menetelmiin, joissa hyödynnetään biologisia periaatteita, jotka ovat mutaatio, risteytys ja yksilönvalinta. Sen tavoitteena on löytää optimaalisia ratkaisuja iteratiivisesti parantamalla populaatiossa olevien yksilöiden laatua sukupolvesta toiseen. Differentiaalievoluution vahvuus on sen kyky käsitellä ongelmia, joissa perinteiset gradienttipohjaiset menetelmät eivät ole käyttökelpoisia, koska tavoitefunktio ei ole derivoituva tai sen hakutila on monimutkainen (Heiskanen, 2015).

Differentiaalievoluution toiminta alkaa satunnaisesti luodulla populaatiolla, joka koostuu useista ratkaisuvektoreista. Näitä vektoreita muokataan algoritmin kolmessa päävaiheessa: mutaatio, risteytys ja valinta. Mutaatiovaiheessa luodaan uusi mutanttivektori v_i, joka lasketaan satunnaisesti valittujen yksilöiden avulla. Mutanttivektori muodostetaan alla annetulla kaavalla (kaava 5):

$$v_i = x_{r1} + F \times (x_{r2} - x_{r3})$$

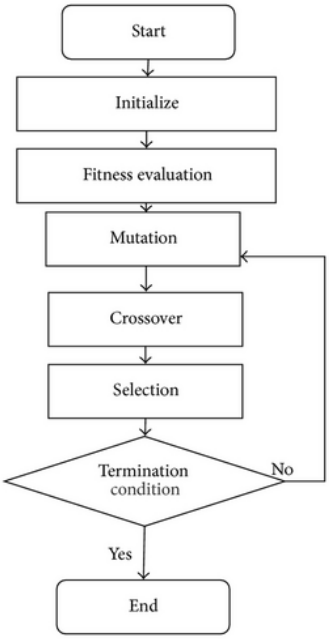
Kaava 5. DE:n mutanttivektori

missä F on mutaatiovakio ja x_{r1}, x_{r2} sekä x_{r3} ovat satunnaisesti valittuja yksilöitä populaatiosta. Tämä vaihe mahdollistaa hakutilan laajan tutkimisen ja uusien potentiaalisten ratkaisujen luomisen (Heiskanen, 2015).

Risteytysvaiheessa mutanttivektori yhdistetään alkuperäiseen yksilöön x_i, joka näin luo uuden ehdokasratkaisun u_i. Tämä tapahtuu todennäköisyyssuhteeseen CR (Crossover Rate) perustuen, minkä avulla määritetään mutanttivektorin vaikutus lopulliseen ehdokasratkaisuun. Lopuksi valintavaiheessa uusi ehdokasratkaisu u_i korvaa

alkuperäisen yksilön x_i , jos se tarjoaa paremman tavoitefunktion arvon. Näin algoritmi varmistaa, että seuraavaan sukupolveen siirtyvät vain parhaat yksilöt (Heiskanen, 2015).

Heiskanen (2015) toteaa differentiaalievoluution suoriutumisen riippuvan vahvasti sen kontrolliparametreista, kuten populaation koosta (NP, Number of Population), mutaatiovakiosta (F) ja risteytyssuhteesta (CR). Näiden parametrien optimointi on kriittistä, sillä esimerkiksi liian pieni populaatio voi johtaa huonoon tutkimiskykyyn, kun taas liian suuri populaatio kasvattaa laskennallista kuormitusta. Lisäksi mutaatiovakion ja risteytyssuhteen arvojen tasapaino vaikuttaa algoritmin kykyyn löytää globaaleja ratkaisuja. Ei ole olemassa yhtä oikeaa kombinaatiota kontrolliparametreille, vaan parametrien kombinaatio riippuu pitkälti ratkaistavasta optimointiongelmasta. Ei ole yhtä oikeaa parametrikombinaatiota, joka ratkaisisi kaikki ongelmat tehokkaasti. Populaation koko NP määrittää yksilöiden määrän populaatiossa ja vaikuttaa algoritmin kykyyn tutkia hakutilaa. Liian pieni populaatio voi jäädä paikallisiin optimeihin, kun taas suuri populaatio parantaa luotettavuutta mutta lisää laskentakustannuksia. Mutaatiovakio F säätelee askelpituutta, jolla hakutilaa tutkitaan. Suurempi F-arvo mahdollistaa laajemman tutkimisen, mutta liian suuri arvo voi hidastaa algoritmia. Tyypillisesti F-arvot vaihtelevat välillä 0,4–0,95. Risteytysvakio CR puolestaan määrittää todennäköisyyden, jolla mutanttivektorin komponentit korvaavat alkuperäisen yksilön komponentit. Pieni CR-arvo suosii alkuperäisen yksilön säilyttämistä, kun taas suuri CR-arvo lisää mutanttivektorin vaikutusta. CR-arvon vaihteluväli on $0 \leq CR \leq 1$ eli se vaihtelee 0 ja 1 välillä, koska kyseessä on todennäköisyys. Alla oleva kuva (kuva 3) havainnollistaa differentiaalievoluutioalgoritmin toimintaa yleisluontoisesti.



Kuva 3. Differentiaalievoluutioalgoritmi (Lähde: WILEY)

3.3 DE neuroverkon koulutuksessa

Ilosen, Kamaraisen ja Lampisen (2003) mukaan differentiaalievoluutio on osoittautunut hyödylliseksi MLP-verkon koulutuksessa. MLP-verkkojen koulutusta pidetään haastavana optimointiongelmana, koska se sisältää useita paikallisia optimipisteitä. Perinteiset gradienttipohjaiset paikalliset optimointimenetelmät eivät usein kykene saavuttamaan globaaleja ratkaisuja, vaan ne saattavat ajautua paikalliseen optimiratkaisuun. Differentiaalievoluution populaatiopohjainen lähestymistapa mahdollistaa hakutilan laajan tutkimisen ja se voi näin tarjota mahdollisia ratkaisuja tähän ongelmaan.

Differentiaalievoluution käyttö neuroverkkojen koulutuksessa tuo merkittäviä etuja mutta, sen suorituskyvyn arviointi ei ole yksinkertaista. Differentiaalievoluution lisätutkimukselle on tarvetta differentiaalievoluution stokastisen luonteen vuoksi. Koulutusalgoritmien vertailussa on otettava huomioon tiettyjä tekijöitä, kuten tehokkuus, laskennallinen taloudellisuus ja algoritmin kyky tuottaa luotettavia tuloksia. Esimerkiksi koulutuksen kestoa voidaan mitata laskenta-ajalla, mutta differentiaalievoluution iteratiivinen luonne voi tehdä vertailusta haastavaa. Tästä

huolimatta differentiaalievoluutio voi auttaa gradienttipohjaisten menetelmien tulosten validoinnissa ja mahdollisesti varmistaa, etteivät ne ole pelkkiä paikallisia optimeja (Ilonen, Kamarainen & Lampinen, 2003).

Myös Camargo, Tissot ja Pozo (2012) toteavat, että differentiaalievoluutio tarjoaa etuja globaalissa hakuprosessissa. Sen suorituskyky verrattuna takaisinkytkentämenetelmään on kuitenkin hitaampi. Tämä johtuu siitä, että differentiaalievoluution tarvitsee luoda useampia neuroverkon instansseja ja käyttää enemmän laskentatehoa suorittaakseen lineaarisia yhdistelmiä. Takaisinkytkentäalgoritmi saavuttaa parhaan tuloksen nopeammin, mutta differentiaalievoluution etu on sen kyvyssä tutkia laajempia hakutiloja. Vaikka takaisinkytkentäalgoritmi osoittautui nopeammaksi ja tehokkaammaksi luottamusmääriä koskevissa testeissä, molemmat algoritmit tuottivat samankaltaisia tuloksia verrattaessa niiden koulutus- ja validointituloksia. Differentiaalievoluution käyttöä neuroverkon koulutuksessa tulee siis tukia tarkemmin sen lupaavan potentiaalin vuoksi.

3.4 Monitavoiteoptimointi

Perkola (2014) kuvaa monitavoiteoptimointia osuvasti käyttämällä esimerkkiä autojen valmistuksesta. Autojen valmistuksessa monet prosessit pyrkivät ristiriitaisiin tavoitteisiin yhtä aikaa. Autojen valmistuksessa on hyvä huomioida auton kestävyys, käytön helppous sekä suorituskyky ja polttoaineen riittävyys. Tähän tilanteeseen ei ole yhtä oikeaa vastausta, vaan täydellistä autoa luodessa päädytään ratkaisuun, joka on niin kallis, ettei kukaan kuluttaja halua sitä ostaa. Esimerkiksi tällaisessa tilanteessa monitavoiteoptimointi on hyödyllinen apuväline, sillä se auttaa kehittämään sellaisen auton, joka huomioi kaikki valmistuskriteerit eikä autosta tule liian kallista.

Monitavoiteoptimointi on menetelmä monimutkaisten ongelmien ratkaisemiseen, kun optimoitavana on samanaikaisesti useita mahdollisesti ristiriitaisia tavoitteita. Perinteinen lähestymistapa tällaisiin ongelmiin on aiemmin tässä luvussa esitelty yksitavoiteoptimointi, joka esimerkiksi käyttää ainoastaan keskineliövirhettä. Vaikka

tämä lähestymistapa on yksinkertainen ja laajalti käytetty, se voi jättää huomiotta ratkaisujen monimuotoisuuden ja rajoittaa algoritmin kykyä löytää tasapainoisia kompromissiratkaisuja, jotka ovat usein tärkeitä monitavoitteisissa tehtävissä. Juuri tähän ongelmaan monitavoiteoptimointi pyrkii löytämään ratkaisua. (Vuontisjärvi, 2023).

Vuontisjärven (2023) mukaan monitavoiteoptimoinnin pyrkimyksenä on löytää niin sanottuja Pareto-optimaalisia ratkaisuja, joissa yhden tavoitteen parantaminen ei ole mahdollista ilman, että jokin toinen tavoite heikkenee. Tämä luo laajan ratkaisujoukon tarkastelun, josta päätöksentekijä voi valita sen, mikä parhaiten vastaa asetettuja vaatimuksia. Monitavoiteoptimointi on hyödyllinen tilanteissa, joissa tavoitteet ovat monimutkaisesti sidoksissa toisiinsa, jossa voi olla tarpeen esimerkiksi optimoida virhefunktio ja verkon rakenteelliset ominaisuudet samanaikaisesti. Tästä hyvä esimerkki on neuroverkkojen koulutus.

Pareto-rintama on tavoiteavaruudessa oleva ”hypersurface”, joka määrittyy voimakkaasti tehokkaiden ratkaisujen joukosta. Tällöin Pareto-rintama koostuu ratkaisuvaihtoehdoista, jotka edustavat niin sanottuja "parhaita kompromisseja", sillä kukaan näistä ratkaisuista ei ole dominoitavissa. Tämä tarkoittaa, ettei ole olemassa toista ratkaisua, joka parantaisi yhtä tavoitetta ilman, että jokin toinen tavoite heikkenee. Pareto-rintama muodostuu joukosta ratkaisuja, joissa eri tavoitteiden välinen tasapaino on optimoitu sekä ne ovat keskenään vertailtavissa sen perusteella, miten hyvin ne vastaavat asetettuihin vaatimuksiin. Pareto-rintaman tarjoamaa joukkoa ratkaisuja voidaan hyödyntää parhaan mahdollisen ratkaisun valinnassa huomioiden kuinka yhden tavoitteen parantaminen vaikuttaa muiden tavoitteiden arvoihin. Tämä antaa avaimet optimaalisen ratkaisun valintaan, joka parhaiten heijastaa kyseisen ongelman kontekstia (Price, Storn ja Lampinen, 2005).

Monitavoiteoptimoinnissa pyritään optimoimaan samanaikaisesti kahta tai useampaa tavoitefunktiota. Monitavoiteoptimointi on matemaattisesti määritelty tavoitefunktioiden vektorina alla olevan ilmaisun kautta (kaava 6).

$$f(x) = [f_1(x), f_2(x), \dots, f_m(x)]$$

Kaava 6. Monitavoiteoptimoinnin määrittely

Missä $x \in R^n$on päätösmuuttujavektori, $f_i(x)$ on i-s tavoitefunktio ja m on tavoitteiden määrä. Ratkaisu on Pareto-optimaalinen, mikäli seuraava alla oleva ehto täyttyy (kaava 7).

$$f_i(x) \leq f_i(x*) \text{ kaikille } i \in [1,2, \dots, m]$$

Kaava 7. Pareto-optimaalisuuden määrittely

Ja indeksillä j (kaava 8), jolle

$$f_j(x) < f_j(x*)$$

Kaava 8. Pareto-optimaalisuuden ehto

Joissa $x*$ on Pareto-optimaalinen ratkaisu ja x on muu ratkaisu.

4 Tutkimusmenetelmät

Tämä luku kuvaa tässä tutkimuksessa käytettävät tutkimusmenetelmät. Ensin perehdytään aiemmin tehtyyn tutkimukseen esimerkkien kautta ja tämän jälkeen katsotaan, miten tämä tutkimus toteutetaan. 4.2-luvussa kuvataan tutkimuksen toteutuksen lisäksi itse algoritmin toteutus ja tämän jälkeen suoritettavat kokeet. Lisäksi määritellään tutkimuksen suorituskykymittarit algoritmille. Näitä mittareita käytetään myöhemmissä vaiheissa algoritmin suorituskyvyn arviointiin. Lopuksi määritellään vielä käytettävät tavoitefunktiot.

4.1 Aiempi tutkimus

Differentiaalievoluution käytöstä yksitavoiteoptimointimenetelmillä on tehty tutkimusta jonkin verran. Nämä tulokset ovat osoittautuneet kohtuullisen lupaaviksi, joten lisätutkimukselle on tarvetta. Monitavoiteoptimoinnin käytöstä differentiaalievoluutioalgoritmissa ei ole tehty tutkimusta ollenkaan, joten tämän tutkimuksen aihe on tärkeä. Tämän tutkimuksen lähtökohtiin kuului juuri tämä olemassa olevan tutkimuksen puute. Tässä luvussa esitellään aiempaa tutkimusta ja niiden tuloksia lyhyesti.

Ilonen, Kamarainen ja Lampinen (2003) tutkivat differentiaalievoluutioalgoritmin käyttöä monikerroksisten perseptroniverkkojen koulutuksessa. Koulutuksessa käytettiin differentiaalievoluutiota, joka oli toteutettu MATLAB-koodina. Testit ajettiin neljällä eri optimointiongelmalla, joissa kussakin oli oma neuroverkon rakenne. Optimointiongelmat sisälsivät funktion approksimaatioita ja kuvion tunnistusta. Tutkimuksen mukaan differentiaalievoluutioalgoritmi osoittautui tehokkaaksi MLP-verkkojen koulutuksessa. Verkon painot ja virheet otettiin huomioon rangaistusjäsennystä (MSEREG) käyttäen. Kokeissa differentiaalievoluution suorituskyky oli verrattavissa gradienttipohjaisiin menetelmiin. Differentiaalievoluution tärkeimmät edut olivat kuitenkin muualla kuin pelkässä koulutuksen suorituskyvyssä. Ensinnäkin differentiaalievoluutiolla ei ole suuria rajoituksia virhefunktioiden käytölle. Toisekseen

menetelmä ei rajoita säännöstelymenetelmien käyttöä. Kolmanneksi se tarjoaa globaaliin minimiin konvergoimisen mahdollisuuden, vaikka konvergenssiaika voi olla pitkä. Lisäksi algoritmin parametreja on helppo säätää. Tutkimus painottaa tarvetta lisätutkimukselle aiheen parissa esittämällä muutamia alla esitettyjä kysymyksiä, joita tulevissa tutkimuksissa voitaisiin käsitellä.

- Miksi jotkin paikalliset optimit ovat differentiaalievoluutiolle houkuttelevampia?
- Miksi differentiaalievoluutio löytää globaalin minimikohdan joissakin ongelmakonfiguraatioissa, kun gradienttipohjaiset menetelmät eivät löydä?
- Mikä on valitun suorituskykyfunktion vaikutus differentiaalievoluution löytämään optimiratkaisuun?

Camargo, Tissot ja Pozo (2012) tutkivat monikerroksisten perseptroniverkkojen kouluttamista käyttäen takaisinkytkentä- ja differentiaalievoluutioalgoritmeja. He vertailivat neljää lähestymistapaa: perinteistä takaisinkytkentää, differentiaalievoluutiota kiinteillä ja mukautuvilla parametreilla sekä hybridimenetelmää, jossa yhdistettiin nämä kaksi algoritmia. Tutkimuksessa käytettiin viittä erilaista tietokantaa MLP-verkkojen kouluttamiseen. Tietokannat jaettiin koulutus-, validointi- ja testijoukkoihin, ja MLP:t saivat satunnaisesti aloituspainot. Tutkimuksessa arvioitiin kunkin lähestymistavan tehokkuutta ja suorituskykyä eri tietokannoilla ja erilaisilla populaatiokoilla. Tutkimuksen tulokset osoittivat, että vaikka takaisinkytkentä oli nopeampi, on differentiaalievoluutio parempi globaalin ratkaisun etsimisessä. Hybridimenetelmä yhdisti molempien lähestymistapojen parhaat puolet ja se mahdollisti tasapainoisen ratkaisun verkon kouluttamiseen. Tutkimus siis puoltaa differentiaalievoluution tai mahdollisesti hybridiratkaisujen lisätutkimuksia.

4.2 Tutkimuksen suunnittelu

Tutkimus toteutetaan ohjelmoimalla DE-algoritmi ja MLP-neuroverkko Python-kielellä. DE-algoritmi toteutetaan tukemaan sekä perinteistä yksitavoiteoptimointia että

monitavoiteoptimointia, jossa lisätään tavoitefunktioita ainakin yksi kappale. MLP-verkon rakenne sisältää tarvittavat piilokerrokset sekä syöttö- ja ulostulokerrokset.

Tälle tutkimukselle on suunniteltu kontrolloidut kokeet. Tämä on perusteltu lähestymistapa, koska algoritmi on stokastinen. Algoritmi ei ole deterministinen, koska se ei anna täsmälleen samoja tuloksia joka ajokerralla. Tämä johtuu useista tekijöistä, kuten alustavan populaation satunnaisuudesta, mutaatiosta ja satunnaisista valinnoista risteytyksessä sekä neuroverkon painojen uudelleen alustuksessa. Muun muassa näitten tekijöiden takia algoritmin ulostulo vaihtelee joka ajokerralla. Testit ajetaan jokaisella asetuksella vähintään kymmeniä, mahdollisesti jopa sata kertaa, jotta saadaan kattavat ja luotettavat tulokset. Tämän lisäksi katsotaan, että tulos suppenee varmasti oikein. Testeissä mitataan algoritmin tehokkuutta, tarkkuutta ja luotettavuutta. Toistojen avulla varmistetaan, että tulokset eivät vaihtele ja tunnistetaan mahdolliset stokastiset vaihtelut.

Tulosten analysointi toteutetaan sekä numeerisesti että graafisesti. Graafien avulla havainnollistetaan esimerkiksi sukupolvien välistä muutosta ajan suhteen ja algoritmin suorituskykyä eri parametrikombinaatioilla. Visualisoinnit auttavat vertailemaan yksitavoite- ja monitavoiteoptimoinnin eroja. Lisäksi on tärkeää analysoida, kuinka monta sukupolvea haluttu luokittelutarkkuus vaatii. Lopuksi analyysin pohjalta tehdään johtopäätökset monitavoiteoptimoinnin vaikutuksista ja sen mahdollisuuksista ratkaista neuronien permutaatio-ongelma. Johtopäätöksissä pohditaan myös tulevia tutkimustarpeita.

4.2.1 DE-algoritmin ja neuroverkon toteutus

Aluksi toteutetaan neuroverkko ja vain yksitavoiteoptimointiin kykenevä DE-algoritmi. Algoritmi toteutetaan Python-ohjelmointikielellä Microsoft Visual Studio -ohjelmointiympäristössä. Kokeet ajetaan ensin yksitavoiteoptimoinnilla, jotta saadaan vertailupohjaa monitavoiteoptimoinnille. Tästä edetään toteuttamaan

monitavoiteoptimointiin kykenevä DE-algoritmi. Algoritmien tarkempi kuvaus annetaan luvussa 5.

Neuroverkko toteutetaan myös Python-ohjelmointikielellä Microsoft Visual Studio -ohjelmointiympäristössä. Neuroverkolle käytetään aluksi vain yhtä tavoitefunktiota ja tavoitefunktioita vaihdetaan ja lisätään, kun siirrytään monitavoiteoptimointiin. Neuroverkko koostuu sisääntulokerroksesta, piilokerroksesta ja ulostulokerroksesta. Syöttökerros sisältää aluksi 4 neuronia. Piilokerroksia on aluksi vain 1 ja piilokerros sisältää 5 neuronia. Ulostulokerros sisältää 3 neuronia. Kokoonpanoa muutetaan tarpeen tullen. Neuroverkon ohjemalliseen toteutukseen perehdytään luvussa 5.

4.2.2 Datakokoelma ja kokeiden toteutus

Tutkimuksessa käytetään IRIS-datakokoelmaa, joka opetetaan differentiaalievoluution avulla luodulle neuroverkolle. Datakokoelma sisältää 150 havaintoa, jotka on jaettu tasaisesti kolmeen luokkaan. Nämä kolme luokkaa ovat kukin oma kukkalajinsa. Kukkalajit ovat Setosa, Versicolor ja Virginica. Jokainen havainto koostuu neljästä ominaisuudesta: sepalin pituus ja leveys sekä terälehden pituus ja leveys. Data on jaettu opetus- testi- ja validointijoukkoihin lähtökohtaisesti seuraavalla tavalla: opetusjoukko 75 %, validointijoukko 20 % ja testijoukko 5 %.

Kokeet suoritetaan aluksi ennalta määritellyillä DE:n parametreilla, jotka on taulukoitu alle. Kontrolliparametrit optimoidaan molemmille optimointimenetelmille tarpeen mukaan. Kontrolliparametreina toimivat populaation koko, mutaatiovakio, risteytysvakio sekä sukupolvien määrä. Populaation koko määrittää, kuinka monta yksilöä kukin algoritmin populaatio sisältää. Mutaatiovakio ilmaisee mutaation määrän eli satunnaisuuden vaikutuksen uusien yksilöiden luomisessa. Risteytysvakio määrittää, kuinka suuri osa uuden yksilön geeneistä kopioidaan vanhemmilta yksilöiltä. Sukupolvien määrä kertoo, kuinka monta sukupolvea yksittäinen ajokerta sisältää. Arvoja vaihdellaan tarpeen tullen. Risteytysvakion ja mutaatiovakion Testit suoritetaan ainakin kymmeniä

(10–30) kertoja. Tarpeen tullen luotettavien tulosten saamiseksi testit saatetaan suorittaa jopa sata kertaa.

Taulukko 1. DE:n parametrit

Parametri	Arvo
Populaation koko	10
Mutaatiovakio	0,7
Risteytysvakio	0,9
Sukupolvien määrä	100

4.2.3 Suorituskykymittarit

Tutkimuksessa käytetään algoritmin arviointiin kolmea suorituskykymittaria, jotka ovat tehokkuus, tarkkuus ja luotettavuus. Tämä alaluku käsittelee niitä ja selventää tarkemmin mitä ne mittaavat.

Tehokkuus mittaa algoritmin kykyä saavuttaa asetetut tavoitteet mahdollisimman vähällä resurssien käytöllä. Tässä tutkimuksessa tehokkuutta arvioidaan koulutuksen aikana tarvittavien iteraatioiden ja kokonaislaskenta-ajan perusteella. Lyhyempi laskenta-aika ja nopeampi konvergenssi viittaavat tehokkaampaan optimointiin. Differentiaalievoluution iteratiivinen luonne tulee todennäköisesti vaikuttamaan suoritusaikaan verrattuna perinteisiin gradienttipohjaisiin menetelmiin.

Tarkkuus viittaa siihen, kuinka hyvin koulutettu neuroverkko pystyy suorittamaan sille annetut luokittelutehtävät. Sitä mitataan yleensä testijoukon tarkkuusprosentilla eli sillä, kuinka monta oikein luokiteltua havaintoa verkko tuottaa kaikista testihavainnoista. Tässä tutkimuksessa tarkkuutta arvioidaan eri optimointimenetelmillä koulutettujen verkkojen välillä ja analysoidaan, kuinka virhefunktio pienenee koulutuksen aikana.

Luotettavuus kuvaa algoritmin kykyä tuottaa johdonmukaisia tuloksia useissa toistetuissa kokeissa. Sitä arvioidaan mittaamalla tulosten varianssia eri ajokertojen välillä. Alhainen varianssi viittaa siihen, että algoritmi toimii vakaasti, kun taas korkea varianssi voi viitata epäluotettavaan optimointiin. Luotettavuus on erityisen tärkeä, kun arvioidaan stokastisten algoritmien suorituskykyä.

4.3 Tavoitefunktiot

Tutkimuksessa käytetään kolmea eri funktiota. Yksi- ja monitavoiteoptimoinneissa käytetään eri tavoitefunktioita, jotka on taulukoitu (taulukko 2) alla. Nämä ovat alustavia tavoitefunktioita ja ne voivat vaihtua tutkimuksen edetessä tarpeen mukaan.

Taulukko 2. Tavoitefunktiot

Menetelmä	Funktio	Kaava
Yksitavoiteoptimointi	Keskineliövirhe	$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y}_i)^2$
Monitavoiteoptimointi	Sigmoid, Softmax	Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$ Softmax: $y_k = \frac{\exp(z_k)}{\sum_{j=1}^{m}\exp(z_j)}$

5 Tutkimuksen toteutus

Tässä luvussa kuvataan tutkimuksen toteutus yksityiskohtaisesti. Ensin tutustutaan tutkimuksen tutkimusympäristöön alaluvussa 5.1, jossa kuvataan käytetty laitteisto ja käytetty ohjelmointiympäristö. Tästä siirrytään ohjelmallisen toteutuksen läpikäymiseen luvussa 5.2, jossa kuvataan käytetty ohjelmarakenne ja toteutustavat. Alaluvussa 5.3 käydään läpi, miten neuroverkon koulutus sujuu ohjelmallisesti. Luvun loppupäässä katsotaan, miten parametrien optimointi toteutettiin ja lopuksi luodaan vielä katsaus ohjelman toimintaan yleisesti sekä tuodaan esiin tärkeimmät havainnot toteutuksessa. Todettakoon vielä, että alkuperäinen suunnitelma oli käyttää keskineliövirhettä yksitavoiteoptimoinnin tavoitefunktiona. Tämä kuitenkin vaihtui käytännön syistä neuroverkon luokittelutarkkuuteen tutkimuksen toteutuksen yhteydessä. Monitavoiteoptimoinnissa tämä luokittelutarkkuus korvaa Sigmoid-funktion. Monitavoiteoptimoinnissa käytetään siis luokittelutarkkuutta ja Softmax-funktiota. Lisäksi monitavoiteoptimoinnin kokeet suoritetaan myös luokittelutarkkuudella sekä minimoivalla hakua ohjaavalla funktiolla f_2 laajan vertailupohjan saavuttamiseksi.

5.1 Tutkimusympäristö

Tutkimus toteutettiin tietokonetta käyttäen Windows 11 -ympäristössä. Tutkimuksessa käytettiin Microsoftin Visual Studio Codea toteutuksen luomiseen ohjelmoimalla. Ohjelma ajettiin ja kokeet suoritettiin myös Visual Studio Coden sisällä. Tutkimuksen toteutuksen luomiseen käytettiin Python-ohjelmointikieltä versiolla 3.10.11. Ohjelmallinen toteutus on kuvattu tarkemmin luvussa 5.2. Tuloksien tallentamisessa käytettiin Microsoftin Excel-ohjelmaa, koska sen avulla datan käsittely on helpompaa ja tarvittavat graafit saadaan piirrettyä ohjelman sisäisesti. Tutkimuksessa käytetyn tietokoneen laitteisto on listattu alla.

- Prosessori: Intel i9-9900k, 8 ydintä, 16 säiettä
- RAM: 32 GB DDR4
- Kovalevy: 1 TB M.2 SSD

- GPU: NVIDIA RTX 2070
- Emolevy: Asus ROG Z-390

5.2 Ohjelmallinen toteutus

Tässä alaluvussa kuvataan tutkimuksen ohjelmallinen toteutus Python-ohjelmointikielellä. Ohjelma toteutettiin säikeistämättömänä eli se suorittaa koodin yksi rivi kerrallaan edeten. Kehittäminen aloitettiin yhdestä tiedostosta, mutta kehityksen edetessä todettiin, että koodin jakaminen kolmeen tiedostoon on järkevämpää koodin määrän takia. Näin ollen toteutus jaettiin kolmeen Python-tiedostoon: main.py, mlp.py ja differential_evolution.py. Näistä main.py sisälsi pääohjelman, jolla hallittiin ohjelman ajoa sekä parametreja. Tiedosto mlp.py sisälsi neuroverkon toteutuksen. Tiedosto differential_evolution.py sisälsi differentiaalievoluutioalgoritmin toteutuksen. Toteutukset on kuvattu tarkemmin tämän alaluvun alaluvuissa 5.2.1, 5.2.2 ja 5.2.3. Ohjelman toteutukset yksi- ja monitavoiteoptimoinnille on liitetty tähän työhön liitteiksi 1 ja 2. Minimoivan funktion f_2 määrittely löytyy liitteestä 3.

5.2.1 Pääohjelman toteutus

Pääohjelma toteutettiin tiedostoon main.py. Pääohjelma yhdistää kaikki ohjelman osat yhdeksi ajettavaksi kokonaisuudeksi. Ohjelma suoritettiin ajamalla main.py tiedosto, josta käyttäjä pystyy myös säätämään neuroverkon ja differentiaalievoluution parametreja. Lisäksi pystytään määrittämään, kuinka monta kertaa optimointi suoritetaan samalla kertaa. Pääohjelmaan on siis muodostettu silmukka, joka ajaa ohjelman uudestaan niin monta kertaa, kuin käyttäjä määrittää num_rums (suoristuskertojen määrä) arvoksi.

Ohjelman toteutuksessa käytettiin useita eri Python-kirjastoja. Ucimlrepo-kirjastoa hyödynnettiin IRIS-datan hakemiseen suoraan Pythonin sisäisesti. Numpy-kirjastoa hyödynnettiin laskentaoperaatioihin ja datan käsittelyyn, kun data oli haettu Ucimlrepo:lla. Pandas-kirjastoa käytettiin tulosten käsittelyyn ja Excel-tiedostojen

hallintaan. Data jaettiin hakemisen jälkeen opetus- validointi ja testausjoukkoihin seuraavasti: opetusjoukko 75 %, validointijoukko 20 %, testausjoukko 5 %.

Pääohjelmassa määritetään käytettävät tavoitefunktiot ja palautetaan tulostusta sekä tallennusta varten niiden arvot optimoinnin päätyttyä. Neuroverkon parametrien säätö tapahtuu muuttamalla input_size, hidden_size ja output_size arvoja neuroverkon alustuksessa pääohjelmassa. Differentiaalievoluution parametreja voidaan säätää muuttamalla optimizer-muuttuja alustusarvoja. Pääohjelma laskee ajokohtaisen suoritusajan ja tulostaa jokaisen ajon tuloksen konsoliin sekä tallentaa samat tulokset Exceliin.

5.2.2 Neuroverkon toteutus

Monikerroksinen perseptroniverkko on implementoitu mlp.py-tiedostossa. Tiedosto alustaa neuroverkon, suorittaa eteenpäinajon ja asettaa verkolle saadut painoarvot ja biasit. Pääohjelma käyttää tätä neuroverkkoa optimoinnissa eli se kutsuu ajettaessa neuroverkon funktioita optimoinnin suorittamiseksi. Neuroverkko koostuu yhdestä sisääntulokerroksesta, yhdestä piilokerroksesta ja yhdestä ulostulokerroksesta. Painot ja biasit alustetaan ohjelmassa välille [-1,1]. Ohjelma ajaa syötteet verkon läpi eteenpäin ja päivittää verkon painoja optimoinnin edetessä. Eteenpäinajo palauttaa verkon ulostulon, jota käytetään optimoinnin jatkamiseksi.

