Nõo Reaalgümnaasium

Joonatan Samuel
11.b klass

Väikeste tehisnärvivõrkude võrgustruktuuri mõju tulemuslikkusele

Praktiline töö

Juhendajad:
Konstantin Tretjakov
Sirje Sild

Nõo 2015

Sisukord

1. Sissejuhatus

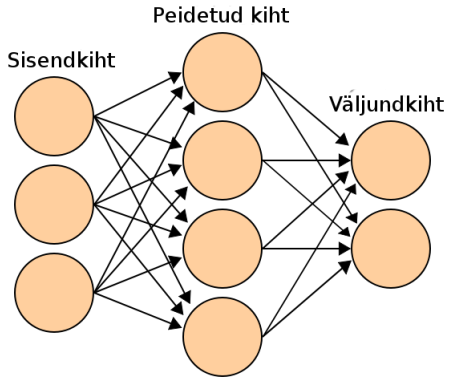
Tehisnärvivõrk on võimekas masinõppe algoritm, mis põhineb bioloogilistel närvivõrkudel. Tehisnärvivõrkude kasutamine reaalsetel andmetel on keeruline, sest tuleb valida palju erinevaid muutujaid arendaja intuitsiooni järgi. Suurt mõju tehisnärvivõrgu võimekusele avaldab arendaja poolt valitud neuronite arv ja nende paigutus võrgus, ehk võrgustruktuur. Ka võrgustruktuur tuleb arendajal endal valida. Seetõttu uuritakse käesolevas töös tehisnärvivõrgu omadusi, et valida parimat närvivõrgu struktuuri. Optimiseerides sedasi tehisnärvivõrkude kasutamist, on võimalik kiirendada märgatavalt algoritmi sisaldavate süsteemide arendamist, sest võrgustruktuuri katseeksituslik valik võib-olla väga aeganõudev.

Töös treenime erinevate võrgustruktuuridega tehisnärvivõrke käsitsi kirjutatud numbreid tuvastama ja seejärel vaatleme igaühe võimekust antud ülesannet sooritada. Samuti üritame vähendada segavate faktorite mõju närvivõrgu tulemuslikkusele ja kindlustada, et tulemuslikkus, mille saame katsest, iseloomustab seda võrku hästi ja on korratav.

Andmestikuks valiti Mixed National Institute of Standards and Technology andmestikul (edaspidi MNIST andmestik), milles on hallskaalal sildistatud pildid numbritest nullist üheksani.

2. Lühiülevaade tehisnärvivõrkudest

Keerulise sõna *„tehisnärvivõrk"* taga on peidus lihtne mõte: kogum *neuroneid*, millest igaüks omab mingisuguseid sisendeid, viib täide mingisugust arvutust ja väljund, kuhu arvutuse vastus saadetakse. Igal ühendusel neuronite vahel on mingisugune *kaal*, millest sõltub, kui palju üks neuron teist mõjutada saab. Kõige esimene kiht võrgust kannab nime *sisendkiht*. See on koht, kuhu saab sisestada andmeid. Pärast sisendkihti on *peidetud kihid*, mis võivad olla omavahel keeruliselt ühendatud. Töös kasutame kõige lihtsamat võimalust - täielikult ühendatud võrku (täielikult ühendatud võrk tähendab, et iga neuron on ühendatud iga järgmise kihi neuroniga). Viimane kiht kannab nimetust *väljundkiht*, see annab meile informatsiooni, mida me tahtsime teada saada sisendandmete kohta. Kõik kihid sisendkihi ja väljundkihi vahel liigituvad peidetud kihtideks. Kogu struktuur üheskoos võib olla väga keerukas, aga ehitusplokkid on alati samad: neuronid ja ühendused nende vahel.



Joonis 1: Täielikult ühendatud tehisnärvivõrk ühe peidetud kihiga.(Burnett, 2006)

Meie, kui süsteemi kokkupanijad, tavaliselt teame, mida me tahame, et see süsteem teeks. See tähendab, et me teame kuidas mõningate sisendandmete jaoks väljund peaks välja nägema. *„Tark“* tehisnärvivõrk on võimeline näitama meile väljundit, mida me eeldame. *„Naiivne“* võrk aga vastupidiselt ei suuda. Näiteks, näidates pilti number neljast loodame, et pärast arvutamist võrk *arvab*, et pildil on number neli, mitte number kaks. Ainuke asi, mis on erinev naiivse ja targa võrgu puhul on neuronite vahelised kaalud. Muutes kaale on võimalik saada naiivsest võrgust tark. Seda protsessi nimetatakse õppimiseks.

Käesolevas töös kajastatud eksperimendi puhul on väljundkihis kümme neuronit - üks iga numbri jaoks. Iga neuroni väljundiks väljundkihis on tõenäosus, kui palju võrk *arvab*, et pildil on sellele neuronile vastav number. Mida suurem on tõenäosus õigel numbril ja väiksem ülejäänud numbritel, seda targem on meie võrk. Ennustuse tegemiseks valime kõige suuremale tõenäosusele vastava numbri.

See, kuidas tehisnärvivõrk oma arvamuseni jõuab ei ole midagi müstilist, vaid üksteise järel kindlas järjekorras teostatud tehted. Iga neuron võtab kõik sisendid ja leiab nende summa. Seejärel kasutab ta saadud summa peal *aktivatsiooni funktsiooni*, mis meie puhul on *sigmoid funktsioon*.

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

Seejärel väljastades enda arvutuse tulemus edasi järgmisele neuronile korrutatakse see läbi neuronite vahelise kaaluga. Siinkohal hakkab protsess aga otsast peale järgmises neuronis ja seda kuni jõutakse väljundkihti. Väljudkihis ei ole aga tulemust kuhugi edasi saata ja vastuseks jääbki viimase aktivatsiooni funktsiooni vastus, mis on väärtus nulli ja ühe vahel. Saadud tulemust annab tõlgendada tõenäosusena.

Seletamaks masinale õppimise ideed defineerime *kaofunktsiooni*. Antud tehisnärvivõrgu neuronite vahelised kaalud arvutab see funktsioon andmete peal väljundid ja seejärel annab väljundiks vea oodatud ja saadud vastuste vahel. Õppimisprotsessi saab näha kui kaofunktsiooni miinimumi otsimist. Teisisõnu on õppimisprotsess püüdlus vähendada viga.

Üks viis miinimumi otsida oleks proovida kõiki võimalikke kaale ja leida väikseim viga kõigi võimaluste seast. Kahjuks on võimalike kaalude kombinatsioonide arv liialt suur. Kuidas

seda teha, on tähtis küsimus ja seda mitte ainult meie süsteemi jaoks. Terve matemaatikaharu nimega optimiseerimine tegeleb selle probleemiga. Populaarne tehnika, mida kasutatakse, kannab nime *gradient-meetod* (Ing.k. gradient descent).

Gradient-meetodi tööpõhimõte on üsnagi triviaalne. Igal õppimisalgoritmi iteratsioonil muudame igat kaalu vähesel määral selles suunas, mis vähendab meie viga. Kui me jõuame kohta, kus mõlemale poole kaalu muutes viga suureneb, oleme jõudnud optimumi. Optimumis võime öelda, et meie tehisnärvivõrk on nii tark, kui ta olla saab. Võib juhtuda, et see optimum ei ole parim ja on mitmeid meetodeid leidmaks globaalset optimumi, aga need võtavad ka rohkem treenimisaega ja seetõttu ei ole meie ülesande jaoks sobilikud.

Käesolevas töös kasutame *gradient-meetodi* asemel *kaasgradient meetodit*(Rasmussen, 2001) (Ing.k. conjugate gradient) mis omab mitmeid eeliseid. Ta leiab enamasti kiiremini optimumi ja ei vaja, et õppimisalgoritmile oleks vaja anda sisendiks ühe kaalu muutmise maksimaalset väärtust. Seeläbi on võimalik vähendada inimviga ja kiirendada õppimisprotsessi. Kuid meetod on põhimõtteliselt sarnane *gradient-meetodiga*.

