LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

MAŠĪNMĀCĪŠANAS PIELIETOJUMS ELEKTRĪBAS MIRKĻA CENAS PAREDZĒŠANAI

BAKALAURA DARBS

Autors: **Sigurds Eglītis**
Studenta apliecības Nr.: se11006
Darba vadītājs: profesors Dr. dat. Jānis Zuters

RĪGA 2016

ANOTĀCIJA

Darba mērķis ir izpētīt mašīnmācīšanās pielietojumu konkrētas problēmas risināšanai. Izvēlētā pētāmā problēma ir elektrības cenas paredzēšana, izmantojot dažādus pieejamos faktorus par noteiktu laika periodu. Darbs tika izstrādāts sadarbojoties ar Viktoriju Bobinaiti, kura izstrādāja savu pētījumu „Modelling Electricity Price Expectations in a Day-Ahead Electricity Market", no šī pētījuma tika iegūti atbilstošie dati.

Darbā tika izveidota datorprogramma, kas pielieto specifisku mašīnmācīšanās veidu – neironu tīklus. Izveidota datorprogramma spēj parametrizēti izveidot neironu tīklus, ļaujot pielāgot gan apmācīšanas procesu, gan mainīt neironu tīkla struktūras parametrus, gan filtrēt pieejamos datus.

Izveidotā datorprogramma tika izmantota, lai analizētu pieejamos datus par ar elektrības tirgu saistītiem faktoriem un meklētu tādus faktorus, ar kuriem būtu iespējams paredzēt elektrības mirkļa cenu.

Atslēgvardi: mašīnmācīšanās, elektrības mirkļa cena, neironu tīkli, Qt

ABSTRACT

The purpose of this work – „Machine learning for the prediction of electricity spot price" – is the exploration of machine learning applications for solving a specific problem. The selected problem was prediction of electricity spot price using various factors collected over a period of time. This work was created by cooperating with Victoria Bobinate and her paper „Modelling Electricity Price Expectations in a Day-Ahead Electricity Market" and the electricity price data used was acquired from this work.

While making this work, a computer program was made that uses a specific type of machine learning – neural nets, to analyze the available data. The program allows user to generate neural nets from parameters, train these generated neural nets using training parameters and preprocess available data, to make training neural network on specific subsets easier.

The created software was then used to analyze the available data about electricity spot price and search for factors that allows to predict the said price.

Keywords: machine learning, electricity spot price, neural networks, Qt

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

IDE – integrētā izstrādes vide (*Integrated development environment*)

att. – attēls vai ilustrācija

MWh – megavatstundas

ODBC – Microsoft datu interfeiss, kas ļauj piekļūt relāciju un nerelāciju datu avotiem (*Open Database Connectivity*)

Qt – C++ IDE

MSE – vidējā kvadrātiskā kļūda (*mean square error*)

IEVADS

Pārdodot un pērkot preces ekonomiskā tirgū ir svarīgi rast priekšstatu par pārdodamā resursa vērtību, kas var būt atkarīga no daudziem faktoriem – preces piedāvājuma, pieprasījuma, preces ražošanu un piegādi ietekmējošiem faktoriem. Preces cenu ietekmējošos faktorus un to svarīgumu nav viegli paredzēt, bet tie ietekmē visizdevīgākos pārdošanas vai pirkšanas brīžus. Viens no šādiem resursiem ir elektrība, kura var tikt pārdota elektrības izsoļu tirgū, kurā vairākas puses izsludina piedāvājumus un pieprasījumus saražotās elektrības daudzumam par noteiktām cenām.

Mēģināt paredzēt šādu informāciju ir iespējams ar dažādām metodēm, viena no kurām ir izmantot mākslīgā intelekta algoritmus. Mākslīgā intelekta algoritmi spēj iemācīties sarežģītas sakarības starp pieejamajiem datiem un izvirzīt minējumus par iespējamo rezultātu. Mākslīgā intelekta priekšrocība ir spēja izmantot algoritmiskas aprēķināšanas pieeju elektroniskās ierīcēs, lai veiktu aprēķinus daudz ātrāk, kā to varētu darīt persona, tādejādi panākot lielāku aprēķinu apjomu, augstāku ticamību un precizitāti.

Šī darba ietvaros izstrādāju datorprogrammu, kas spēj apmācīt neironu tīklu balstoties uz pieejamajiem datiem par elektrības cenām, kuras izmantoju no Viktorijas Bobinaites pētījuma „Modelling Electricity Price Expectations in a Day-Ahead Electricity Market", iegūt statistiku par apmācīšanas procesu, saglabāt iegūtās derīgās konfigurācijas un izmantot apmācīto neironu tīklu, lai iegūtu minējumus par iespējamām vērtībām elektrības cenām.

Bakalaura darba mērķi:

1. Izpētīt mašīnmācīšanās, neironu tīklu pielietojumu, konkrēto datu apstrādei.
2. Pielietot iegūtās zināšanas konkrētajai problēmai un radīt risinājumu, kas spēj apstrādāt pieejamos datus.
3. Veikt datu pirmsapstrādi, lai filtrētu pieejamos datus ar noteiktiem filtrēšanas noteikumiem un radītu failus, kuri ļauj novērtēt dažādu faktoru grupu ietekmi.
4. Atrast tādas neironu tīklu konfigurācijas, kuras pietiekami precīzi ļauj paredzēt pieejamos datus.
5. Salīdzināt iegūtos rezultātus ar Viktorijas Bobinaites pētījuma rezultātiem un veikt izstrādātā neironu tīkla paredzētu rezultātu salīdzinājumu ar sagaidītajām vērtībām.

1. MĀKSLĪGAIS INTELEKTS

Mākslīgā intelekta izpēte ir izpētes lauks, kurā tiek mēģināts saprast to, kā radīt inteliģentu būtņu īpašības. Dažas no mākslīgā intelekta definīcijām [1]:

- Sistēmas, kas domā kā cilvēki.
- Sistēmas, kas uzvedas kā cilvēki.
- Sistēmas, kas domā racionāli.
- Sistēmas, kas uzvedas racionāli.

Galvenie mākslīgā intelekta izpētes lauki ir zināšanu apstrāde, plānošana, mācīšanās, valodas apstrāde, objektu atpazīšana. Radīt mākslīgo intelektu ir ne tikai noderīgi mūsu pašu – cilvēku – intelekta saprašanai, bet tam ir arī ļoti praktiski un plaši pielietojumi, kas var palīdzēt algoritmiski atrisināt noteiktas problēmas, un tādējādi uzlabot mūsu dzīves.

Viens no plaši pielietotiem mākslīgā intelekta veidiem ir mašīnmācīšanās, kurā, datora sistēmai vai programmatūrai intelekts tiek veidots ar dažādiem mācīšanās algoritmiem. Mašīnmācīšanās algoritmi ļauj sistēmai iegūt jaunas zināšanas balstoties uz tai pieejamajiem datiem un labojot kļūdainus pieņēmumus, tādejādi uzlabojot spēju spriest par situācijām un reaģēt atbilstoši datiem, kuras sistēma vēl nav apstrādājusi. Šie algoritmi ļauj radīt datorsistēmas, kuras nav pieciešams iepriekš programmēt konkrētai situācijai, bet ir iespējams paļauties uz to, ka tās apgūs jaunu informāciju un pielāgos reakciju atbilstoši sagaidītajai atbildei.

Pagaidām īstu mākslīgo intelektu nav izdevies radīt, bet noteiktas intelekta īpašību imitācijas ir iespējams sasniegt. Daudz ir izdevies panākt pētot bioloģiskās dzīvu būtņu īpašības – kādi ir smadzeņu neironu darbības pamatprincipi, kā tie ir savienoti savā starpā un ar citiem orgāniem. Pētījumi smadzeņu uzbūvē ir ļāvuši radīt mākslīgos neironu tīklus, kuri ļoti vienkāršotā veidā imitē bioloģisko nervu sistēmu spēju iemācīties un reaģēt.

1.1. Mašīnmācīšanās algoritmi

Mašīnmācīšanās algoritmi ļauj risināt problēmu, kas rodas mēģinot apvienot algoritmisku programmēšanu, kas sevī ietver programmatūras izveidošanu, kas veic vienu noteiktu uzdevumu, un ideju par mācīšanos, pielāgošanos nezināmai situācijai. Pielāgošanās iepriekš nezināmai situācijai nozīmē, ka arī pašam sistēmas veidotājam var nebūt īsti zināmi principi, kā sistēmai būtu jāreaģē uz noteiktiem ievades datiem, piemēram, programmētājs

visticamāk nezinās, kā pareizi identificēt plaušu vēzi rentgena attēlos, bet apmācīta mašīnmācīšanās datorprogramma spēs veikt šādu klasifikāciju [2].

Mašīnmācīšanos pielieto dažādu problēmu risināšanai. Sākot ar e-pasta mēstuļu filtrēšanu, meklēšanu internetā, personalizētu reklāmu piemeklēšanai, līdz pat tādām problēmām kā - viltojumu noteikšana, akcīžu tirdzniecība, zāļu izgatavošana, nestandarta situāciju noteikšana. Šādas problēmas var sadalīt dažādās problēmu sfēras - klasificēšana (datu sadalīšana noteiktās klasēs), regresijas analīze (funkcijas atrašana, kas aproksimē datus), grupēšana (datu objektu līdzīgu grupu sameklēšana), varbūtiskas blīvuma funkcijas novērtējums, datu dimensiju skaita samazināšana, iezīmju izgūšana no pieejamajiem datiem.

Mašīnmācīšanās algoritmu apmācīšana var notikt dažādi, noteikti algoritmi nekad nebeidz veikt mācīšanos un pielāgošanos jauniem faktoriem, citiem algoritmiem ir nepieciešama priekšapmācīšana. Tomēr visiem kopējā īpašība ir tas, ka datu daudzums un kvalitāte var stipri ietekmēt iegūto minējumu precizitāti. Ja datu ir pārāk maz, tad mašīnmācīšanās algoritms var tos iegaumēt un nespēt adekvāti reaģēt uz jauniem un neredzētiem datiem, bet ja to ir par daudz, vai ir izvēlēta nepareiza algoritma konfigurācija, apmācīšanas ilgums var būt pārāk liels un gala precizitāte var būt pārāk zema, lai to varētu praktiski pielietot.