5.2.3 Differentiaalievoluutioalgoritmin toteutus

Differentiaalievoluutioalgoritmi on implementoitu tiedostossa differential_evolution.py. Tiedostossa suoritetaan ensin differentiaalievoluution alustus. Tämän jälkeen siirrytään itse optimoinnin suoritukseen määrittelemällä populaatio ja optimointiin liittyvät muuttujat. Sitten suoritetaan itse sukupolvikohtainen optimointiprosessi, joka toistetaan määriteltyjen parametrien mukaan. Lopuksi tulostetaan optimointien tuloksena saatu ohjelman lopputulos eli tulostetaan ohjelman löytämä paras arvo tavoitefunktiolle. Toteutuksen lopussa määritellään Excel-tiedostoon DataFrame, joka määrittää, miten

tiedot tallennetaan Exceliin. Täältä viedään tiedot pääohjelmaan, joka suorittaa suurimman osan tuloksille tehtävistä operaatioista.

5.2.4 Yksi- ja monitavoiteoptimoinnin toteutus

Yksitavoiteoptimoinnin ja monitavoiteoptimoinnin toteutukset eroavat jonkin verran toisistaan. Toteutukset tehtiin erillisinä molemmille optimointitavoille eli yhteensä toteutettiin kuusi kooditiedostoa. Suurimpana erona yksi- ja monitavoiteoptimointien toteutuksessa on käytettyjen tavoitefunktioiden määrä. Monitavoiteoptimointiin lisättiin luokittelutarkkuuden lisäksi Softmax-funktio. Näin ollen myös optimointiprosessia tuli päivittää, koska tarvittiin implementaatio myös toisen funktion arvon seuraamiselle ja tulostamiselle optimointiprosessissa. Myös Exceliin tallennusta piti laajentaa monitavoiteoptimoinnille, koska tavoitefunktioita oli kaksi yhden sijaan. Molemmista funktioista määritettiin paras arvo, parhaan arvon keskiarvo sekä keskihajonta ajojen välillä. Datajoukoista määritettiin niiden virheiden suuruus ja tarkkuus. Myöhemmin lisättiin myös toteutus funktiolle f_2.

5.3 Parametrien optimointi

Parametrien optimointi suoritettiin ennen varsinaisia kokeita, jotta molemmille optimointitavoille saatiin mahdollisimmat hyvät lähtöparametrit niiden optimaalisen toiminnan kannalta. Parametreille kokeiltiin kymmeniä eri kombinaatioita, joista paras valittiin kokeisiin. Parametrien optimointiin lähdettäessä käytetyt parametrit löytyvät alla olevasta taulukosta (Taulukko 3).

Taulukko 3. DE:n lähtöparametrit

Parametri	Arvo
Populaation koko	10
Mutaatiovakio	0,7
Risteytysvakio	0,9
Sukupolvien määrä	100

Parametreja muuttamalla testattiin, miten tavoitefunktion- tai funktioiden arvo muuttuu optimointiprosessin päätyttyä. Tällä tavalla löydettiin alkuarvoja paremmat parametrien arvot, joilla saatiin parannettua ajojen keskiarvotulosta ja pienennettyä tulosten keskihajontaa. Lisäksi suoritusaika oli merkittävä tekijä, koska isompi populaatio koko ja suurempi sukupolvien lukumäärä kasvattivat algoritmin suoritusaikaa huomattavasti. Alla olevissa taulukoissa (Taulukko 4 ja Taulukko 5) on esitetty saadut optimaaliset parametrit yksi- sekä monitavoiteoptimoinnille. Kokeet on suoritettu näillä parametreilla.

Taulukko 4. Yksitavoiteoptimoinnin DE parametrit

Yksitavoiteoptimointi

Parametri	Arvo
Populaation koko	50
Mutaatiovakio	0,7
Risteytysvakio	0,9
Sukupolvien määrä	200

Yksitavoiteoptimoinnin tapauksessa populaation kokoa kasvatettiin, koska huomattiin, että luokittelutarkkuuden optimiarvo paranee siten vielä entisestään. Risteytys- ja mutaatiovakioiden osalta testauksen lähtöparametrit vaikuttivat olevan paras valinta kokeiden parametreiksi, kun katsottiin algoritmin luokittelutarkkuuden keskiarvoa ja keskihajontaa. Sukupolvien määrän tapauksessa ratkaisu parani usein vielä 100 tai 150

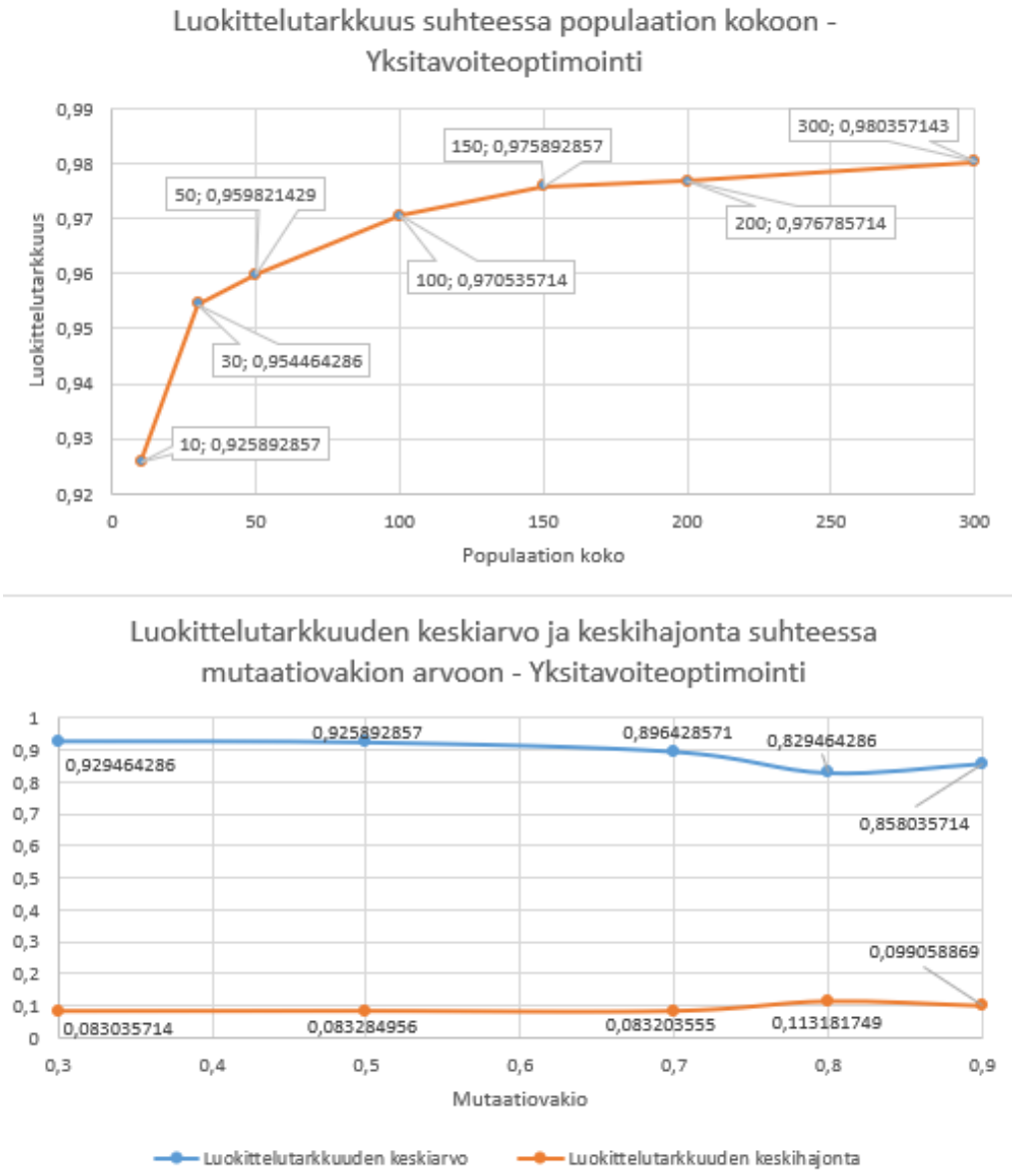
sukupolven jälkeen, mutta ei enää 200 sukupolven jälkeen, joten tämän sukupolvien määräksi 200. Nämä valinnat on perusteltu kuvilla 4 ja 5.

Taulukko 5. Monitavoiteoptimoinnin DE parametrit

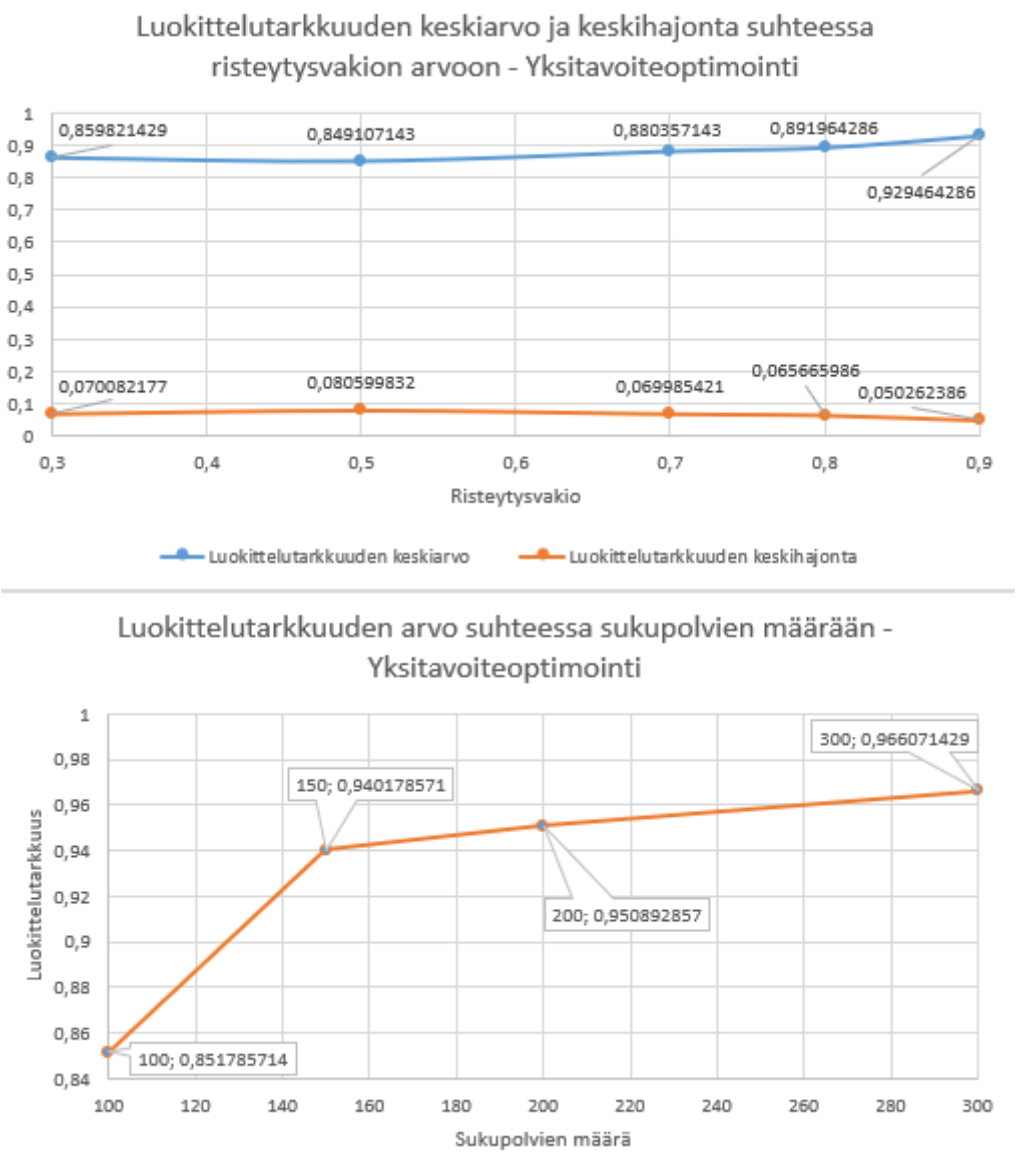
Monitavoiteoptimointi	
Parametri	**Arvo**
Populaation koko	200
Mutaatiovakio	0,8
Risteytysvakio	0,9
Sukupolvien määrä	150

Monitavoiteoptimoinnissa populaation kooksi valikoitui 200 yksilöä, koska ratkaisu vaikutti paranevan merkittävästi suuremmalla populaation koolla. Tätä isommat populaation koot olisivat enää parantaneet tuloksia marginaalisesti ja optimoinnin suoritusaika olisi kasvanut liian suureksi kokeiden käytännöllisyyden kannalta. Mutaatiovakio kasvattaminen vaikutti parantavan keskiarvoista luokittelutarkkuuden loppuarvoa hieman, joten sitä kasvatettiin arvosta 0,7 arvoon 0,8. Risteytysvakio pidettiin samana, koska sillä ei havaittu olevan suurta vaikutusta kumpaakaan suuntaan säädettäessä. Sukupolvien määräksi tuli 150, koska ratkaisu parani usein 100 sukupolven jälkeen hieman, mutta ei lähes koskaan enää 150 sukupolven jälkeen. Monitavoiteoptimoinnin parametrivalinnat on perusteltu kuvissa 6 ja 7.

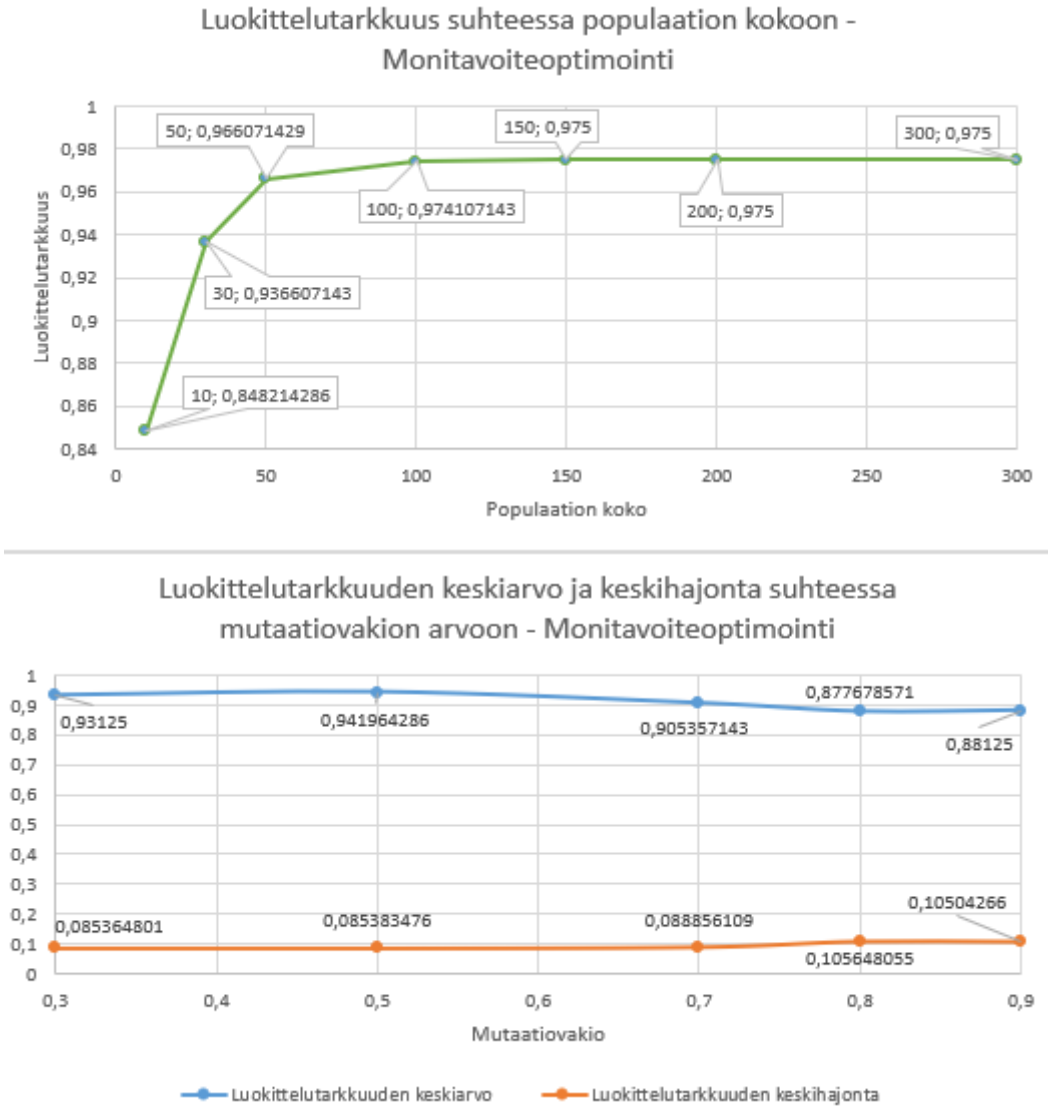
Alla olevissa kuvissa 4, 5, 6 ja 7 on perusteltu kontrolliparametrien valintaa molemmilla optimointimenetelmillä saatujen parametrien optimoinnin tulosten perusteella. Kuvaajista voidaan pääosin todeta edellä mainitut perustelut oikeiksi ja parametrien arvot riittävän optimaalisiksi. Arvoa 200 isompia populaation kokoja ei käytetä tutkimuksen suorittamisessa, vaikka ne parantaisivat optimoinnin tulosta marginaalisesti, koska ohjelman suoritusaika ja vaadittavat resurssit kasvavat liian suuriksi. Sukupolvien määrän osalta voidaan todeta sama, joskin sukupolvissa tuloksen parannus on vielä marginaalisempi.



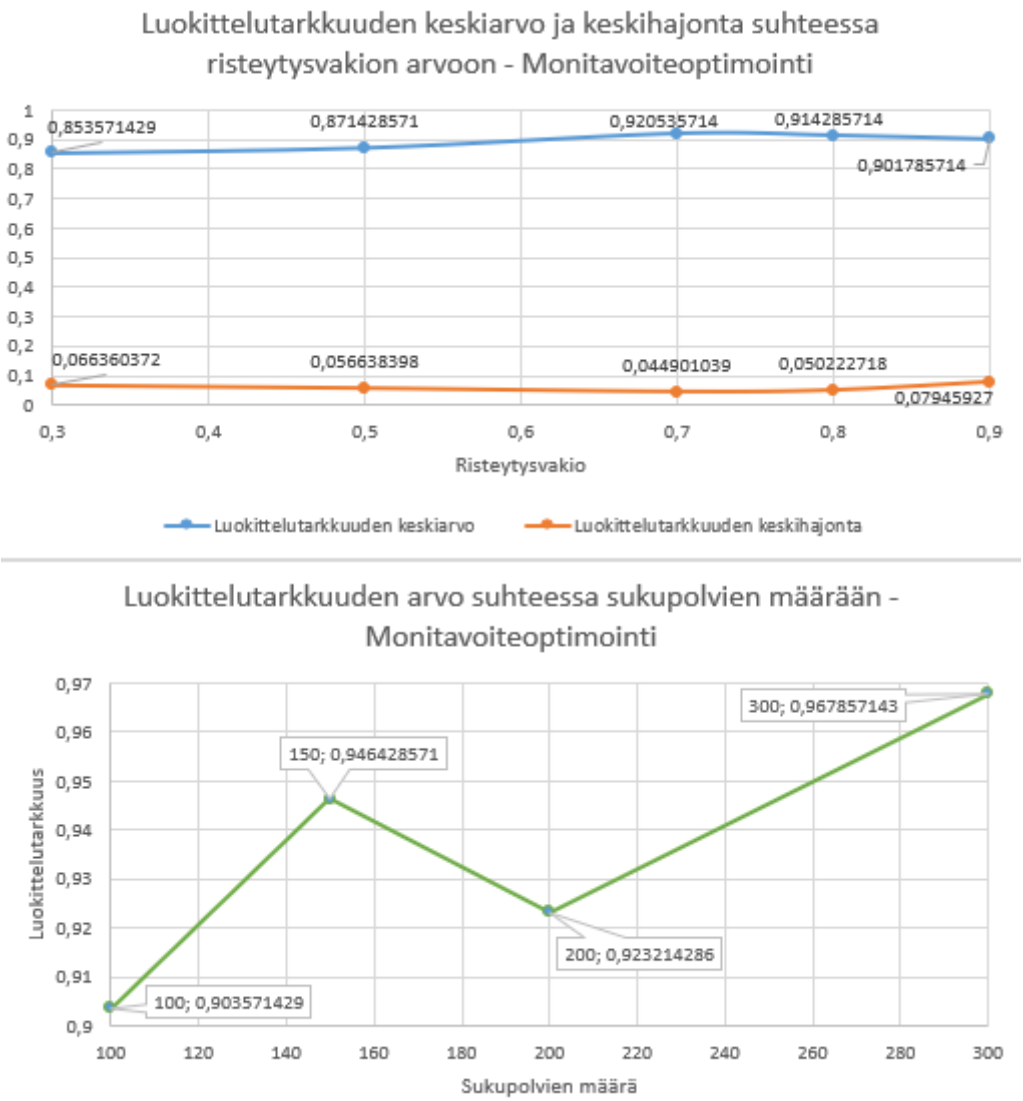
Kuva 4. Populaation koon ja mutaatiovakion optimointi - Yksitavoiteoptimointi



Kuva 5. Risteytysvakion ja sukupolvien määrän optimointi - Yksitavoiteoptimointi



Kuva 6. Populaation koon ja mutaatiovakion optimointi - Monitavoiteoptimointi



Kuva 7. Risteytysvakion ja sukupolvien määrän optimointi - Monitavoiteoptimointi

5.4 Ohjelman toiminta ja havainnot toteutuksessa

Ohjelman ajaminen aloitetaan avaamalla main.py-tiedosto, josta määritetään halutut parametrit ajotapahtumaan ja suoritettavien ajojen määrä. Suurin osa testiajoista suoritettiin 10 ajokerralla, jotta suoritusaika ei olisi liian suuri. Itse kokeet suoritettiin taas 100 ajokerralla, jotta saadaan luotettavampia tuloksia, koska kyseessä on stokastinen algoritmi. Parametrien ja ajokertojen määrän valitsemisen jälkeen ohjelman voi ajaa painamalla run-painiketta. Toimiessaan oikein ohjelma tulostaa hetken kuluttua ensimmäisen ajokerran tulokset alla olevan kuvan 8 esittämällä tavalla. Lopuksi ajo

tulostaa keskiarvon sekä keskihajonnan ja tiedon siitä, että tietojen tallentaminen onnistui. Tämä on esitetty kuvassa 9. Alla oleva on ajettu yksitavoiteoptimoinnin toteutuksella.

```
Suoritetaan ajo 1/10...
Ajo 1: Optimal Value: 0.9732142857142857, Execution Time: 1.5244 seconds

Verkon painoarvot vektorina:
[ 0.4755373  -0.01794235 -0.27414789 -1.          0.81401029 -1.
 -1.         -0.16967663 -0.70446688 -0.22359311 -1.          1.
  0.03566786  1.         -0.16169067 -0.88181002  1.          1.
  1.         -1.          0.6794846  -0.39167688  0.56808074 -0.04867151
  1.          1.         -0.41539994 -0.78449475 -1.         -0.155644
  0.92575744  1.          1.          1.         -1.         -0.09958236
  0.60722582  1.          0.56539772 -1.          1.          0.83941833
  0.89553466]
Validation Accuracy: 0.9667, Test Accuracy: 1.0000, Training Error: 0.0268
```

Kuva 8. Yksitavoiteoptimoinnin tulostus

```
Suoritetaan ajo 10/10...
Ajo 10: Optimal Value: 0.9821428571428571, Execution Time: 1.1850 seconds

Verkon painoarvot vektorina:
[ 1.          0.49315469 -1.         -0.61519443 -1.          1.
  1.          0.13202382 -0.93172038  0.98634331 -1.         -1.
  1.          1.         -1.         -1.         -1.          1.
  1.         -0.77716267  1.         -1.         -1.         -1.
 -0.77784658  1.         -1.         -1.          1.          1.
  1.         -1.         -1.         -0.29164812 -1.         -1.
  0.4         1.         -1.         -1.         -1.         -1.
  0.52857535]
Validation Accuracy: 1.0000, Test Accuracy: 1.0000, Training Error: 0.0179

Kaikki painoarvot tallennettu tiedostoon 'kaikki_painoarvot.xlsx'.
Kaikki tulokset tallennettu tiedostoon 'kaikki_tulokset.xlsx'.
Optimal Value - Keskiarvo: 0.9804, Keskihajonta: 0.0054
```

Kuva 9. Yksitavoiteoptimoinnin tulostus lopussa

Jokaiselta ajokerralta voidaan lukea ajokohtaisesti saatu luokittelutarkkuuden arvo (Optimal Value), suoritusaika (Execution Time), validointijoukon tarkkuus (Validation Accuracy), testijoukon tarkkuus (Test Accuracy) ja opetusjoukon virhe (Training Error). Ajon lopusta tiedoista voidaan lukea luokittelutarkkuuden keskiarvo ja keskihajonta.

Voidaan huomata ohjelman antavan tiedon siitä, että painoarvot on onnistuneesti tallennettu tiedostoon ”kaikki_painoarvot.xlsx” sekä kaikki tulokset on tallennettu onnistuneesti tiedostoon ”kaikki_tulokset.xlsx”. Tämä on merkki ohjelman onnistuneesta ajosta.

Alla on esitetty vastaavat ajot monitavoiteoptimoinnin toteutuksella (kuva 10 ja kuva 11).

```
Suoritetaan ajo 1/10...
Ajo 1: Accuracy: 0.9732142857142857, Softmax: 0.9141833203083765, Execution Time: 7.9886 seconds
Training Error: 0.0268, Validation Accuracy: 0.9667, Test Accuracy: 1.0000
Painoarvot ajo 1:
[ 0.86438683  0.42497925 -0.40540459 -0.095069   -1.          0.68234489
 -0.9674602  -0.97850705  1.          1.         -1.         -0.6
  1.         -1.          0.33401405 -1.         -1.          0.99732636
 -0.18601393 -1.          1.          0.79309588 -1.         -0.46633546
  0.5892013   0.83388887  0.81884177 -1.          0.6        -1.
 -0.75116407 -1.          1.          0.6069236   1.         -1.
 -1.          0.55335817  0.9321658  -0.51246734 -1.          0.64524302
  1.        ]
```

Kuva 10. Monitavoiteoptimoinnin tulostus

```
Suoritetaan ajo 10/10...
Ajo 10: Accuracy: 0.9821428571428571, Softmax: 0.9995824437923643, Execution Time: 6.0340 seconds
Training Error: 0.0179, Validation Accuracy: 1.0000, Test Accuracy: 1.0000
Painoarvot ajo 10:
[-1.          1.         -1.         -1.         -1.          1.
  0.6        -0.6         0.58437512 -1.         -1.         -1.
  0.0330973   0.91632422  1.         -1.         -1.         -0.24286799
  1.          1.          1.          1.          1.         -0.7143825
 -0.6126989   1.         -1.         -1.          1.          1.
 -1.         -1.          1.         -0.67291406 -0.58216724  0.93859643
  1.         -1.         -1.          1.         -0.6         0.36445824
 -1.        ]

Painoarvot tallennettu tiedostoon 'kaikki_painoarvot.xlsx'.

Kaikki tulokset tallennettu tiedostoon 'kaikki_tulokset.xlsx'.
Accuracy - Keskiarvo: 0.9777, Keskihajonta: 0.0045
Softmax - Keskiarvo: 0.9233, Keskihajonta: 0.0505
```

Kuva 11. Monitavoiteoptimoinnin tulostus lopussa

Näistä voidaan huomata, että ohjelma tulostaa samat tiedot kuin yksitavoiteoptimoinnin toteutuksessa, mutta lisäksi ohjelma tulostaa vielä toisenkin tavoitefunktion arvon eli Softmaxin arvon. Ohjelma tulostaa keskiarvon ja keskihajonnan molemmille

tavoitefunktioille. Ohjelma tulostaa tuttuun tapaan myös sen, että tallennukset tiedostoihin onnistuivat. Funktion f_2 tapauksessa taas tulostetaan Softmaxin sijaan funktion f_2 arvo, keskiarvo ja keskihajonta.

Ohjelma vaikuttaa toimivan oikein ja luotettavasti, koska sen ajo menee läpi virheittä sekä tulokset vaikuttavat olevan johdonmukaisia. Tästä voidaan edetä saatujen tulosten tarkasteluun, mitkä analysoidaan luvussa 6.

6 Tulokset

Tässä luvussa perehdytään tutkimuksesta saatuihin tuloksiin. Ensin katsotaan läpi ennalta määritellyt suorituskykymittarit, jotka ovat tehokkuus, luokittelutarkkuus ja luotettavuus. Suorituskykymittareiden osalta tarkastellaan, miten optimointimenetelmät sekä algoritmi suoriutuivat suhteessa suorituskykymittareihin. Tämän jälkeen vertaillaan alaluvussa 6.2, miten nämä algoritmit suoriutuivat suhteessa toisiinsa. Luvussa 6.3 luodaan yhteenveto yksitavoiteoptimoinnin sekä monitavoiteoptimoinnin tuloksista. Luvussa 6.4 tarkastellaan vielä tuloksia monitavoiteoptimoinnista minimoivalla tavoitefunktiolla f_2. Luvussa 6.5 katsotaan lyhyesti tuloksia monitavoiteoptimoinnin ajolle 500:lla sukupolvella luokittelutarkkuudella ja funktiolla f_2. Luvussa 6.6 tehdään syväanalyysia kaikille toteutuksille ja etenkin monitavoiteoptimoinnille.

Molemmat optimointialgoritmit ajettiin 100 kertaa putkeen läpi, josta kerättiin yksittäisten ajokertojen data talteen. Tästä laskettiin myös keskiarvoisia tuloksia sekä määritettiin keskihajontaa tuloksille, jotta saadaan tietää, miten paljon tulokset eroavat toisistaan. Tulokset kerättiin talteen Excel-tiedostoihin. Tuloksina tässä työssä käytetään ohjelman tulosteita algoritmien suoriutumisesta. Tuloksia ovat tavoitefunktion tai -funktioiden paras löydetty arvo, datajoukkojen virheet sekä luokittelutarkkuus, verkon painoarvot, suoritusaika ja funktion evaluaatioiden lukumäärä. Näistä tuloksista muodostetaan tässä luvussa visualisointeja tulosten havainnollistamiseksi.

6.1 Algoritmien suorituskyky

Tämän alaluvun tarkoitus on analysoida yksi- ja monitavoiteoptimointimenetelmiä kolmesta eri suorituskykyä mittaavasta näkökulmasta. Näiden suorituskykymittareiden avulla voidaan arvioida, miten tässä tutkimuksessa käytetyt optimointimenetelmät suoriutuivat MLP-neuroverkon kouluttamisesta differentiaalievoluutiolla. Tarkasteltavat suorituskykymittarit ovat tehokkuus, luokittelutarkkuus ja luotettavuus. Näitä suorituskykymittareita on käytetty myös edellisissä aihetta koskevissa tutkimuksissa,

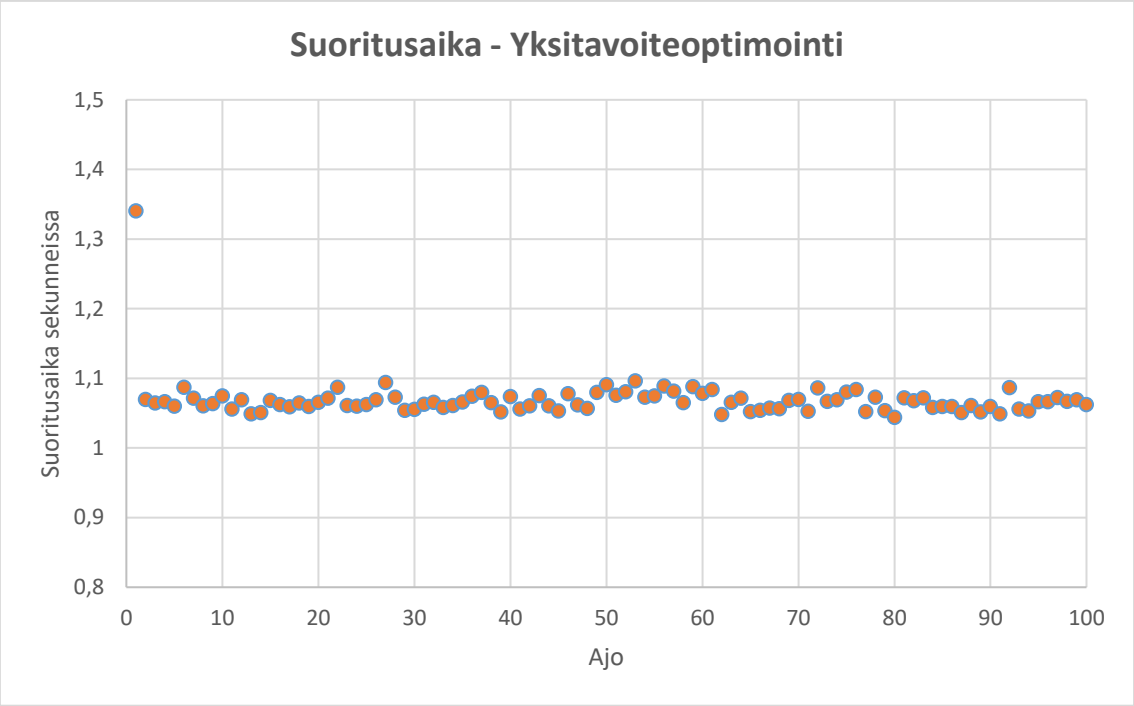
joten niiden käyttö on myös nyt täten perusteltua. Tämän luvun päämäärä on pyrkiä selvittämään, kumpi käytetyistä optimointimenetelmistä soveltuu tarkoitukseensa paremmin. Monitavoiteoptimointi on osoittautunut lupaavaksi, joten etenkin sen tarkasteleminen tässä luvussa on oleellista.