3. Andmete iseloomustus ja eeltöötlus

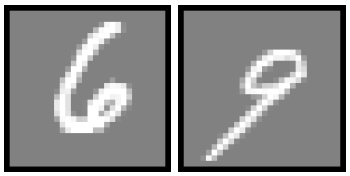
3.1. Andmete valikust

Töös kasutame MNIST(LeCun, Cortes, and Burges, 1998) käsitsi kirjutatud numbrite andmesikku, mis on laialt kasutatud(LeCun, Cortes, and Burges, 1998)(Labusch, Barth, and Martinetz, 2008)(Ciresan, U.Meier, and Schmidhuber, 2012)(Lauer, Suen, and Bloch, 2007), võrdlemaks erinevate masinõppe lahenduste võimekust. Seda seetõttu, et andmestik on lihtsasti kättesaadav, ei vaja eeltöötlemist ja on piisavalt suur. Samuti on kerge saada aru numbrite ära tundmise idee keerukusest - pea kõigil on ette tulnud frustreerivat situatsiooni kellegi teise käsitsi kirjutatud numbreid lugedes. Sellegipoolest inimene, kes kirjutas selle numbri mõistab seda, sest ta ise on seda palju näinud ja on õppinud enda käekirja tuvastama.

Joonis 2: Erineva käekirjaga kirjutatud number kahed andmestikust

3.2. Masinõppe andmestike liigitusi

Treeningandmestikuks nimetatakse andmeid, mille alusel õpetatakse masinõppe algoritmi. Tehisnärvivõrgu puhul kasutame me treeningandmeid, et leida kaofunktsiooni väärtust ja gradientvektori väärtust, mille alusel me muudame neuronite vahelisi kaale. Treeningnäide on treeningandmestiku üks sildi ja treeningandmete paar. MNIST andmestiku puhul on treeningandmeteks pikslite heledused ühes numbripildis ja sellele pildile vastav silt.

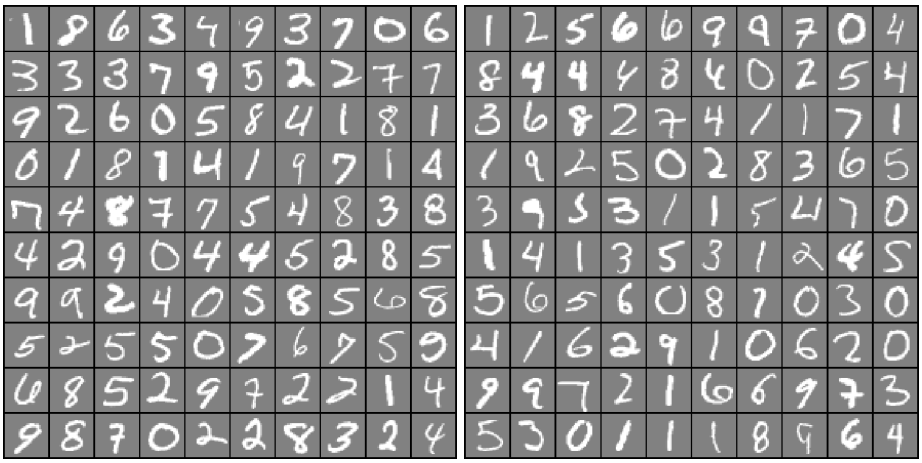


Joonis 3: Treening näited siltidega vastavalt ”6” ja ”9”

Testandmestikuks nimetatakse aga andmekogumit, mis muidu on treeningandmestikuga identne, kuid mida ei kasutata õppimisel. Testandmestikku kasutatakse kontrollimaks, et ei toimu andmestikule spetsialiseerumist. Spetsialiseerumist võib vaadata kui tehisnärvivõrgu üritust kõiki treeningnäiteid meelde jätta, selle asemel, et leida neid läbivat ühist joont - sellest tulenevalt tehisnärvivõrgu arusaam ei üldistu uutele näidetele. Selle vastu saab kasutada mitmeid meetodeid nagu regulariseerimine ja võrgustruktuuri lihtsustamine, kuid samuti aitab ka rohkemate treeningandmete kasutamine.

3.3. Andmete iseloomustus

Kokku on terves andmestikus 60 000 treening- ja 10 000 testnäidet, kuid kasutati andmestikust suvaliselt valitud 5 000 treeningnäidet ja 400 testnäidet. Kõigil numbripiltidel oli iga piksel hallskaala vahemikus $[0;255]$, selle viisime vahemikku $[0;1]$ lihtsama treenimise eesmärgil. Andmestik on osaliselt juba eeltöödeldud – kõik numbrid on tehtud sama suureks ja seejärel asetatud pildi keskele. Treening- ja testandmed pärinevad mõlemad umbes 250 kirjutajalt, kuid andmestikud on koostatud nõnda, et ühe inimese kirjutatud numbrid ei esine mõlemas andmestikus. Selline andmestiku koostamine ei võimalda masinõppe algoritmil ära harjuda andmestiku eripäradega, nagu seda on näiteks inimeste käekiri, ja annab parema ülevaate algoritmi üldistamisvõime võimekusest.



Joonis 4: 100 suvalist treening- ja testnäidet

4. Probleemipüstitus

Kasutades tehisnärvivõrke mingitel andmetel on keeruline ette teada, kui mitut peidetud kihti ja neuronit igas kihis on vaja. Mida rohkem peidetud kihte kasutada seda suuremat mittelineaarsust andmestikus võrk suudab jäljendada, kuid suureneb ka treenimisaeg(Hochreiter, Y. Bengio, and Schmidhuber, 2001). Kasutades paljusid neuroneid igas kihis võib aga olla ebasoovitav, sest andmetöötluse ressursimahukus suureneb iga neuroniga. Muutes suurte võrkude kasutamise keeruliseks vähese ressursiga seadmetel, nagu telefonid ja tahvelarvutid, või reaalajalisel infotöötlusel, nagu näiteks videotöötlus või audiotöötlus.

Mõistmaks tehisnärvivõrgu struktuuri mõju võrgu võimekusele, treenime erinevate võrgustruktuuridega tehisnärvivõrke ära tundma käsitsi kirjutatud numbreid, kus numbrid on vahemikus nullist üheksani. Pärast treenimist kontrollime võrgu õpitud arusaama numbritest testandmestikul.

Kiirendamaks treenimist, kasutame ainult osa suurest MNIST andmestikust. Valisime suvaliselt 5 000 treeningnäidet andmestiku treeningnäidetest[1]. Väiksemat andmestikku kasutades võib treening kergemini viia andmetele spetsialiseerumiseni, ehk närvivõrk võib liialt kiivalt vaadata andmestikus esinevat müra ja teisi iseärasusi nagu näiteks käekiri selle asemel, et järgida andmestiku fundamentaalseid erinevusi. Kogumaks tehisnärvivõrgu tõelist arusaama numbritest, vaatleme täpsust numbrite tuvastamisel mitte treeningandmetel, vaid testandmestikul, kuhu kuulub 400 testnäidet.

Vähendamaks treeningandmetele spetsialiseerumist, kasutame võrgu treenimisel regulariseerimist, mis on matemaatiline trikk, et treenimisel hoida arusaam numbritest lihtsamana. Kuna pole üheselt määratav, kui lihtsa või keeruka arusaama numbritest tehisnärvivõrk peab saavutama, proovime logaritmiliselt läbi viis astet ja valime parima tulemuse.