1.2. Mašīnmācīšanās algoritmu apmācīšanas veidi

Mašīnmācīšanās algoritmi veic noteiktas sistēmas apmācīšanu, bet apmācīšanas process var atšķirties dažādos algoritmos. Daži no apmācīšanas veidiem [3]:

- Pārraudzītā mācīšanās – algoritmi, kuri mēģina atrast tādu funkciju, kas no dotajiem ievades datiem ļauj iegūt sagaidāmos datus. Šajā mācīšanās veidā ietilpst klasificēšanas problēmas, kurās, ievadītie dati tiek sadalīti iepriekš noteiktās klasēs, ņemot vērā piemērus klasificēšanai.
- Nepārraudzītā mācīšanās – algoritmi, kuri mēģina izveidot modeli ievades datiem un dara to nebalstoties uz piemēriem.
- Puspārraudzītā mācīšanās – algoritmi, kuri kombinē pārraudzītās un nepārraudzītās mācīšanās algoritmus, lai gan izveidotu modeli datiem, gan balstītos uz dotiem piemēriem.

- Stimulētā mācīšanās – algoritmi, kuri veic novērojumus, un mēģina nonākt līdz secinājumiem, par to, kāda būtu pareiza atbildes reakcija. Apmācīšana notiek mērot darbības sekas.
- Transdukcija – līdzīga pārraudzītās mācīšanās algoritmiem, bet stigri nenosaka funkciju, kuru nepieciešams pielāgot noteiktiem datiem. Tā vietā censās paredzēt jaunus izvades datus balsoties uz treniņa ievades un izvades datiem, un jauniem jauniem ievades datiem.
- Mācīšanās mācīties – algoritmi, kuri iemācās induktīvu novirzi, balstoties uz iepriekšējo pieredzi.

Populārākie mācīšanās veidi tiek apskatīti nākamajās nodaļās.

1.2.1. Pārraudzītā mācīšanās

Pārraudzītās mācīšanās algoritmi ir vieni no populārākajiem mašīnmācīšanās algoritmiem, jo tie ļauj risināt problēmas, kurās ir jāpanāk, lai dators iemācās sakarības datos, kuriem ir skaidri zināms sagaidāmais rezultāts vai klasificēšanas klases, piemēram burtu vai ciparu atpazīšana, un šīs problēmas ir viegli definēt no savāktiem datiem.

Pārraudzītā mācīšanās ir vispopulārākā metode neironu tīklu un lēmumu pieņemšanas koku apmācīšanai. Abu šo mākslīgā intelekta algoritmu apmācīšana ir ļoti atkarīga no sākotnēji dotās informācijas. Neironu tīklu gadījumā ir nepieciešams novērtēt kļūdu jeb novirzi no sagaidāmā rezultāta, balstoties uz kuru, tālāk tiek mēģināts koriģēt savstarpējos savienojumus starp neironiem mākslīgajā neironu tīklā. Lēmumu pieņemšānas koku gadījumā sākotnēji dotās klasifikācijas tiek izmantotas, lai noteiktu, kuri datu atribūti ir visnozīmīgākie un dod visvairāk informācijas.

Pirms veikt pārraudzīto mācīšanos, ir nepieciešams savākt datus par pētamo problēmu. Ja ir iespējams, var izmantot „eksperta metodi", kurā, balstoties uz cilvēka intuīciju, kurš ir saistīts ar pētāmās problēmas sfēru, tiek atlasīti atribūti un faktori, kuri būtu jāmēra. Ja šāds eksperts nav pieejams, tad var nākties vienkārši mēģināt savākt visu pieejamo informāciju cerībā, ka izdosies atrast pareizo informāciju mērīšanai. Šādi savāktai informācijai gan ir trūkums, ka tā saturēs daudz nederīgas informācijas un var daudzkārt palielināt sistēmas apmācīšanas ilgumu un palielināt iespēju, ka radīsies kļūdas.

Lieko informāciju un faktorus ir iespējams izņemt vai apvienot, un šo procesu sauc par raksturiezīmju apakškopas atrašanu. Atrodot raksturiezīmju apakškopas ar mazāk liekas informācijas ir iespējams palielināt datizraces algoritmu ātrdarbību un precizitāti.

Piemēri pārraudzītās mācīšanās pielietojumam – bioinformātika, rokraksta atpazīšana, objektu atpazīšana un datoru redze, runas atpazīšana.

1.2.2. Nepārraudzītā mācīšanās

Nepārraudzītās mācīšanās algoritmi cenšās izpildīt sarežģītāku mērķi kā pārraudzītās mācīšanās algoritmi – tiek mēģināts panākt, ka sistēma iemācās izdarīt kaut ko, par ko sistēmas radītājam nav ne jausmas, kā to izdarīt vai izmērīt. Parasti ir divas pieejas nepārraudzītai mācīšanās metodei. Pirmā ir mēģināt atalgot sistēmu par minējumiem, kuri ir veiksmīgi, un rada skaidrību par pētāmo datu iespējamo modeli. Šāda sistēmas apmācīšana rada noteiktu atalgojuma modeli, no kura sistēma vienmēr zin, kādu atalgojumu sagaidīt par noteiktām darbībām, iespējams, pat neradot nekādu datu modeli par dotajiem datiem. Otrā tipa nepārraudzītā mācīšanās ir grupēšana, kurā tiek meklētas līdzības apstrādājamajos ievades datos. Datu grupēšana tiek balstīta uz pieņēmumu, ka līdzīgo datu grupas arī aprakstīs un parādīs apslēpto datu modeli [4].

Nepārraudzītās mācīšanās algoritmi tiek veidoti tā, lai tie no datiem iegūtu noteiktu struktūru. Šīs struktūras kvalitāte tiek aprakstīta ar izmaksu funkciju, kuru sistēma cenšas minimizēt. Par labu iegūto struktūru tiek uzskatītas tādas datu struktūras, kuras labi apraksta apstrādājamo datu modeli, kurām ir pēc iespējas zemāka izmaksu funkcija, un kuras ir iespējams iegūt no vairākām datu apakškopām.

Gan pārraudzītās, gan nepārraudzītās macīšanās algoritmi var nonākt tādā apmācīšanas stadijā, kurā tie šķietami uzrāda ļoti precīzus rezultātus apmācīšanai paredzētajos datos, bet saskaroties ar jauniem datiem, kļūst ļoti neprecīzi. Šādu stāvokli sauc par pārpielāgošanos. Mašīnmācīšanās algoritms būtībā iegaumē datus, nevis sakarības tajos, kas noved pie nepareizas reakcijas. Nepārraudzītās mācīšanās gadījumā par to liecina robustuma trūkums datu modelī, jeb datu modeli iegūstot no dažādām pieejamo datu apakškopām, iegūtie datu modeļi savā starpā stipri atšķiras.

Piemēri nepārraudzītās mācīšanās algoritmu pielietojumam – sociālo tīklu analīze, cilvēku sadalīšana grupās, balstoties uz internetā veiktajiem pirkumiem.

1.2.3. Stimulētā mācīšanās

Stimulētā mācīšanās notiek apmācāmajai sistēmai mijiedarbojoties ar apkārtējās vides modeli un saņemot pozitīvu vai negatīvu atbildes reakciju veicot noteiktas darbības. Stimulētās mācīšanās gadījumā sistēmai var būt pieejama gan pilnīga informācija, par to, ko tā var novērot, gan tikai daļēja, kā arī var būt dažādi ierobežojumi darbībām, ko tā spēj noteiktos brīžos veikt. Simulēto mācīšanos var veikt gan tad, ja ir jau iegūts tās vides modelis vai tās simulācija, kurā sistēmai nāksies risināt problēmu, gan arī tad, ja vienīgais veids, kā iegūt informāciju par vidi ir ar to mijiedarboties.

Stimulētā mācīšanās var tikt veikta dažādos veidos, kuri izveido uzvedības politikas – noteiktus vēlamās uzvedības plānus, kuri satur pie laba rezultāta novedošas darbību secības. Visvienkāršākais veids ir veikt pilno darbību politiku pārlasi, kurā tiek apskatītas katras iespējamās darbību secības gala rezultāts un tiek meklēta labākā. Tomēr pilnā pārlasa prasa daudz laika, no kura liela daļa var tikt iztērēta apskatos sliktas darbību secības. Eksistē labāki risinājumi, kā piemēram laiciskās atšķirības metode, Monte Karlo metodes, tiešā politikas meklēšana.

Stimulēto mācīšanos var labi pielietot mašīnmācīšanās problēmām, kurās ir jāveic izvēle starp īstermiņa ieguvumu un ilgtermiņa ieguvumu, piemēram, darbību plānošanā, robotu uzvedībā, galda spēlēs (šahā, dambretē, go).

1.3. Mākslīgie neironu tīkli

Mākslīgie neironu tīkli ir tādas sistēmas, kuras ir spējīgas aproksimēt nelineāras funkcijas un strukturāli atgādina dabīgās nervu sistēmas - cilvēku smadzeņu un to šūnu jeb neironu izkārtojumu. Mākslīgā neironu tīklā īstu neironu funkcijas modelē ar apstrādes elementiem, kuri reaģē līdzīgi kā bioloģiskie neironi, kas saņemot noteiktu impulsu, aktivizējas un pastiprina vai slāpē signālu citiem savienotiem neironiem. Mākslīgie neironu tīkli spēj izpildīt darbības paralēli, atšķirībā no algoritmiska uzdevuma sadalījuma apakšuzdevumos, kuri tiek izpildīti secīgi [5].