6.1.1 Tehokkuus

Tehokkuuden tarkoitus on arvioida optimointialgoritmien suorituskykyä optimointiprosessin aikana. Tehokkuuden mittaus perustuu seuraaviin tuloksissa saatuihin tuloksiin, jotka ovat suoritusaika, funktion evaluointien lukumäärä ja funktion arvon kehittyminen. Tehokkuus suorituskykymittarina pyrkii vastaamaan alla annettuihin kysymyksiin.

- Kuinka nopeasti algoritmi saavuttaa optimaalisen ratkaisun?
- Kuinka monta funktion evaluointia optimaalisen ratkaisun saavuttaminen vaatii?

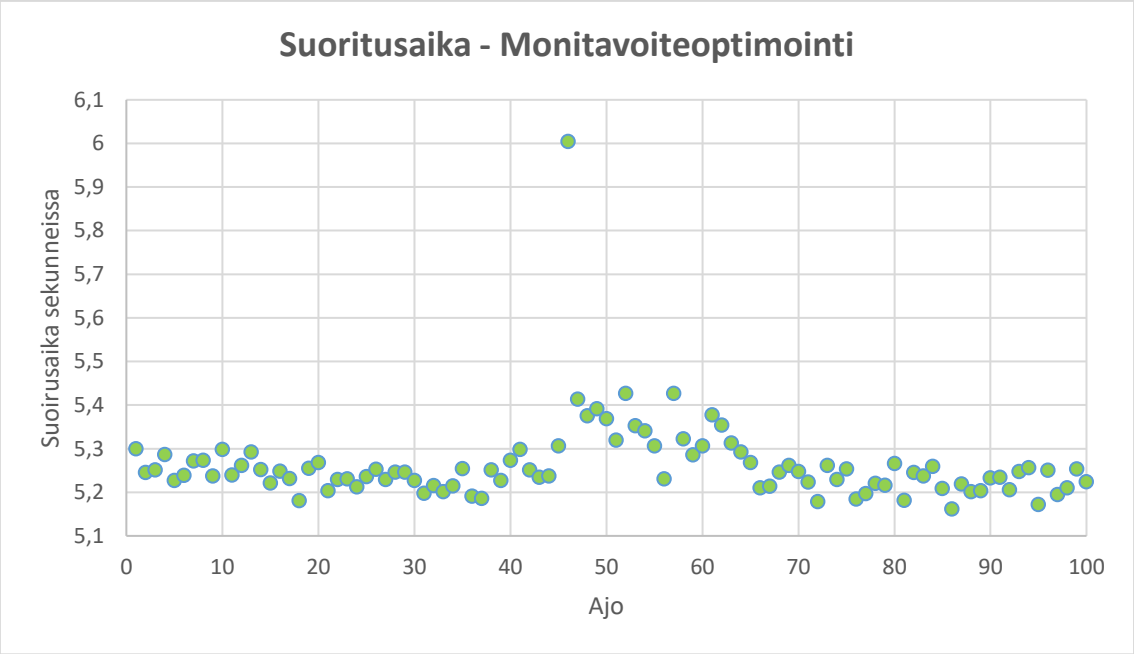
Tarkastelleen ensimmäiseksi molempien optimointialgoritmien suoritusaikaa. Alla esitetyssä kuvaajassa on nähtävissä yksitavoiteoptimoinnin keskimääräinen suoritusaika 100 ajokerralla.



Kuvio 1. Suoritusaika – Yksitavoiteoptimointi

Kuvaajasta voidaan lukea, että yksitavoiteoptimoinnin suoritusaika asettui keskiarvoisesti vaihteluvälille [1,04–1,1] sekuntia. Poikkeuksena tästä ensimmäinen ajo, joka selittyy esimerkiksi muiden Windows-prosessien vaatimilla resursseilla. 100 ajokertaan kului siis aikaa karkeasti laskettuna noin 107 sekuntia eli karkeasti pyöristettynä noin 2 minuuttia.

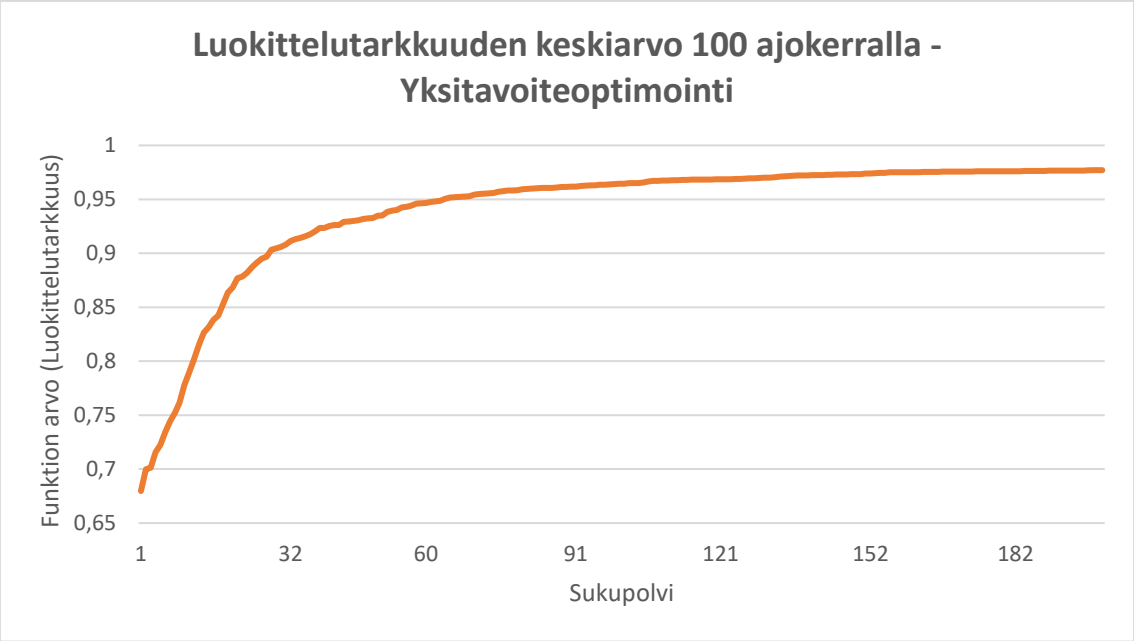
Alla esitetyssä kuvaajassa on nähtävissä monitavoiteoptimoinnin keskimääräinen suoritusaika 100 ajokerralla.



Kuvio 2. Suoritusaika – Monitavoiteoptimointi

Kuvaajasta voidaan lukea, että monitavoiteoptimoinnin suoritusaika asettui keskiarvoisesti vaihteluvälille [5,15–5,45] sekuntia. Poikkeuksena tästä 46. ajo, joka selittyy esimerkiksi muiden Windows-prosessien vaatimilla resursseilla. 100 ajokertaan kului siis aikaa karkeasti laskettuna noin 530 sekuntia eli karkeasti pyöristettynä noin 9 minuuttia.

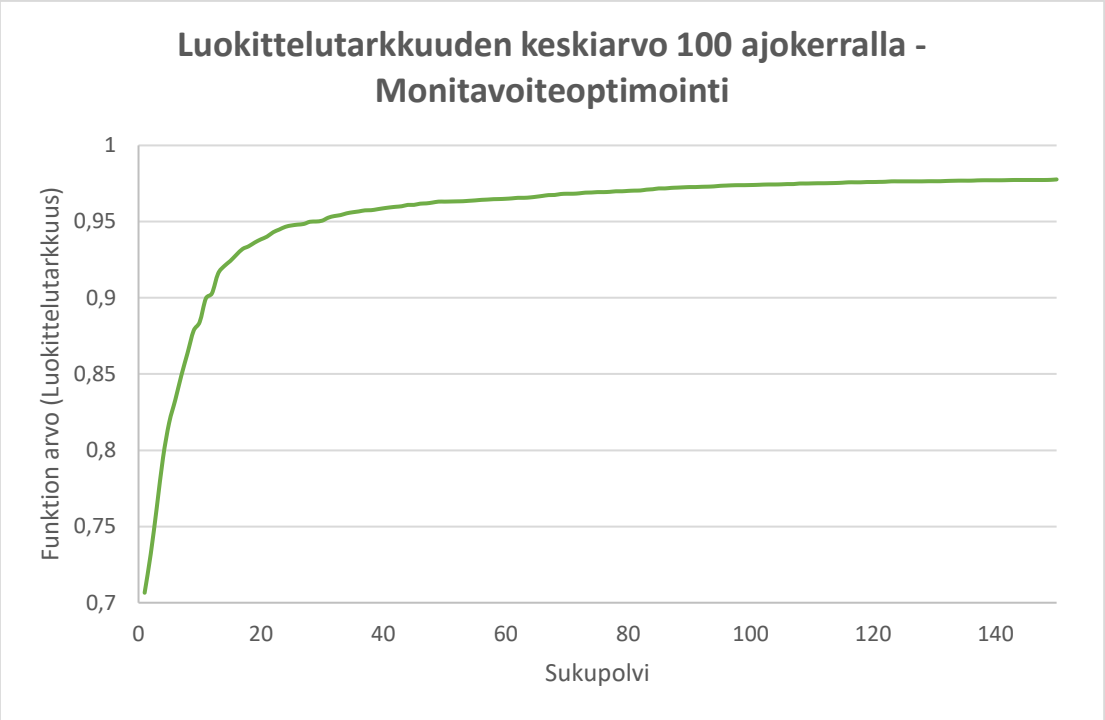
Seuraavaksi tarkastellaan tavoitefunktion arvon kehittymistä sukupolvikohtaisesti. Monitavoiteoptimoinnissa katsotaan molempien tavoitefunktioiden kehittymistä. Alla esitetyssä kuvaajassa (kuvio 3) on nähtävissä yksitavoiteoptimoinnin funktion arvon kehitys sukupolvien määrän kasvaessa. Yksitavoiteoptimoinnissa käytettiin DE:n parametrina 200 sukupolvea. Sukupolvet on esitetty x-akselilla ja funktionarvo y-akselilla.



Kuvio 3. Luokittelutarkkuuden arvon kehittyminen – Yksitavoiteoptimointi

Luokittelutarkkuuden keskiarvo on laskettu kaikkien 100 ajon kehittyvistä sukupolvikohtaisista arvoista. Se esittää täten keskiarvoisesti koko tutkimuksen dataa yksittäisten ajokertojen sukupolvien tasolla. Keskiarvo on määritetty Python-skriptillä, joka lukee kaikki 100 yksittäisen ajon Excel-datatiedostoa ja laskee uuteen Excel-tiedostoon keskiarvon jokaisen tiedoston jokaisen sukupolven arvosta. Näin muodostettiin kahden sarakkeen tiedosto, joka summaa kaikki yksittäiset ajokerrat. Kuvaajasta voimme huomata, että funktion arvo saavuttaa optimiarvonsa noin 150 sukupolven jälkeen. Funktion arvo kehittyy ensimmäiset 60 sukupolvea voimakkaasti, jonka jälkeen kehitys tasaantuu ja hiljalleen pysähtyy 150 sukupolven jälkeen. Funktion ensimmäisen sukupolven arvo on noin 0,67, josta se kehittyy 0,98 paikkeille.

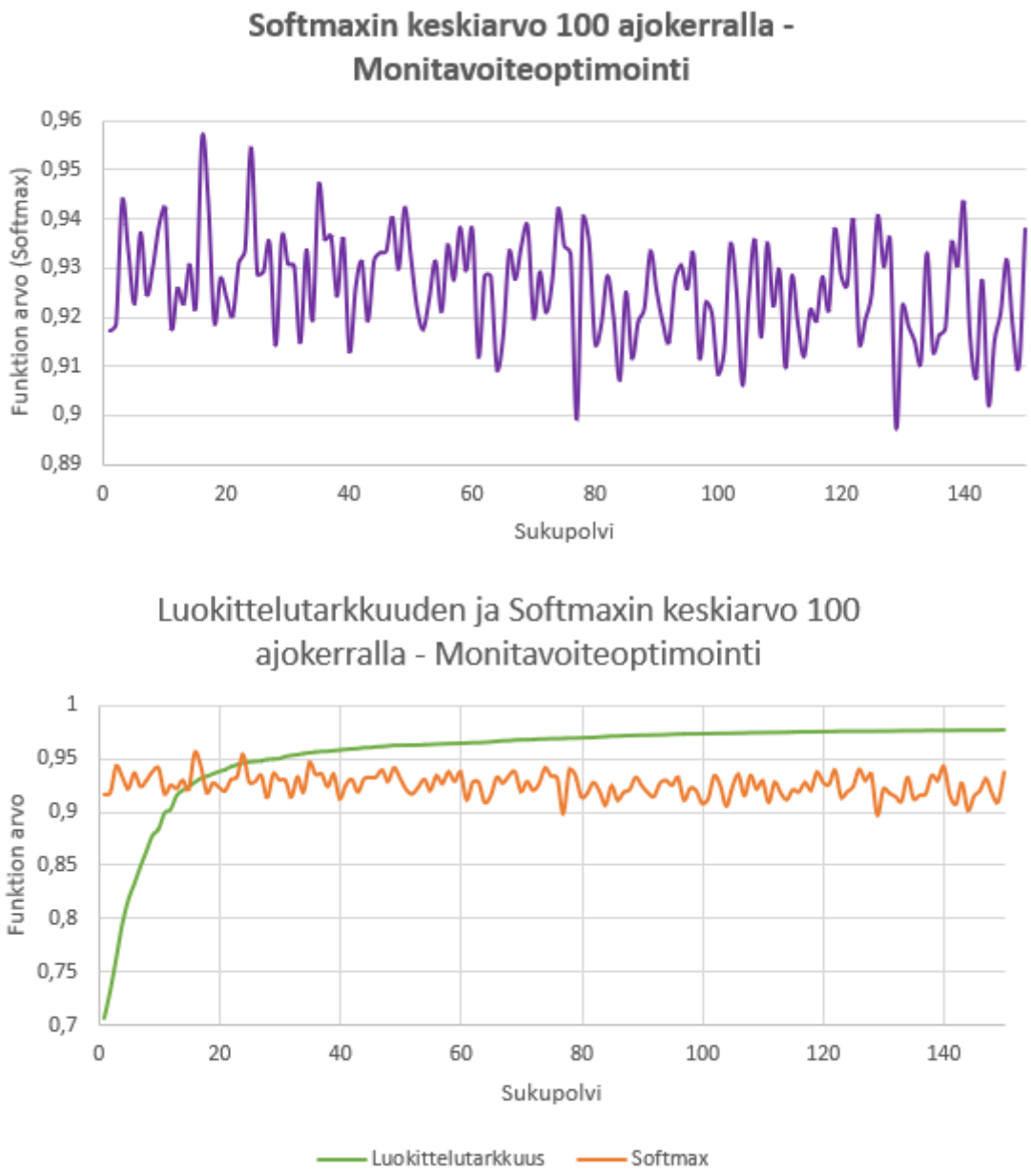
Alla esitetyssä kuvaajassa (kuvio 4) on nähtävissä monitavoiteoptimoinnin luokittelutarkkuuden funktion arvon kehitys sukupolvien määrän kasvaessa. Monitavoiteoptimoinnissa käytettiin DE:n parametrina 150 sukupolvea. Sukupolvet on esitetty x-akselilla ja funktion arvo y-akselilla.



Kuvio 4. Luokittelutarkkuuden arvon kehittyminen – Monitavoiteoptimointi

Monitavoiteoptimoinnin osalta luokittelutarkkuuden keskiarvo on määritetty samalla menetelmällä, kuin yksitavoiteoptimoinnissa. Yllä oleva kuvaaja siis esittää kaikkien ajojen keskiarvoa luokittelutarkkuuden funktion arvosta 100 ajokerralla. Kuvaajasta voimme huomata, että luokittelutarkkuus saavuttaa optimiarvonsa noin 100 sukupolven jälkeen. Luokittelutarkkuuden arvo muuttuu merkittävästi noin 60 sukupolveen asti, jonka jälkeen funktion arvon muuttuminen hidastuu. Funktion ensimmäisen sukupolven arvo on noin 0,71, josta se kehittyy 0,98 paikkeille.

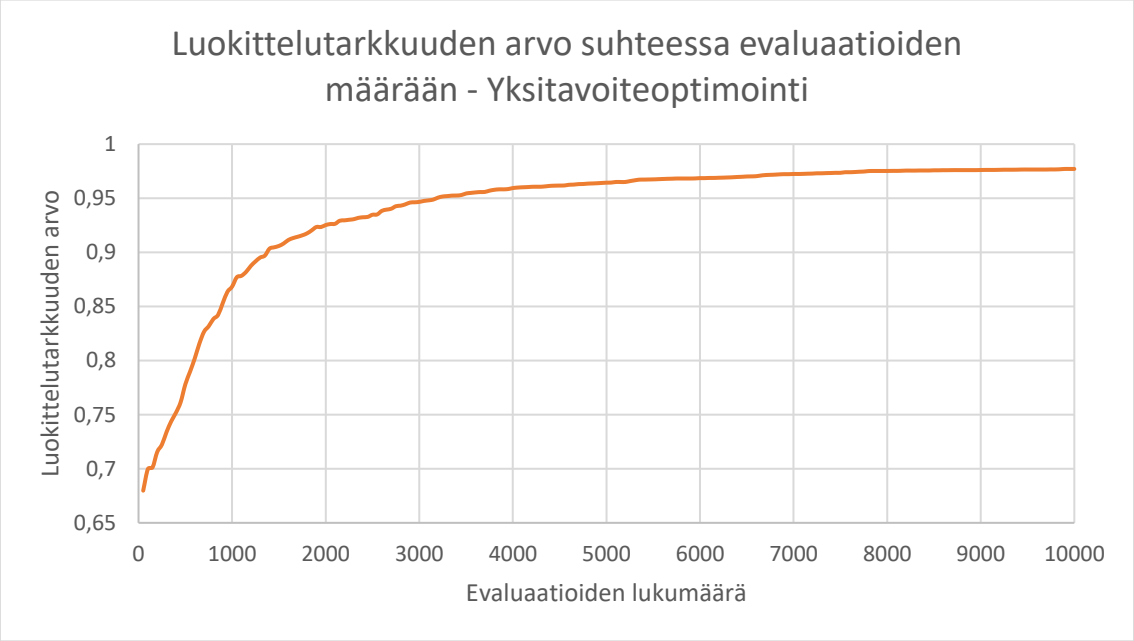
Alla esitetyssä kuvaajassa (kuvio 5) on nähtävissä monitavoiteoptimoinnin Softmax-funktion arvon kehitys sukupolvien määrän kasvaessa.



Kuvio 5. Softmaxin arvon kehittyminen ja molempien funktioiden arvot– Monitavoiteoptimointi

Myös Softmaxin keskiarvo on määritetty samaan tapaan kuin luokittelutarkkuuksien keskiarvot. Softmaxin funktion arvon vaihteluväli on karkeasti [0,895–0,96]. Se liikkuu siis jatkuvasti lähellä arvoa 1.

Seuraavaksi keskitytään funktion evaluointien lukumäärän. Evaluointien lukumäärää analysoitaessa huomioidaan vain luokittelutarkkuuden arvo ja se, kuinka monta evaluaatiota luokittelutarkkuuden optimiarvon saavuttaminen vaatii. Alla esitetyssä kuvaajassa (kuvio 6) on esitetty yksitavoiteoptimoinnin luokittelutarkkuuden arvon kehittyminen suhteessa evaluointien lukumäärään siihen mennessä.

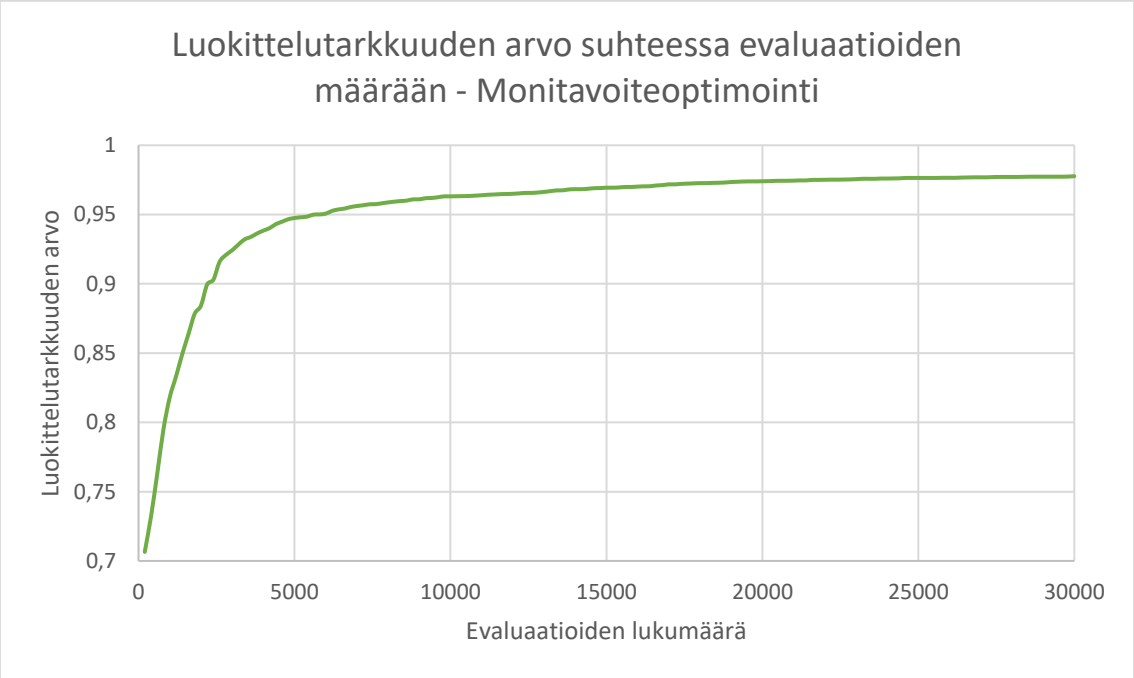


Kuvio 6. Luokittelutarkkuuden ja evaluointien määrän suhde– Yksitavoiteoptimointi

Yksitavoiteoptimoinnin tapauksessa luokittelutarkkuus vaikuttaa saavuttavan kohtuullisen tasaisen arvon noin 8000 evaluoinnin kohdalla. Tässä kohdassa luokittelutarkkuuden arvo on noin 0,97. 10000 evaluaatiossa luokittelutarkkuuden arvo on 0,97054. Suurin muutos luokittelutarkkuuden arvossa näyttää tapahtuvan ennen 5000 evaluaatiota. Tämä evaluaatioiden lukumäärä suhteessa luokittelutarkkuuteen on laskettu myös muodostaen 100 ajon keskiarvon avulla keskiarvoinen luokittelutarkkuuden kehitys.

Tämän jälkeen tutustutaan luokittelutarkkuuteen suhteessa evaluaatioiden määrään monitavoiteoptimoinnissa. Alla olevassa kuvaajassa (kuvio 7) on esitetty

luokittelutarkkuuden arvo suhteessa evaluaatioiden lukumäärään monitavoiteoptimoinnissa.



Kuvio 7. Luokittelutarkkuuden ja evaluointien määrän suhde– Monitavoiteoptimointi

Kuvaajasta voidaan huomata, että luokittelutarkkuuden arvo vaikuttaa saavuttavan jokseenkin tasaisen arvon 20000 evaluaation kohdalla. Tässä kohdassa luokittelutarkkuuden arvo on noin 0,97. 30000 evaluaatiossa luokittelutarkkuuden arvo saavuttaa arvon 0,977679. Suurin muutos luokittelutarkkuuden arvossa vaikuttaa tapahtuvan ennen 10000 evaluaatiota. Tämä evaluaatioiden lukumäärä suhteessa luokittelutarkkuuteen on laskettu muodostaen 100 ajon keskiarvon avulla keskiarvoinen luokittelutarkkuuden kehitys.

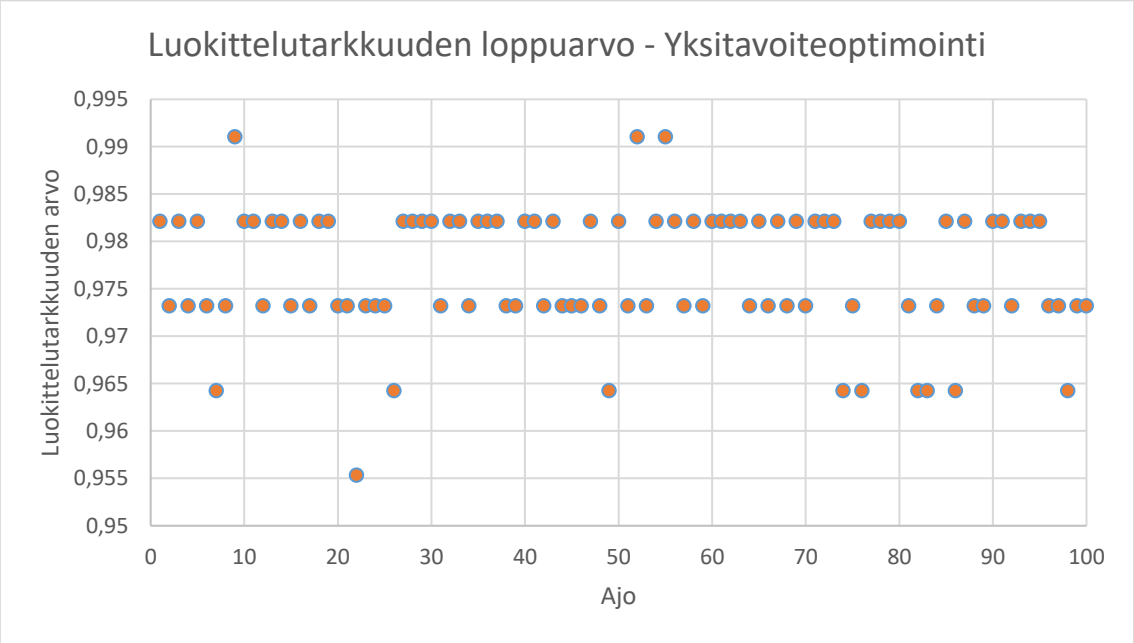
6.1.2 Luokittelutarkkuus

Luokittelutarkkuuden tehtävä on arvioida neuroverkon kykyä luokitella sille opetettavaa datakokoelmaa. Tästä hyvänä indikaattorina toimii luokittelutarkkuuden arvo, joka on saatu tuloksena molempia optimointimenetelmiä ajettaessa. Käytetään tämän lisäksi luokittelutarkkuuden arvioinnissa opetus- testi- ja validointijoukonvirhettä.

Luokittelutarkkuus suorituskykymittarina pyrkii vastaamaan seuraavaan alla annettuun kysymykseen.

- Kuinka laadukkaasti neuroverkko oppii luokittelemaan datasetin?

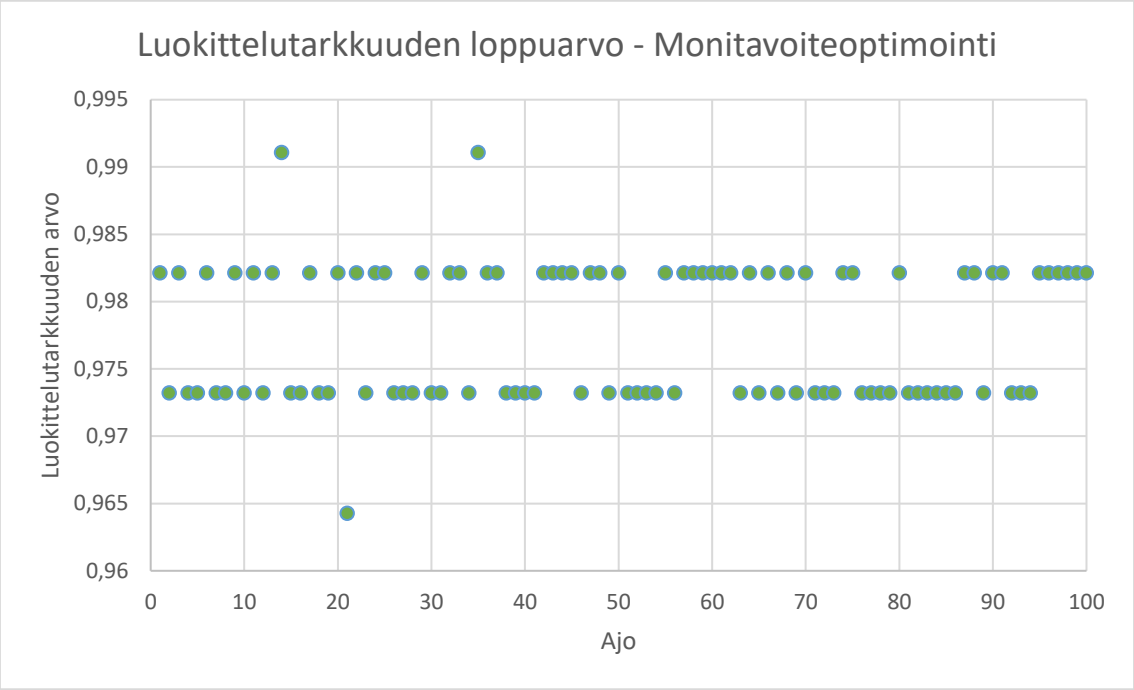
Lähdetään ensimmäiseksi tarkastelemaan luokittelutarkkuuden loppuarvoja kaikkien ajokertojen kesken molemmissa optimointimenetelmistä. Luodaan siis visualisointi luokittelutarkkuuden loppuarvoista yhdistäen ne ajon järjestysnumeroon. Tämä tieto saadaan Exceliin tallennetuista kaikkien ajojen tuloksista. Myös tässä käytetään samoja 100 ajokerran tuloksia.



Kuvio 8. Luokittelutarkkuuden loppuarvo ajojen yli – Yksitavoiteoptimointi

Kuviosta 8 voidaan huomata, että luokittelutarkkuus saa arvoja väliltä [0,955–0,995], joten voidaan todeta, että luokittelutarkkuus on hyvin korkea yksitavoiteoptimoinnin tapauksessa. Luokittelutarkkuus vaikuttaa saavan pääsääntöisesti kahta eri loppuarvoa, jotka ovat alempi 0,9732 ja korkeampi 0,9821. Suunnilleen 12 pistettä aiheuttaa hajontaa noiden kahden pääsääntöisen pisteen lisäksi. Näistä 12 pisteestä 9 kappaletta

saa pienemmän luokittelutarkkuuden arvon kuin 0,9732 ja 3 kappaletta saa suuremman luokittelutarkkuuden arvon kuin 0,9821.