[1] Andmestikud, programmid ja tulemused on kättesaadavad:
https://github.com/Jonksar/Neural-network-research

5. Töökäik ja töö meetodid

Kõik programeerimine tehti programeerimiskeeles Octave, mis on kõrgema taseme keel mõeldud põhiliselt numbriliste kalkulatsioonide jaoks. Tihti kasutatakse seda lineaaralgebra, statistilise analüüsi ja teiste numbriliste simulatsioonide ning eksperimentide jaoks. (Octave, 2015)

Treenimaks võrke koostas autor programmi, mis arvutab kaofunktsiooni väärtuse ja gradiendi[2]. Selline programm koostati eraldi kõigi erinevate peidetud kihtide arvu jaoks, kuid kõik on analoogsed. Seejärel on tarvis valida optimiseerimise algoritm, mis gradient-vektori abil otsib kaofunktsioon miinumini ehk õpib numbreid tundma. Optimiseerimis algoritmi valikuks treenis autor erineva struktuuriga võrke ja vaatas aega, mis läheb täieliku koondumiseni. Leiti, et kaasgradient meetod töötab kiiremini gradient-meetodist, samuti on suureks eeliseks gradient-meetodi ees automaatne treeningsammu suuruse valik. Seejärel on tarvis leida õppimist lõpetav tingimus, milleks sai kaofunktsiooni muut kahe peidetud kihiliste võrkude puhul 2×10^{-3} üle 80 treeningtsükli ja kolme peidetud kihiliste võrkude puhul 2×10^{-3} üle 120 treeningtsükli. Sellised piirid leidis autor treenides erinevaid ekstreemse (kümme neuronit või mõni neuron igas võrgu kihis) struktuuriga võrke ja erinevate regulariseerimisastmetega võrke. Leiti, et ühe sammu kaofunktsiooni langemise vaatamisel juhtub tihti treenimise katkestamine, eriti suure regulariseerimise ja väheste neuronite arvu puhul. Samuti tuleb kaasgradienti kasutates pidada silmas, et esialgu teeb kaasgradient väiksemaid samme ja seetõttu treenides vähemate iteratsioonide arvu kaupa kiirus väheneb. Nende probleemide vältimiseks vaadeldakse viimase 80 või 120 iteratsiooni kaofunktsiooni muutu. Seejärel koostati programm, mis suutis ühe numbri järgi paika panna võrgustruktuuri, regulariseerimis parameetri lambda, treenida võrke, leida võrgu täpsus andmetel ning seejärel salvestada tulemus failina. Seejärel treeniti kõik 5500[3] erinevat tehisnärvivõrku. Autori kirjutatud programmid on kaasas lisadena.

Tehisnärvivõrkude treenimine toimus Tartu Ülikooli arvutusklastri "Alligaator" peal. Treenimiseks kasutati kõiki Alligaatori 80 protsessorituuma. Tuumad on 8 Intel Xeon E7 - 2860 CPU-d @ 2.27Ghz (*High Performance Computing Center, University of Tartu*). Treenimiseks kulus kahe kihiliste võrkude (kokku 500) jaoks tund ja kolme kihiliste (kokku 5 000) jaoks kolmteist tundi. Selline treenimisaja pikkus saadi failide tekkimise ajahetkede vahe järgi. Saadud treenimisaeg on hea pakkumine arvutusaja ülemisele piirile, mis läheb 5 500 võrgu treenimiseks Alligaatori kõikide tuumade peal.

Ka edasine andmete analüüs toimus Octave keskkonnas. Autor vaatles erinevaid andmete omadusi ja valis välja andmeid paremini iseloomustavad graafikud, seejärel genereeriti joonised ning pildid. Kõik andmetöötlus toimus Octave'i funktsioonidega nagu seda on näiteks *average,*

[2]Maatriksid, mis iseloomustuvad kaofunktiooni osatuletist iga kaalu kohta; Kasutatakse õppimise ajal kaalude muutmiseks.

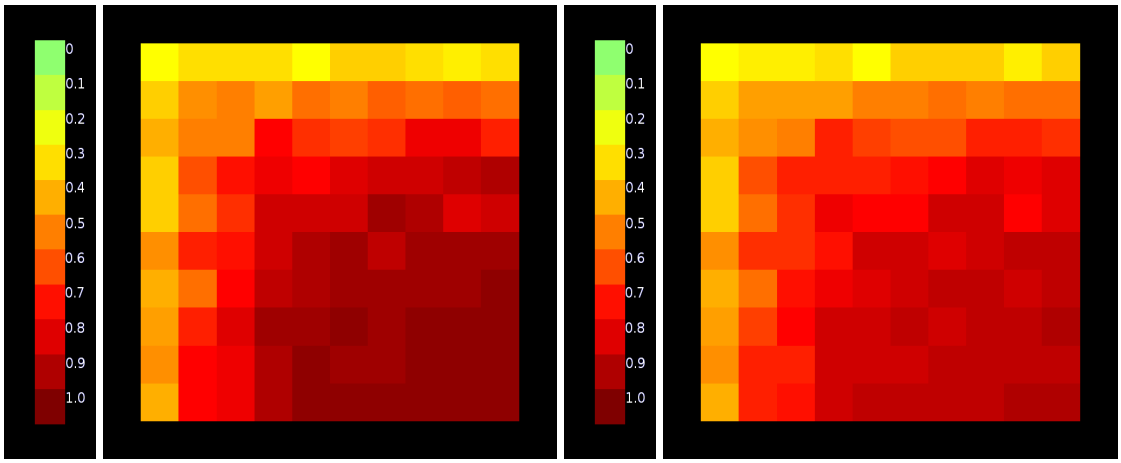
[3]5 lambdat$\times$1 000 kolme, 5 lambdat $\times$ 100 kahe peidetud kihiliste võrgustruktuuride korral; Kombinatsioonide arv on $\prod w_l$, kus w_l on maksimaalne neuronite arv kihis l.

mean, max, maxcum, min, gradient.

6. Tulemused ja nende analüüs

6.1. Kahe peidetud kihiliga võrgustruktuuride analüüs

Kahe peidetud kihiga võrgustruktuuride täpsuse protsent on toodud järgmisel joonisel. Horisontaalset telge pidi on esimese kihi neuronite arv ja vertikaalset telge pidi teise kihi neuronite arv. Kusjuures graafik algab ülevalt vasakult - seal on võrgustruktuur, kus esimeses ja teises kihis mõlemas on üks neuron.

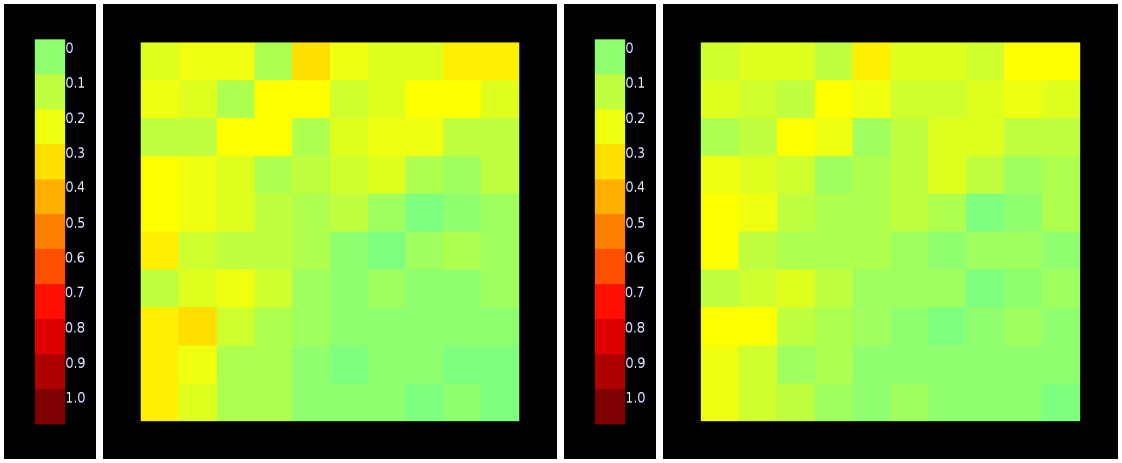


Joonis 5: Treening ja test tulemused vastavalt kahe peidetud kihiliste võrgustruktuuride jaoks

On näha, et mõlemate kihtide neuronite arvust oleneb võrgu võimekus sama palju. Seda seetõttu, et joonis on diagonaali pidi sümmeetriline. Samuti on huvitav asjaolu, et kui ühes kihis on vähe neuroneid ja teises palju ei ole võrgu võimekus hea, kuigi neuronite arv on võrdlemisi suur. Seda võib näha, kui kogu informatsiooni kokku surumist väheste neuronite kätte, kuid neil ei ole võimekust kogu informatsiooni talletada - tekib informatsiooni kadu ja seetõttu ei ole võrk võimeline enam hästi numbreid tuvastama. Nagu oodatud on näha, et treeningandmedetel on keskmine täpsus suurem, kui testandmetel. Treeningandmete keskmine täpsus oli 71.304% ja testandmetel 65.940%. Minimaalne täpsuse väärtus mõlema andmestiku puhul oli üsna lähedane ja samuti lähedane 10%'le, mida oodata suvalisel pakkumisel. Treeningandmestikul oli minimaalne täpsus 22.640% ja testandmestikul 24.250%. Maksimaalne täpsus oli aga 97.260% ja 91.250% vastavalt treening- ja testandmestikul. Selle tulemuse saavutas võrgustruktuur, mille peidetud kihtide suurused on vastavalt 9 ja 10.