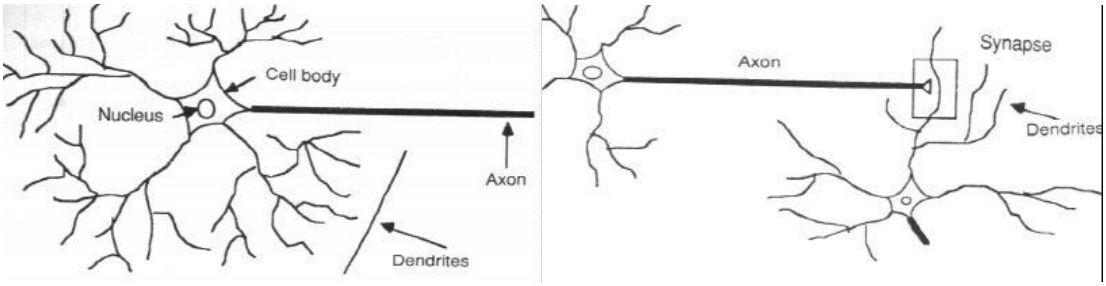
Mākslīgo neironu apstrādes elementus parasti grupē noteiktos slāņos, kuri vai nu saņem sākotnējo informāciju, nodod to citiem neironu slāņiem, vai arī satur mākslīgā neironu tīkla galarezultātu. Savienojumi starp neironiem tiek modelēti kā svaru vērtības, kas apsraksta katra sasaistītā neirona efektu uz nākamā neirona vērtību. Vienvirziena neironu tīklos, informācija

tiek ievadīta ievades slānī, un tiek nodota tālāk pa neironu tīklu, kamēr tiek izrēķinātas izvades slāņa vērtības.

Lai gan mākslīgo neironu tīklu ideja ir smēlusies iedvesmu bioloģiskajās nervu sistēmās, modernās imlementācijas lielāku nozīmi piešķir nevis līdzībai ar bioloģiskajiem neironu tīkliem, bet gan statistisku metožu un signāla apstrādes metožu pielietojumam.

1.3.1. Cilvēka smadzeņu neironu uzbūve

Lai gan pilnībā nav izprasts veids, kā notiek mācīšanās un kognitīvu īpašību iegūšana bioloģiskajos neironu tīklos, bioloģiskā atsevišķu neironu uzvedība un uzbūve ir zināma. Neirona šūnai ir kodols, šūnas ķermenis, dendrīti un aksons. Aksons, garš un taisns neirona šūnas atzarojums, savienojas ar citas neironu šūnas dendrītu, veidojot ķīmisku savienojumu, kuru sauc par sinapsi, un nodod tālāk elektrisku signālu, kuru rada neirona aktivizācija. Neironu var aktivizēt citas neironu šūnas, kuras spēj uztvert signālus no apkārtējās vides (receptori), vai arī cita neirona sūtītais signāls [6].

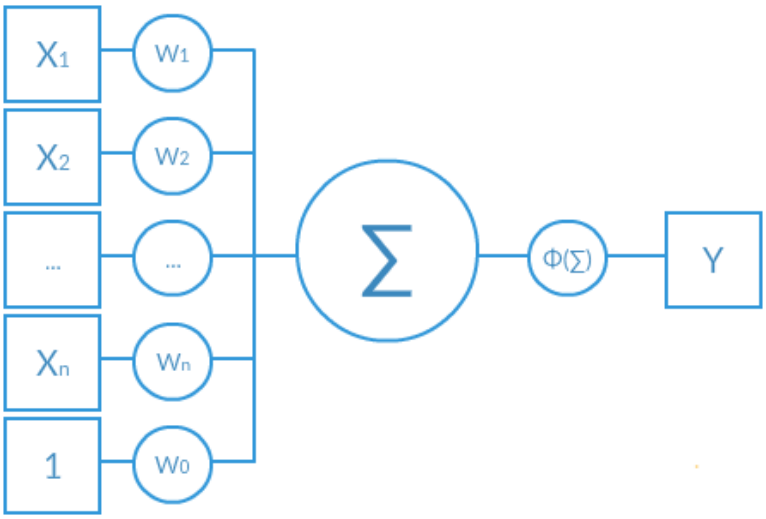


1.1. att. **Neirona un neironu savienojuma shematisks attēlojums [7]**

Katram neironam var būt signāli, kas ne tika ipastiprina tā aktivizāciju, bet arī slāpē to. Neirons aktivizējas tikai tad, ja tā aktivizējošie signāli ir stiprāki par tā slēpējošiem signāliem. Reāli bioloģiskie neironi reaģē viens ar otru arī dažādos sarežģītos veidos, piemēram, savienojumi ar citiem neironiem var būt nevis ķīmisku sinapšu veidā, bet arī ar elektriskas atstarpes savienojumiem, kā arī neirona savienojumi mēdz mainīt savas īpašības laikā, piemēram neiroplasticitāti, jaunu savienojumu veidošanos. Tomēr, lai modelētu neirona darbības principus pietiek arī ar vienkāršotu bioloģiskā neirona modeli, kurā galvenais uzsvars tiek likts uz neirona spēju summēt slāpējošus un aktivizējošus signālus un veikt citu neironu aktivizāciju, kādā noteiktā veidā.

1.3.2. Mākslīgais neirons

Mākslīgā neironu tīkla visvienkāršākais pamatelements ir mākslīgais neirons. Mākslīgais neirons sastāv no noteikta skaita ievaddatiem $X_1, X_2, \dots, X_n$, katram no kuriem ir piesaistīts nozīmīgums jeb svars $W_1, W_2, \dots, W_n$, noslieces svara W_0, un aktivizācijas funkcijas $\Phi(\sum)$, kas ģenerē neirona apstrādes rezultātu. Neirona darbība vienmēr notiek virzienā no ievaddatiem uz neirona rezultāta Y aprēķināšanu. Mākslīgiem neironiem parasti tiek piešķirts arī papildus noslieces svars W_0, kurš kā ievadi vienmēr saņem 1. Tas ir nepieciešams, lai būtu iespējams mainīt ne tikai aktivizācijas funkcijas slīpumu, bet arī panākt aktivizācijas funkcijas nobīdi [8]

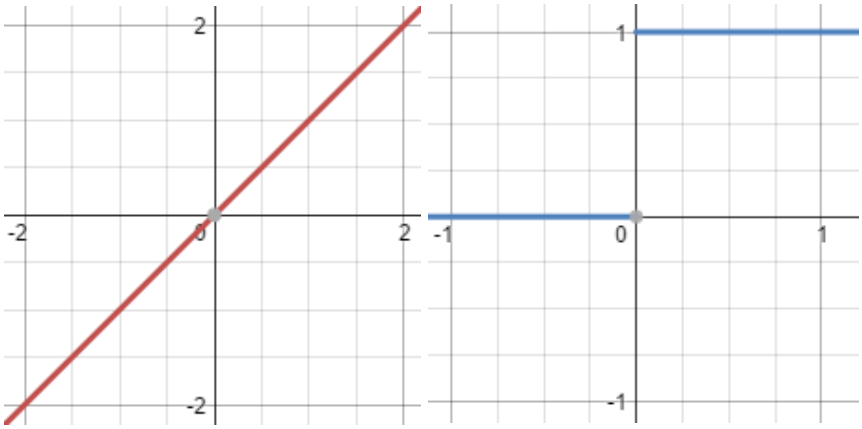


1.2. att. **Mākslīgā neirona shēma**

Ievaddatus apstrādā pareizinot katru ievaddatu mainīgo X_i ar tam saistīto svaru W_i, un aprēķinot kopējo summu $\sum_{i=0}^{n} W_i X_i$. Tālāk šo summu apstrādā ar aktivizācijas funkciju, kas nosaka, kāda vērtība būs neirona izvades mainīgajā $Y = \Phi(\sum_{i=0}^{n} W_i X_i)$. Aktivizācijas funkcijas var būt dažādas, bet apmācīšanu var labāk veikt tad, ja tām piemīt noteiktas īpašības. Ja aktivizācijas funkcija ir nelineāra, tad neironu tīkls spēs aproksimēt universālas funkcijas. Lai neironu tīkls spētu izmantot gradienta mācīšanās metodes, kurās mazākā kļūda tiek meklēta pārvietojoties pa kļūdas funkciju un meklējot minimumu, aktivizācijas funkcijai ir jābūt nepārtraukti atvasināmai. Citas svarīgas aktivizācijas funkcijas īpašības – monotonitāte, gludums, funkcijas tuvošanās nullei, ja parametra vērtība tuvojas nullei [8].

1.3.3. Tipiskās mākslīgā neirona aktivizācijas funkcijas

Visvienkāršākās mākslīgā neirona aktivizācijas funkcijas ir identitātes funkcija un binārā soļa funkcijas. Identitātes funkcija $\Phi(x) = x$ ļauj neironam atgriezt svērto ievades datu summu, kā izejas signālu un ir noderīga veicot mākslīgā neironu tīkla apmācīšanu reālu skaitļu vērtību iegūšanai. Binārā soļa funkcija $\Phi(x) = \begin{cases} 0, x < 0; \\ 1, \ x \geq 0; \end{cases}$ savukārt ļauj viegli nodalīt gadījumus, kuros neirona aktivizācija ir notikusi, un kuros nē, tādēļ šo funkciju var izmantot klasifikācijas problēmu risināšanai [8].



1.3. att. **Lineāra (pa kreisi) un binārā soļa (pa labi) aktivizācijas funkcijas**

Labāka alternatīva binārajai soļa funkcijai ir Sigmoida funkcija.

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

Atšķirībā no binārās soļa funkcijas tā ir nepārtraukta, kas ļauj atrast tās atvasinājumu visos šīs funkcijas punktos, turklāt pareizi veidojot algoritmisku implementāciju ir iespējams samazināt nepieciešamo darbību apjomu, salīdzinot ar citām aktivizācijas funkcijām, jo Sigomida funkcijas atvasinājums veidojas no šīs pašas funkcijas vērtības.

$$\Phi'(x) = \Phi(x)(1 - \Phi(x))$$

Detalizēta dažādu aktivizācijas funkciju un to atvasinājumu tabula.