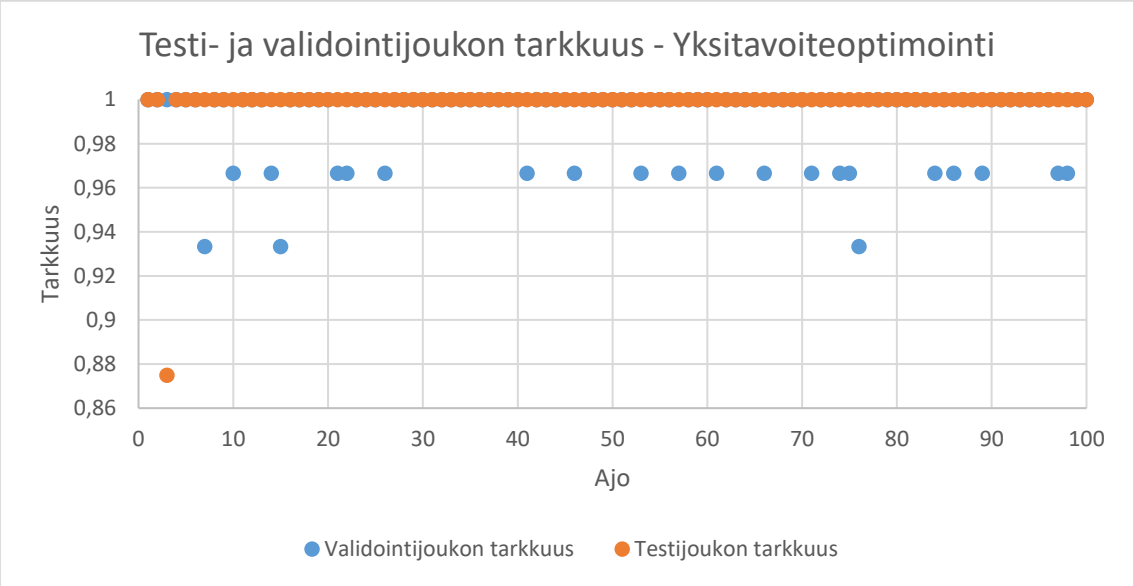


Kuvio 9. Luokittelutarkkuuden loppuarvo ajojen yli – Monitavoiteoptimointi

Kuviosta 9 voidaan lukea, että luokittelutarkkuus monitavoiteoptimoinnissa saa arvoja väliltä [0,96–0,995]. Myös tämä luokittelutarkkuus on hyvin korkea. Vaikuttaa taas siltä, että luokittelutarkkuus saa suurimmalta osalla ajoista pääasiassa kahta eri arvoa, jotka ovat suurempi 0,9821 ja pienempi 0,9732. Hajontaa näyttää tapahtuvan vähän, sillä vain kolme pistettä on kahden valtaa pitävän tuloksen ulkopuolella. Yksi näistä pisteistä on pienempi kuin 0,9732 ja kaksi pistettä saavat suuremmat luokittelutarkkuuden arvot kuin 0,9821.

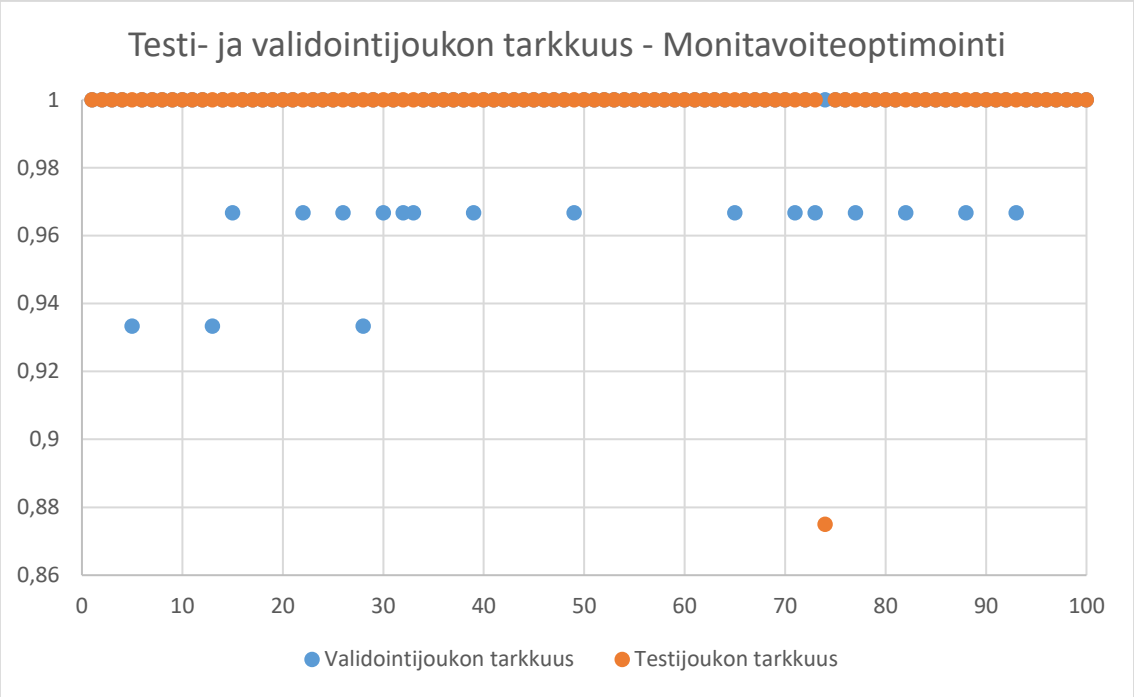
Seuraavaksi tarkastellaan testi- ja validointidatajoukon tarkkuutta luokittelutarkkuuden mittarina. Tämä tarkastelu tehdään molempien optimointimenetelmien kannalta. Kuten aiemmasta muistetaan, data jaettiin kolmen edellä mainitun datajoukon kesken seuraavalla jaolla, joka on opetusjoukko 75 %, validointijoukko 20 % ja testijoukko 5 %.

Alla kuviossa 10 on esitetty yksitavoiteoptimoinnin testi- ja validointijoukkojen tarkkuus suhteessa ajokertoihin.



Kuvio 10. Datajoukkojen virhe – Yksitavoiteoptimointi

Kuviosta 10 voimme huomata, että validointi- sekä testijoukon tarkkuus on pääsääntöisesti jokaisen ajon kohdalla 1 eli tarkin mahdollinen. Pientä vaihtelua tarkkuudessa voidaan havaita etenkin validointijoukon osalta, mutta tämä toistuu keskimääräisesti vain noin joka 10 ajo. Testi joukko on ajoa 3 lukuun ottamatta aina arvossa 1. Ajossa 3 testijoukon tarkkuus saa arvon 0,875. Alla kuviossa 11 on esitetty monitavoiteoptimoinnin testi- ja validointijoukkojen tarkkuus suhteessa ajokertoihin.



Kuvio 11. Datajoukkojen virhe – Monitavoiteoptimointi

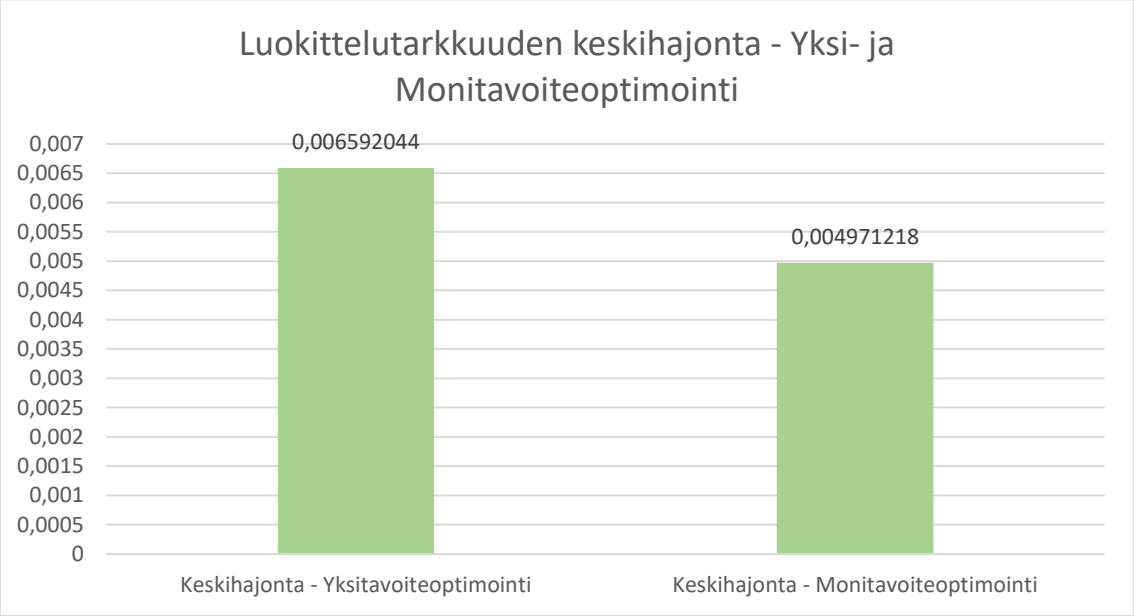
Kuviosta 11 voidaan huomata, että molempien joukkojen tarkkuus on lähes kaikilla ajokerroilla 1. Myös tässä tapauksessa validointijoukon tarkkuus saa 1 poikkeavan arvon noin 10 ajokerran välein. Monitavoiteoptimoinnin tapauksessa testijoukon tarkkuus käy arvossa 0,875 kerran ajolla 74. Tämä oli viimeinen tarkasteltava mittari luokittelutarkkuuden osalta. Tästä siirrytään tarkastelemaan optimointialgoritmeja luotettavuuden näkökulmasta luvussa 6.1.3.

6.1.3 Luotettavuus

Luotettavuuden tarkoituksena on tarkastella optimointialgoritmien johdonmukaisuutta sekä toistettavuutta. Luotettavuuden puolesta hyvänä indikaattorina toimii ajojen tulosten keskihajonta. Keskihajonta kertoo, kuinka paljon eri ajokerroilla saadut tulokset poikkeavat toisistaan eli miten toistettavia tulokset ovat. Luotettavuus kertoo paljon algoritmin käyttökelpoisuudesta, sillä tuloksien tulee olla toistettavia, jotta voidaan tehdä johdonmukaisia päätelmiä tulosten perusteella. Luotettavuus suorituskykymittarina pyrkii vastaamaan seuraaviin alla oleviin kysymyksiin.

- Kuinka paljon saadut tulokset eroavat toisistaan?
- Kuinka toistettavia saadut tulokset ovat?

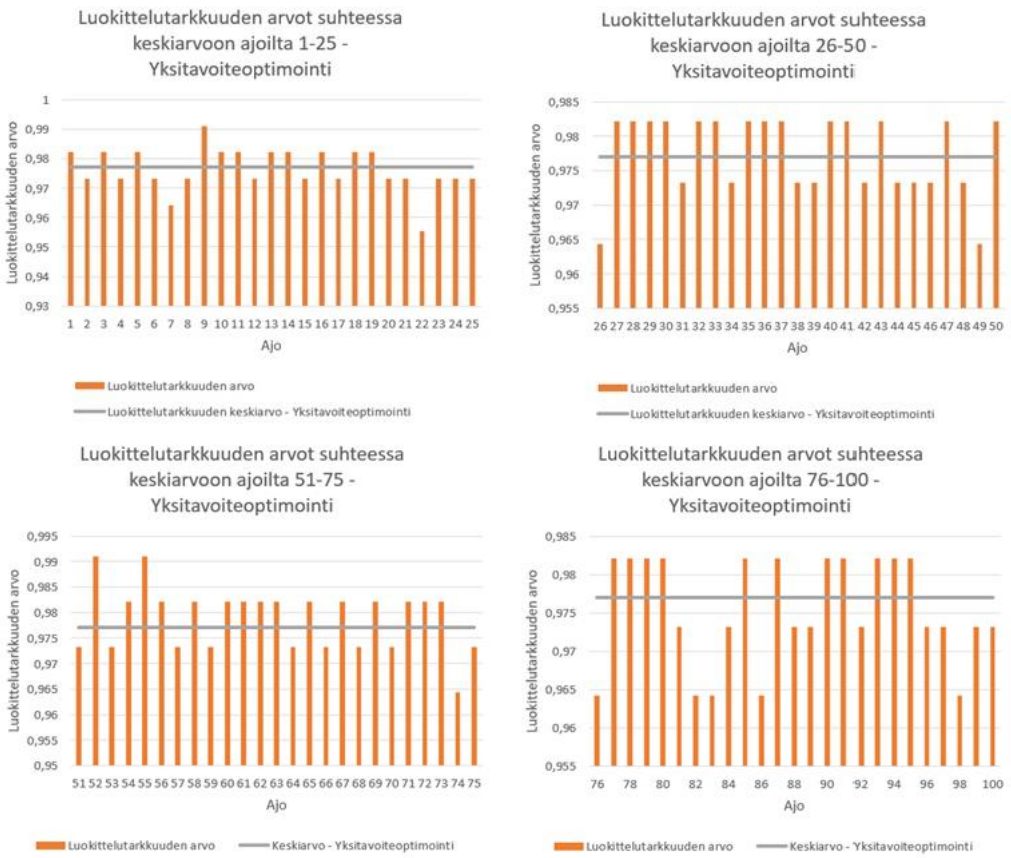
Lähdetään tarkastelemaan ajojen keskihajontaa. Keskihajonta on laskettu sekä yksi- että monitavoiteoptimoinnille ajojen päätteeksi saaduista tuloksista. Tästä voidaan päätellä, miten toistettavia kunkin menetelmän 100 ajokertaa ovat. Keskihajonta on laskettu valmiiksi kaikkien tulosten Excel-tiedostoon, josta visualisointi luodaan tässä osiossa. Alla olevasta kuviosta 12 on nähtävissä yksi- ja monitavoiteoptimoinnin tulosten keskihajonta.



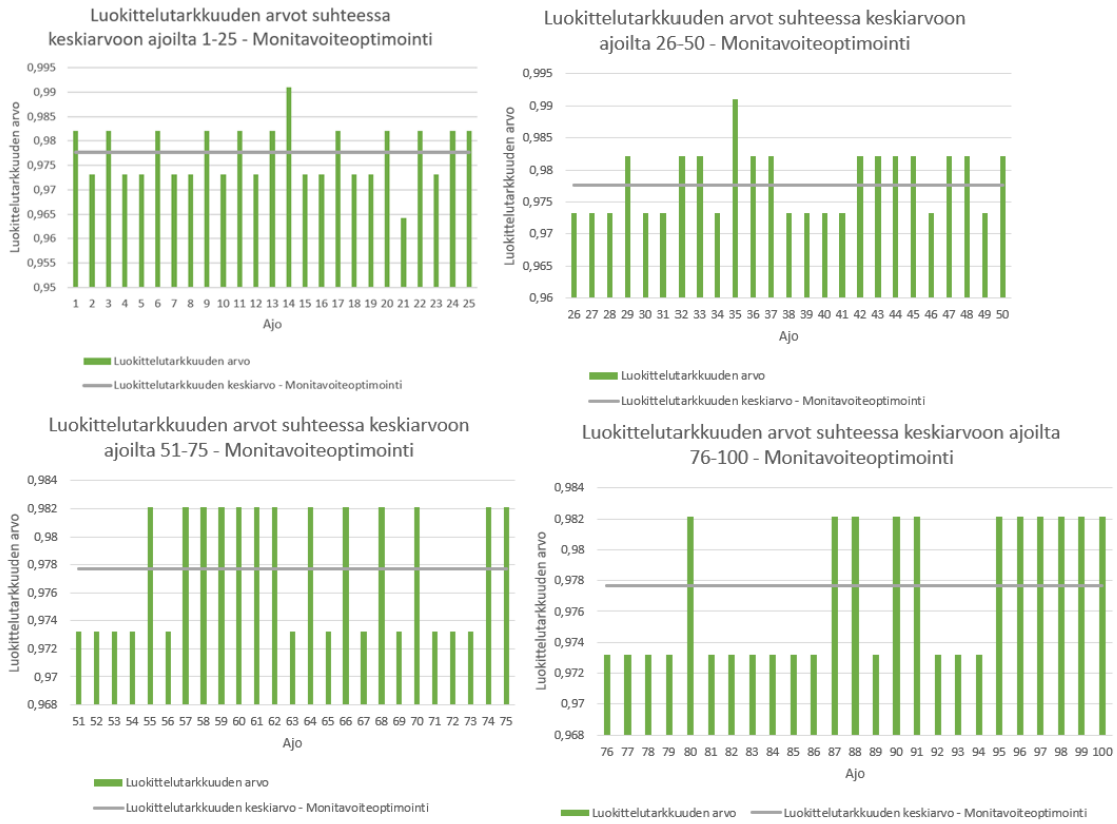
Kuvio 12. Luokittelutarkkuuden keskihajonta – Yksi- ja monitavoiteoptimointi

Kuviosta 12 voidaan havaita keskihajonnan olevan molemmilla optimointimenetelmillä kohtuullisen pieni. Tästä voidaan todeta, että molemmat algoritmi tuottavat tarpeeksi luotettavia ja toistettavia tuloksia eli ne toimivat hyvin. Yksitavoiteoptimoinnin keskihajonta saa arvon 0,006592 ja monitavoiteoptimoinnin keskihajonta saa arvon 0,004971. Monitavoiteoptimoinnin keskihajonta on siis pienempi eli sen tulokset ovat vielä luotettavampia kuin yksitavoiteoptimoinnin tulokset. Seuraavaksi jatketaan luotettavuuden analysointia vertaamalla luokittelutarkkuuden arvoja jokaisella

ajokerralla luokittelutarkkuuden keskiarvoon molemmilla optimointimenetelmillä. Tämän visualisointi on esitetty kuvioissa 13 ja 14.



Kuvio 13. Keskiarvo suhteessa kaikkiin tuloksiin 100 ajolta – Yksitavoiteoptimointi



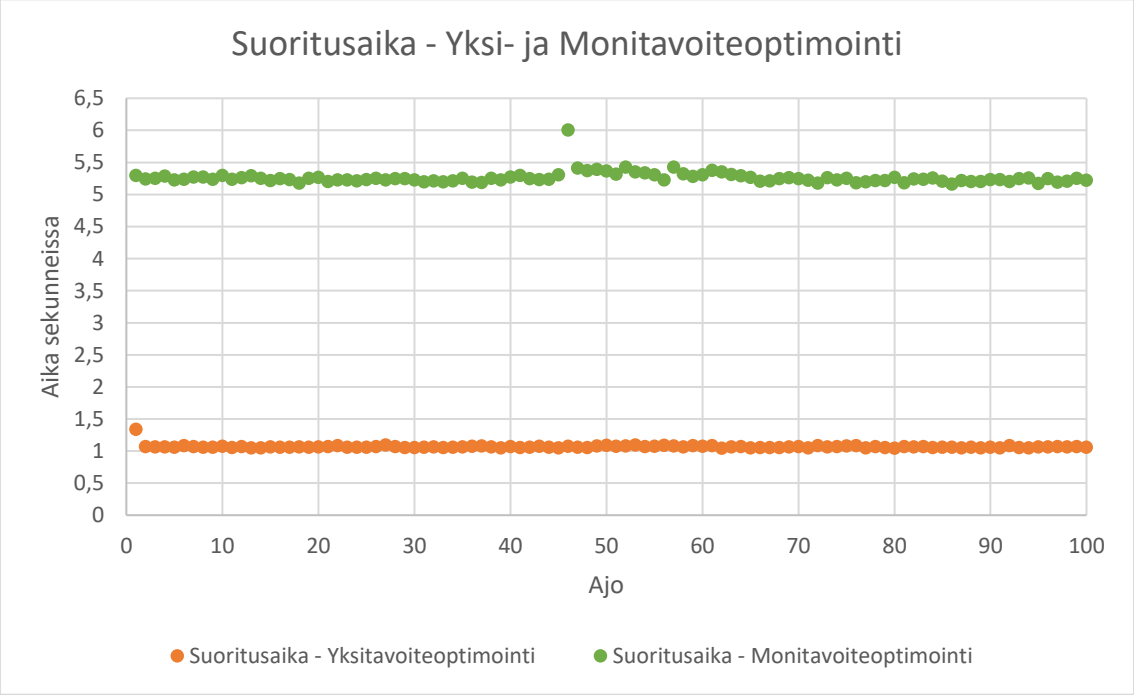
Kuvio 14. Keskiarvo suhteessa kaikkiin tuloksiin ajoilta 100 ajolta – Monitavoiteoptimointi

Kuvioista 13 ja 14 voidaan todeta tulosten hajonnan olevan kohtuullisen pientä, joten keskihajonta voidaan todeta paikkansa pitäväksi ja näin ollen algoritmien tuottamat tulokset tarpeellisen luotettaviksi. Kuvioista voidaan huomata, että monitavoiteoptimoinnilla hajonta on pienempää, mutta taas toistuvampaa, kun verrataan keskiarvotuloksen viivaan. Keskiarvotulos on esitetty harmaalla viivalla jokaisessa visualisoinnissa.

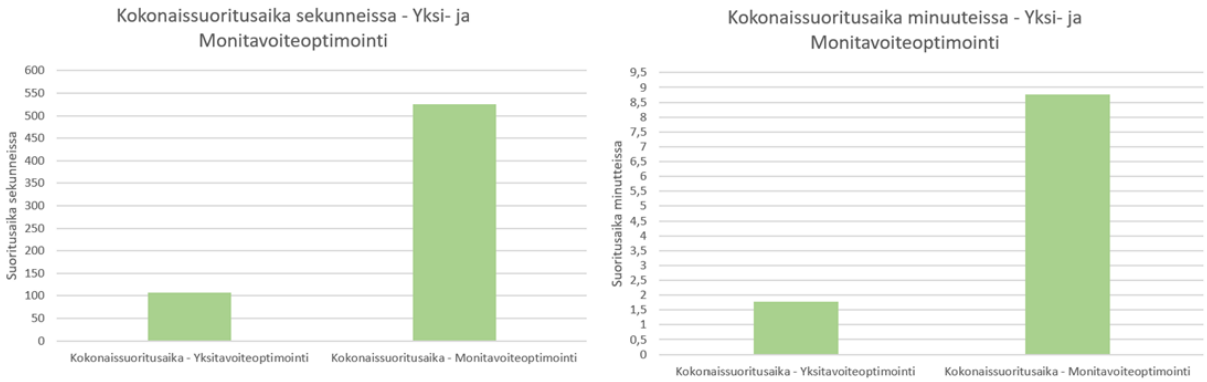
6.2 Optimointimenetelmien vertailu

Tässä alaluvussa keskitytään yksi- ja monitavoiteoptimointimenetelmien vertailuun. Menetelmien vertailu on tämän työn päätavoite. Tässä vertailussa käytetään apuna edellisessä luvussa 6.1 piirrettyjä kuvaajia ja yhdistellään niitä, jotta voidaan todeta yksi- ja monitavoiteoptimoinnin erot ja yhtäläisyydet suoraan visualisointien avulla.

Lähdetään ensiksi liikkeelle vertaamalla menetelmien suoritusaikoja alla olevalla kuviolla 15. Kuviossa 16 on esitetty molempien menetelmien kokonaissuoritusajat sekunneissa sekä minuuteissa.



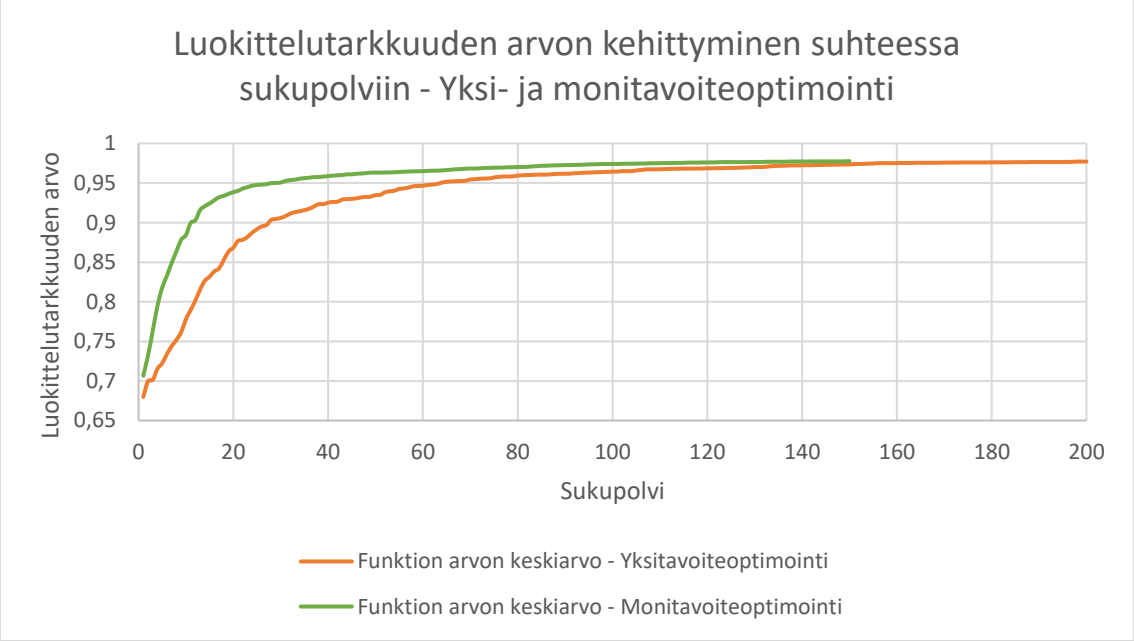
Kuvio 15. Yksi- ja monitavoiteoptimointien suoritusajat



Kuvio 16. Yksi- ja monitavoiteoptimointien kokonaissuoritusajat

Kuvioista 15 ja 16 voidaan nähdä, että monitavoiteoptimoinnin suoritusaika yksitavoiteoptimointiin nähden on noin 5-kertainen. Kuten aikaisemmin tässä luvussa jo todettiin, kuvaajista voidaan huomata kuviossa 15 pientä poikkeavuutta

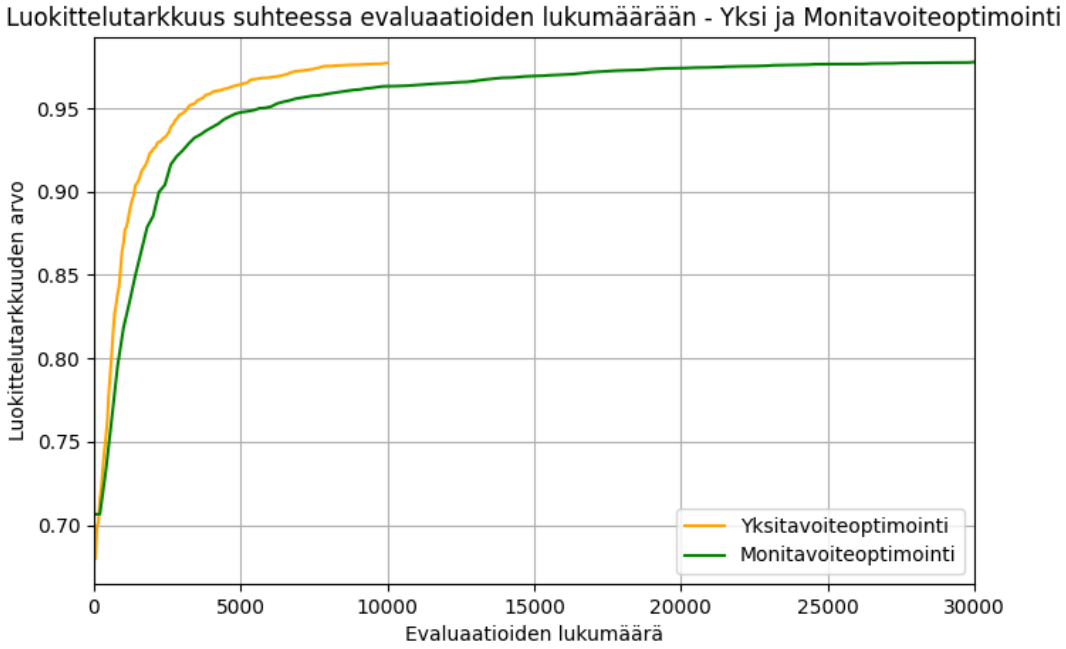
yksitavoiteoptimoinnin ajokerralla 46 ja monitavoiteoptimoinnin ajokerralla 1. Tämä poikkeavuus selittyy muiden Windows-prosessien vaatimilla resursseilla eli hetkellisesti tämän ohjelman suoritukselle on ollut vähemmän resursseja käytettävissä. Yksitavoiteoptimoinnissa yhden ajon suorittaminen vie hieman yli 1 sekunnin, kun taas monitavoiteoptimoinnissa yhden ajon suorittaminen vie yli 5 sekuntia. Myös kokonaissuoritusajassa on merkittävää eroa, sillä yksitavoiteoptimoinnilla 100 ajon suorittaminen kestää noin 2 minuuttia, kun taas monitavoiteoptimoinnilla siihen menee hieman alle 9 minuuttia. Suurempi suoritusaika monitavoiteoptimoinnissa johtuu muun muassa siitä, että optimoitavana on kaksi tavoitefunktiota sekä siitä, että differentiaalievoluution parametreina on käytetty esimerkiksi isompaa populaation kokoa kuin yksitavoiteoptimoinnissa. Täten voidaan todeta, että suoritusajassa yksitavoiteoptimointi on tehokkaampi menetelmä kuin monitavoiteoptimointi.



Kuvio 17. Yksi- ja monitavoiteoptimointien luokittelutarkkuus suhteessa sukupolviin

Kuviosta 17 voi havainnoida minkä verran sukupolvia optimituloksen saavuttaminen vaatii. Monitavoiteoptimointi pääsee sukupolvien määrän nähden selkeästi nopeammin optimiratkaisuun kuin yksitavoiteoptimointi. Yksitavoiteoptimoinnille saman optimiratkaisun lopullinen saavuttaminen vaatii noin neljänneksen enemmän sukupolvia

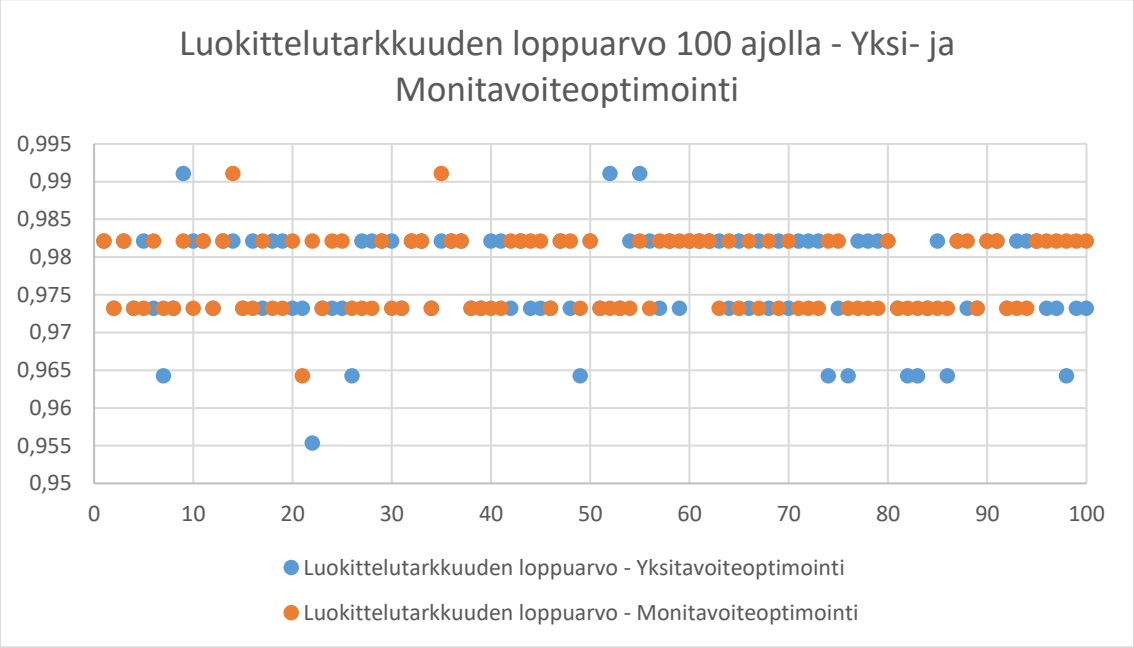
kuin monitavoiteoptimoinnilla. Monitavoiteoptimointi saavuttaa yli 0,95 luokittelutarkkuuden arvon jo noin 30 sukupolven jälkeen, kun yksitavoiteoptimoinnilta se vaatii lähes 70 sukupolvea. Koska monitavoiteoptimointi saavuttaa tästä näkökulmasta paremman ratkaisun nopeammin, voidaan todeta, että tässä suhteessa monitavoiteoptimointi on tehokkaampi optimointimenetelmä kuin yksitavoiteoptimointi.