Järgmisel joonisel on näha muutuse horisontaalse ja vertikaalse telje muutuse summat. See aitab mõista, missugused võrgustruktuurid on head enda lähedaste võrgustruktuuride suhtes. Mida suurem on muutuse väärtus, seda suuremat kasu toob võrgustruktuuri vähene

muutmine. Samuti on see hea indikaator, missuguseid võrgustruktuure vältida - tuleks valida võrgustruktuure mis asetsevad piisavalt kaugel minimaalsest võrgustruktuurist ja mille muutuste summa on minimaalne. See aitab valida lokaalse miinimumiga võrgustruktuuri.

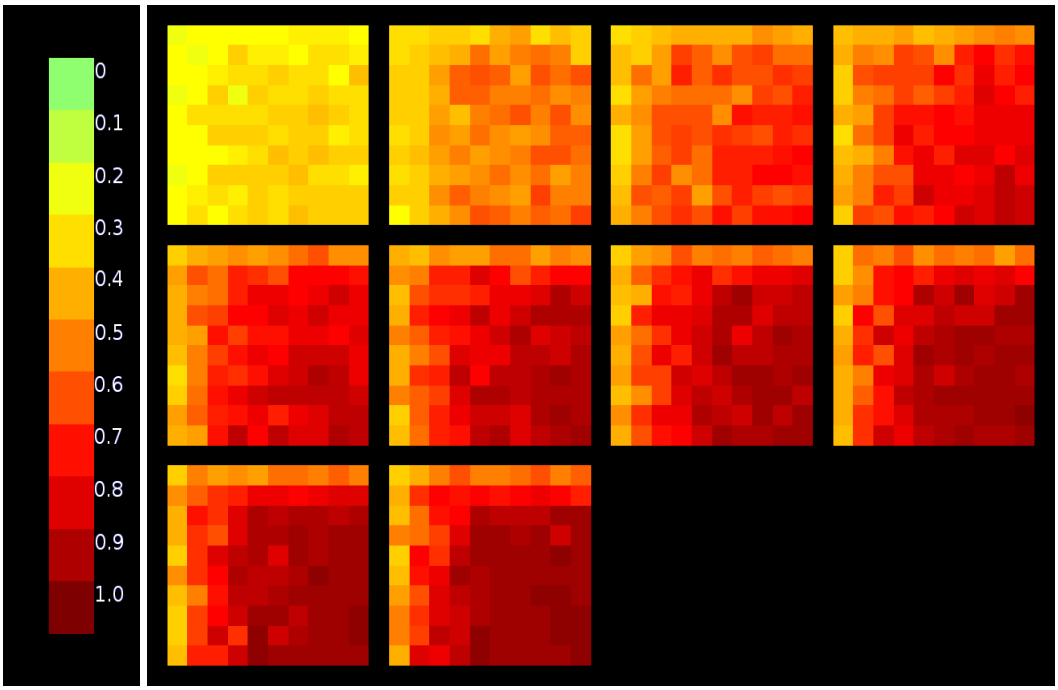


Joonis 6: Treening ja test tulemuste horisontaalse ja vertikaalse telje muutuse summa

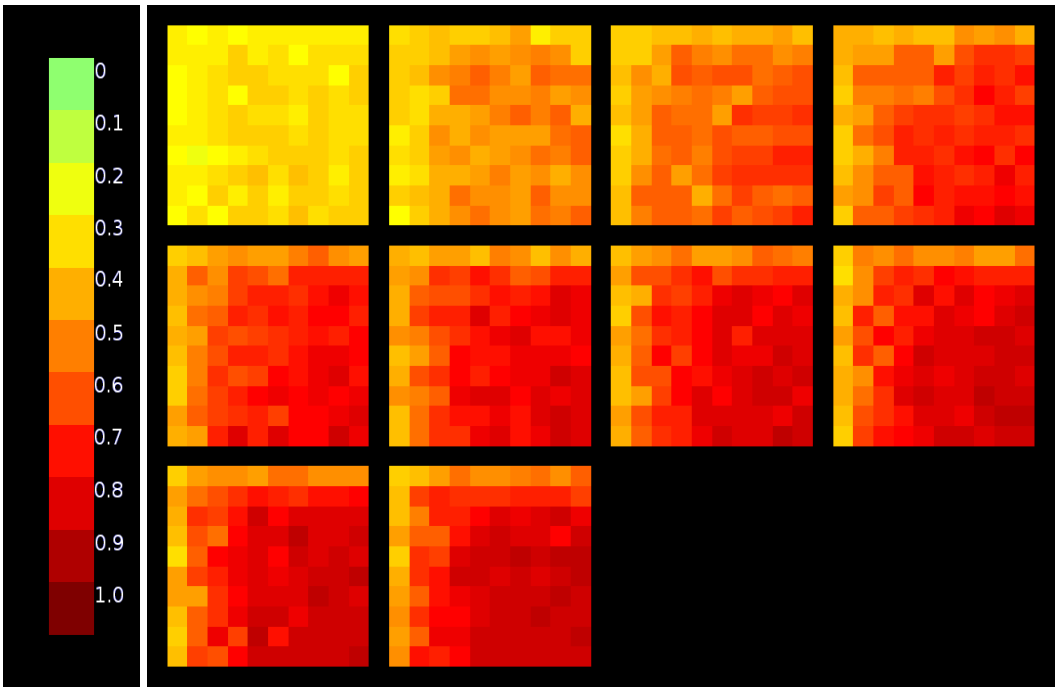
Näeme ka siin, et kasutada ei tohiks väheste neuronitega kihte. Pigem tuleks kasutada numbrite tuvastamiseks rohkem, kui viie neuroniliste kihtidega võrke. See-eest üldistust luua, et see kõikide ülesannete jaoks nõnda on ei ole võimalik, vaid pigem tuleks vaadata tendentsi hoiduda üliväheste neuronite arvuga kihtidest.

6.2. Kolme peidetud kihiga võrgustruktuuride analüüs

Nagu ka kahe peidetud kihiliste võrgustruktuuride jooniste korral on ka kolme peidetud kihiliste võrgustruktuuride jooniste korral horisontaalset telge pidi igal ruudul esimese kihi neuronite arv ja vertikaalset telge pidi teise kihi neuronite arv. Kuna aga on vajalik visualiseerida ka kolmandat peidetud kihti, siis igal pildil on erinev kolmanda kihi neuronite arv. Vasakul üleval oleva pildil on kolmanda kihi neuroneid üks ja neuronite arv suureneb igal järgmisel pildil.



Joonis 7: Treening andmetel täpsus kolme peidetud kihiliste võrgustruktuuridega.