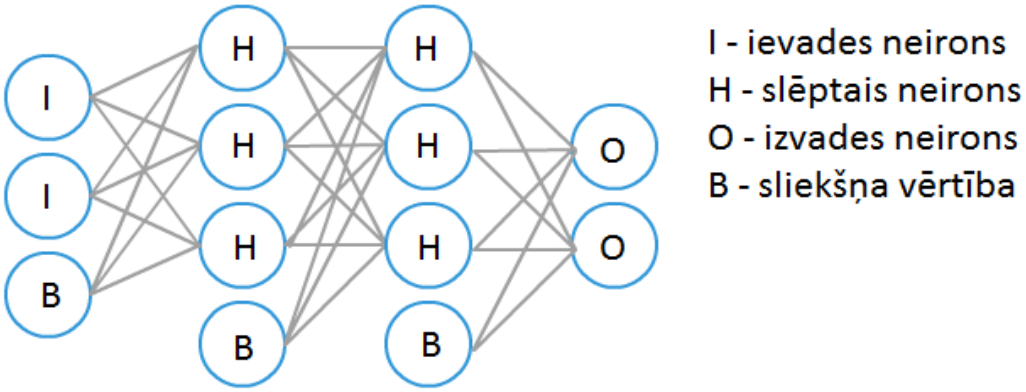
1.1.Tabula

Dažādas mākslīgā neirona aktivizācijas funkcijas

Aktivizācijas funkcija	Formula	Atvasinājums
Lineārā	$\Phi(x) = x$	$\Phi'(x) = 1$
Binārā soļa	$\Phi(x) = \begin{cases} 0, & x < 0; \\ 1, & x \geq 0; \end{cases}$	$\Phi'(x) = \begin{cases} 0, & x \neq 0; \\ citādi\ nav \end{cases}$
Sigmoida	$\Phi(x) = \frac{1}{1 + e^{-x}}$	$\Phi'(x) = \Phi(x)(1 - \Phi(x))$
Parametrizēta Sigmoida	$\Phi(x) = \frac{1}{1 + e^{-\beta x}}$	$\Phi'(x) = \beta\Phi(x)(1 - \Phi(x))$
TanH	$\Phi(x) = \frac{2}{1 + e^{-2x}} - 1$	$\Phi'(x) = 1 - \Phi(x)^2$
ArcTan	$\Phi(x) = \tan^{-1}(x)$	$\Phi'(x) = \frac{1}{x^2 + 1}$

1.3.4. Neironu tīkla slāņi un uzbūve

Mākslīgos neironu tīklus var sadalīt trīs kategorijās pēc to arhitektūras – vienslāņu vienvirziena, vairākslāņu vienvirziena un rekurentie neironu tīkli. Neironi vienslāņa un vairākslāņu arhitektūrās ir izvietoti vairākos slāņos, kuri katrs veic savu funkciju. Neironu tīkla ievades dati tiek ievadīti ievades slānī, kurā vērtības tiek vienkārši saglabātas katrā no neironiem, slēptie slāņi apstrādā iepriekšējā slāņa neironu vērtības un sagatavo tās nākamajam slānim, bet izejas slāņa neironi satur mākslīgā neironu tīkla apstrādātās vērtības. Katrā slānī, izņemot izvades slāni, parasti implementējot neironu tīklu tiek izveidots vēl viens papildus neirons, kura vērtība vienmēr ir 1, un kurš darbojas kā visu nākamā slāņa neironu sliekšņa svara vērtība. Vienslāņu vienvirziena neironu tīkli atšķiras no vairākslāņu vienvirziena neironu tīkliem ar to, ka tiem nav slēpto slāņu, bet ir tikai ievades un izvades slāņi.



1.4. att. **Vairākslāņu vienvirziena mākslīgā neironu tīkla shēma**

Savukārt rekurento neironu tīklu arhitektūrā, neironu tīklu veidojošie neironi var būt saslēgti ne tikai vienā virzienā, no ievades datiem uz izvades datiem, bet arī veidojot vienvirziena ciklus. Šādi neironu tīkli spēj iemācīties ne tikai pareizu izvades datu vērtības atbilstoši ievades datiem, bet arī secību datos, kas ir noderīgi, piemēram, pielietojot neironu tīklus teksta apstrādei.

1.3.5. Mākslīgā neironu tīkla parametri

Daži no parametriem, kurus var mainīt mākslīgā mākslīgajos neironu tīklos – mācīšanās ātrums, mācīšanās inerce, ivades datu troksnis, treniņa un pārbaudes tolerances.

1.3.5.1. Mācīšanās ātrums

Mācīšanās ātrums ir faktors, kuru piemēro mākslīgā neirona tīkla svaru izmaiņām veicot neironu tīkla apmācīšanu. Mazs mācīšanās ātrums palielina apmacīšanās ilgumu, toties ļaut neironu tīklam mācīties precīzāk. Pārāk mazu mācīšanās ātrumu izvēle var novest pie tā, ka neironu tīkls nonāk apmācīšanās lokālajā minimumā – konfigurācijā, kurā mazas neironu tīkla svaru vērtības tikai palielina neirona tīkla neprecizitāti kopumā, bet nav sasniegta vislabākā konfigurācija. Izvēloties lielas neironu tīkla mācīšanās ātruma vērtības, ir iespējams panākt, ka neironu tīkls mācās ātrāk, tomēr tad pastāv iespēja, ka neironu tīkls mācās pārāk haotiski, un nespēj nonākt pie optimālas konfigurācijas, pat ja tāda pastāv.

1.3.5.2. Mācīšanās inerce

Mācīšanās inerce norāda to daudzumu no mākslīgā neirona svaru izmaiņām, kuras tiek saglabāta turpinot neironu tīkla apmācīšanu. Šī parametra efekts ir tāds, ka neironu tīkls var „paātrināt" savu apmācīšanos noteiktā virzienā, kā arī kavē neironu tīkla virzīšanos nevēlamu konfigurāciju virzienā.

1.3.5.3. Ievades datu troksnis

Piešķirot neliela apjoma nejaušu troksni neironu tīkla ievades datiem var tikt uzlabota neironu tīkla spēja neiestrēgt lokālos kļūdu minimumos, kā arī panākta izvairīšanās no neironu tīkla pārapmācīšanās.

1.4. Neironu tīklu apmācīšana un apstāšanās nosacījumi

Tipiski neironu tīklu apmācīšana tiek viekta, aprēķinot katram no pieejamajiem datu elementiem neironu tīkla rezultātu un salīdzinot to ar sagaidāmo rezultātu. No sagaidāmā un reāli iegūtā rezultāta iegūst neironu tīkla radīto kļūdu, kura tālāk tiek izplatīta ar atpakaļizplatīšanās algoritmu. Atpakaļizplatīšanās algoritms ļauj aprēķināt katra neirona svara ietekmi uz rezultātu, un attiecīgi nepieciešamās korekcijas. Izrēķinātās svaru korekcijas tiek veiktas neironu tīklā, un apstrāde pāriet pie nākamā datu elementa un turpinās līdz tiek sasniegti neironu tīklam piemēroti apstrādes beigu nosacījumi.

Ir dažādi nosacījumi, pēc kuriem var spriest, vai neironu tīkla apmācīšana varētu būt jābeidz. Pirmais un vienkāršākais ir vienkārši pietiekami ilga apmācīšana, kuru mēra epohās. Katra epoha ir viens pilns treniņa datu apstrādes cikls, attiecīgi, neironu tīkla apmācīšanu varētu beigt, piemēram, pēc 1000 šādiem cikliem, jo iespējams, ir sasniegts labākais iespējamais apmācīšanas rezultāts un tālāka apmācīšana var radīt tikai neirona tīkla pārapmācīšanu.

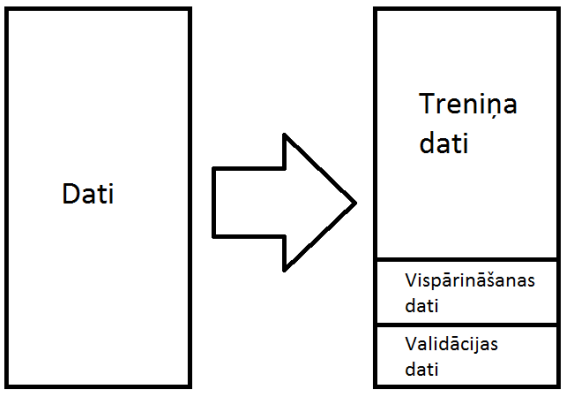
Vēl ir iespējams mērīt neironu tīkla precizitāti un vidējo kvadrātisko novirzi no sagaidāmajiem rezultātiem. Neirona precizitāte ir to gadījumu proporcija, kuros neironu tīkls ir klasificējis datus pareizi, vai arī pietiekami tuvu, regresijas apmācīšanas gadījumā, tipiski šī apmācīšanas nosacījuma lielums varētu būt 0,95. Vidējā kvadrātiskā novirze (MSE) savukārt ir lielums, kuru rēķina pēc formulas

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(P_i - E_i)^2,$$

kur P_i ir treniņa datu grupas elementa neironu tīkla izrēķinātā vērtība, bet E_i ir sagaidītā vērtība rezultātam. Tā skaitliski parāda, cik ļoti neironu tīkls kļūdās savos rezultātos kopumā, un gadījumos, kuros ir zināms, kāds kļūdas lielums vairs nav nozīmīgs, var kalpot kā apmācīšanas beigšanas nosacījums.

1.4.1. Datu sadalīšana

Lai veiktu pilnīgāku mākslīgā neironu tīkla apmācīšanu un iegūtā rezultāta novērtēšanu, pieejamos datus nepieciešams sadalīt trīs apakškopās – treniņa datu kopā (training set), vispārināšanas datu kopā (generalization set) un validācijas datu kopā (validation set). Sadalīšanu datu kopās vislabāk ir veikt maksimāli nejaušā veidā, lai neiestājas situācijas, kurās veicot neironu tīkla apmācīšanu atkārtoti, neironu tīkls mācās pārāk vienveidīgi.



1.2. att. **Shematisks attēlojums datu sadalīšanai**

Treniņa datu kopā iedalītie datu elementi tiek izmantoti neironu tīkla apmācīšanai, un pilna neironu tīkla apmācīšana vienreiz ar šo kopu tiek saukta par epohu. Tipiski šīs datu kopas apjoms ir apmēram 60-80% no visiem pieejamajiem datiem.

Vispārināšanas datu kopā tiek iedalīti dati, kurus izmanto, lai novērtētu neironu tīkla apmācīšanās progresu, katras epohas beigās un noteiktu, kā neironu tīkls spēj tikt galā ar iepriekš neredzētiem un netrennētiem datiem. Šīs datu kopas apjoms no kopējiem datiem ir apmēram 10-20%.