Kuvio 18. Yksi- ja monitavoiteoptimointien luokittelutarkkuuden keskiarvo suhteessa evaluaatioihin

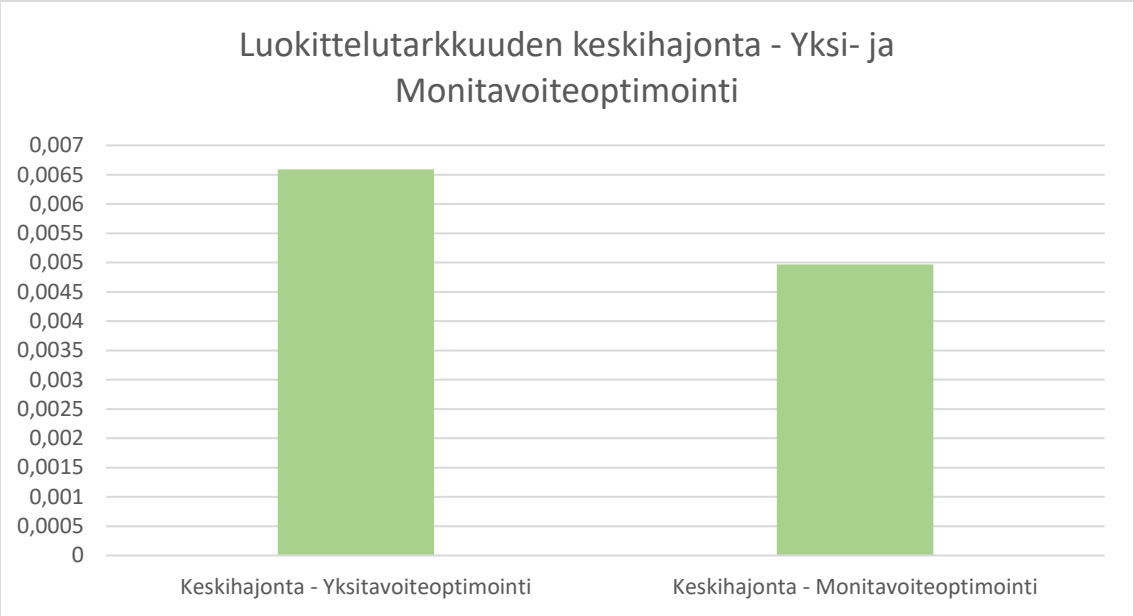
Kuviosta 18 voidaan lukea funktion arvon kehitys molemmille optimointimenetelmillä evaluaatioiden lukumäärän nähden. Tässä suhteessa yksitavoiteoptimointi vaikuttaa tehokkaammalta optimointimenetelmältä, koska se saavuttaa 0,95 luokittelutarkkuuden arvon noin 3000 evaluaation jälkeen. Monitavoiteoptimoinnilta tämän saman 0,95 luokittelutarkkuuden arvon saavuttaminen vaatii hieman yli 5000 evaluaatiota. Tässä suhteessa yksitavoiteoptimointi vaikuttaa olevan tehokkaampi menetelmä. Funktion arvo vaikuttaa paranevan ainakin monitavoiteoptimoinnilla vielä 30000 evaluaation jälkeen, mutta vain osalla ajokerroista. Parannus on myös marginaalisen pieni, joten

evaluaatioiden lisääminen suhteessa suoritusajan lisääntymiseen ei ole enää järkevää käytettävissä olevilla resursseilla. Luokittelutarkkuuden arvo parantuu vain joitakin sadasosia. Tästä tehdään kuitenkin vielä kokeet luvussa 6.5.



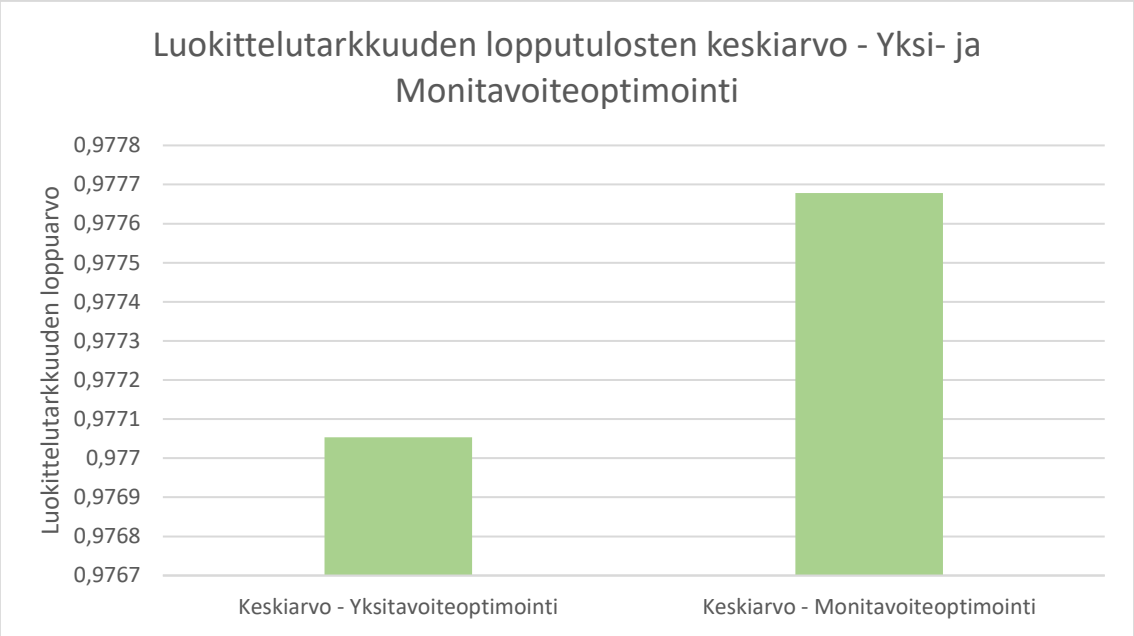
Kuvio 19. Yksi- ja monitavoiteoptimointien luokittelutarkkuuden loppuarvo

Kuviosta 19 voidaan tarkastella optimointimenetelmien paremmuutta luokittelutarkkuuden näkökulmasta. Vaikuttaa siltä, että molemmat optimointimenetelmät saavat suurimman osan ajokerroista pääosin kahta eri luokittelutarkkuuden arvoa, jotka ovat 0,9732 ja 0,9821. Pääosa luokittelutarkkuuksista on prosenteissa siis noin 97 % sekä 98 %. Puhutaan siis jo hyvin korkeasta luokittelutarkkuuden arvosta molempien optimointimenetelmien osalta. Kuvaajan perusteella erot ovat melko marginaalisia, mutta monitavoiteoptimointi vaikuttaa saavan hieman parempia arvoja, koska yksitavoiteoptimoinnissa esiintyy enemmän hajontaa arvon 0,9732 alapuolella. Täten voidaan todeta tämän ja myös monitavoiteoptimoinnin lopullisen luokittelutarkkuuden keskiarvon perusteella, että monitavoiteoptimointi sai tässä tutkimuksessa parempia luokittelutarkkuuden arvoja kuin yksitavoiteoptimointi.



Kuvio 20. Yksi- ja monitavoiteoptimointien tulosten keskihajonta

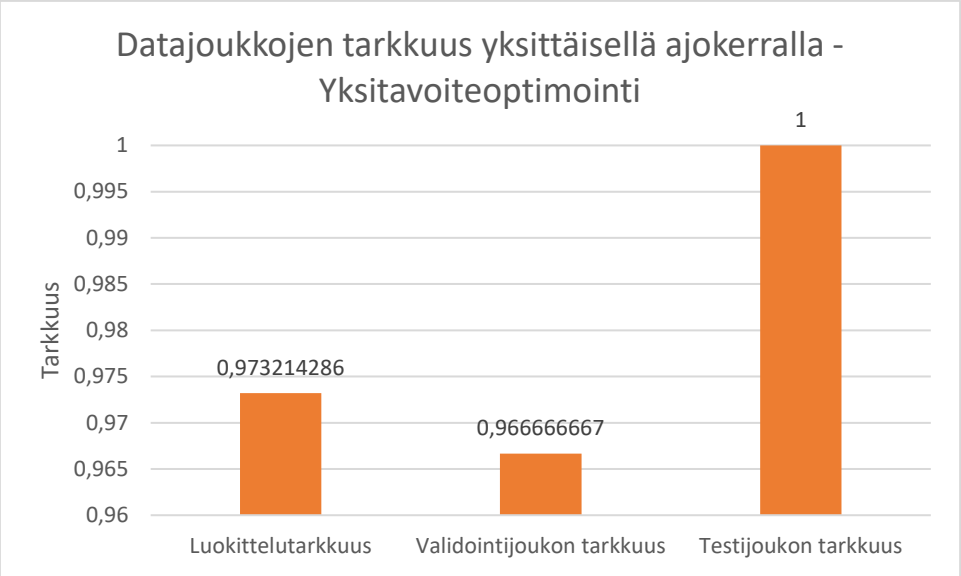
Kuviosta 20 voidaan havaita, että keskihajonta molemmilla optimointimenetelmillä on kohtuullisen pientä, joten algoritmien tulokset ovat hyvin toistettavia. Monitavoiteoptimoinnin keskihajonta on pienempi kuin yksitavoiteoptimoinnin. Pienempi keskihajonta tarkoittaa parempaa tulosta luotettavuuden kannalta, koska saadut tulokset vaihtelevat vähemmän ajokertojen välillä. Suurempi keskihajonta taas vähentäisi luotettavuutta. Täten näihin keskihajonnan tuloksiin perustuen voidaan todeta, että monitavoiteoptimointi on luotettavampi optimointimenetelmä kuin yksitavoiteoptimointi. Ero keskihajonnassa optimointimenetelmien kesken on noin 30 % monitavoiteoptimoinnin hyväksi.



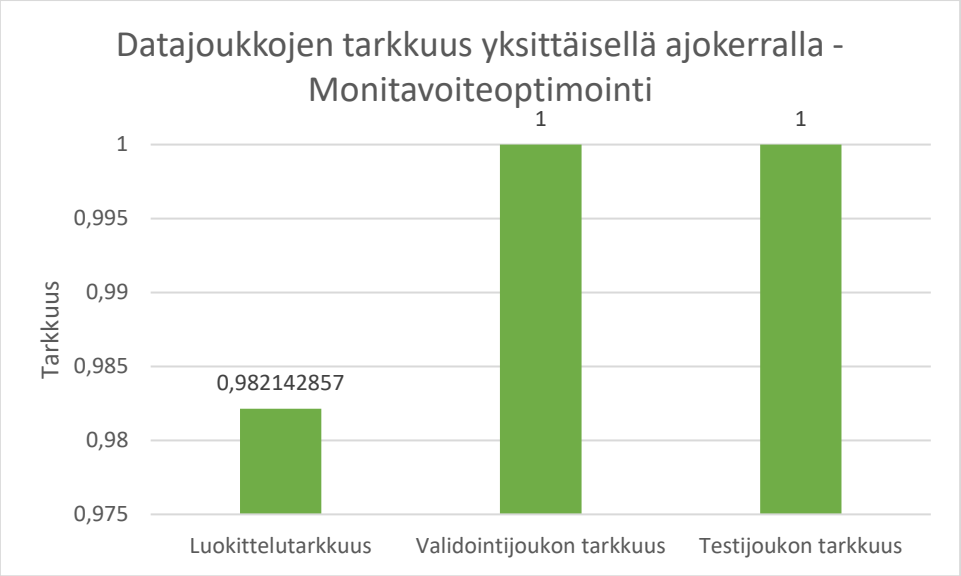
Kuvio 21. Yksi- ja monitavoiteoptimointien lopputuloksen keskiarvo

Kuviosta 21 voidaan tarkastella luokittelutarkkuuden lopputulosten keskiarvoa molemmilla optimointimenetelmillä. Voidaan heti huomata, että monitavoiteoptimoinnin tulosten keskiarvo on isompi, joten se on tässä tilanteessa täten parempi. Suurempi luokittelutarkkuuden keskiarvo tarkoittaa suurempaa luokittelutarkkuutta neuroverkon koulutusprosessissa. Yksitavoiteoptimoinnin osalta luokittelutarkkuuden keskiarvo on 0,9771, kun taas monitavoiteoptimoinnin osalta luokittelutarkkuuden keskiarvo on 0,9777. Täten tulokset puhuvat monitavoiteoptimoinnin paremmuuden puolesta, vaikkakin hyvin marginaalisesti.

Seuraavaksi tarkastellaan molempien optimointimenetelmien luokittelutarkkuutta, opetusjoukon virhettä ja validoinnin virhettä yksittäisestä satunnaisesti valitusta ajokerrasta. Ajokerroiksi valikoituivat yksitavoiteoptimoinnissa ajo 21 ja monitavoiteoptimoinnissa ajo 37. Kuvioissa 22 ja 23 on esitetty datajoukkojen tarkkuus yksi- ja monitavoiteoptimoinnille.



Kuvio 22. Datajoukkojen tarkkuus – Yksitavoiteoptimointi

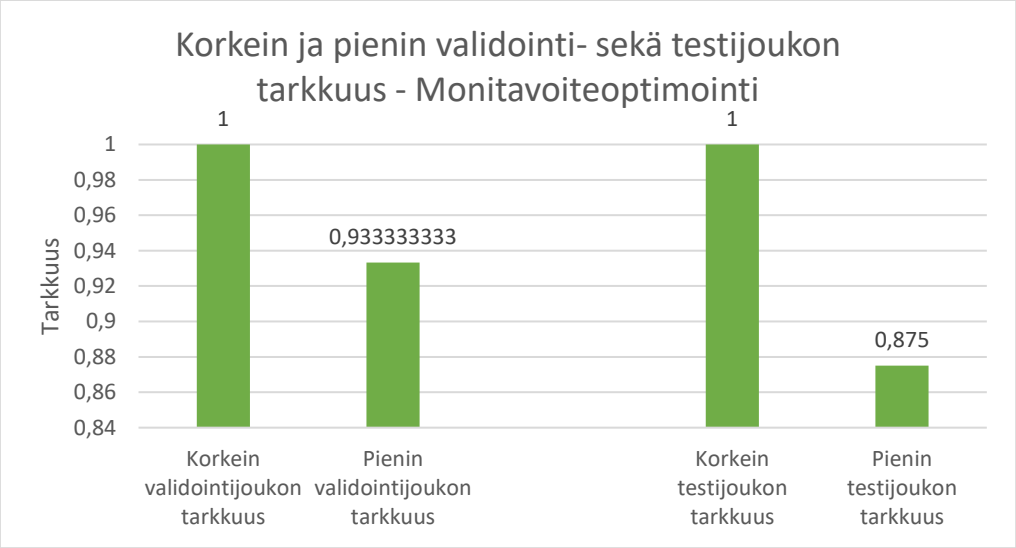


Kuvio 23. Datajoukkojen tarkkuus yksittäisellä ajokerralla - Monitavoiteoptimointi

Kuvioista 22 ja 23 voidaan huomata, että monitavoiteoptimointi saa paremman luokittelutarkkuuden arvon. Validointijoukon tarkkuus on myös parempi monitavoiteoptimoinnilla, koska se saa arvon 1 ja yksitavoiteoptimoinnin validointijoukon tarkkuus saa arvon 0,966. Testijoukon tarkkuus on molemmilla

optimointimenetelmillä 1. Seuraavaksi tarkastellaan validointi ja testijoukon korkeinta ja huonointa tarkkuutta molemmilla optimointimenetelmillä.

Kuvio 24. Korkein ja pienin validointi- sekä testijoukon tarkkuus – Yksitavoiteoptimointi



Kuvio 25. Korkein ja pienin validointi- sekä testijoukon tarkkuus – Monitavoiteoptimointi

Kuvioista 24 ja 25 voidaan lukea korkein ja pienin validointi- sekä testijoukon tarkkuus molemmille optimointimenetelmille. Voidaan huomata, että molempien menetelmien

osalta tulokset ovat samoja tässä kuvaajassa. Korkein arvo validointi- sekä testijoukon tarkkuudelle molemmilla menetelmillä on 1. Pienin validointijoukon tarkkuus molemmilla menetelmillä on 0,933. Pienin testijoukon tarkkuus molemmilla menetelmillä on 0,875. Taulukoidaan vielä tarvittava evaluaatioiden määrä vähintään 80 % luokittelutarkkuuden saavuttamiseen.

Taulukko 6. Evaluaatioiden määrä 80 % luokittelutarkkuuden saavuttamiseksi - Yksitavoiteoptimointi

Ajo - Yksitavoiteoptimointi	Luokittelutarkkuus	Evaluaatiot
1	0,857143	1050
25	0,892857	50
50	0,910714	600
75	0,883929	1000
100	0,830357	1050

Taulukko 7. Evaluaatioiden määrä 80 % luokittelutarkkuuden saavuttamiseksi - Monitavoiteoptimointi

Ajo - Monitavoiteoptimointi	Luokittelutarkkuus	Evaluaatiot
1	0,857143	600
25	0,901786	400
50	0,857143	1600
75	0,830357	600
100	0,883929	1600

Taulukoista 6 ja 7 voidaan lukea yksittäisiltä ajokerroilta evaluaatioiden vaaditusta lukumäärästä vähintään 80 % luokittelutarkkuuden saavuttamiseen. Ohjelma toteutettiin niin, että se tallettaa evaluaatioiden määrän aina, kun luokittelutarkkuuden arvo muuttuu. Yllä esitetyt luokittelutarkkuuden arvot ovat ensimmäisiä yli 80 % luokittelutarkkuuden arvoja tietyiltä ajokerroilta. Tähän taulukointiin valittiin viisi ajokertaa, jotka olivat molemmilla optimointimenetelmillä ajot 1, 25, 50, 75 ja 100.

Voidaan huomata, että monitavoiteoptimoinnin ollessa kompleksisempi algoritmi, se ei kuitenkaan vaadi huomattavasti enempää tai se vaatii jopa vähemmän evaluaatioita 80 % luokittelutarkkuuden saavuttamiseen. Tulosten käsittely ja tulkinta jatkuu luvussa 6.3, jossa summataan yhteen saatuja tuloksia ja niistä tehtyjä päätelmiä.

6.3 Tulosten tulkinta

Tässä luvussa luodaan yleiskatsaus saatuihin tuloksiin ja niistä tehtäviin päätelmiin. Tuloksille asetettiin luvussa 6.1 kolme suorituskykymittaria, jotka ovat tehokkuus, luokittelutarkkuus ja luotettavuus. Alla olevassa taulukossa 4 on taulukoituna tämän tutkimuksen suorituskykymittarit ja niihin liittyvät suorituskyvyn arvioinnin kriteerit, joita hyödynnettiin esimerkiksi edellisessä luvussa 6.2. Lisäksi tehdään päätelmä siitä, kumpi optimointimenetelmä oli tämän suorituskykymittarin perusteella parempi. Taulukkoon on merkitty sen optimointimenetelmän osalta X-merkki, joka suoriutui kyseisen kriteerin osalta paremmin.

Taulukko 8. Optimointimenetelmien suoriutuminen

Suorituskykykriteerit	**Yksitavoiteoptimointi**	**Monitavoiteoptimointi**
Suoritusaika	**X**	
Funktion arvon kehitys (sukupolvet)		**X**
Funktion arvon kehitys (evaluaatiot)	**X**	
Luokittelutarkkuuden loppuarvo 100 ajolla		**X**
Luokittelutarkkuuden loppuarvon keskihajonta		**X**
Luokittelutarkkuuden loppuarvon keskiarvo		**X**

Taulukosta 8 voidaan huomata, että yksitavoiteoptimointi voittaa monitavoiteoptimoinnin vain suoritusajassa ja funktion kehityksen evaluaatioissa eli yksitavoiteoptimointi pärjää pienemmillä resursseilla kuin monitavoiteoptimointi. Monitavoiteoptimointi taas kykenee tehokkaampaan funktion arvon kehitykseen suhteessa sukupolvien määrään, vaikkakin tähän vaadittava laskentateho on isompi. Monitavoiteoptimointi saa myös paremman lopputuloksen, kun katsotaan 100 ajon loppuarvon graafia sekä monitavoiteoptimoinnin loppuarvojen keskiarvoa. Myös keskihajonnan osalta monitavoiteoptimointi suoriutuu paremmin eli sen loppuarvot eroavat vähemmän toisistaan kuin yksitavoiteoptimoinnin loppuarvot. Summataan seuraavaksi tulos tehokkuuden, luokittelutarkkuuden ja luotettavuuden kannalta. Alla on esitetty taulukko 9, johon on koottu suorituskykymittarit ja niiden alla paremmin suoriutunut optimointimenetelmä on merkitty taulukkoon X-merkillä.

Taulukko 9. Optimointimenetelmät suhteessa suorituskykymittareihin

Suorituskykymittari	**Yksitavoiteoptimointi**	**Monitavoiteoptimointi**
Tehokkuus	**X**	
Luokittelutarkkuus		**X**
Luotettavuus		**X**

Tehokkuuden osalta yksitavoiteoptimointi vaikuttaa paremmalta vaihtoehdolta, vaikka monitavoiteoptimointi pärjääkin sukupolvien määrässä mitatussa funktion arvon kehityksessä hyvin. Monitavoiteoptimoinnin suoritusaikaa voisi toki saada pienennettyä, jos populaation kokoa pienennetään. Luokittelutarkkuuden osalta monitavoiteoptimointi on parempi suoriutuja, koska se saa hieman parempia luokittelutarkkuuden arvoja. Monitavoiteoptimointi voittaa myös luotettavuudessa, koska sen tulokset ovat keskihajonnaltaan pienempiä ja näin ollen paremmin toistettavia sekä täten luotettavampia kuin yksitavoiteoptimoinnin tulokset. Jo näihin tuloksiin perustuen voidaan todeta, että lisätutkimukselle monitavoiteoptimoinnin käytöstä monikerroksisen perseptroniverkon koulutuksessa differentiaalievoluutiolla on tarvetta.

6.4 Monitavoiteoptimointi hakua ohjaavalla tavoitefunktiolla f_2

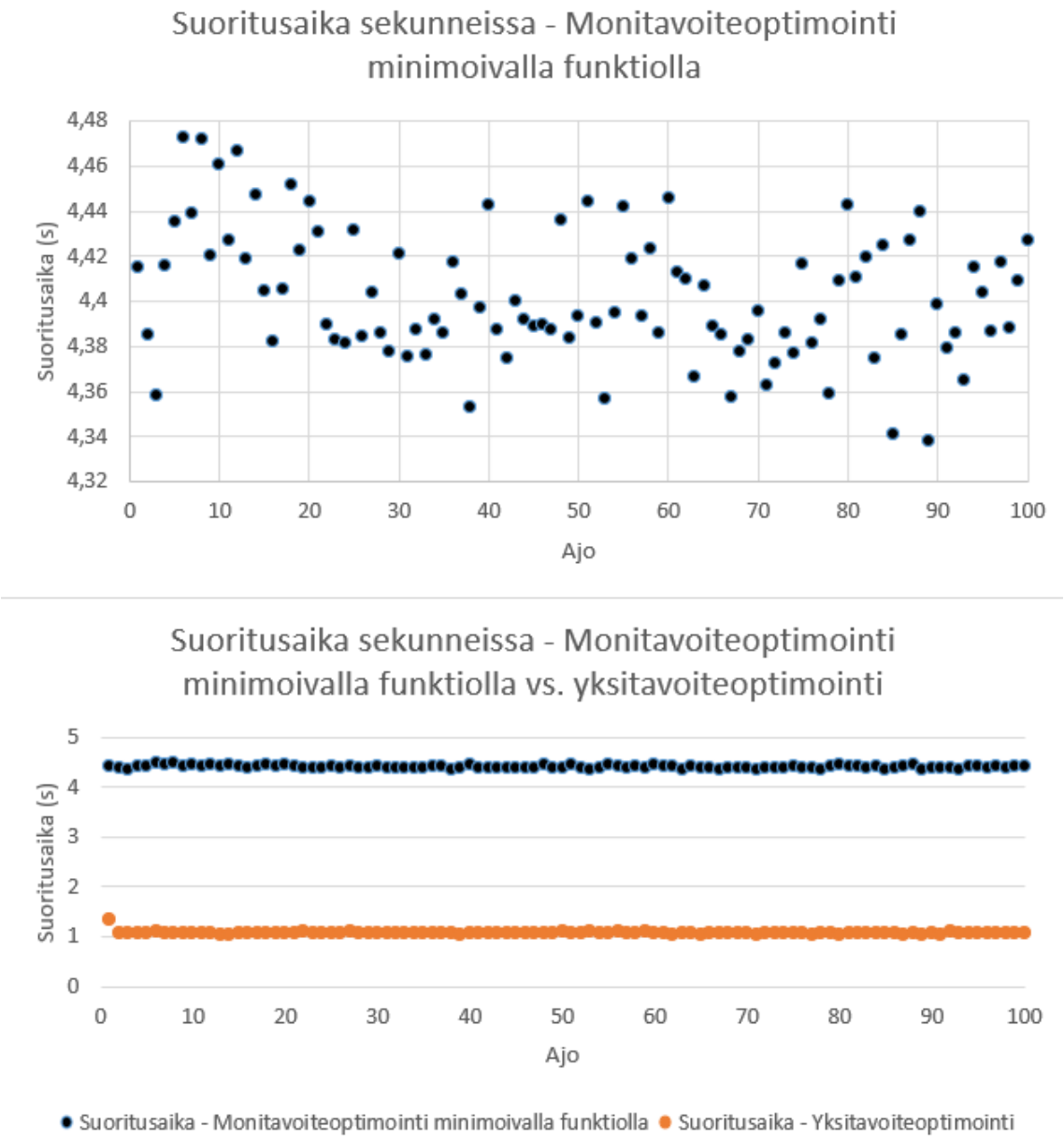
Tässä luvussa suoritetaan kokeet vielä toisella monitavoiteoptimoinnin toteutuksella. Tämä toteutus on tarpeellinen, jotta saadaan kattava kuva monitavoiteoptimoinnin toiminnasta differentiaalievoluutiolla. Saadut tulokset visualisoidaan sekä verrataan saatuja tuloksia yksitavoiteoptimoinnin tuloksiin. Softmax-tavoitefunktio vaihdetaan minimoivaan neuroverkon piilokerroksen aktivaatiosummia laskevaan tavoitefunktioon. Tämän kyseisen funktion tarkoitus on tasapainottaa saatua ratkaisua eli luokittelutarkkuutta, joka tarkoittaa sitä, että tämä funktio pyrkii eri tavoitteeseen, kuin luokittelutarkkuus. Funktion tavoitteena on ohjata ratkaisujen hakua oikeaan suuntaan. Kyseinen tavoitefunktio f_2 on määritelty alla (kaava 9).

$$f_2 = -\sum_{i=2}^{n} \min\left(S_i - S_{i-1}, 0\right)$$

Kaava 9. Hakua ohjaava funktio f_2

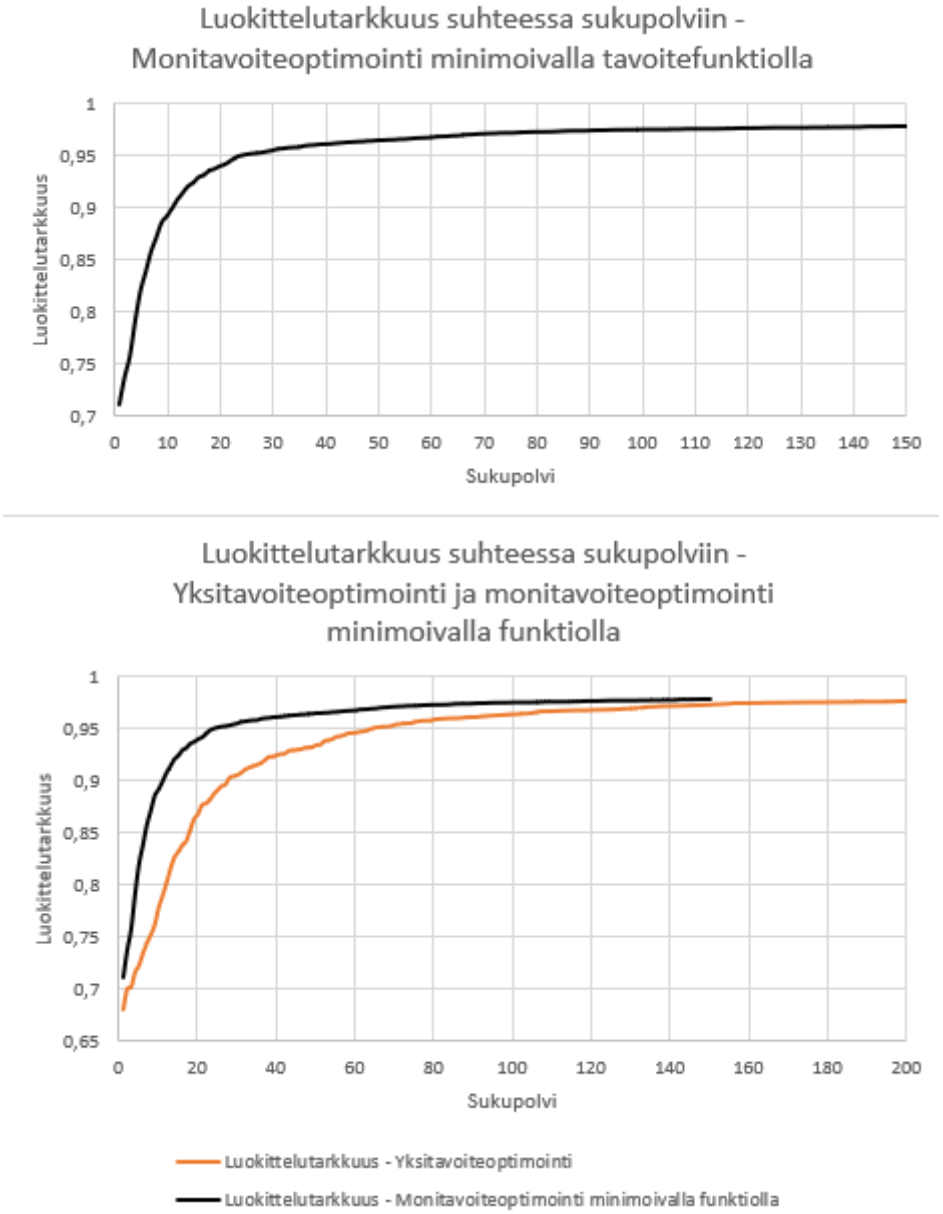
Missä n on piilokerroksen neuronien määrä ja S_i on piilokerroksen i neuronien aktivaatiosumma.

Tarkastellaan aluksi suoritusaikaa sekä verrataan sitä yksitavoiteoptimoinnin suoritusaikaan. Suoritusaika on esitetty alla kuviossa 26.