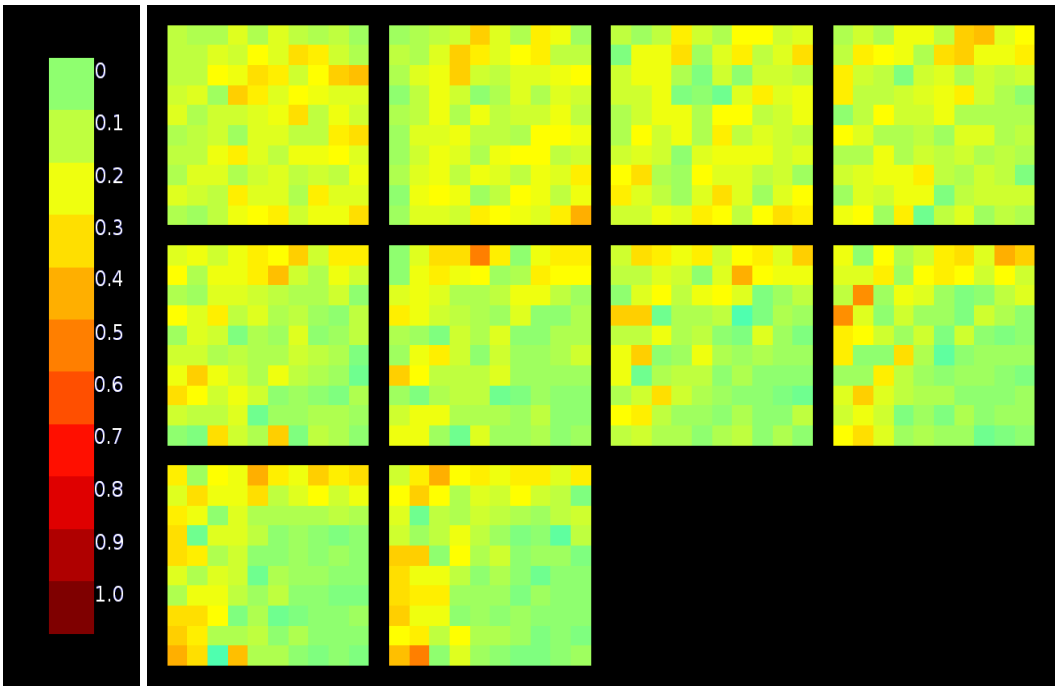


Joonis 8: Test andmetel täpsus kolme peidetud kihiliste võrgustruktuuridega.

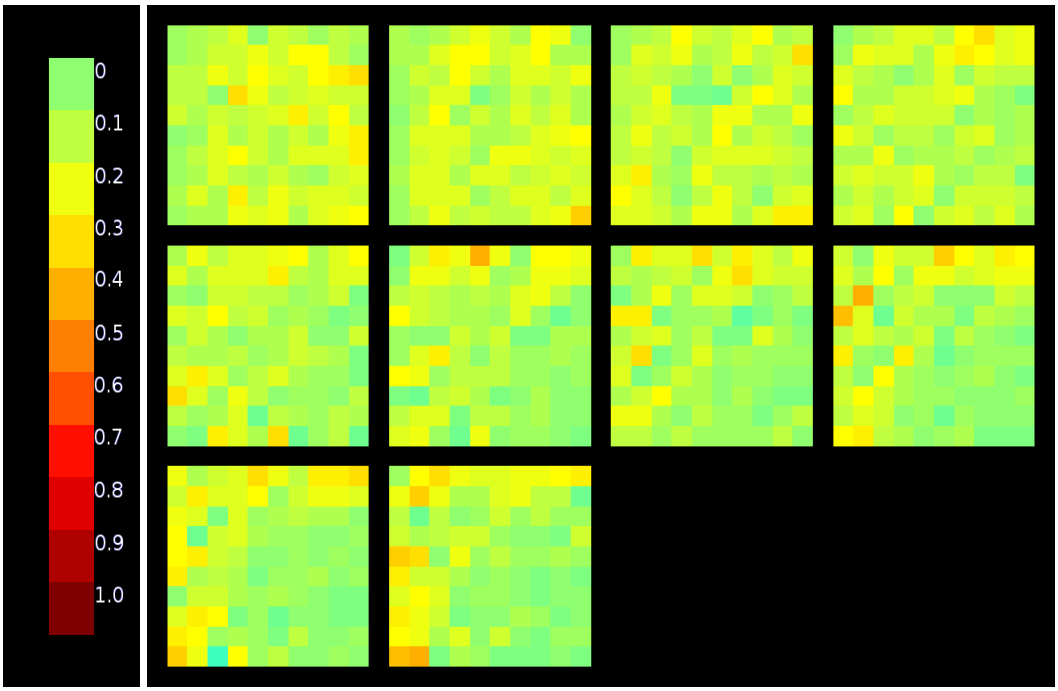
Nagu ka kahe kihiliste puhul on näha, et võrgu strukutuuri muutus on sümmeetriline, ehk kõik kihid on võrdväärsed. Samuti esineb ka siin informatsiooni kadu väheste neuronite tõttu vabalt valitud kihis.

Keskmine täpsuse protsent kolme peidetud kihiliste võrgustruktuuride puhul on treeningandmestikul 63.536% ja testandmestikul 58.328%. Maksimum on kolmekihiliste

võrgustruktuuride puhul treeningandmestikul 96.780% ja 88.750% testandmestikul. See on üllatavalt väiksem kahekihiliste maksimumist, mis võib tähendada liialt varajast õpetamise peatamist. Selle tulemus saavutas võrk, mille peidetud kihtide neuronite arv on vastavalt 10, 5 ja 7.



Joonis 9: Treeningandmetel täpsus kolme peidetud kihiliste võrgustruktuuridega.



Joonis 10: Testandmetel muutuste summa kolme peidetud kihiliste võrgustruktuuridega

Erinev kahe peidetud kihiliste võrgustruktuuride analüüsile on see, et muutuste summa kol-

me peidetud kihiliste võrgustruktuuride puhul on liialt mürane, et üldist põhjapanevat tõde väita, kuid mida suurem on kolmanda kihi neuronite arv, seda selgemalt tundub välja tulevat samasugune informatsioon - hoiduda tuleks ekstreemselt väikestest peidetud kihtidest.

7. Kokkuvõte

Kõrge tulemuslikkusega tehisnärvivõrgud on tavaliselt paljude kihtidega ja suurel arvul neuronitega. See-eest on vahel vajalik ka vähese arvutusvõimsusega seadmetel kasutadada tehisnärvivõrke. Sel juhul võib insener olla sunnitud kasutama väheste kihtide või väheste neuronitega tehisnärvivõrku. Töö eesmärgiks oli kasutades läbiproovimist tuua selgust väikeste tehisnärvivõrkude võrgustruktuuri valikusse.

Leiti, et väikeste tehisnärvivõrkude puhul MNIST andmestikul ei ole suurt võimekuse muutumist olenevalt kihtide arvust. Samuti, et vea suurus sõltub sama palju kõikide kihtide neuronitest, mis on üllatav, sest tihti kasutatakse tehisnärvivõrgu kihi neuronite arvu määramiseks lineaarselt langevat jada. Kahe kihiliste võrgustruktuuride parima tulemuslikkuse saavutas võrgustruktuur, mille peidetud kihtide arv on vastavalt 9 ja 10 tulemusega 97.260% treeningandmestikul ja 91.250% testandmetel. Kolme kihiliste korral oli selleks aga 10, 5 ja 7. See võrk saavutas treeningandmetel täpsuse 96.780% ja testandmetel 88.750%

Kuna käesolev töö kinnitas, et tehisnärvivõrgu tulemuslikust sõltuvalt võrgustruktuurist võib näha, kui sujuvat funktsiooni on tulevikus võimalik treenida ainult mõningaid võrke ja vahepealsed võrgustruktuurid lähendada. Samuti on huvitav treenimisele kulunud aja kulu vastavalt võrgustruktuurist ja suuremate võrkude puhul võrgustruktuuri muutmisel tulemuslikuse muutuse summa analüüs.