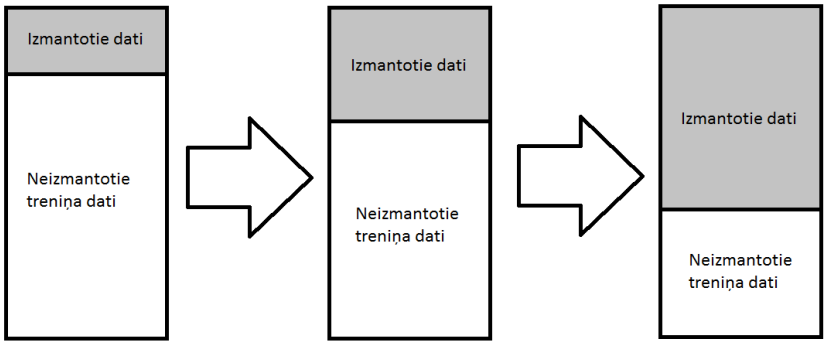
Validacijas datu kopā iedalītie dati tiek izmantoti, lai noteiktu neironu tīkla kopējo precizitāti, pēc tam, kad neironu tīkla apmācīšana ir beigusies. Šīs kopas apjoms no kopējiem datiem ir apmēram 10-20%.

1.4.2. Advancētas datu sadalīšanas metodes

Mēģinot apmācīt neironu tīklus ar lieliem datu apjomiem, neironu tīkla apmācīšana paliek lēna, bet ir iespējams pielietot dažas datu sadalīšanas metodes, kuras šo procesu paātrina.

1.4.2.1. Apmācīšanas datu kopas palielināšana

Ir iespējams trenēt neironu tīklus neizmantojot visu pieejamo treniņa datu kopu uzreiz, bet gan tikai tās daļu, kas tiek pamazām papildināta, līdz tiek aptverta visa treniņa datu kopa. Šāda treniņu datu kopas apstrāde ļauj panākt to, ka neironu tīklam ir iespēja sākt mācīšanos no ātrāk apstrādājamas datu kopas, un iziet cauri vairākām apmācīšanās iterācijām, pirms tas nonāk stadijā, kurā tiek katras iterācijas apmācīšanas garums ir salīdzinoši liels.

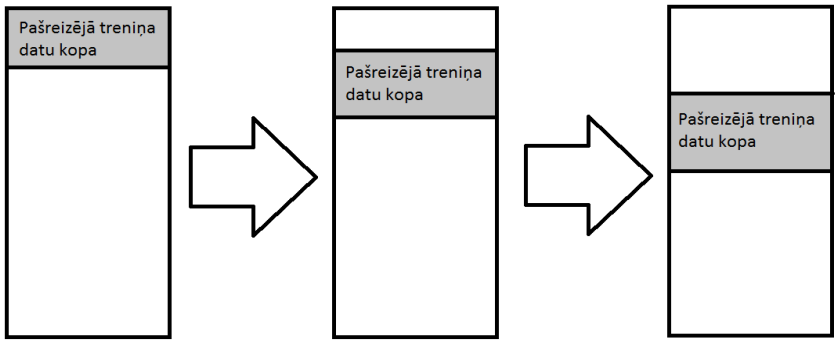


1.5. att. **Shematisks attēlojums apmācīšanas datu kopas palielināšanai**

1.4.2.2. Apmācīšanas datu kopas virzīšana

Apmācīšanas datu kopas virzīšana notiek izvēloties noteikta lieluma apakškopu no visiem apmācīšanai izmantotajiem datiem, un „virzot" šo apakškopu pāri visiem pieejamajiem treniņa datiem, pievienojot daļu jaunus elementus, daļu atmetot un daļu paturot. Tādā veidā

tiek panākts tas, ka katra apmācīšanas iterācija ir īsa, bet apmācīšanai tiek izmantoti visi treniņu datu kopas elementi.



1.6. att. **Shematisks attēlojums apmācīšanas datu kopas virzīšanai**

1.5. Kļūdu rēķināšana un backpropagation algoritms

Neironu tīkla koriģēšana un apmācīšana notiek balstoties uz izrēķinātā rezultāta un sagaidāmā rezultāta starpības jeb kļūdas izplatīšanu neironu tīklā un neironu tīkla svaru koriģēšanas atbilstoši tam, cik ļoti katrs svars ir radījis novirzi no sagaidāmās vērtības. To cik ļoti katrs svars ir iespaidojis kļūdu rašanos, ir iespējams izzināt ar tā saucamo backpropagation algoritmu [9].

Backpropagation algoritma darbības pamatsoļi ir sekojoši:

1. Tiek aprēķināti neironu tīkla izejas dati. Kur y ir neirona vērtība, ϕ ir aktivizācijas funkcija, k ir neirona indekss slānī, n ir maksimālais neironu skaits iepriekšējā slānīm, w_{ik} ir svars starp iepriekšējā slāņa i-to un pašreizējā slāņa k-to elementu un x ir neirona vērtība.

$$y_k = \phi\left(\sum_{i=1}^{n} w_{ik} x_i\right)$$

2. Katram no izejas neironiem tiek aprēķināta novirze no sagaidāmās vērtības. $\delta_k = p_k - x_k$, p ir paredzētā vērtība.
3. Tiek rēķinātas katra slēptā slāņu un ieejas slāņa neironu kļūdas. Katra no šo slāņu neironu kļūdām veidojas, saskaitot nākamā slāņa visu saistīto neironu kļūdu, neironu vērtību.

$\delta_k = (\sum_{i=1}^{n} w_{ki}\delta_i)$, ja neirons nav izejas slānī (δ_i ir nākamā slāņa neirona kļūda, w_{ki} ir k-tā neirona un nākamā slāņa neirona i sasaistes svars).

4. Kad pilnīgi visu neironu kļūdas ir izrēķinātas, tiek rēķinātas katra neironu savienojuma koriģētais svars w'_{ik}, kur i un k ir neironu svara indeksi, δ_k ir nākamā slāņā neirona kļūda, $\frac{d\phi_i(e)}{de}$ ir i-tā neirona aktivizācijas funkcijas atvasinājuma vērtība, y_i ir

$$w'_{ik} = w_{ik} + \eta\delta_k \frac{d\phi_i(e)}{de} y_i$$

5. Izrēķinātās izmaiņas tiek piešķirtas

$$w_{ik} = w'_{ik}$$

1.6. Neironu tīklu ierobežojumi

Galvenais mākslīgo neironu ierobežojums ir tas, ka tie būtībā darbojas kā sistēmas, kuru saturs nav zināms, jeb melnās kastes sistēmas. Sistēmas apmācības laikā iegūtā konfigurācija var nedot īpaši skaidrāku priekšstatu par to, kāds tad īsti ir apmācībai izmantoto datu modelis, bet vienkārši dot labus minējumus par sagaidāmo rezultātu. Turklāt, šiem neironu tīkla aprēķinātajiem rezultātiem var nebūt absolūta precizitāte – neironu tīkls aprēķinās sagaidāmās vērtības, bet to precizitāti var ietekmēt nepilnīgi dati par pētāmo problēmu, nepareizi veikta neironu tīkla apmācīšana un konfigurēšana.

Izvēloties neironu tīkla parametrus ir jārēķinās ar to, ka neironu tīkls ar pārāk mazu neironu skaitu būs ar mazāku informācijas kapacitāti, kā neironu tīkls ar lielāku neironu skaitu, jo neironu tīklā ir mazāk iespējamo konfigurāciju un attiecīgi arī ar to ir iespējams modelēt mazāk sarežģītas problēmas. No otras puses, izvēloties pārāk lielu neironu skaitu, palielinās iespēja, ka neironu tīkls nonāks stāvoklī, kurā tas iemācās testa datus, nevis sakarības starp tiem.

2. IZVEIDOTĀS SISTĒMAS APRAKSTS

Veicot pētījumu, veidoju datorprogrammu programmēšanas vidē Qt, kas implementē neironu tīklu tā, lai būtu iespējams apstrādāt pieejamos datus par elektrības cenām. Izveidotā datorprogramma spēj parametrizēti sadalīt pieejamos datus treniņa, vispārināšanas un pārbaudes datu kopās, veikt neironu tīkla apmācīšanu un apmācīšanas rezultātu izvadi, veikt datu paredzēšanu, balstoties uz treniņā iegūtajam zināšanām, kā arī eksportēt neirona tīkla svarus vēlākai izmantošanai.

Apstrādājamos datus ir iespējams filtrēt, izvēloties, kā katra kolonna tiks apstrādāta un veidot priekšapstrādātus failus, kuros atrodas tikai izvēlētajā veidā filtrētie dati. Datu filtrēšanu var veikt izvēloties katrai datu kolonnai vienu no vairākiem apstrādes veidiem – neapstrādāt, iekļaut kā ieejas datus, iekļaut noteiktu skaitu iepriekšējo vērtību kā atsevišķas ievades datu kolonnas, iekļaut kā kolonnu, kuru nepieciešams paredzēt, iekļaut kā kolonnu informācijai, kura neietekmē ieejas vai izejas datus. Izvēloties iespēju iekļaut iepriekšējās vērtības kā atsevišķas kolonnas, lietotājam ir iespēja izvēlēties arī kolonnu skaitu.

Programmas veidotais neironu tīkls un tā apmācīšana var tikt parametrizēts mainot mācīšanās ātrumu, mācīšanās inerci, pieļaujamo vidējo kvadrātisko kļūdu, maksimālo epohu skaitu, slēptā slāņa neironu skaitu. Ieejas un izejas slāņa izmēri tiek noteikti izmantojot priekšapstrādātajā failā esošo informāciju. Veicot neironu tīkla atkārtotu apmācīšanu, tā konfigurācija tiek ģenerēta neņemot vērā iepriekšējās apmācīšanas reizes. Neironu slāņi neironu tīklā ir ar dažādām aktivizācijas funkcijām, ievades slānī un izvades slānī – identitātes funkcija, bet slēptajā slānī parametrizēta Sigmoida funkcija, kurai tiek piemērots parametrs $\beta = 0.00001$, lai panāktu, ka aktivizācijas funkcija pārāk lielām un pārāk mazām parametra vērtībām nekļūtu par 1 vai 0, kas rada problēmas mācīšanās gaitā – mācīšanās paliek ļoti lēna, jo Sigmoida funkcijas atvasinājums šajos gadījumos kļūst par 0.