Kuvio 26. Suoritusaika – Monitavoiteoptimointi minimoivalla funktiolla

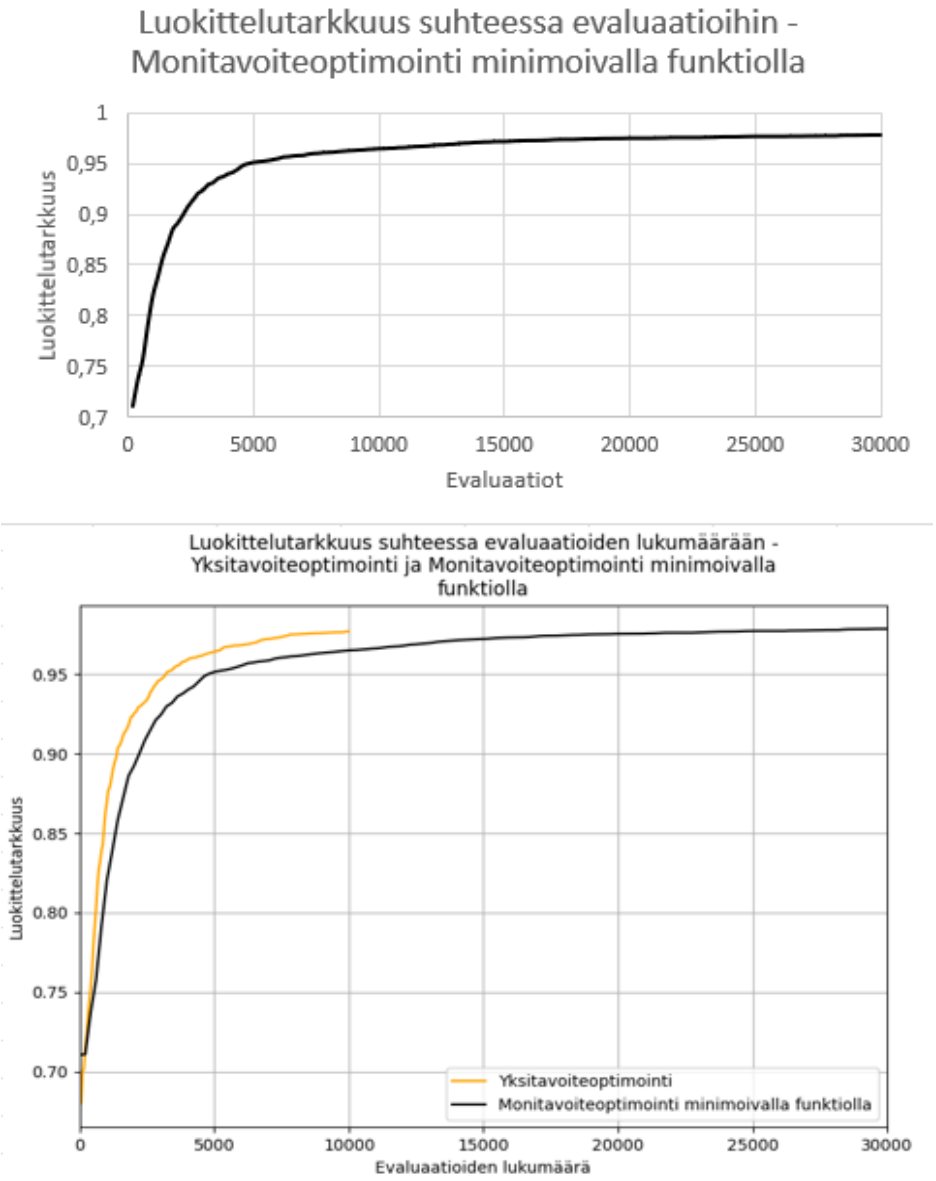
Kuviosta 26 voidaan huomata, että suoritusaika monitavoiteoptimoinnilla funktiolla f_2 vaihtelee välillä 4,32–4,48 sekuntia. Suoritusaika on siis hieman pienempi kuin monitavoiteoptimoinnilla Softmax-funktiolla. Voidaan myös huomata, että suoritusaika on noin 4 kertaa suurempi kuin yksitavoiteoptimoinnilla. Tarkastellaan seuraavaksi luokittelutarkkuutta suhteessa sukupolviin.



Kuvio 27. Luokittelutarkkuus suhteessa sukupolviin - Monitavoiteoptimointi minimoivalla funktiolla

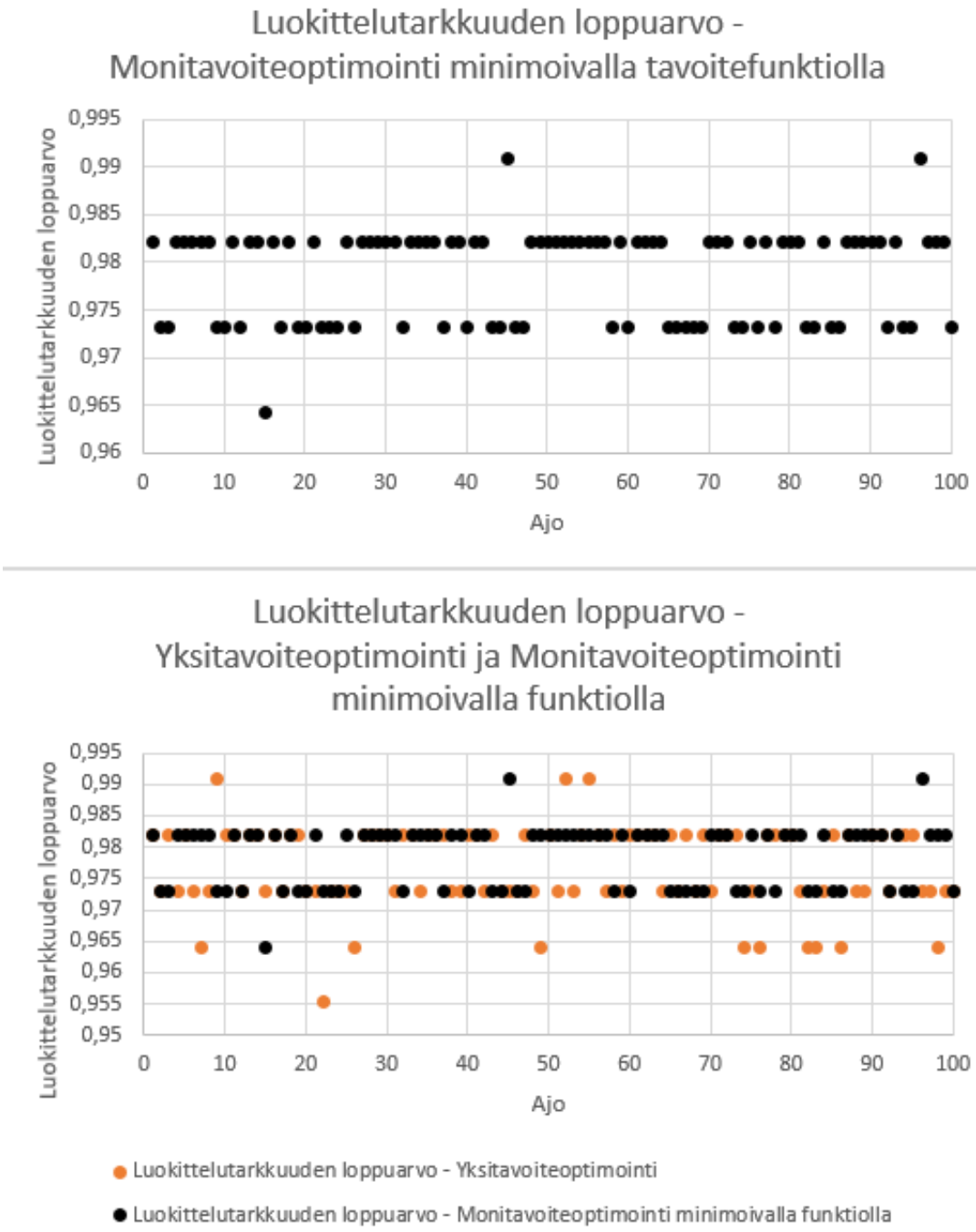
Kuviossa 27 on laskettu sukupolvikohtainen keskiarvo kaikilta 100 ajokerralta ja muodostettu siitä keskiarvoinen kehityskäyrä optimoinnille. Kuviosta nähdään, että luokittelutarkkuus saa heti ensimmäisellä sukupolvella yli 0,7 tarkkuuden. Jo ennen 30. sukupolvea saavutetaan luokittelutarkkuuden arvo 0,95, joka on jo hyvä luokittelutarkkuus. Keskiarvoisesti päädytään noin 0,98 luokittelutarkkuuteen 150 sukupolven jälkeen. Voidaan myös huomata, että monitavoiteoptimointi funktiolla f_2

saavuttaa paremman luokittelutarkkuuden arvon nopeammin sukupolviin nähden kuin yksitavoiteoptimointi. Nähdään, että yksitavoiteoptimointi vaatii 0,95 luokittelutarkkuuden saavuttamiseksi yli 60 sukupolvea, kun taas monitavoiteoptimointi funktiolla f_2 saavuttaa sen alle puolella sukupolvien määrällä yksitavoiteoptimointiin verrattuna. Seuraavaksi tarkastellaan luokittelutarkkuuden arvon kehitystä keskiarvoisesti suhteessa evaluaatioiden lukumäärään.



Kuvio 28. Luokittelutarkkuus suhteessa evaluaatioihin - Monitavoiteoptimointi minimoivalla funktiolla

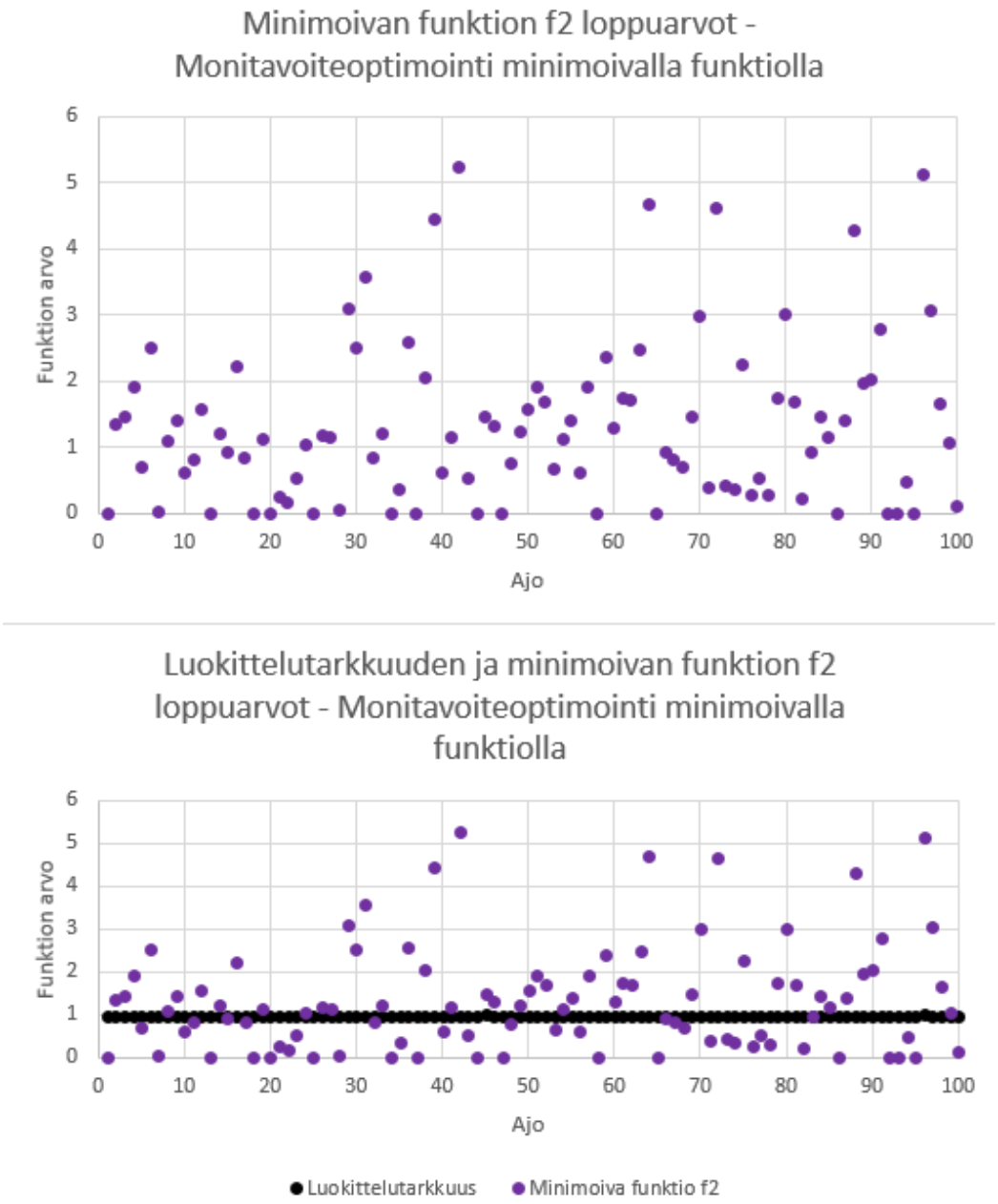
Kuviosta 28 nähdään, että monitavoiteoptimointi funktiolla f_2 saavuttaa 0,95 luokittelutarkkuuden noin 5000 funktion evaluaatiolla. 0,8 luokittelutarkkuus saavutetaan jo vajaassa 1000 evaluaatiossa. Yksitavoiteoptimointi saavuttaa nämä tarkkuudet nopeammin evaluaatioihin nähden. Yksitavoiteoptimoinnilla 0,8 luokittelutarkkuus saadaan jo reilulla 500 evaluaatiolla. 0,95 luokittelutarkkuus taas saavutetaan noin 3000 evaluaatiolla. Seuraavaksi katsotaan luokittelutarkkuuden loppuarvoja.



Kuvio 29. Luokittelutarkkuuden loppuarvo - Monitavoiteoptimointi minimoivalla funktiolla

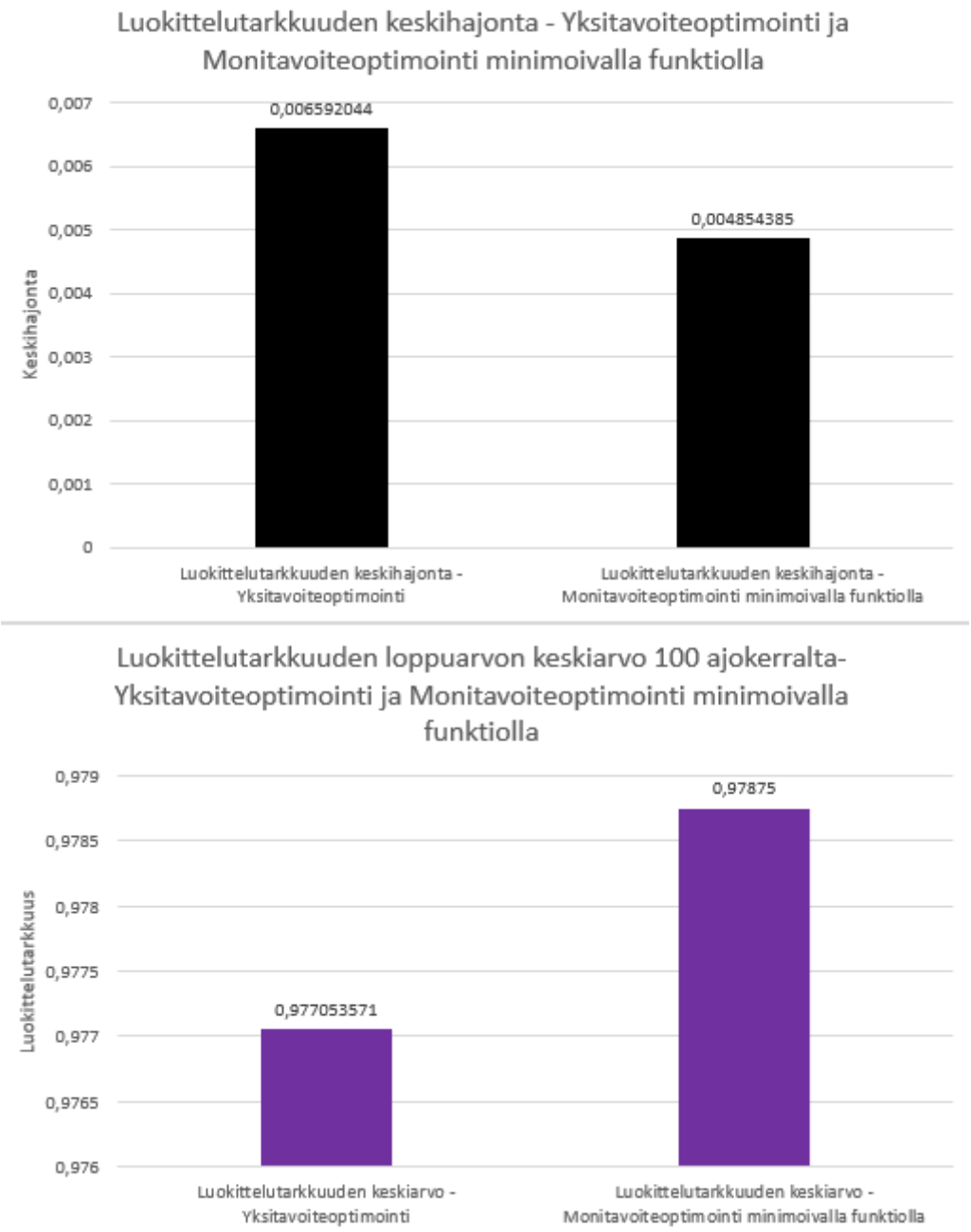
Kuviosta 29 nähdään, että luokittelutarkkuus saa pääasiassa kahta loppuarvoa. Nämä arvot ovat 0,9732 ja 0,9821. Kahdella ajokerralla saavutetaan jopa yli 0,99 luokittelutarkkuus, joka todennäköisesti on luokittelutarkkuus 1. Tämä pieni virhe laskennasta johtuu luultavasti ohjelman liukulukujen laskennan tietotyypistä. Kahdella

ajokerralla saavutetaan todennäköisesti siis paras mahdollinen luokittelutarkkuus. Yksitavoiteoptimointi kykenee tähän korkeaan luokittelutarkkuuteen myös kolmella ajokerralla. Muuten yksitavoiteoptimointi saa myös pääasiassa luokittelutarkkuuden arvoja 0,9732 ja 0,9821. Yksitavoiteoptimoinnilla on toki suurempaa hajontaa alaspäin. Tarkastellaan seuraavaksi funktion f_2 loppuarvoja.



Kuvio 30. Minimoivan funktion f2 loppuarvot

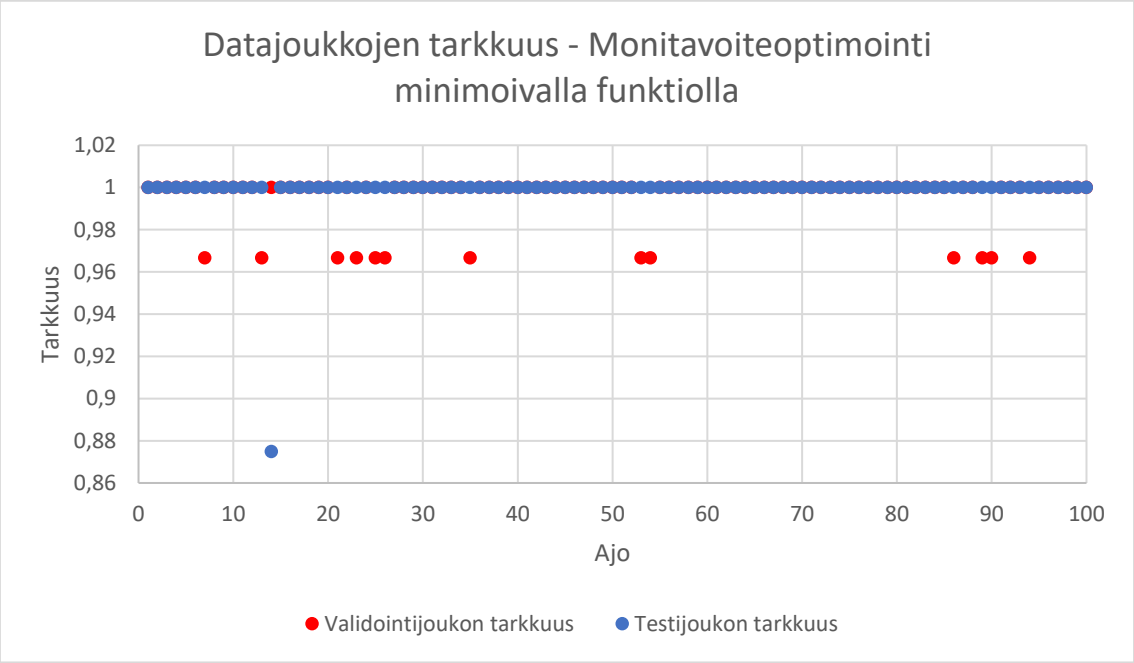
Kuviosta 30 voidaan nähdä, että funktio f_2 saa karkeasti arvoja väliltä 0–5,5. Loppuarvojen hajonta on siis suuri. Kuviosta voidaan myös huomata funktion f_2 arvot verrattuna luokittelutarkkuuden arvoihin nähden ja todeta, että funktion f_2 arvojen hajonta on erittäin suuri verrattuna luokittelutarkkuuden arvoihin. Algoritmi pyrkii minimoimaan funktion f_2 arvoa ja arvo 0 saavutetaan 17 eri ajokerralla. Katsotaan seuraavaksi luokittelutarkkuuden keskihajontaa funktiolla f_2.



Kuvio 31. Luokittelutarkkuuden keskihajonta ja keskiarvo - Monitavoiteoptimointi minimoivalla funktiolla

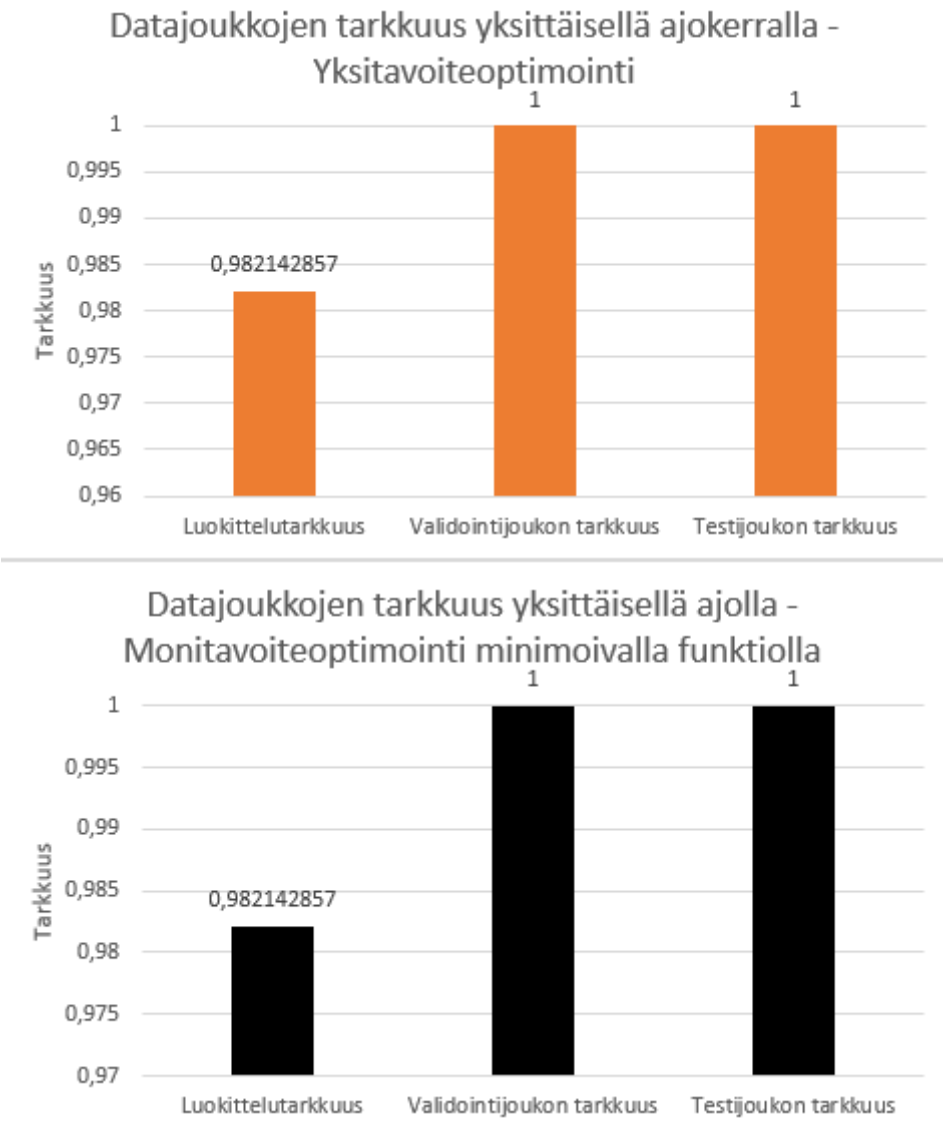
Kuviosta 31 voidaan huomata, että keskihajonta monitavoiteoptimoinnilla funktiolla f_2 on myös hyvin pientä, joten tulokset vaikuttavat luotettavilta sekä ne ovat toistettavia. Luokittelutarkkuuden keskihajonta monitavoiteoptimoinnilla funktiolla f_2 on myös pienempää yksitavoiteoptimoinnilla. Myös monitavoiteoptimointi funktiolla f_2

saavuttaa keskiarvoisesti paremman luokittelutarkkuuden loppuarvon, vaikkakin marginaalisesti. Tarkastellaan sitten datajoukkojen tarkkuutta.



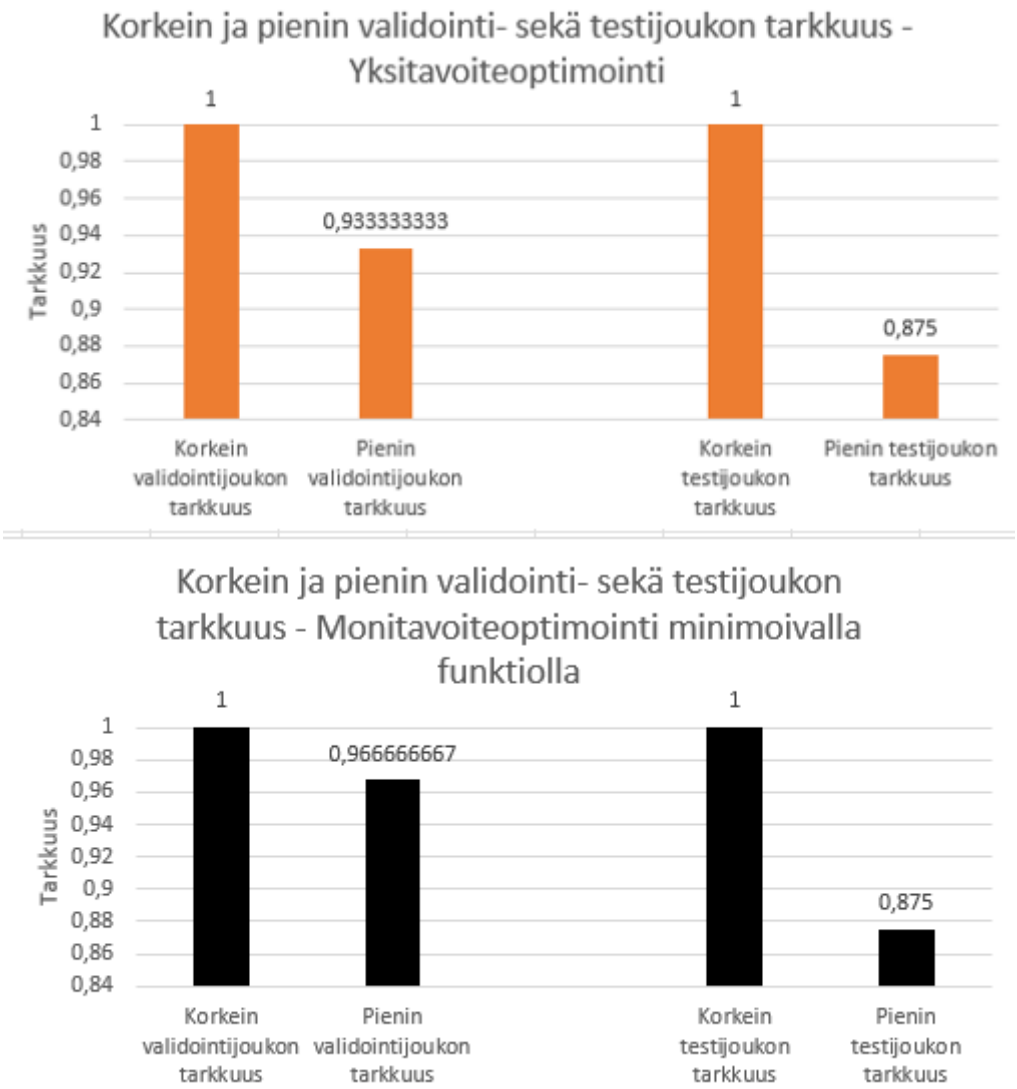
Kuvio 32. Datajoukkojen tarkkuus - Monitavoiteoptimointi minimoivalla funktiolla

Kuviosta 32 voidaan lukea validointijoukon ja testijoukon tarkkuus. Huomataan, että molemmat datajoukot saavat pääsääntöisesti tarkkuuden arvoa 1 eli data luokitellaan oikein. Validointijoukossa on pientä hajontaa, koska osalla ajoista saavutetaan tarkkuus 0,9667. Testijoukossa on vain yksi ajokerta, joka poikkeaa arvosta 1 ja tämä on ajo 14 tarkkuuden arvolla 0,875. Datajoukot saavuttavat siis korkeita tarkkuuden arvoja myös monitavoiteoptimoinnilla funktiolla f_2 . Tarkastellaan seuraavaksi datajoukkojen tarkkuutta yksittäisillä ajokerroilla.



Kuvio 33. Datajoukkojen tarkkuus yksittäisellä ajokerralla - Monitavoiteoptimointi minimoivalla funktiolla

Kuviosta 33 voidaan lukea datajoukkojen tarkkuuksia yksittäisillä satunnaisesti valituilla ajokerroilla. Yksitavoiteoptimoinnille käytettiin ajokertaa 21 ja monitavoiteoptimoinnille funktiolla f_2 käytettiin ajokertaa 68. Voidaan huomata, että molempien optimointimenetelmien kuvaajat ovat arvoiltaan identtiset. Molemmat saavuttavat siis jokaisessa datajoukossa samat arvot ja päätyvät samaan optimipisteeseen 0,982142857. Tässä pisteessä on todennäköisesti vain 1 datan piste väärin luokiteltuna. Tarkastellaan sitten korkeinta ja pienintä validointi- sekä testijoukon tarkkuutta.



Kuvio 34. Korkein ja pienin validointi- sekä testijoukon tarkkuus - Monitavoiteoptimointi minimoivalla funktiolla

Kuviosta 34 voidaan lukea korkein ja pienin validointi- sekä testijoukon tarkkuus monitavoiteoptimoinnille funktiolla f_2. Voidaan huomata, että monitavoiteoptimointi saa parempia validointijoukon arvoja kuin yksitavoiteoptimointi, sillä sen validointijoukon pienin arvo on suurempi. Testijoukon osalta korkein ja pienin tarkkuus ovat identtisiä molemmilla menetelmillä. Pienempien tarkkuuksien osalta voidaan myös todeta, että tarkkuudet ovat kohtuullisen korkeita. Tarkastellaan sitten taulukoituna sitä, miten nopeasti monitavoiteoptimointi funktiolla f_2 saavuttaa 80 % luokittelutarkkuuden evaluaatioihin nähden.

Taulukko 10. Evaluointien määrä 80 % luokittelutarkkuuden saavuttamiseksi - Monitavoiteoptimointi minimoivalla funktiolla

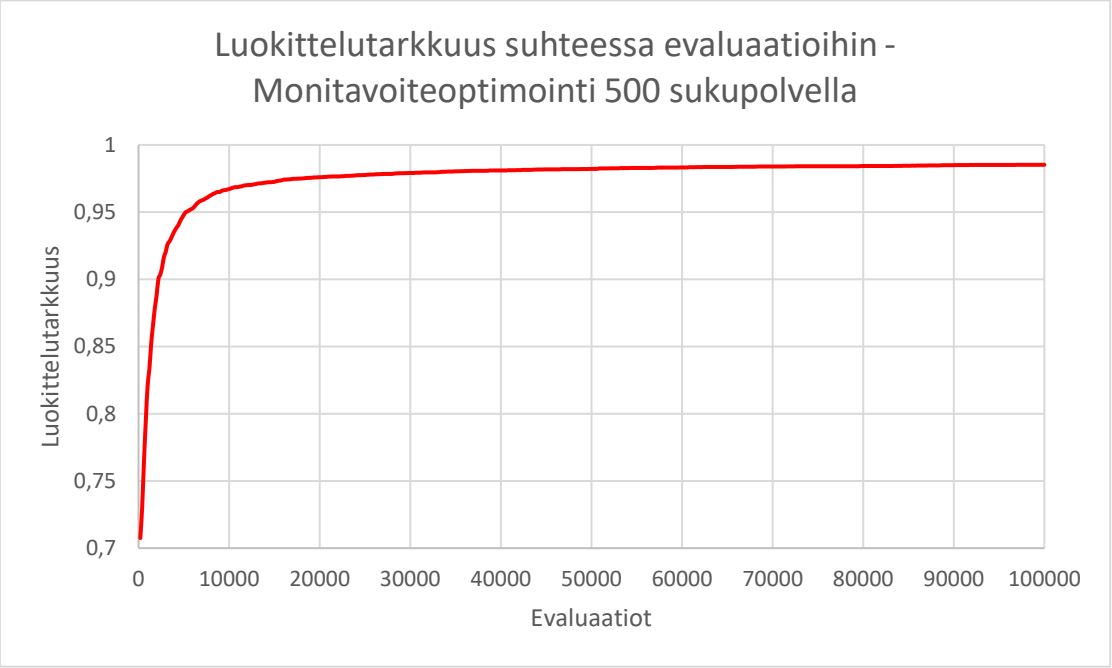
Ajo – Monitavoiteoptimointi minimoivalla funktiolla	Luokittelutarkkuus	Evaluaatiot
1	0,803571	1800
25	0,848214	600
50	0,883929	200
75	0,821429	600
100	0,875	1000

Taulukosta 10 voidaan lukea 80 % luokittelutarkkuuteen vaadittavia evaluaatioiden määriä eri ajokerroilla. Tähän valittiin myös ajot 1, 25, 50, 75 ja 100, kuten aiemmassakin optimointimenetelmien vertailussa. Tämä taulukon perusteella vaikuttaisi siltä, että monitavoiteoptimointi funktiolla f_2 saavuttaa 80 % luokittelutarkkuuden pienemmillä evaluaatioiden määrillä kuin monitavoiteoptimointi Softmax-funktiolla. Erot ovat myös kohtalaisen pieniä yksitavoiteoptimointiin nähden, joten laskennallisesti tämän menetelmän käyttäminen voi olla kannattavaa paremman luokittelutarkkuuden saavuttamiseksi.