8. Resume

The usage of artificial neural networks can be difficult in practice because many different parameters have to be manually-tuned by the system developer. In this work we study the effect of the choice of network structure in small networks on their performance. This choice may be important in real-time data analysis on low computation ability devices such as mobile devices.

All possible neural networks with two or three hidden layers and up to 10 neurons in a layer were trained with varying degree of regularization to prevent over-fitting the dataset. The best performing regularization degree is chosen to be representative of a network structures ability to learn from training set.

Testing of artificial neural networks ability to learn is carried out on Mixed Nations Institute of Standards and Technology dataset. Dataset contains 60 000 examples of training data and

10 000 examples of test data. Dataset itself contains labeled grayscale images of numbers ranging from zero to nine and is frequently used to test different machine learning techniques. 5 000 training examples and 400 test examples were chosen at random in order to reduce training time. The conditions to stop learning were chosen by training artificial neural networks with extreme (very little or many neurons in every layer)network architecture. The error function change of at least 2×10^{-3} over 80 conjugate gradient training iterations for two hidden layered architectures and over 120 conjugate gradient training iterations for three layered network architectures were chosen.

All the programming required was carried out in programming language Octave. Main contribution of the author was to compose programs to find data necessary to train neural network weights such as gradient vector and cost function value and automate the process to such degree that thousands of network architectures could be trained automatically. All of the post-processing of data was also carried out in Octave environment.

The invariance of hidden layer depth was found by observing that the precision plot against different network structures is symmetrical up to the noise by the diagonal. This effect was observed both on two and three layered architecture. Very low performance was spotted when any one of the layers had a low number of neurons. This may be caused by information loss caused by allowing only a few neurons to hold the information about input data at any time during forward propagation. Plotting the sum of the gradient of precision showed clearly for two layered networks the necessity to not use a low number of neurons in either layer. For three layered network the pattern was less evident and became more apparent larger the third layer became. Data plots of neural network accuracy seem to suggest that the accuracy is smooth in terms of the network structure. It meaning that in subsequent training the networks could be trained more sparsely and it would still be representative. Also it would be more informative to test the neural networks training and testing times to help select the best network for any application.

Kasutatud kirjandus

Burnett, C.M.L. (2006). *An example artificial neural network with a hidden layer*.

Ciresan, D.C., U.Meier, and J. Schmidhuber (2012). "Multi-column Deep Neural Networks for Image Classification". In: *CoRR* abs/1202.2745. URL: http://arxiv.org/abs/1202.2745.

High Performance Computing Center, University of Tartu. http://www.hpc.ut.ee/alligaator_usage.

Hochreiter, S., P. Frasconi Y. Bengio, and J. Schmidhuber (2001). "A Field Guide to Dynamical Recurrent Neural Networks." In: ed. by John F. Kolen and Stefan C. Kremer. IEEE Press. Chap. Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies, pp. 237–243.

Labusch, Kai, Erhardt Barth, and Thomas Martinetz (2008). *Simple Method for High-Performance Digit Recognition Based on Sparse Coding*.

Lauer, Fabien, Ching Y. Suen, and Gérard Bloch (2007). "A trainable feature extractor for handwritten digit recognition". In: *Pattern Recognition*. URL: https://hal.archives-ouvertes.fr/hal-00018426.

LeCun, Y., C. Cortes, and C. J. C. Burges (1998). *THE MNIST DATABASE of handwritten digits*. http://yann.lecun.com/exdb/mnist/.

Octave (2015). *Brief Introduction to Octave*. https://www.gnu.org/software/octave/doc/interpreter/Introduction.html.

Rasmussen, C.E. (2001). *Conjugate Gradient*. http://learning.eng.cam.ac.uk/carl/code/minimize/minimize.m.

9. Lisad

9.1. *Lisa 1.* Kahe kihilise tehisnärvivõrgu kaofunktsioon

```
function [J grad] = nnCostFunction2(nn_params, ...
                                    input_layer_size, ...
                                    hidden_layer_size, ...
                                    hidden_layer_size2, ...
                                    num_labels, ...
                                    X, y, lambda)

    %   NNCOSTFUNCTION Implements the neural network cost function for a two
        hidden layer
    %   neural network which performs classification
    %   [J grad] = NNCOSTFUNCTON(nn_params, hidden_layer_size, num_labels, ...
    %   X, y, lambda) computes the cost and gradient of the neural network.
        The
    %   parameters for the neural network are "unrolled" into the vector
    %   nn_params and need to be converted back into the weight matrices.

    % Reshape nn_params back into the parameters Theta1 and Theta2, the
        weight matrices
    % for our 2 layer neural network
    ils = input_layer_size;
    hls = hidden_layer_size;
    hls2 = hidden_layer_size2;
    ols = num_labels; % output layer size

    sizeT1 = (ils +1) * hls;
    sizeT2 = (hls +1 ) * hls2;
    sizeT3 = (hls2 +1) * ols;

    Theta1 = reshape(nn_params(1:sizeT1), ...
    hls, (ils + 1));

    Theta2 = reshape(nn_params((sizeT1 + 1):(sizeT1+sizeT2)), ...
    hls2, (hls + 1));

    Theta3 = reshape(nn_params((1 + sizeT1 + sizeT2):(sizeT1 + sizeT2 +
        sizeT3)), ...
    ols, (hls2 + 1));
```

```
% Setup some useful variables
m = size(X, 1);

% Return values
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));
Theta3_grad = zeros(size(Theta3));

s = 0;

% Loops thorugh all the training examples
for i=1:m,

      % Feedforward
      x_i = [1 X(i,:)]';

      y_i = zeros(num_labels, 1);
      y_i(y(i)) = 1;

      z_2 = Theta1 * x_i;
      a_2 = [1; sigmoid(z_2)];
      z_3 = Theta2 * a_2;
      a_3 = [1; sigmoid(z_3)];
      z_4 = Theta3 * a_3;
      h = sigmoid(z_4);

      % Delta computation
      d_4 = h - y_i;
      d_3 = Theta3' * d_4;
      d_3 = d_3(2:end) .* sigmoidGradient(z_3);
      d_2 = Theta2' * d_3;
      d_2 = d_2(2:end) .* sigmoidGradient(z_2);


      % Computing gradient without regularisation
      Theta3_grad = Theta3_grad + (d_4 * a_3');
      Theta2_grad = Theta2_grad + (d_3 * a_2');
      Theta1_grad = Theta1_grad + (d_2 * x_i');
```

```
        s = s + sum(-y_i .* log(h) .- (1.-y_i) .* log(1.-h));

    end


    % Removing constant parameters from regularisation
    t_1 = Theta1(:, 2:end);
    t_2 = Theta2(:, 2:end);
    t_3 = Theta3(:, 2:end);

    % Computing regularisation term
    Theta1_grad = [Theta1_grad(:,1)/m ((Theta1_grad(:,2:end)/m) + (lambda/m)
        * t_1)];
    Theta2_grad = [Theta2_grad(:,1)/m ((Theta2_grad(:,2:end)/m) + (lambda/m)
        * t_2)];
    Theta3_grad = [Theta3_grad(:,1)/m ((Theta3_grad(:,2:end)/m) + (lambda/m)
        * t_3)];

    t_1 = sum(sum(t_1.^2));
    t_2 = sum(sum(t_2.^2));
    t_3 = sum(sum(t_3.^2));
    r = (lambda/(2*m))*(t_1 + t_2 + t_3);

    % Adding regularisation to the cost
    J = (1/m) * s + r;

    % Unroll gradients for returning
    grad = [Theta1_grad(:) ; Theta2_grad(:); Theta3_grad(:)];

end
```