2.1. Pieejamie dati

Pētījumam tika izmantoti dati par elektrības cenām no 2013. gada 3. jūnija līdz 2016. gada 10. oktobrim. Zemāk aprakstītas pieejamās datu kolonnas.

2.1. tabula

Pieejamo datu kolonnu apkopojums

Nosaukums	**Datu tips**	**Formāts/mērvienība**
Sezona	Teksts	„Summer", „Autumn", „Winter", „Spring"
Dienas tips	Teksts	„Working days", „Weekend"
Datums	Teksts	<diena>-<mēnesis>-<gads>
Laiks	Teksts	<sākuma stunda> - <beigu stunda>
Mirkļa cena	Skaitlis	EUR/MWh
Patēriņš LV	Skaitlis	MWh
Patēriņš LT	Skaitlis	MWh
Paredzētais patēriņš LV	Skaitlis	MWh
Paredzētais patēriņš LT	Skaitlis	MWh
Elspot apjoms pārdošanai, LV	Skaitlis	MWh
Elspot apjoms pirkšanai, LV	Skaitlis	MWh
Elspot apjoms pārdošanai, LT	Skaitlis	MWh
Elspot apjoms pirkšanai, LT	Skaitlis	MWh
Saražotais apjoms, LV	Skaitlis	MWh
Saražotais apjoms, LT	Skaitlis	MWh
Saražotā apjoma prognoze, LV	Skaitlis	MWh
Saražotā apjoma prognoze, LT	Skaitlis	MWh
Elektrības plūsma, LV	Skaitlis	MWh
Elektrības plūsma, LT	Skaitlis	MWh
Saražotais mazos elektrības ģeneratoros (<10MW)	Skaitlis	MWh
Saražotais hidroelektrostacijās	Skaitlis	MWh
Saražotais termālajās elektrostacijās	Skaitlis	MWh
Saražotais vēja ģeneratoros	Skaitlis	MWh

Season	Type of day	Date	Hours	Spot_price, EUR/MWh	Consumption in Latvia, MWh	Consumption in Lithuania, MWh
Summer	Working days	03-06-2013	00 - 01	31.02	572	752
Summer	Working days	03-06-2013	01 - 02	30.05	548	728
Summer	Working days	03-06-2013	02 - 03	30.04	537	721
Summer	Working days	03-06-2013	03 - 04	30.03	517	701
Summer	Working days	03-06-2013	04 - 05	28.13	528	733
Summer	Working days	03-06-2013	05 - 06	40.81	615	900
Summer	Working days	03-06-2013	06 - 07	64.24	754	1076
Summer	Working days	03-06-2013	07 - 08	59.93	873	1211
Summer	Working days	03-06-2013	08 - 09	61.91	934	1262
Summer	Working days	03-06-2013	09 - 10	63.86	967	1274
Summer	Working days	03-06-2013	10 - 11	62.4	961	1269
Summer	Working days	03-06-2013	11 - 12	59.17	937	1260
Summer	Working days	03-06-2013	12 - 13	64.6	963	1273
Summer	Working days	03-06-2013	13 - 14	60.89	956	1256
Summer	Working days	03-06-2013	14 - 15	51.78	944	1248
Summer	Working days	03-06-2013	15 - 16	45.38	923	1210
Summer	Working days	03-06-2013	16 - 17	46.64	891	1169

2.1. att. **Piemērs pieejamajiem datiem**

2.2. Programmas moduļu apraksts

Izveidotā programma sastāv no četriem galvenajiem moduļiem, kuri, katrs apstrādā noteiktu daļu neironu tīkla apmācīšanas un datu sagatavošanas procesa.

Galvenie moduļi ir veidoti atbilstoši katrai darbībai, kuru lietotājam nepieciešams veikt ar sistēmu:

- Modulis „Datu glabāšana”
- Modulis „Datu apakškopas”
- Modulis „Datu apmācīšanas process”
- Modulis „Informācijas žurnalēšana”

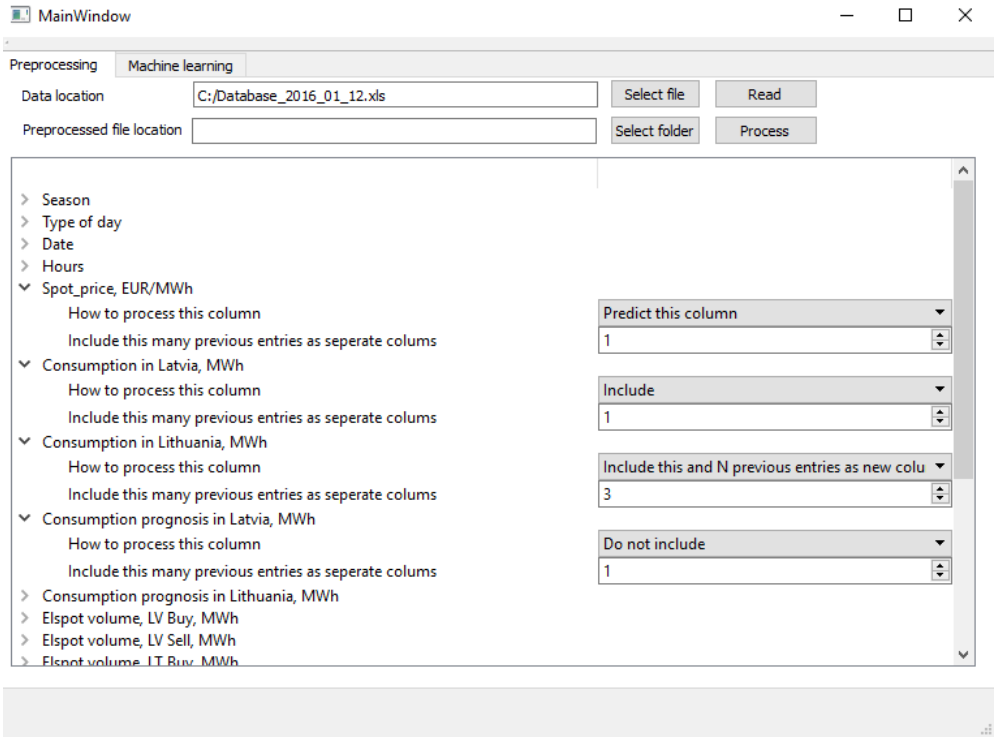
2.2.1. Modulis „Datu glabāšana un priekšapstrāde”

2.2.1.1. Nolūks

Datu glabāšanas modulis paredzēts datu ielasīšanai no sākotnējā datu faila formātā „.xls” izmantojot ODBC, uzglabāšanai izvēlētos apstrādes veidus katrai kolonnai, un priekšapstrādes datu eksporta veikšanai.

2.2.1.2. Funkcija

Modulis sāk darboties, kad lietotājs izvēlās apstrādājamo datu failu un nospiež darbības pogu „Read", kas atrodas programmas sadaļas „Preprocessing" „Data location" rindiņā. Ja ir izvēlēts datu fails un to ir iespējams atvērt ar ODBC metodi kā SQL datubāzi, tad tiek apstrādātas visas šī faila kolonnas un izveidoti apstrādāšanas nosacījumu ieraksti sadaļas „Preprocessing" tabulā.



2.2. att. **Lietotāja saskarne modulim „Datu glabāšana un priekšapstrāde"**

Izvēloties apstrādes darbību „Process", „Preprocessed file location" izvēlētajā sistēmas failu mapē tiks izveidots priekšapstrādāts fails pēc nosacījumiem, kādi ir norādīti priekšapstrādes filtru tabulā.

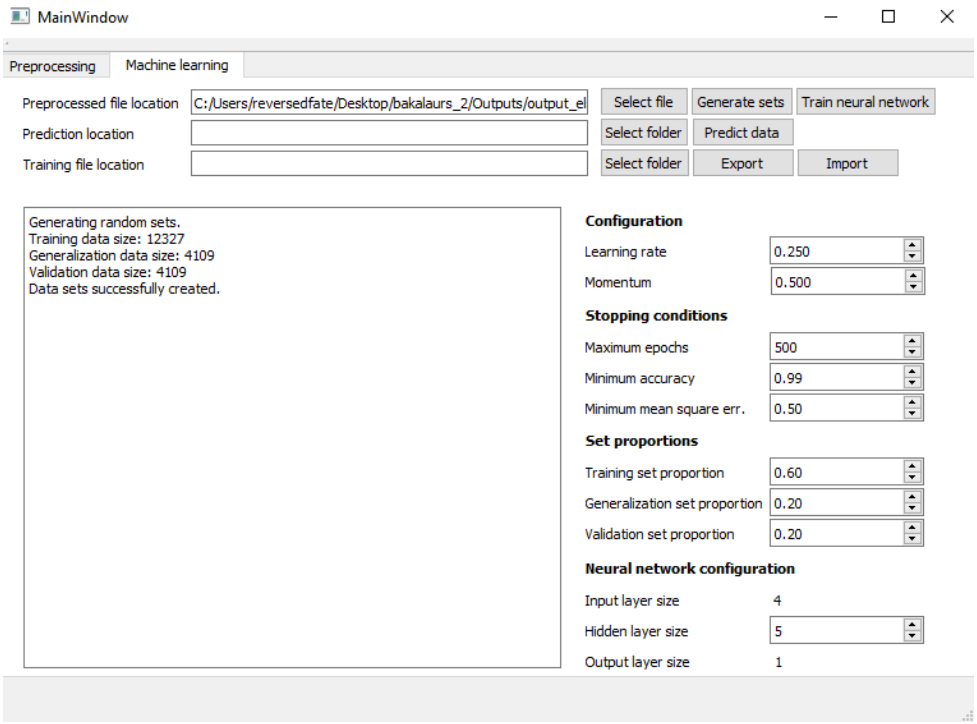
2.2.2. Modulis „Datu apakškopas"

2.2.2.1. Nolūks

Modulis „Datu apakškopas" paredzēts no priekšapstrādes sagatavotā faila ielasīto datu sadalīšanai treniņa, vispārināšanas un validācijas datu kopās un to tālākai glabāšanai, kamēr tiek veikta neironu tīkla apmācīšana.