Tulosten perusteella voidaan todeta sama kuin monitavoiteoptimoinnissa Softmax-funktiolla eli yksitavoiteoptimointi on suoritusajaltaan tehokkaampi optimointimenetelmä kuin monitavoiteoptimointi funktiolla f_2. Toisaalta suoritusaikaa olisi voitu pienentää pienentämällä esimerkiksi populaation kokoa. Monitavoiteoptimointi funktiolla f_2 tuottaa kuitenkin luokittelutarkkuuden arvolta parempia sekä luotettavampia tuloksia. Luotettavuus perustuu pienempään keskihajontaan kuin yksitavoiteoptimoinnilla. Erot ovat kuitenkin melko marginaalisia. Tarkastellaan vielä luvussa 6.5, miten sukupolvien ja evaluaatioiden määrän lisääminen vaikuttaa monitavoiteoptimoinnin tuloksiin funktiolla f_2.

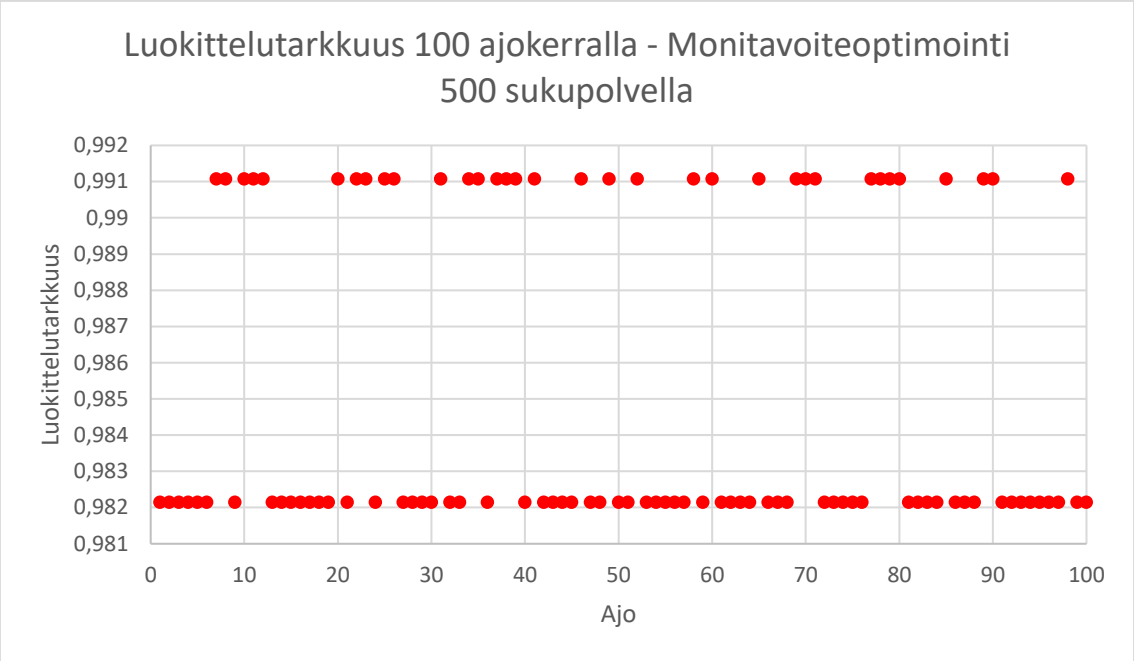
6.5 Monitavoiteoptimointi hakua ohjaavalla funktiolla f_2 500:lla sukupolvella

Tämän luvun tavoite on yksinkertaisesti vastata kysymykseen: paraneeko luokittelutarkkuuden arvo vielä 30000 evaluaation jälkeen? Monitavoiteoptimointi ajettiin läpi muuten samalla parametriyhdistelmällä kuin aiemmin, mutta sukupolvien määräksi asetettiin 500, jolloin tavoitefunktioita evaluoitiin 100000 kertaa. Tavoitefunktioina käytettiin aiempia luokittelutarkkuutta sekä funktiota f_2.



Kuvio 35. Luokittelutarkkuus suhteessa evaluaatioihin - Monitavoiteoptimointi 500 sukupolvella

Kuviossa 35 on esitetty luokittelutarkkuuden arvon kehitys keskiarvoisesti 100 ajokerralla suhteessa evaluaatioiden määrään. Kuvaajasta voidaan huomata, että luokittelutarkkuuden arvo paranee vielä 30000 evaluaation jälkeen. Luokittelutarkkuus saavuttaa lopulta arvon 0,9852. 30000 evaluaatiolla tulos oli 0,9788. Tulos 100000 evaluaatiolla on siis parempi kuin 30000 evaluaatiolla.



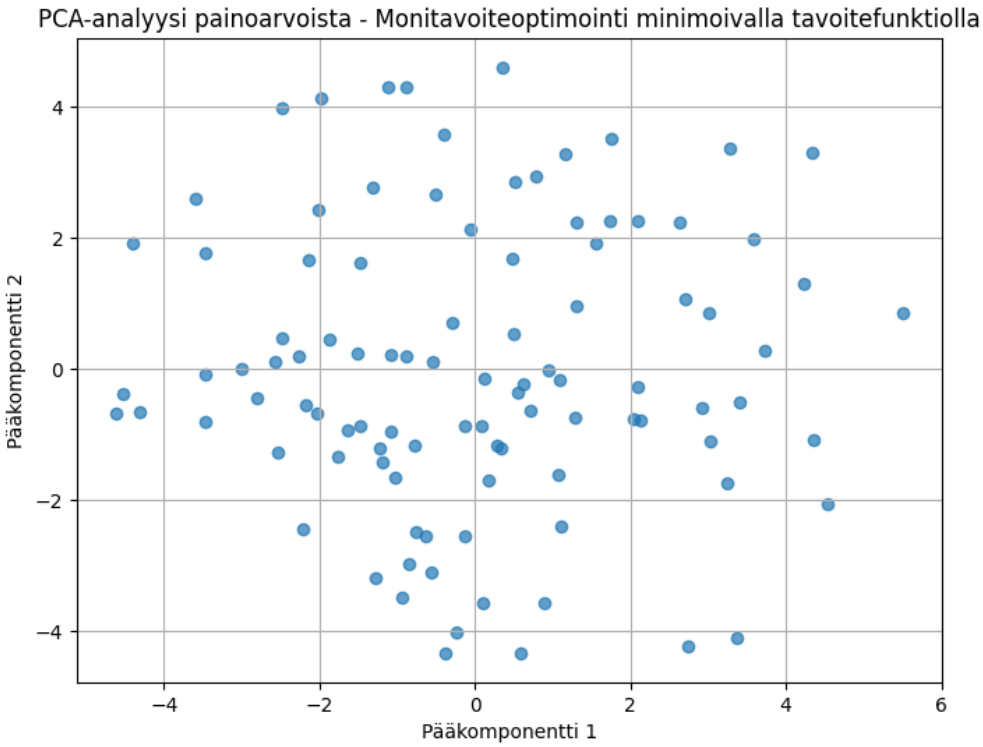
Kuvio 36. Luokittelutarkkuus 100 ajokerralla - Monitavoiteoptimointi 500 sukupolvella

Kuviosta 36 voidaan lukea luokittelutarkkuuden arvot 100 ajokerralla, kun monitavoiteoptimoinnissa käytettiin 500 sukupolvea ja 100000 funktioiden evaluaatiota. Huomataan, että luokittelutarkkuus saa vain kahta arvoa, jotka ovat 0,9821 sekä 0,9911. Tulos parani vielä siis entisestään, kun verrataan edellisiin monitavoiteoptimoinnin kokeisiin. Hajontaa tuloksissa esiintyy erittäin vähän, koska tulokset keskittyvät vain kahteen ratkaisuun. Voidaan todeta, että optimoinnin tulos paranee 30000 evaluaation jälkeen. Tehdään lopuksi vielä syväanalyysia tuloksille neuroverkon painoarvoilla sekä muodostetaan Pareto-rintama monitavoiteoptimoinnille.

6.6 Tulosten syväanalyysi

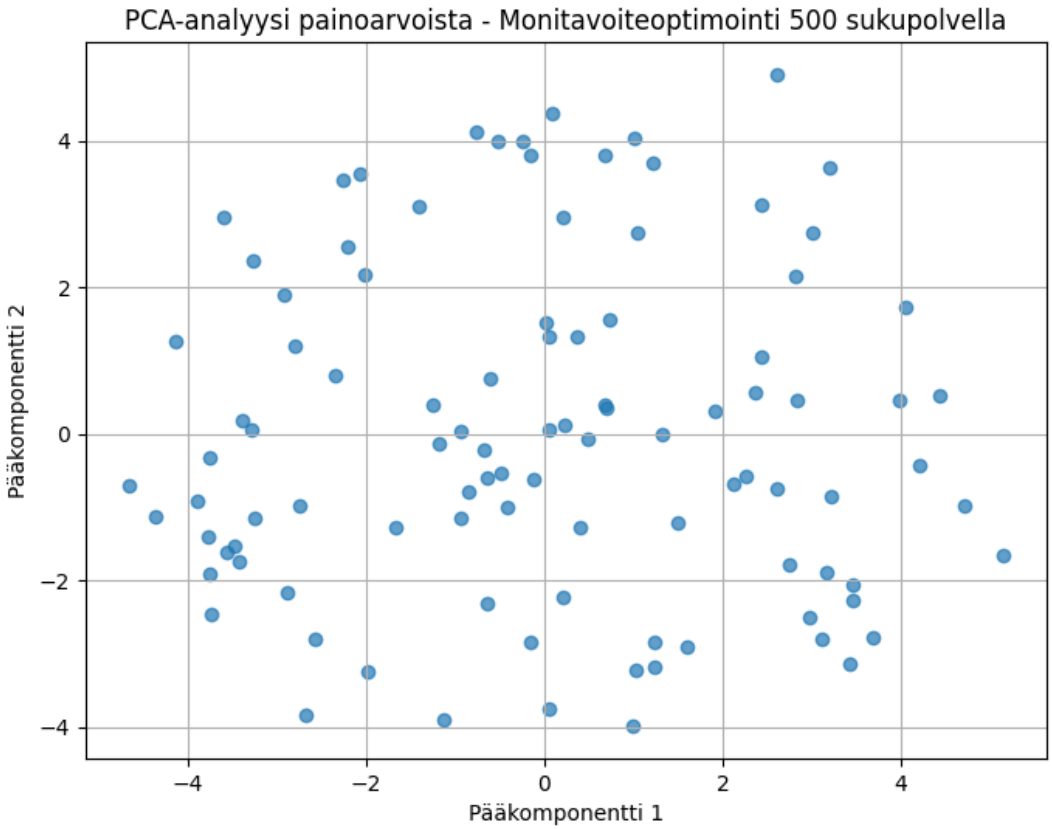
Tässä alaluvussa tehdään syvempää analyysia tutkimuksessa saaduille tuloksille kuvaajien muodossa. Ensiksi tehdään PCA-analyysi molemmille optimointimenetelmille. PCA-analyysin avulla yritetään selvittää mahdollisia rakenteellisia riippuvaisuuksia saaduissa verkon painoarvoissa. Lisäksi ajetaan monitavoiteoptimoinnin painoarvoille vielä Hungarian-algoritmi, jonka avulla voidaan löytää optimaalisempi järjestys

painoarvoille. Lopuksi muodostetaan Pareto-rintama monitavoiteoptimoinnille. Pareto-rintama pyrkii huomioimaan molemmat käytetyt tavoitefunktiot ja muodostamaan näistä pisteistä Pareto-rintaman.



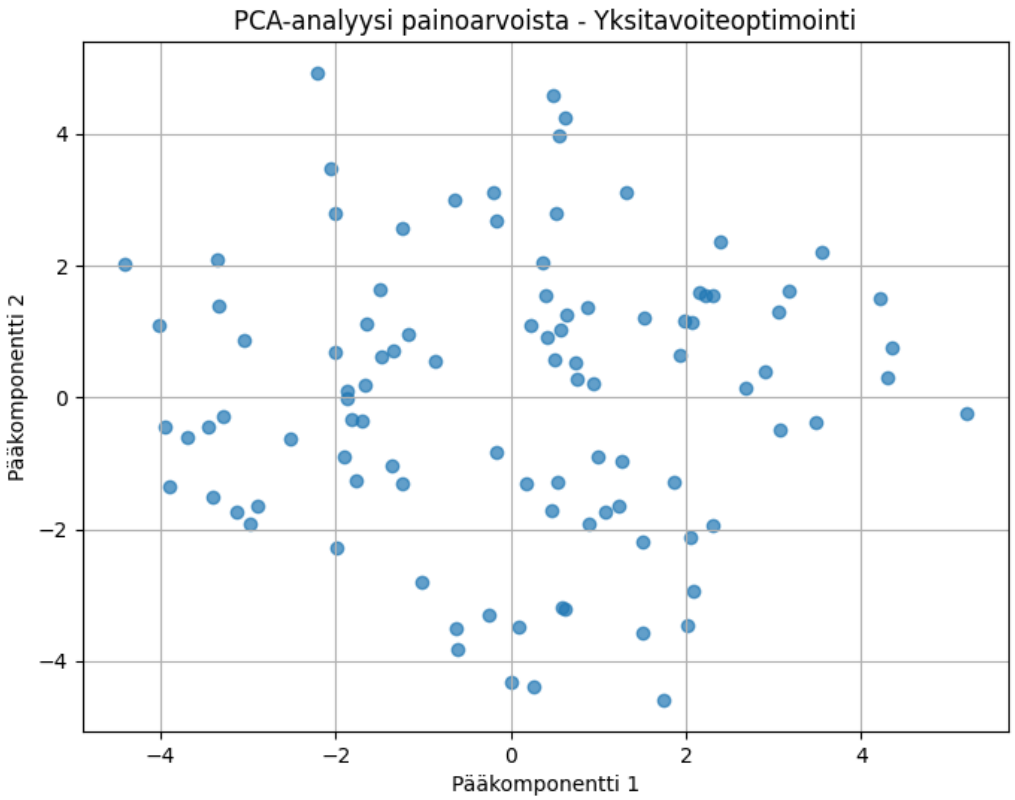
Kuvio 37. PCA-analyysi painoarvoista - Monitavoiteoptimointi minimoivalla funktiolla

Kuviosta 37 voidaan tarkastella monitavoiteoptimoinnin PCA-kuvaajaa. Kuvio on muodostettu neuroverkon painoarvojen avulla. Nähdään, että jotkin pisteistä muodostavat klustereita eli ne ovat lähellä toisiaan. Tämä viittaa siihen, että ne ovat jotenkin sidoksissa toisiinsa. Selkeää klusterirakennetta ei esiinny. Yleisesti pisteet ovat melko hajallaan, joka voi viitata siihen, että neuronien permutaatio-ongelma ei ole kovin vahvasti läsnä.



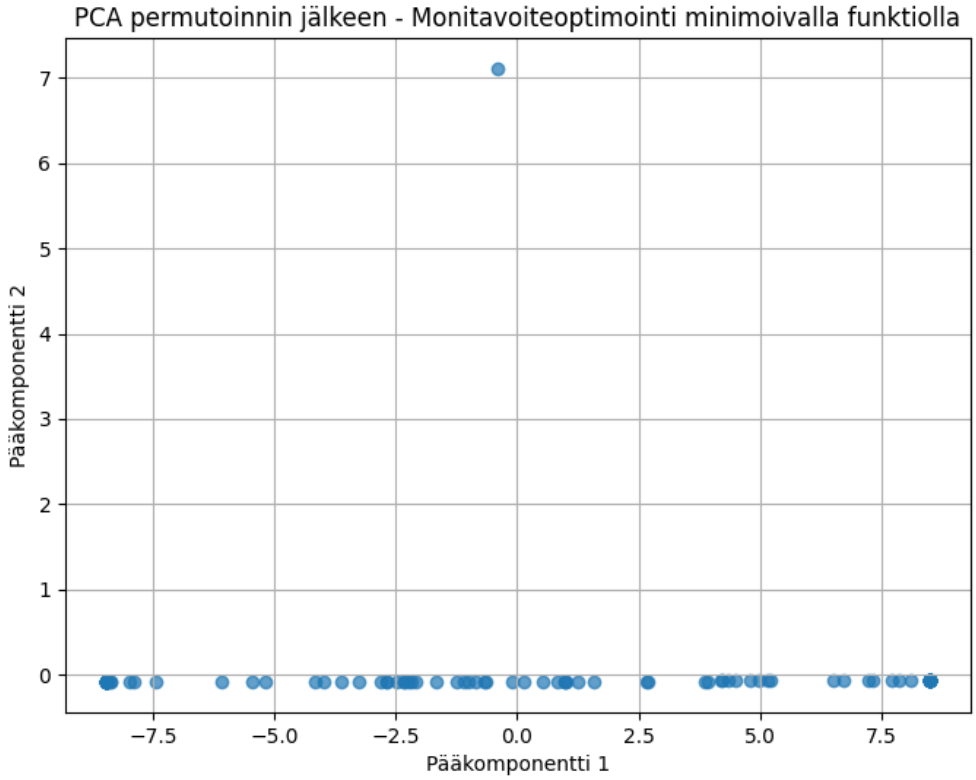
Kuvio 38. PCA-analyysi painoarvoista - Monitavoiteoptimointi 500 sukupolvella

Kuviosta 38 voidaan huomata, että data on hajanaisempaa kuin monitavoiteoptimoinnin minimoivan tavoitefunktion PCA-analyysissä. Sukupolvien määrän lisääntyminen ei kuitenkaan merkittävästi muuttanut PCA:n rakennetta ja ratkaisuavaruus on edelleen kohtuullisen laaja. Monitavoiteoptimointi 500 sukupolvella päätyi vain kahteen optimiratkaisuun, mutta PCA:n perusteella painoarvojakaumassa on vaihtelua ajokertojen välillä. Tämä voi johtua verkon erilaisista permutaatioista, vaikka loppuratkaisu olisikin sama.



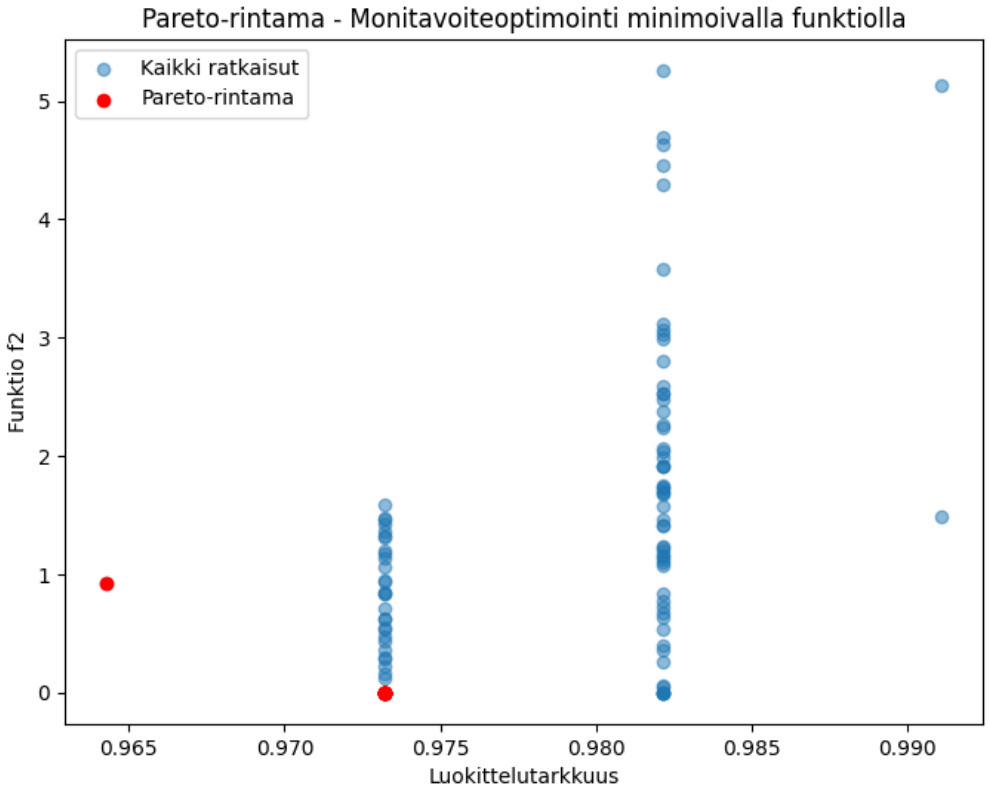
Kuvio 39. PCA-analyysi painoarvoista - Yksitavoiteoptimointi

Kuviosta 39 voidaan nähdä PCA-analyysi yksitavoiteoptimoinnin painoarvoille. Klusteroitumista voidaan havaita hieman enemmän kuin monitavoiteoptimoinnilla. Näin ollen permutaatio-ongelma saattaa ainakin osassa tapauksista olla läsnä. Erot monitavoiteoptimointiin ovat kuitenkin melko pieniä. Kuvion perusteella voidaan todeta, että algoritmi saattaa jumittua joihinkin tiettyihin ratkaisuihin, joka voi olla viite ainakin osittaisesta permutaatio-ongelman läsnäolosta, mutta merkittävää eroa monitavoiteoptimointiin ei vaikuta olevan.



Kuvio 40. PCA-analyysi Hungarian-algoritmin jälkeen - Monitavoiteoptimointi minimoivalla funktiolla

Kuviosta 40 voidaan tarkastella PCA-analyysia monitavoiteoptimoinnille minimoivalla funktiolla, kun neuroverkon loppupainoarvoille on tehty Hungarian-algoritmilla uudelleen järjestely. Nyt vaikuttaa siltä, että tulokset ovat lineaarisia ja permutaatio-ongelmaa ei välttämättä juurikaan esiinny. Hungarian-algoritmi järjestelee verkon uudestaan löytääkseen optimaalisemman ratkaisun. Tulokset vaikuttavat olevan riippuvaisia toisistaan. Hungarian-algoritmi näyttää paljastavan erilaiset permutaatiot, joten voidaan todeta, että tutkimuksessa käytetty algoritmi ei löytänyt parasta mahdollista ratkaisua. Hungarian-algoritmi suoritettiin testimielessä, jotta nähdään, mihin datan lisäkäsittelyllä voidaan päätyä. Hungarian-algoritmia voisi jopa harkita lisättäväksi optimointialgoritmiin tulevaisuuden tutkimuksessa, jotta nähdään, mikä sen vaikutus on.



Kuvio 41. Pareto-rintama - Monitavoiteoptimointi minimoivalla funktiolla

Kuviossa 41 on esitetty Pareto-rintama monitavoiteoptimoinnille minimoivalla funktiolla. Pareto-rintama on esitetty punaisilla pisteillä. Nämä pisteet ovat ratkaisuja, joissa toisen funktion tavoitetta ei voi parantaa heikentämättä toisen funktion tavoitetta. Pareto-rintama löytyy luokittelutarkkuuden arvojen väliltä 0,960–0,975. Vaikuttaa siltä, että esimerkiksi luokittelutarkkuuden arvon 0,99 saavat ratkaisut saavutetaan sillä kustannuksella, että ratkaisusta tulee dominoituva muilla tavoitteilla. Tavoitefunktio f_2 vaikuttaa ohjaavan ratkaisua haluttuun suuntaan. Voidaan huomata, että Pareto-rintaman pisteet ovat keskittyneet alhaisiin funktion f_2 loppuarvoihin. Näissä pisteissä monitavoiteoptimointi vaikuttaa löytäneen ratkaisun, jossa molemmat tavoitefunktiot tulevat lähes yhtä hyvin huomioiduksi. Tulokset eivät kuitenkaan osoita, että monitavoiteoptimointi olisi selvästi tehokkaampi menetelmä kuin yksitavoiteoptimointi, vaikka marginaalisia eroja löytyykin monitavoiteoptimoinnin hyväksi.

7 Johtopäätökset

Tämän tutkimuksen tavoitteena oli selvittää, voiko DE-algoritmin suorituskykyä parantaa MLP-verkon koulutuksessa monitavoiteoptimoinnin avulla. Tavoitteena oli myös tutkia, ratkaiseeko monitavoiteoptimoinnin käyttö neuronien permutaatio-ongelman. Tutkimuskysymys muotoiltiin seuraavasti: “Miten monitavoiteoptimointi voi parantaa differentiaalievoluutioalgoritmin suorituskykyä monikerroksisen perseptroniverkon koulutuksessa?”. Tutkimuksessa toteutettiin Python-ohjelmointikielellä kokeelliset ohjelmistot MLP-verkon yksitavoiteoptimointiin sekä monitavoiteoptimointiin. Ohjelmistoja voidaan hyödyntää myös jatkossa ja kehittää edelleen. Kokeet ajettiin 100 kertaa molemmilla optimointimenetelmillä, jotta stokastisuus saataisiin minimoitua. Monitavoiteoptimoinnille käytettiin useampaa erilaista toteutusta. Tulokset osoittavat, että monitavoiteoptimointi tarjoaa joitakin etuja perinteisempään yksitavoiteoptimointimenetelmään verrattuna.

Tutkimuksen hypoteesi oli, että monitavoiteoptimointi voi tehostaa DE-algoritmia MLP-verkon koulutuksessa. Tämä hypoteesi voidaan todeta osittain oikeaksi, sillä monitavoiteoptimointi suoriutui yksitavoiteoptimointia paremmin luokittelutarkkuuden sekä luotettavuuden osalta. Tehokkuuden kannalta yksitavoiteoptimointi oli parempi, sillä sen suoritusaika oli pienempi sekä se vaati vähemmän tavoitefunktion evaluaatioita optimiarvon saavuttamiseksi. Tehokkuuden osalta monitavoiteoptimointi oli nopeampi saavuttamaan optimiratkaisun ainoastaan silloin, kun verrattiin luokittelutarkkuuden funktion arvon kehitystä läpi käytyihin DE-algoritmin sukupolviin. Monitavoiteoptimointi ei niinkään parantanut algoritmin tehokkuutta, vaan sen tuloksia ja tulosten luotettavuutta. Tutkimusongelmaa ei saatu ratkaistua täysin kokonaan, mutta tutkimuksesta saatiin vertailukelpoisia tuloksia ja ratkaisun kehittäminen tulevaisuudessa näin ollen helpottuu. Molemmilla optimointimenetelmillä päästiin hyvään luokittelutarkkuuteen ja keskihajontaan. Alla olevassa myös luvussa 6.3 esitetyssä taulukossa (taulukko 11) tiivistettynä optimointialgoritmien paremmuus suorituskykymittareiden suhteen. Paremmin suoriutunut optimointialgoritmi on merkitty X-merkillä.

Taulukko 11. Optimointimenetelmät suhteessa suorituskykymittareihin

Suorituskykymittari	Yksitavoiteoptimointi	Monitavoiteoptimointi
Tehokkuus	**X**	
Luokittelutarkkuus		**X**
Luotettavuus		**X**

Tutkimuksen tuloksia monitavoiteoptimoinnin osalta voidaan pitää hyvänä edistysaskeleena monikerroksisen perseptroniverkon optimoinnille DE-algoritmilla, mutta tulosten perusteella erot yksitavoiteoptimointiin verrattuna olivat jokseenkin marginaalisia. Monitavoiteoptimoinnissa käytettiin jo nyt kohtuullisen suurta DE:n populaation kokoa, jonka takia optimoinnin suorittaminen vei keskiarvoisesti kymmeniä minuutteja eri monitavoiteoptimoinnin toteutuksilla. Populaation koon kasvattaminen tästä suuremmaksi kasvattaisi suoritusaikaa jopa useisiin tunteihin, joten se rajoittaa ainakin peruslaitteistolla tehtävää lisätutkimusta. Jos monitavoiteoptimoinnin osalta saataisiin suoritusaikaa ja vaadittavia laskentaresursseja pienennettyä, optimoinnin suorituksessa voisi tapahtua mahdollisesti jopa merkittävää edistystä, sillä optimoinnissa voitaisiin käyttää suurempia DE:n parametreja. Myös laskentatehon kasvaminen ja näin ollen suoritusaikojen pienentyminen voisivat lisätä monitavoiteoptimoinnin soveltuvuutta. Funktion f_2 toteutuksen kontrolliparametrit eivät välttämättä olleet vertailukelpoisia muihin toteutuksiin nähden, joten tämä olisi hyvä varmistaa jatkotutkimuksessa optimaalisilla parametreilla.

Monitavoiteoptimoinnilla 500 sukupolvella päästiin kuitenkin jo huomattavan korkeisiin luokittelutarkkuuden arvoihin ja optimointiprosessi tuotti vain kahta ratkaisua 0,98 ja 0,99 sadalla ajokerralla. Tämä viestii erittäin korkeasta luokittelutarkkuudesta sekä algoritmin luotettavuudesta. Etenkin sukupolvien määrän ja populaation koon kasvattaminen voisi parantaa optimointiprosessin tuloksia vielä entisestään. Tämän tutkimuksen osalta tätä suuremmat parametrit tuskin tuovat

suoritusaikaan nähden enää tarpeeksi merkittävää etua niiden kasvattamisen perusteeksi. Pareto-rintaman ja Hungarian-algoritmin osalta voidaan huomata, että monitavoiteoptimointi vaikuttaa ohjaavan ratkaisua haluttuun suuntaan eli kohti yhtä globaalia optimiratkaisua. Tätä ratkaisua ei kuitenkaan vielä välttämättä saavutettu, mutta tutkimuksen tulokset ovat hyvä lähtökohta jatkotutkimukselle.

Neuronien permutaatio-ongelman osalta todettakoon, että monitavoiteoptimoinnin tulokset eivät anna suoria viitteitä siitä, että monitavoiteoptimoinnin käyttö vähentäisi merkittävästi tai poistaisi kokonaan neuronien permutaatio-ongelman ilmenemistä 30000 funktion evaluaatiolla. Myöskään 100000 funktion evaluaatiolla ei huomattu merkittävää parannusta permutaatio-ongelman vähenemisen kannalta. Toki monitavoiteoptimoinnin tulokset olivat mahdollisesti ainakin osittain lähellä globaalia optimiratkaisua, koska luokittelutarkkuuden arvot olivat erittäin korkeita ja lähellä arvoa 1. Yksitavoiteoptimointi jäi hieman alempiin luokittelutarkkuuden arvoihin, joka voi viitata yksitavoiteoptimoinnin jäävän joissain tapauksissa jumiin paikallisiin optimiarvoihin. Monitavoiteoptimointi päätyi kaikissa toteutuksissa lähinnä kahteen luokittelutarkkuuden arvoon, mutta nämä arvot saavutettiin kuitenkin vähintään hieman toisistaan poikkeavilla neuroverkkojen rakenteilla, joka taas viittaa ainakin vähäiseen permutaatio-ongelman läsnäoloon. Toisaalta luokittelutarkkuus on erittäin korkea, joka voi viitata siihen, että monitavoiteoptimointi saavuttaa luokittelutarkkuuden arvoja, jotka ovat ainakin lähellä globaalia optimiratkaisua. On kuitenkin myös hyvä muistaa verkon mahdollinen ylioppiminen luokittelutarkkuuden arvon tarkastelemisen osalta. PCA-analyysissä monitavoiteoptimoinnin painojakauma osoittautui hajanaisemmaksi yksitavoiteoptimointiin verrattuna. Monitavoiteoptimoinnin hajanaisempi painojakauma voi olla seurausta monitavoiteoptimoinnin tuottamasta laajemmasta ratkaisujoukosta, vaikka luokittelutarkkuuden arvojen hajonta olikin erittäin pientä. Tämä saattaa korostaa verkon rakenteellisia eroja PCA-analyysissä, vaikka todellisuudessa permutaatio-ongelma ei välttämättä olisi lisääntynyt tai se olisi jopa vähentynyt. Tulosten perusteella voidaan päätellä, että monitavoiteoptimointi ei välttämättä vähennä permutaatio-ongelman huonoja vaikutuksia, mutta se saattaa päätyä lähelle globaalia

optimiratkaisua. Monitavoiteoptimointi voi kuitenkin auttaa optimointiprosessia välttämään paikallisia optimeja, vaikka se ei ratkaisekaan permutaatio-ongelmaa täydellisesti. Permutaatio-ongelma ei kuitenkaan ole yksiselitteinen ja se voi johtua useista eri tekijöistä. Permutaatio-ongelmalle voi löytyä jatkotutkimuksessa myös muita ennalta tuntemattomia syitä.