9.2. *Lisa 2.* Kahe kihilise tehisnärvivõrgu abil numbrite tuvastamine

```
function p = predict(Theta1, Theta2, Theta3, X)
     %PREDICT Predict the label of an input given a trained neural network
     %   p = PREDICT(Theta1, Theta2, X) outputs the predicted label of X given
         the
     %   trained weights of a neural network (Theta1, Theta2)

     % Useful values
     m = size(X, 1);

     % You need to return the following variables correctly
     p = zeros(size(X, 1), 1);

     a1 = [ones(m,1), X];
     z2 = a1 * Theta1';
     a2 = [ones(size(z2,1),1), sigmoid(z2)];
     z3 = a2 *Theta2';
     a3 = [ones(size(z3,1), 1), sigmoid(z3)];
     z4 = a3 * Theta3';
     a4 = sigmoid(z4);

     [predict_max, index_max] = max(a4, [], 2);

     p = index_max;


     %=========================================================================

end
```

9.3. *Lisa 3.* Kahe kihiliste võrgustruktuuride treenimist läbiviiv programm

```
function [testdata, traindata] = NNU(X, Xtest, y, ytest, input)

load ordermatrix;

% Variables we will be using
lambdam = [0.01, 0.1, 1, 10, 100];
trainerror = zeros(numel(lambdam), 1);
testerror = zeros(numel(lambdam), 1);
threshold = 2e-3;
lambda = lambdam(order_matrix(input, 3));

% Defines what structure are we using
input_layer_size = 784; % 28 * 28 Input Images of Digits
num_labels = 10;        % 10 labels, from 1 to 10
hidden_layer_size = order_matrix(input, 1);
hidden_layer_size2 = order_matrix(input, 2);

% Useful variables
m = size(X, 1);
sizeT1 = (input_layer_size + 1) * hidden_layer_size;
sizeT2 = (hidden_layer_size + 1) * hidden_layer_size2;
sizeT3 = (hidden_layer_size2 + 1) * num_labels;

% Shorthand names
ils = input_layer_size;
hls = hidden_layer_size;
hls2 = hidden_layer_size2;
ols = num_labels; % output layer size

pred = [];
test_pred = [];

printf("Input %i defines network with Hidden layer size: %i 2nd Hls: %i
    Lambda: %i\n",
input, hidden_layer_size, hidden_layer_size2, lambda)

fprintf('\nGenerating initial Neural Network Parameters ...\n')
```

```
nn_params = zeros(1, sizeT1 + sizeT2 + sizeT3);

initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, hidden_layer_size2);
initial_Theta3 = randInitializeWeights(hidden_layer_size2, num_labels);
% Unroll parameters
initial_nn_params = [initial_Theta1(:); initial_Theta2(:); initial_Theta3(:)];
fprintf('\nTraining Neural Network...\n')

options = optimset('MaxIter', 120);

% Create "short hand" for the cost function to be minimized
costFunction = @(p) nnCostFunction2(p, ...
input_layer_size, ...
hidden_layer_size, ...
hidden_layer_size2, ...
num_labels, X, y, lambda);

% Now, costFunction is a function that takes in only one argument (the
% neural network parameters) train first with the initial parameters,
% save results to nn_params.
[cost, dummy] = costFunction(initial_nn_params);

nn_params = initial_nn_params;

i = 0;
do;

i += 1;
prev_cost = cost;
[nn_params, cost] = fmincg(costFunction, nn_params, options);


% Unrolling parameters
Theta1 = reshape(nn_params(1:sizeT1), ...
hls, (ils + 1));

Theta2 = reshape(nn_params((sizeT1 + 1):(sizeT1+sizeT2)), ...
hls2, (hls + 1));
```

```
Theta3 = reshape(nn_params((1 + sizeT1 + sizeT2):(sizeT1 + sizeT2 + sizeT3)),
    ...
ols, (hls2 + 1));

pred = predict(Theta1, Theta2, Theta3, X);
test_pred = predict(Theta1, Theta2, Theta3, Xtest);

trainerror = mean(double(pred == y));
testerror = mean(double(test_pred == ytest));
fprintf('\nTraining Set Accuracy: %f', trainerror * 100);
fprintf('\nTest Set Accuracy: %f\n', testerror * 100);

until(prev_cost - cost) < threshold || i < 20;

printf("Saving file %s\n", sprintf("NN2h-%d-%d-L%d", hls, hls2,
    order_matrix(input, 3)))
save(sprintf("NN2h-%d-%d-L%d", hls, hls2, order_matrix(input, 3)),
    "trainerror", "testerror")

endfunction
```

9.4. *Lisa 4.* Kolme kihilise tehisnärvivõrgu kaofunktsioon

```
function [J grad] = nnCostFunction3(nn_params, ...
                input_layer_size, ...
                hidden_layer_size, ...
                hidden_layer_size2, ...
                hidden_layer_size3, ...
num_labels, ...
X, y, lambda)
%NNCOSTFUNCTION Implements the neural network cost function for a three hidden
    layer
%neural network which performs classification
%   [J grad] = NNCOSTFUNCTON(nn_params, hidden_layer_size, num_labels, ...
%   X, y, lambda) computes the cost and gradient of the neural network. The
%   parameters for the neural network are "unrolled" into the vector
%   nn_params and need to be converted back into the weight matrices.

% Reshape nn_params back into the parameters Theta1 and Theta2, the weight
    matrices
% for our 2 layer neural network


% Useful variables
ils = input_layer_size;
hls = hidden_layer_size;
hls2 = hidden_layer_size2;
hls3 = hidden_layer_size3;
ols = num_labels; % output layer size

m = size(X, 1);

sizeT1 = (ils +1) * hls;
sizeT2 = (hls +1) * hls2;
sizeT3 = (hls2 +1) * hls3;
sizeT4 = (hls3 +1) * ols;

% Unrolling parameters
Theta1 = reshape(nn_params(1:sizeT1), hls, (ils + 1));

Theta2 = reshape(nn_params((sizeT1 + 1):(sizeT1+sizeT2)), hls2, (hls + 1));
```

```
Theta3 = reshape(nn_params((1 + sizeT1 + sizeT2):(sizeT1 + sizeT2 + sizeT3)),
    hls3, (hls2 + 1));

Theta4 = reshape(nn_params((1 + sizeT1 + sizeT2 + sizeT3):(sizeT1 + sizeT2 +
    sizeT3 + sizeT4)), ols, (hls3 + 1));


% return the following variables
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));
Theta3_grad = zeros(size(Theta3));
Theta4_grad = zeros(size(Theta4));

s = 0;

% feedforward and backpropagation part, one example at a time.
for i=1:m,

x_i = [1 X(i,:)]';

y_i = zeros(num_labels, 1);
y_i(y(i)) = 1;

z_2 = Theta1 * x_i;
a_2 = [1; sigmoid(z_2)];
z_3 = Theta2 * a_2;
a_3 = [1; sigmoid(z_3)];
z_4 = Theta3 * a_3;
a_4 = [1; sigmoid(z_4)];
z_5 = Theta4 * a_4;
h = sigmoid(z_5);

d_5 = h - y_i;
d_4 = Theta4' * d_5;
d_4 = d_4(2:end) .* sigmoidGradient(z_4);
d_3 = Theta3' * d_4;
d_3 = d_3(2:end) .* sigmoidGradient(z_3);
d_2 = Theta2' * d_3;
d_2 = d_2(2:end) .* sigmoidGradient(z_2);
```

```
Theta4_grad = Theta4_grad + (d_5 * a_4');
Theta3_grad = Theta3_grad + (d_4 * a_3');
Theta2_grad = Theta2_grad + (d_3 * a_2');
Theta1_grad = Theta1_grad + (d_2 * x_i');

s = s + sum(-y_i .* log(h) .- (1.-y_i) .* log(1.-h));

end


% exclude bias unit
t_1 = Theta1(:,2:end);
t_2 = Theta2(:,2:end);
t_3 = Theta3(:,2:end);
t_4 = Theta4(:,2:end);


% Regularize parameters
Theta1_grad = [Theta1_grad(:,1)/m ((Theta1_grad(:,2:end)/m) + (lambda/m) *
    t_1)];
Theta2_grad = [Theta2_grad(:,1)/m ((Theta2_grad(:,2:end)/m) + (lambda/m) *
    t_2)];
Theta3_grad = [Theta3_grad(:,1)/m ((Theta3_grad(:,2:end)/m) + (lambda/m) *
    t_3)];
Theta4_grad = [Theta4_grad(:,1)/m ((Theta4_grad(:,2:end)/m) + (lambda/m) *
    t_4)];

% Regulare cost function
t_1 = sum(sum(t_1.^2));
t_2 = sum(sum(t_2.^2));
t_3 = sum(sum(t_3.^2));
t_4 = sum(sum(t_4.^2));
r = (lambda/(2*m))*(t_1 + t_2 + t_3 + t_4);

% Cost for returning.
J = (1/m) * s + r;

% Unroll gradients
grad = [Theta1_grad(:) ; Theta2_grad(:); Theta3_grad(:); Theta4_grad(:)];

end
```