2.2.2.2. Funkcija

Modulis sāk darboties tad, kad lietotājs izvēlas priekšapstrādāto datu failu programmas sadaļā „Machine learning” un nospiež darbības pogu „Generate sets”. Atbilstoši izvēlētajām datu apakškopu proporcijām tiek pēc nejaušības principa sadalīts priekšapstrādātā datu faila saturs, rezultātā izvadot informāciju par galējo elementu skaitu katrā no datu kopām. Ja neizdodas atvērt failu, tiek izvadīta informācija par kļūdu.



2.3. att. **Piemērs programmas darbībai veicot datu sadalīšanu apakškopās**

2.2.3. Modulis „Datu apmācīšanas process”

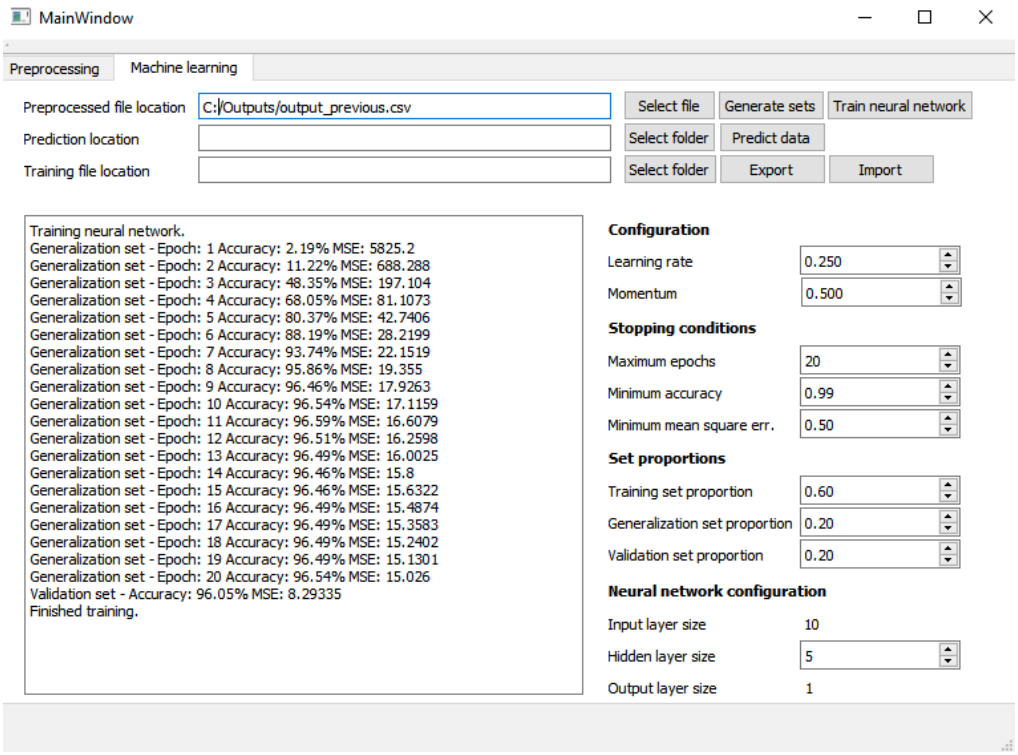
2.2.3.1. Nolūks

Modulis „Datu apmācīšanas process” paredzēts darbībām ar nerionu tīklu – neironu tīkla apmācīšanai, vērtību paredzēšanai, konfigurācijas importam un eksportam.

2.2.3.2. Funkcija

Modulis darbojas nospiežot darbības pogas „Train neural network”, „Predict data”, „Export”, „Import”.

Nospiežot darbības pogu „Train neural network”, tiek uzģenerēts jauns neironu tīkls, izmantojot sadaļā „Neural network configuration” norādītos ievades, slēptā un izvades slāņā izmērus un tiek veikta šī neironu tīkla apmācīšana izmantojot sadaļās „Configuration” un „Stopping configuration” norādītos parametrus. Tiek izvadīta informācija par apmācīšanas progresu, parādot katras epohas pašreizējo kļūdu un vidējo kvadrātisko novirzi, par pamatu ņemot ģeneralizācijas kopu un galējā precizitāte, par pamatu ņemot validācijas kopu. Ja nav izveidotas datu apakškopas, tiek parādīta atbilstoša informācija.



2.4. att. **Piemērs apmacīšanas procesam**

Nospiežot darbības pogu „Predict data” tiek veikta visu datu kopu paredzēšana veidojot „Prediction location” laukā norādītajā sistēmas datu mapē jaunu failu formātā „prediction_<laika zīmogs>.csv”. Šādi ģenerētajā failā pirmajā rindā atradīsies informācija par ievades datu kolonnu skaitu un izvades datu kolonnu skaitu pirmajā rindā. Nākamajās rindās seko dati noteiktā secībā – vispirms visi ievades dati, tad visi paredzētie dati, un visbeidzot dati, kuri trennējot tika uzskatīti par pareizajām neironu tīkla izejas datiem.

10	1										
27.01	25.4	26.01	25.1	25.38	25.51	25.47	26.01	26.02	38.09	26.9819	27.01
58.37	55.05	52.09	36.07	36.03	35.04	35.03	35.03	34	34.18	58.4554	58.37
42.25	42.28	31.25	34.98	49.93	49.96	46.56	47.2	42.83	42.25	42.2905	42.25
88.05	88.03	88.04	88.04	88.08	80.04	48.01	42.31	40.35	40.18	88.0977	88.05
47.27	47.23	41.15	42.13	46.18	47.2	46.11	46.15	46.2	46.18	47.3237	47.27
78.26	78.28	78.27	76.1	78.25	76.01	55.09	31.51	31.22	31.41	78.3244	78.26
48.04	33.06	32.02	31.09	31.1	32.01	32.08	33.03	38.03	54.05	48.1352	48.04
54.45	55.04	54.52	60.01	65.06	77.07	77.09	90.04	100.06	97.1	54.5037	54.45
31.06	31.05	31.03	31.06	31.09	46.05	47.12	47.83	47.67	43.62	31.0609	31.06
48.07	48.09	53	60.03	60.02	60.02	60.04	60.07	60.05	51.03	48.13	48.07
33.04	33.07	43.06	53.05	77.51	75.08	77.52	77.53	77.56	77.55	33.057	33.04

2.5. att. **Piemērs izeksportētajiem paredzējuma datiem ar 10 ievades datiem, 1 neirona tīkla izvadīto rezultātu un 1 sagaidīto rezultātu**

Nospiežot darbības pogu „Export”, tiek veikta neironu tīkla, ja tāds ir izveidots, neironu savstarpējo svaru eksportēšana failā „configuration_<laika zīmogs>.csv”. Šis fails atradīsies „Training file location” lauka norādītajā sistēmas datu mapē. Faila saturu veido neironu tīkla ievades, slēptā un izejas slāņā izmēri pirmajā faila rindā, nākamajās faila rindās tiek saglabāti neironu tīkla svari. Nospiežot darbības pogu „Import” tiks veikta neironu tīkla izveidošana no iepriekš izeksportētajiem neironu tīkla svariem tālākai izmantošanai datu paredzēšanai.

10	5	1		
236.912	295.406	-325.144	-123.617	-180.863
21.2327	0.333189	33.5935	-28.8887	-18.2422
11.6907	-3.33059	21.6888	-24.7314	-14.7711
7.99562	-4.10432	17.9171	-25.6715	-15.7747
1.17387	-7.66466	11.6946	-25.0648	-16.9968
-2.61181	-10.322	6.37313	-25.9373	-17.8451
-8.69332	-14.7567	1.884	-27.4582	-20.3403
-12.5861	-17.7755	-1.15106	-30.014	-23.4775
-17.547	-22.9861	-6.29992	-31.3882	-26.7346
-20.5511	-26.1465	-8.91827	-33.0771	-29.3623
-1.34925	-0.9506	0.091052	-0.62628	-0.47382
298.035				
393.918				
-472.644				
-160.501				
-239.971				
90.3276				

2.6. att. **Piemērs izeksportētai neironu tīkla konfigurācijai neironu tīklam ar 10 neironiem ievades slānī, 5 slēptajiem neironiem un 1 izvades slāņa neironu**

2.3. Datu analīze

Izmantojot izveidoto datorprogrammu, veicu dažādus mēģinājumus iegūt labas neironu tīkla konfigurācijas un ieejas datus. Pieejamos datus sadalīju sešos priekšapstrādes failos, katrā no kuriem glabājās noteikta tipa faktori.

- „output_consumption.csv", kurā kā apstrādājamie dati tika iekļautas visas kolonnas, kuras saturēja informāciju par patēriņu Latvijā un Lietuvā un kā sagaidāmie dati – kolonna „Mirkļa cena".
- „output_elspot.csv", kurā kā apstrādājamie dati tika iekļautas visas kolonnas, kuras satur informāciju par Elspot pirkšanas un pārdošanas apjomiem un kā sagaidāmie dati – kolonna „Mirkļa cena".
- „output_net.csv", kurā kā apstrādājamie dati tika iekļautas visas kolonnas, kuras saturēja informāciju par elektrības plūsmu Latvijā un Lietuvā un kā sagaidāmie dati – kolonna „Mirkļa cena".
- „output_previous.csv", kurā kā apstrādājamie dati ir iekļautas „Mirkļa cena" kolonnas 49 iepriekšējās vērtības un kā paredzamā vērtība – pašreizējā vērtība kolonnā „Mirkļa cena".
- „output_production.csv", kurā kā apstrādājamie dati ir norādītas visas kolonnas, kuras attiecas uz elektrības saražoto daudzumu Latvijā un Lietuvā.
- „output_production_types.csv", kurā kā apstrādājamie dati ir norādīti saražotās elektrības daudzuma apjomi pa ražošanas tipiem – mazajās elektrības ražotnēs, hidroelektrostacijās, termoelektrostacijās un vēja ģenerātoros. Kā sagaidāmie dati tika izmantota kolonna „Mirkļa cena".
- „output_various.csv", kurā kā apstrādājamie dati tika iekļautas kolonnas „Patēriņš LV", „Patēriņš LT", „Saražotais LV", „Saražotais LT", „LV bilances plūsma", „LT bilances plūsma", „Saražotais hidroelektrostacijās", „Saražotais termoelektrostacijās", „Saražotais vēja ģeneratoros", „Saražotais mazos elektrības ģeneratoros (<10MW)" un sagaidāmie izejas dati – „Mirkļa cena".