Tulevaisuuden kannalta monitavoiteoptimoinnin lisätutkimus on tämän tutkimuksen tulosten perusteella suositeltavaa. Monitavoiteoptimointi vaikuttaa omaavan potentiaalia etenkin verkon luokittelutarkkuuden parantamisessa sekä tulosten luotettavuudessa. Myös permutaatio-ongelman osalta monitavoiteoptimoinnin hyödyntämistä on syytä tutkia lisää. Tämä tutkimus antoi joitakin viitteitä permutaatio-ongelman vähenemisestä monitavoiteoptimoinnissa vaikkei se suoranaisesti poistanutkaan permutaatio-ongelman ilmenemistä. Toki nämä seikat olisi hyvä varmistaa uudelleen erilaisella datakokoelmalla sekä optimoimalla parametriasetukset uudestaan. Erityisesti monitavoiteoptimointi voisi höytyä vielä suuremmasta populaation koosta sekä suuremmasta määrästä sukupolvia ja funktioiden evaluaatioita. Populaation koon kasvattaminen on suositeltavaa etenkin, mikäli käytössä on suuremmat laskentaresurssit, koska jo tässä tutkimuksessa monitavoiteoptimointi oli laskennallisesti huomattavasti raskaampi kuin yksitavoiteoptimointi. Toisaalta mikäli käytössä on samat laskentaresurssit, voisi populaation koon pienentäminen johtaa mahdollisesti parempaan monitavoiteoptimoinnin suoritusaikaan ja täten parantaa algoritmin tehokkuutta. Mikäli monitavoiteoptimointi saadaan kaikkien kolmen suorituskyvyn kriteerin kannalta paremmaksi kuin yksitavoiteoptimointi, on monitavoiteoptimoinnin laajempi käyttöönotto erittäin perusteltavaa. Monitavoiteoptimointi vaikuttaa nyt jo olevan yksitavoiteoptimointia lupaavampi optimointimenetelmä. Jatkossa voisi olla hyödyllistä tutkia monitavoiteoptimoinnin soveltamista myös monimutkaisempiin neuroverkkorakenteisiin. Lisäksi optimointiprosessin stokastisuutta voisi tarkastella syvällisemmin esimerkiksi tätä tutkimusta suuremmalla määrällä toistoja, joskin se vaatii suurempia laskentaresursseja suoritusajan pienentämiseksi. Myös tämän tutkimuksen tuloksia sekä dataa voidaan hyödyntää jatkotutkimuksessa.

Lähteet

Heiskanen, M. (2015). *Differentiaalievoluutioalgoritmin kontrolliparametrien valinta*. Noudettu 1.11.2024 osoitteesta https://osuva.uwasa.fi/bitstream/handle/10024/510/osuva_6603.pdf?sequence=1&isAllowed=y

Ilonen, J., Kamarainen, J.-K., & Lampinen, J. (2003). Differential Evolution Training Algorithm for Feed-Forward Neural Networks. *Neural Processing Letters, 17*(1), 93–105. Noudettu 22.11.2024 osoitteesta https://webpages.tuni.fi/vision/public_data/publications/NPL2003.pdf

Särkiniemi, E. (2020). *Itseoppivan tekoälyn kouluttaminen geneettisellä algoritmilla*. Kaakkois-Suomen ammattikorkeakoulu. Noudettu 22.11.2024 osoitteesta https://www.theseus.fi/bitstream/handle/10024/338838/Eero_Sarkiniemi.pdf?sequence=2

Vuontisjärvi, T. (2023). *Monitavoiteoptimointi*. Oulun yliopisto, Matemaattisten tieteiden yksikkö. Noudettu 23.11.2024 osoitteesta https://oulurepo.oulu.fi/bitstream/handle/10024/43010/nbnfioulu-202310183147.pdf?sequence=1

Juvonen, J. (2020). Neuroverkot taloudellisen aikasarjan ennustamisessa: Empiirinen tutkimus S&P 500 -indeksillä. Pro gradu -tutkielma. Turun kauppakorkeakoulu. Noudettu 27.11.2024 osoitteesta: https://www.utupub.fi/bitstream/handle/10024/149320/Neuroverkot%20taloudellisen%20aikasarjan%20ennustamisessa.pdf?sequence=1

Kairamo, E. (2022). *Neuroverkkojen hyödyntäminen osakemarkkinoiden hintakehityksen analysoinnissa*. Lappeenrannan–Lahden teknillinen yliopisto LUT. Noudettu 2.12.2024

osoitteesta https://lutpub.lut.fi/bitstream/handle/10024/163831/TutkielmaValmis_Kairamo_Elviira.pdf?sequence=1&isAllowed=y

Toivanen, P. (2013). *MLP-verkon arkkitehtuurin optimointi geneettisellä algoritmilla*. Tampereen yliopisto, Informaatiotieteiden yksikkö. Pro gradu -tutkielma. Noudettu 2.12.2024
osoitteesta https://trepo.tuni.fi/bitstream/handle/10024/84654/gradu06780.pdf?sequence=1&isAllowed=y

Haasiomäki, M.-P. (2019). *Äänien luokitteleminen neuroverkoilla*. Jyväskylän yliopisto, Informaatioteknologian tiedekunta, Kokkolan yliopistokeskus Chydenius. Pro gradu -tutkielma. Noudettu 2.12.2024 osoitteesta https://urn.fi/URN:NBN:fi:jyu-201911295070.

Haikonen, P. (2017). Vesijohto- ja jätevesiverkoston saneerausten priorisoinnin kehittäminen. Diplomityö. Tampereen teknillinen yliopisto. Noudettu 3.12.2024 osoitteesta:
https://trepo.tuni.fi/bitstream/handle/123456789/25663/Haikonen.pdf?sequence=4&isAllowed=y

Savic, D. (2002). *Single-objective vs. Multiobjective Optimisation for Integrated Decision Support*. Noudettu 3.12.2024 osoitteesta https://scholarsarchive.byu.edu/cgi/viewcontent.cgi?article=3734&context=iemssconference

Camargo, T., Tissot, J. & Pozo, F. (2012). *Use of Backpropagation and Differential Evolution Algorithms to Training MLPs*. Noudettu 6.12.2024 osoitteesta https://www.researchgate.net/publication/261072741_Use_of_Backpropagation_and_Differential_Evolution_Algorithms_to_Training_MLPs

Perkola, T. (2014). Ohjelmistoarkkitehtuurin monitavoiteoptimointi. Tampereen yliopisto, Informaatiotieteiden yksikkö. Pro gradu -tutkielma. Noudettu 7.12.2024 osoitteesta https://trepo.tuni.fi/bitstream/handle/10024/95498/GRADU-1401884618.pdf?sequence=1

Price, K., Storn, R., & Lampinen, J. (2005). *Differential Evolution: A Practical Approach to Global Optimization*. (Book) Natural Computing Series. Springer-Verlag. ISBN 3-540-20950-6. Viitattu 8.12.2024.

Spronck, P. (1996). *Elegance: Genetic Algorithms in Neural Reinforcement Control*. Delft University of Technology, Faculty of Technical Mathematics and Informatics. Graduation thesis. Viitattu 9.3.2025.

Zankinski, I. (2017). *Effects of the Neuron Permutation Problem on Training Artificial Neural Networks with Genetic Algorithms*. In I. Dimov, I. Faragó, & L. Vulkov (Eds.), *Lecture Notes in Computer Science* (Vol. 10187, pp. 777–782). Springer. Conference paper. Noudettu 9.3.2025 osoitteesta https://www.researchgate.net/publication/316049327

Yang, J.-M. (2001). *A Robust Evolutionary Algorithm for Training Neural Networks*. *Neural Computing and Applications*. Article. Noudettu 11.3.2025 osoitteesta https://www.researchgate.net/publication/220372937

Liitteet

Liite 1. Yksitavoiteoptimoinnin toteutus

```
#main.py

import numpy as np
from sklearn.preprocessing import LabelEncoder
from ucimlrepo import fetch_ucirepo
from differential_evolution import DifferentialEvolution
from mlp import MLP
import time
import pandas as pd

# Määritä suorituskertojen määrä
num_runs = 100

# Hae IRIS-datasarja UCI-repositorysta
iris = fetch_ucirepo(id=53)
X = iris.data.features.to_numpy()
y = iris.data.targets.to_numpy()

# Muunna luokat numeeriseen muotoon
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y.ravel())

# Jaa data opetus-, validointi- ja testijoukkoihin (75%, 20%, 5%)
np.random.seed(42)
indices = np.random.permutation(len(X))
train_size = int(0.75 * len(X))
valid_size = int(0.20 * len(X))

# Luo indeksit eri joukkoja varten
train_indices = indices[:train_size]
valid_indices = indices[train_size:train_size + valid_size]
test_indices = indices[train_size + valid_size:]

# Jaottele data indeksien mukaan
X_train, y_train = X[train_indices], y_encoded[train_indices]
X_valid, y_valid = X[valid_indices], y_encoded[valid_indices]
X_test, y_test = X[test_indices], y_encoded[test_indices]

# Alusta MLP-verkko
mlp = MLP(input_size=4, hidden_size=5, output_size=3)
```

```
# Määritä painojen rajat
input_hidden_size = 4 * 5
hidden_output_size = 5 * 3
bias_hidden_size = 5
bias_output_size = 3
total_weights = input_hidden_size + hidden_output_size + bias_hidden_size
+ bias_output_size
bounds = [(-1, 1)] * total_weights

# Tavoitefunktion määritys
def objective_function(weights):
    mlp.set_weights(weights)
    predictions = mlp.forward(X_train)

    if predictions.ndim == 1 or predictions.shape[1] == 1:
        predicted_classes = (predictions > 0.5).astype(int).flatten()
        true_classes = y_train
    else:
        predicted_classes = np.argmax(predictions, axis=1)
        true_classes = y_train

    accuracy = np.mean(predicted_classes == true_classes)
    return -accuracy

# Tulosten tallentamiseen
all_weights = []
all_results = []

# Suorita optimointi useita kertoja
for run in range(num_runs):
    print(f"\nSuoritetaan ajo {run + 1}/{num_runs}...")


    log_file = f"YksitavoiteAjo_{run + 1}_tulokset.xlsx"
    optimizer = DifferentialEvolution(
        func=objective_function,
        bounds=bounds,
        population_size=50,
        mutation_factor=0.7,
        crossover_rate=0.9,
        generations=200,
        log_file=log_file
    )

    # Mittaa optimoinnin suoritusaika
    start_time = time.perf_counter()
```

```
    optimal_solution, optimal_value, generation_data =
optimizer.optimize(return_log=True)
    end_time = time.perf_counter()

    execution_time = end_time - start_time

    # Tulosta yksittäisen ajon tulokset
    print(f"Ajo {run + 1}: Optimal Value: {-optimal_value}, Execution
Time: {execution_time:.4f} seconds")

    # Tulosta verkon painoarvot
    print("\nVerkon painoarvot vektorina:")
    print(optimal_solution)

    # Tallenna painoarvot listaan
    all_weights.append(optimal_solution)

    # Arvioidaan validointi- ja testitarkkuus
    mlp.set_weights(optimal_solution)
    valid_predictions = mlp.forward(X_valid)
    valid_predicted_classes = np.argmax(valid_predictions, axis=1)
    validation_accuracy = np.mean(valid_predicted_classes == y_valid)

    test_predictions = mlp.forward(X_test)
    test_predicted_classes = np.argmax(test_predictions, axis=1)
    test_accuracy = np.mean(test_predicted_classes == y_test)

    # Arvioidaan opetusjoukon virhe
    train_predictions = mlp.forward(X_train)
    train_predicted_classes = np.argmax(train_predictions, axis=1)
    train_error = 1 - np.mean(train_predicted_classes == y_train)

    print(f"Validation Accuracy: {validation_accuracy:.4f}, Test
Accuracy: {test_accuracy:.4f}, Training Error: {train_error:.4f}")

    # Tallenna sukupolvien tiedot Exceliin
    generation_df = pd.DataFrame(generation_data, columns=["Generation",
"Function Value", "Evaluations"])
    generation_df.to_excel(log_file, index=False)

    # Tallenna ajon tulokset
    all_results.append({
        "Run": run + 1,
        "Optimal Value": -optimal_value,
        "Execution Time": execution_time,
        "Validation Accuracy": validation_accuracy,
        "Test Accuracy": test_accuracy,
```

```
        "Training Error": train_error
    })

# Painoarvojen tallennus Exceliin
weights_file = "kaikki_painoarvot.xlsx"
weights_df = pd.DataFrame(all_weights)
weights_df.to_excel(weights_file, index=False, header=[f"Weight_{i}" for
i in range(len(bounds))])

# Kaikkien tulosten tallennus Exceliin
total_results_file = "kaikki_tulokset.xlsx"
results_df = pd.DataFrame(all_results)
results_df.to_excel(total_results_file, index=False)

# Lisää DE:n parametrit DataFrameen
results_df[" "] = ""
results_df["Population Size"] = optimizer.population_size
results_df["Mutation Factor"] = optimizer.mutation_factor
results_df["Crossover Rate"] = optimizer.crossover_rate
results_df["Generations"] = optimizer.generations

# Laske keskiarvo ja keskihajonta "Optimal Value" -sarjasta
optimal_values = [result["Optimal Value"] for result in all_results]
mean_optimal_value = np.mean(optimal_values)
std_optimal_value = np.std(optimal_values)

# Lisää keskiarvo ja keskihajonta viimeiselle riville
mean_std_row = pd.DataFrame({
    "Run": ["Mean/Std"],
    "Optimal Value": [""],
    "Execution Time": [""],
    "Validation Accuracy": [""],
    "Test Accuracy": [""],
    "Training Error": [""],
    " ": [""],
    "Mean OV": [mean_optimal_value],
    "Std OV": [std_optimal_value],
    "Population Size": [""],
    "Mutation Factor": [""],
    "Crossover Rate": [""],
    "Generations": [""]
})

# Lisää tämä rivi DataFrameen
results_df = pd.concat([results_df, mean_std_row], ignore_index=True)

# Tallenna ajon tulokset ja DE:n parametrit Exceliin
```

```
results_df.to_excel(total_results_file, index=False)

# Tulosta palaute tallennuksen onnistumisesta
print(f"\nKaikki painoarvot tallennettu tiedostoon '{weights_file}'.")
print(f"Kaikki tulokset tallennettu tiedostoon '{total_results_file}'.")
print(f"Optimal Value - Keskiarvo: {mean_optimal_value:.4f},
Keskihajonta: {std_optimal_value:.4f}")
```

```
#mlp.py

import numpy as np

class MLP:

    #Alustaa MLP:n ja sen parametrit
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.weights_input_hidden = np.random.uniform(-1, 1, (input_size,
hidden_size))
        self.bias_hidden = np.random.uniform(-1, 1, hidden_size)
        self.weights_hidden_output = np.random.uniform(-1, 1,
(hidden_size, output_size))
        self.bias_output = np.random.uniform(-1, 1, output_size)

    #Suorita verkon eteenpäinsyöttö
    def forward(self, x):
        hidden = np.dot(x, self.weights_input_hidden) + self.bias_hidden
        output = np.dot(hidden, self.weights_hidden_output) +
self.bias_output
        return output

    #Painojen ja biasien asettaminen
    def set_weights(self, weights_and_biases):
        input_hidden_size = self.input_size * self.hidden_size
        hidden_output_size = self.hidden_size * self.output_size

        self.weights_input_hidden =
weights_and_biases[:input_hidden_size].reshape(self.input_size,
self.hidden_size)
        self.bias_hidden =
weights_and_biases[input_hidden_size:input_hidden_size +
self.hidden_size]
```

```
        self.weights_hidden_output = weights_and_biases[input_hidden_size
+ self.hidden_size:input_hidden_size + self.hidden_size +
hidden_output_size].reshape(self.hidden_size, self.output_size)
        self.bias_output = weights_and_biases[-self.output_size:]
```

```
#differential_evolution.py

import numpy as np
import pandas as pd

class DifferentialEvolution:

    # Alustaa differentiaalievoluution parametrit
    def __init__(self, func, bounds, population_size, mutation_factor,
crossover_rate, generations, log_file):
        self.func = func
        self.bounds = bounds
        self.population_size = population_size
        self.mutation_factor = mutation_factor
        self.crossover_rate = crossover_rate
        self.generations = generations
        self.log_file = log_file #Tallennuksen tiedostopolku

    # Suorittaa optimoinnin differentiaalievoluutiolla
    def optimize(self, return_log=False):
        np.random.seed(None)
        dim = len(self.bounds)
        population = np.random.rand(self.population_size, dim)
        for i, (low, high) in enumerate(self.bounds):
            population[:, i] = low + population[:, i] * (high - low)

        best_solution = None
        best_score = float('inf')
        eval_count = 0
        data = []
        prev_best_score = None

        # Optimointiprosessi
        for generation in range(self.generations):
            for i in range(self.population_size):
                indices = [idx for idx in range(self.population_size) if
idx != i]
                a, b, c = population[np.random.choice(indices, 3,
replace=False)]
                mutant = a + self.mutation_factor * (b - c)
```

```
                mutant = np.clip(mutant, [low for low, _ in self.bounds],
[high for _, high in self.bounds])

                crossover = np.random.rand(dim) < self.crossover_rate
                trial = np.where(crossover, mutant, population[i])

                trial_score = self.func(trial)
                current_score = self.func(population[i])
                eval_count += 1

                if trial_score < current_score:
                    population[i] = trial
                    if trial_score < best_score:
                        best_solution = trial
                        best_score = trial_score

            # Jos funktion arvo muuttuu, kirjataan evaluointien määrä
            if prev_best_score is None or best_score != prev_best_score:
                data.append([generation + 1, -best_score, eval_count])
            else:
                data.append([generation + 1, -best_score, None])

            # Päivitetään edellinen paras arvo
            prev_best_score = best_score

        # Tallennetaan tiedot Exceliin
        df = pd.DataFrame(data, columns=["Generation", "Function Value",
"Evaluations"])
        df.to_excel(self.log_file, index=False)

        # Palautetaan paras ratkaisu
        if return_log:
            return best_solution, best_score, data
        return best_solution, best_score
```

Liite 2. Monitavoiteoptimoinnin toteutus

```
# main.py

import numpy as np
import pandas as pd
import time
from differential_evolution import DifferentialEvolution
from mlp import MLP
from sklearn.preprocessing import LabelEncoder
from ucimlrepo import fetch_ucirepo

# Suorituskertojen määrä
num_runs = 100

# Ladataan IRIS-datasetti
iris = fetch_ucirepo(id=53)
X = iris.data.features.to_numpy()
y = iris.data.targets.to_numpy()

# Koodataan luokat numeerisiksi arvoiksi
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y.ravel())

# Datasetin jako esimerkiksi: 75% opetus, 20% validointi, 5% testaus
np.random.seed(42)
indices = np.random.permutation(len(X))
train_size = int(0.75 * len(X))
valid_size = int(0.20 * len(X))

#Alustetaan indeksit
train_indices = indices[:train_size]
valid_indices = indices[train_size:train_size + valid_size]
test_indices = indices[train_size + valid_size:]

#Määritetään joukot
X_train, y_train = X[train_indices], y_encoded[train_indices]
X_valid, y_valid = X[valid_indices], y_encoded[valid_indices]
X_test, y_test = X[test_indices], y_encoded[test_indices]

# Alusta MLP
mlp = MLP(input_size=4, hidden_size=5, output_size=3)

# Määritellään painojen rajat
input_hidden_size = 4 * 5
hidden_output_size = 5 * 3
```

```
bias_hidden_size = 5
bias_output_size = 3
total_weights = input_hidden_size + hidden_output_size + bias_hidden_size
+ bias_output_size
bounds = [(-1, 1)] * total_weights

# Määritellään tavoitefunktiot
def objective_function(weights):
    mlp.set_weights(weights)
    predictions = mlp.forward(X_train)

    # Lasketaan luokittelutarkkuus
    predicted_classes = np.argmax(predictions, axis=1)
    accuracy = np.mean(predicted_classes == y_train)

    # Lasketaan softmax-arvot
    softmax_values = np.exp(predictions) / np.sum(np.exp(predictions),
axis=1, keepdims=True)
    softmax = np.mean(np.max(softmax_values, axis=1))

    return -accuracy, -softmax

# Tähän tiedostoon kirjataan kaikki tulokset
log_file = "kaikki_tulokset.xlsx"

# Tulosten tallentamiseen
all_results = []
all_weights = []
all_de_params = []
accuracy_values = []
softmax_values = []

# Suoritetaan useita ajoja
for run in range(num_runs):
    print(f"\nSuoritetaan ajo {run + 1}/{num_runs}...")

    optimizer = DifferentialEvolution(
        func=objective_function,
        bounds=bounds,
        population_size=200,
        mutation_factor=0.8,
        crossover_rate=0.9,
        generations=150,
        log_file=f"MonitavoiteAjo_{run + 1}_tulokset.xlsx"  #
Sukupolvikohtaiset arvot tallennetaan tähän tiedostoon
    )
```

```
    # Mittaa suoritusaika
    start_time = time.perf_counter()
    optimal_solution, (optimal_value_accuracy, optimal_value_softmax) =
optimizer.optimize()
    end_time = time.perf_counter()

    execution_time = end_time - start_time

    # Tallennetaan DE:n parametrit ja tulokset
    accuracy_values.append(-optimal_value_accuracy)
    softmax_values.append(-optimal_value_softmax)

    all_de_params.append({
        "Population Size": optimizer.population_size,
        "Mutation Factor": optimizer.mutation_factor,
        "Crossover Rate": optimizer.crossover_rate,
        "Generations": optimizer.generations
    })

    # Tulosta ja tallenna yksittäisen ajon tulokset
    print(f"Ajo {run + 1}: Accuracy: {-optimal_value_accuracy}, Softmax:
{-optimal_value_softmax}, Execution Time: {execution_time:.4f} seconds")

    # Arvioidaan validointi-, testitarkkuus ja opetusjoukon virhe
    mlp.set_weights(optimal_solution)

    # Opetusjoukon virhe
    train_predictions = mlp.forward(X_train)
    train_predicted_classes = np.argmax(train_predictions, axis=1)
    train_error = 1 - np.mean(train_predicted_classes == y_train)

    # Validointitarkkuus
    valid_predictions = mlp.forward(X_valid)
    valid_predicted_classes = np.argmax(valid_predictions, axis=1)
    validation_accuracy = np.mean(valid_predicted_classes == y_valid)

    # Testitarkkuus
    test_predictions = mlp.forward(X_test)
    test_predicted_classes = np.argmax(test_predictions, axis=1)
    test_accuracy = np.mean(test_predicted_classes == y_test)

    print(f"Training Error: {train_error:.4f}, Validation Accuracy:
{validation_accuracy:.4f}, Test Accuracy: {test_accuracy:.4f}")

    # Tallenna painoarvot
    all_weights.append(optimal_solution)
```

```
    # Tulosta painoarvot konsoliin
    print(f"Painoarvot ajo {run + 1}:")
    print(optimal_solution)

    # Tallenna tulokset listaan
    all_results.append({
        "Run": run + 1,
        "Optimal Accuracy": -optimal_value_accuracy,
        "Optimal Softmax": -optimal_value_softmax,
        "Execution Time": execution_time,
        "Training Error": train_error,
        "Validation Accuracy": validation_accuracy,
        "Test Accuracy": test_accuracy
    })

# Lasketaan keskiarvot ja keskihajonnat
mean_accuracy = np.mean(accuracy_values)
std_accuracy = np.std(accuracy_values)
mean_softmax = np.mean(softmax_values)
std_softmax = np.std(softmax_values)

# Lisää DE:n parametrit ensimmäiselle riville
de_param_row = {
    "Run": "",
    "Optimal Accuracy": "",
    "Optimal Softmax": "",
    "Execution Time": "",
    "Training Error": "",
    "Validation Accuracy": "",
    "Test Accuracy": "",
    "": "",
    "Population Size": all_de_params[0]["Population Size"],
    "Mutation Factor": all_de_params[0]["Mutation Factor"],
    "Crossover Rate": all_de_params[0]["Crossover Rate"],
    "Generations": all_de_params[0]["Generations"],
    "": "",
    "Mean Accuracy":  "",
    "Std Accuracy": "",
    "Mean Softmax": "",
    "Std Softmax": ""
}

# Lisää DE-parametrit ja muut tulokset DataFrameen
results_df = pd.DataFrame(all_results)
de_param_df = pd.DataFrame([de_param_row])

# Yhdistä tulokset ja DE-parametrit
```

```
final_df = pd.concat([de_param_df, results_df], ignore_index=True)

# Lisää keskiarvot ja keskihajonnat ensimmäiselle riville
final_df.loc[0, "Mean Accuracy"] = mean_accuracy
final_df.loc[0, "Std Accuracy"] = std_accuracy
final_df.loc[0, "Mean Softmax"] = mean_softmax
final_df.loc[0, "Std Softmax"] = std_softmax

# Tallennetaan kaikki tulokset Exceliin
with pd.ExcelWriter(log_file) as writer:
    final_df.to_excel(writer, index=False)

# Tallenna kaikki painoarvot erilliseen Excel-tiedostoon
weights_df = pd.DataFrame(all_weights)
weights_file = "kaikki_painoarvot.xlsx"

with pd.ExcelWriter(weights_file) as writer:
    weights_df.to_excel(writer, index=False)

#Palaute tallennuksen onnistumisesta
print("\nPainoarvot tallennettu tiedostoon 'kaikki_painoarvot.xlsx'.")
print("\nKaikki tulokset tallennettu tiedostoon 'kaikki_tulokset.xlsx'.")
print(f"Accuracy - Keskiarvo: {mean_accuracy:.4f}, Keskihajonta:
{std_accuracy:.4f}")
print(f"Softmax - Keskiarvo: {mean_softmax:.4f}, Keskihajonta:
{std_softmax:.4f}")
```

```
#mlp.py

import numpy as np

class MLP:

    #Alustetaan MLP
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.weights_input_hidden = np.random.uniform(-1, 1, (input_size,
hidden_size))
        self.bias_hidden = np.random.uniform(-1, 1, hidden_size)
        self.weights_hidden_output = np.random.uniform(-1, 1,
(hidden_size, output_size))
        self.bias_output = np.random.uniform(-1, 1, output_size)

    # Verkon eteenpäinajo
```

```
    def forward(self, x):
        hidden = np.dot(x, self.weights_input_hidden) + self.bias_hidden
        output = np.dot(hidden, self.weights_hidden_output) +
self.bias_output
        return output

    # Asettaa verkon painot ja biasit
    def set_weights(self, weights_and_biases):
        input_hidden_size = self.input_size * self.hidden_size
        hidden_output_size = self.hidden_size * self.output_size

        self.weights_input_hidden =
weights_and_biases[:input_hidden_size].reshape(self.input_size,
self.hidden_size)
        self.bias_hidden =
weights_and_biases[input_hidden_size:input_hidden_size +
self.hidden_size]
        self.weights_hidden_output = weights_and_biases[input_hidden_size
+ self.hidden_size:input_hidden_size + self.hidden_size +
hidden_output_size].reshape(self.hidden_size, self.output_size)
        self.bias_output = weights_and_biases[-self.output_size:]
```

```
#differential_evolution.py

import numpy as np
import pandas as pd

class DifferentialEvolution:

    # Alustaa differentiaalievoluution parametrit
    def __init__(self, func, bounds, population_size, mutation_factor,
crossover_rate, generations, log_file):
        self.func = func
        self.bounds = bounds
        self.population_size = population_size
        self.mutation_factor = mutation_factor
        self.crossover_rate = crossover_rate
        self.generations = generations
        self.log_file = log_file
        self.generation_data = []
        self.best_improvements = []
        self.prev_accuracy = None  # Luokitteluarkkuus
        self.prev_softmax = None  # Softmax
        self.eval_count = 0  # Evaluointien määrä

    # Suorittaa optimoinnin
```

```
    def optimize(self):
        np.random.seed(None)
        dim = len(self.bounds)
        population = np.random.rand(self.population_size, dim)
        for i, (low, high) in enumerate(self.bounds):
            population[:, i] = low + population[:, i] * (high - low)

        best_solution = None
        best_score = float('inf')
        eval_count = 0
        data = []
        prev_best_score = None

        for generation in range(self.generations):
            for i in range(self.population_size):
                indices = [idx for idx in range(self.population_size) if
idx != i]
                a, b, c = population[np.random.choice(indices, 3,
replace=False)]
                mutant = a + self.mutation_factor * (b - c)
                mutant = np.clip(mutant, [low for low, _ in self.bounds],
[high for _, high in self.bounds])

                crossover = np.random.rand(dim) < self.crossover_rate
                trial = np.where(crossover, mutant, population[i])

                trial_score_accuracy, trial_score_softmax =
self.func(trial)
                current_score_accuracy, current_score_softmax =
self.func(population[i])
                self.eval_count += 1  # Kasvatetaan evaluointien määrää
                eval_count += 1

                # Tarkista, onko luokittelutarkkuus tai softmax muuttunut
verrattuna edellisiin
                if (self.prev_accuracy is None or trial_score_accuracy !=
self.prev_accuracy or
                    self.prev_softmax is None or trial_score_softmax !=
self.prev_softmax):

                    # Tallennetaan arviointien määrä vain silloin, kun
funktion arvo muuttuu
                    self.best_improvements.append({
                        "Generation": generation + 1,
                        "Evaluations": self.eval_count,
                        "Best Accuracy": -trial_score_accuracy,
                        "Best Softmax": -trial_score_softmax
```

```
                    })

                    # Päivitetään edelliset arvot
                    self.prev_accuracy = trial_score_accuracy
                    self.prev_softmax = trial_score_softmax

                # Päivitä paras ratkaisu, jos löytyy parempi
                if trial_score_accuracy < current_score_accuracy:
                    population[i] = trial
                    if trial_score_accuracy < best_score:
                        best_solution = trial
                        best_score = trial_score_accuracy

            # Tallennetaan sukupolvikohtaiset arvot vain, kun arvo on
muuttunut
            if self.prev_accuracy is None or trial_score_accuracy !=
self.prev_accuracy:
                data.append([generation + 1, -trial_score_accuracy,
eval_count])
            else:
                data.append([generation + 1, -trial_score_accuracy,
None])

            # Tallennetaan generaatiokohtaiset tiedot
            self.generation_data.append({
                "Generation": generation + 1,
                "Accuracy": -best_score,
                "Softmax": -trial_score_softmax,
                "Evaluations": self.eval_count
            })

        # Tallennetaan sukupolvikohtaiset arvot tiedostoon
        generations_data = pd.DataFrame(self.generation_data,
columns=["Generation", "Accuracy", "Softmax", "Evaluations"])
        generations_data.to_excel(self.log_file, index=False,
sheet_name="Generations")

        # Palautetaan tulokset
        return best_solution, (best_score, trial_score_softmax)
```

Liite 3. Hakua ohjaavan funktion f_2 määrittely

```
#main.py
# Määritellään tavoitefunktiot
def objective_function(weights):
    mlp.set_weights(weights)
    predictions = mlp.forward(X_train)

    # Luokittelutarkkuus
    predicted_classes = np.argmax(predictions, axis=1)
    accuracy = np.mean(predicted_classes == y_train)

    # f2-funktion laskeminen
    S = np.sum(mlp.weights_input_hidden, axis=0)
    f2 = -np.sum(np.minimum(np.diff(S), 0))

    return -accuracy, f2
```