9.5. *Lisa 5.* Kolme kihiliste tehisnärvivõrgu abil numbrite tuvastamine

```
function p = predict(Theta1, Theta2, Theta3, Theta4, X)
     %PREDICT Predict the label of an input given a trained neural network
     %   p = PREDICT(Theta1, Theta2, Theta3, Theta4, X) outputs the predicted
         label of X given the
     %   trained weights of a neural network (Theta1, Theta2, Theta3, Theta4)

     % Useful values
     m = size(X, 1);

     % Return values; predictions.
     p = zeros(size(X, 1), 1);

     % Forward propagation
     a1 = [ones(m,1), X];
     z2 = a1 * Theta1';
     a2 = [ones(size(z2,1),1), sigmoid(z2)];
     z3 = a2 *Theta2';
     a3 = [ones(size(z3,1), 1), sigmoid(z3)];
     z4 = a3 * Theta3';
     a4 = [ones(size(z4,1),1), sigmoid(z4)];
     z5 = a4 * Theta4';
     a5 = sigmoid(z5);

     [predict_max, index_max] = max(a5, [], 2);

     p = index_max;

     %
         ==========================================================================

end
```

9.6. *Lisa 6.* Kolme kihiliste võrgustruktuuride treenimist läbiviiv programm

```
function [testdata, traindata] = NNU(X, Xtest, y, ytest, input)

load ordermatrix;

% Variables we will be using
lambdam = [0.01, 0.1, 1, 10, 100];
trainerror = zeros(numel(lambdam), 1);
testerror = zeros(numel(lambdam), 1);
threshold = 2e-3;
lambda = lambdam(order_matrix(input, 4));

% Defines what structure are we using
input_layer_size = 784; % 28 * 28 Input Images of Digits
num_labels = 10;        % 10 labels, from 1 to 10
hidden_layer_size = order_matrix(input, 1);
hidden_layer_size2 = order_matrix(input, 2);
hidden_layer_size3 = order_matrix(input, 3);

% Useful variables
m = size(X, 1);
sizeT1 = (input_layer_size + 1) * hidden_layer_size;
sizeT2 = (hidden_layer_size + 1) * hidden_layer_size2;
sizeT3 = (hidden_layer_size2 + 1) * hidden_layer_size3;
sizeT4 = (hidden_layer_size3 + 1) * num_labels;

% Shorthand names
ils = input_layer_size;
hls = hidden_layer_size;
hls2 = hidden_layer_size2;
hls3 = hidden_layer_size3;
ols = num_labels; % output layer size

pred = [];
test_pred = [];

printf("Input %i defines network with Hidden layer size: %i 2nd Hls: %i 3rd
    Hls: %i Lambda: %i\n",
input + 500, hidden_layer_size, hidden_layer_size2, hidden_layer_size3, lambda)
```

```
fprintf('\nGenerating initial Neural Network Parameters ...\n')


nn_params = zeros(1, sizeT1 + sizeT2 + sizeT3 + sizeT4);

initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, hidden_layer_size2);
initial_Theta3 = randInitializeWeights(hidden_layer_size2, hidden_layer_size3);
initial_Theta4 = randInitializeWeights(hidden_layer_size3, num_labels);

% Unroll parameters
initial_nn_params = [initial_Theta1(:); initial_Theta2(:); initial_Theta3(:);
   initial_Theta4(:)];
fprintf('\nTraining Neural Network...\n')

options = optimset('MaxIter', 120);

% Create "short hand" for the cost function to be minimized
costFunction = @(p) nnCostFunction3(p, ...
input_layer_size, ...
hidden_layer_size, ...
hidden_layer_size2, ...
hidden_layer_size3, ...
num_labels, X, y, lambda);

% Now, costFunction is a function that takes in only one argument (the
% neural network parameters) train first with the initial parameters,
% save results to nn_params.
[cost, dummy] = costFunction(initial_nn_params);

nn_params = initial_nn_params;

i = 0;
do;

i += 1;
prev_cost = cost;
[nn_params, cost] = fmincg(costFunction, nn_params, options);
```

```
% Unrolling parameters
Theta1 = reshape(nn_params(1:sizeT1), ...
hls, (ils + 1));

Theta2 = reshape(nn_params((sizeT1 + 1):(sizeT1+sizeT2)), ...
hls2, (hls + 1));

Theta3 = reshape(nn_params((1 + sizeT1 + sizeT2):(sizeT1 + sizeT2 + sizeT3)),
    ...
hls3, (hls2 + 1));

Theta4 = reshape(nn_params((1 + sizeT1 + sizeT2 + sizeT3):(sizeT1 + sizeT2 +
    sizeT3 + sizeT4)), ...
ols, (hls3 + 1));

pred = predict(Theta1, Theta2, Theta3, Theta4, X);
test_pred = predict(Theta1, Theta2, Theta3, Theta4, Xtest);

trainerror = mean(double(pred == y));
testerror = mean(double(test_pred == ytest));
fprintf('\nTraining Set Accuracy: %f', trainerror * 100);
fprintf('\nTest Set Accuracy: %f\n', testerror * 100);

until(prev_cost - cost) < threshold || i < 30;

printf("Saving file %s", sprintf("NN3h-%d-%d-%d-L%d", hls, hls2, hls3,
    order_matrix(input, 4)))
save(sprintf("NN3h-%d-%d-%d-L%d", hls, hls2, hls3, order_matrix(input, 4)),
    "trainerror", "testerror")

endfunction
```

9.7. *Lisa 7.* Sigmoid funktsioon

```
function g = sigmoid(z)
      %SIGMOID Compute sigmoid functoon
      %   J = SIGMOID(z) computes the sigmoid of z.

      g = 1.0 ./ (1.0 + exp(-z));
end
```

9.8. *Lisa 8.* Esialgne randoomsete kaalude valik

```
function W = randInitializeWeights(Lin, Lout)
      %RANDINITIALIZEWEIGHTS Randomly initialize the weights of a layer with
         L_in
      %incoming connections and L_out outgoing connections
      %   W = RANDINITIALIZEWEIGHTS(L_in, L_out) randomly initializes the
         weights
      %   of a layer with L_in incoming connections and L_out outgoing
      %   connections.
      %
      %   Note that W is set to a matrix of size(L_out, 1 + L_in) as
      %   the column row of W handles the "bias" terms
      %

      % Return values
      W = zeros(Lout, 1 + Lin);

      epsiloninit = 0.12;
      W = rand(Lout, 1 + Lin) * 2 * epsiloninit - epsiloninit;

end
```

9.9. *Lisa 9.* Programm käivitamaks treenimisprotsessi ühe käsurea käsuga

```
#! /usr/bin/octave -qf

# Neural network research bash script
arg_list = argv ();

printf("Training artificial neural network; Given input was %s\n",
    arg_list{1});

load traindata5000;
load testdata400;

more on;
warning('off', 'Octave:possible-matlab-short-circuit-operator');

if str2num(arg_list{1}) <= 500;
      cd NN2h
      NNU(X, Xtest, y, ytest, str2num(arg_list{1}));
else
      cd NN3h
      NNU(X, Xtest, y, ytest, str2num(arg_list{1}) - 500);
endif
```