Katram no šiem sagatavotajiem failiem tika veikta neironu tīkla apmācīšana vismaz desmit reizes, saglabājot labāko iegūto rezultātu, ar dažādiem parametriem:

- 3000 maksimālās epohas, 99% sagaidāmā precizitāte, 5 minimālā vidējā kvadrātiskā kļūda, 3 slēptie neironi, 0.25 mācīšanās ātrums, 0.5 inerce.

- 3000 maksimālās epohas, 99% sagaidāmā precizitāte, 5 minimālā vidējā kvadrātiskā kļūda, 5 slēptie neironi, 0.25 mācīšanās ātrums, 0.5 inerce.
- 3000 maksimālās epohas, 99% sagaidāmā precizitāte, 5 minimālā vidējā kvadrātiskā kļūda, 8 slēptie neironi, 0.25 mācīšanās ātrums, 0.5 inerce.
- 3000 maksimālās epohas, 99% sagaidāmā precizitāte, 5 minimālā vidējā kvadrātiskā kļūda, 10 slēptie neironi, 0.25 mācīšanās ātrums, 0.5 inerce.

Tālāk seko apkopotie rezultāti, katram no pārbaudītajiem failiem un katram eksperimentu veidam. Tabulā kā katra eksperimenta rezultāts ir norādīta galējā precizitāte izmantojot validācijas datu kopu pēc neironu tīkla apmācīšanas beigām, kas var iestāties sasniedzot maksimālo epohu skaitu, pietiekamu precizitāti vai vidējo kvadrātisko kļūdu izmantojot vispārināšanas datu kopu. Iegūtā precizitāte apraksta to gadījumu proporciju, kurās neironu tīkla radītā vērtība atšķīrās no sagaidītās vērtības ne vairāk kā par 10%.

2.2. tabula

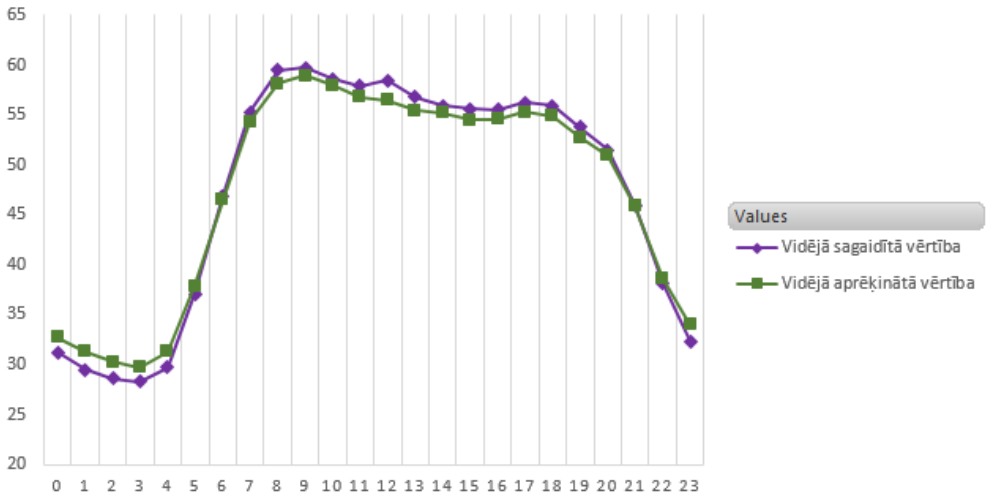
Dažādu sagatavoto izvades failu iegūto rezultātu apkopojums

Fails	**H = 3**	**H = 5**	**H = 8**	**H = 10**
output_consumption.csv	29.69% MSE: 240.998	24.09% MSE: 283.459	29.52% MSE: 194.673	29.73% MSE: 192.679
output_elspot.csv	25.67% MSE: 381.705	22.17% MSE: 363.263	20.44% MSE: 302.429	24.84% MSE: 348.127
output_net.csv	22.29% MSE: 402.762	25.01% MSE: 371.956	21.29% MSE: 404.502	21.31% MSE: 406.123
output_previous.csv	60.6% MSE: 82.7087	69.88% MSE: 100.512	66.69% MSE: 55.8242	46.01% MSE: 98.8558
output_production.csv	16.3% MSE: 417.809	26.72% MSE: 303.476	28.25% MSE: 497.322	27.5% MSE: 282.466
output_production_types.csv	23.46% MSE: 308.318	25.01% MSE: 287.224	18.71% MSE: 380.047	26.72% MSE: 243.805
output_various.csv	32.44% MSE: 193.478	27.03% MSE: 292.245	25.69% MSE: 211.011	31.66% MSE: 185.884

Veiktajos eksperimentos vienīgie rezultāti ar augstāku precizitāti un mazu vidējo kvadrātisko kļūdu bija tie, kuros tika mēģināts izmantot priekšapstrādāto failu „output_previous.csv”, ar iepriekšējām mirkļa cenas vērtībām. Izmantojot citus faktorus, netika iegūti viennozīmīgi precīzi rezultāti. Tas liecina par to, ka pagātnes cena iespaido pašreizējo cenu vairāk nekā kādi citi faktori tiešā veidā. Šādi rezultāti saskan ar Viktorijas Bobinaites pētījumu par šiem pašiem datiem [10]. Šajā pētījumā galvenie secinājumi saistībā ar elektrības mirkļa cenu nosakošajiem faktoriem ir tādi, ka mirkļa cenu vislabāk ir iespējams paredzēt no iepriekšējām elektrības mirkļa cenas vērtībām, kuras tiek apkopotas par noteiktu iepriekšējo periodu – apmēram 480 iepriekšējās vērtības.

2.4. Iegūto konfigurāciju izmantošana mirkļa cenas paredzēšanai

Izmantojot labāko iegūto neironu tīkla konfigurāciju, veicu datu apstrādi, lai iegūtu paredzējumus datiem un salīdzinātu tos ar failā pieejamajām vērtībām. Izmantojot neironu tīkla konfigurāciju ar 8 slēptajiem neironiem un 49 iepriekšējām elektrības mirkļa cenas vertības, aprēķināju sagaidīto pašreizējo elektrības mirkļa cenu. Iegūtos datus apkopoju, gan sadalot vērtības grupās pa mēnešiem, gan apskatot vērtības kopumā, un aplūkojot katras dienas stundas vidējo sagaidīto vērtību un vidējo aprēķināto vērtību.



2.7. att. **Iegūtais apkopojums vidējai sagaidītajai vērtībai un vidējai aprēķinātajai vērtībai**

Kā redzams kopējā vidējo vērtību apkopojumā, tad vērtības kopumā neironu tīkls paredz ar augstu precizitāti, bet tas, ka aprēķinātā precizitāte ir tikai 69.88%, liecina par to, ka tikai balstoties uz iepriekšējām elektrības mirkļa cenas vērtībām būs izņēmuma gadījumi, kuros šo cenu ietekmēs kāds faktors, kurš neveidojas no mirkļa cenas pagātnes datiem, bet arī

nav pieejams sākotnējos datos par elektrības mirkļa cenu un iespējamajami to ietekmējošajiem faktoriem.

Rezultātu apkopojumā pa mēnešiem (sk. 1. pielikumu), novērojama līdzīga tendence – vidēji neironu tīkls spēj pietiekami precīzi paredzēt informāciju, ar retiem izņēmumiem, lai gan konkrētos gadījumos iespējamas situācijas, kurās elektrības mirkļā cenu ietekmē nepieejami faktori.

3. SECINĀJUMI

Izpētot mākslīgos neironu tīklus tika gūts priekšstats par to darbības principiem, pielietojamību un konfigurējamību. Tika apkopota informācija par neironu tīklu apmācīšanas algoritmiem, aktivizācijas funkcijām un matemātisko pamatojumu neironu tīkla darbībai. Tika apkopota informācija par neironu tīklu algoritmu dažādām struktūrām un tipiem.

Izpētītā teorija tika pielietota, lai izveidotu datorprogrammu, kas implementē mākslīgo neironu tīklu un spēj apstrādāt datus noteiktā formātā. Izveidotā programma spēj veikt datu priekšapstrādi, filtrējot datus dažādos veidos, parametrizēti veidot un apmācīt neironu tīklu, veikt iegūtās apmācīšanas eksportēšanu un importēšanu, un veikt datu paredzēšanu.

Pieejamie dati par elektrības datiem tika apstrādāti ar izveidoto programmu, lai izveidotu dažādus priekšapstrādātos failus, kuros pieejamie dati tika sadalīti pēc faktoru tipiem. Izmantojot šos failus tika veikta neironu tīkla apmācīšana un tika iegūti paredzēto datu precizitātes novērtējumi. Iegūtie rezultāti tika salīdzināti ar Viktorijas Bobinaites pētījuma rezultātiem, kuros tika norādīti nozīmīgi faktori, ar kuriem izdevies iegūt labas neironu tīklu konfigurācijas, šādas derīgas konfigurācijas izdevās iegūt arī ar šajā bakalaura darba izstrādāto programmu.

Tika salīdzinātas dažādas neironu tīklu konfigurāciju un priekšapstrādāto failu kombinācijas, no kurām tika izvēlēta konfigurācija, kura apstrādāja pieejamos datus ar vislielāko precizitāti. Izmantojot izvēlēto neironu tīkla konfigurāciju tika ģenerēta paredzētās vērtības, kuras tika apstrādātas apkopojot informāciju par mēnešiem un konkrētām stundām, tādā veidā iegūstot salīdzinājumu ar ģenerētajām vērtībām un sagaidītajām vērtībām. Iegūtais salīdzinājums parāda, ka lai gan trenētā neironu tīkla precizitāte ir apmēram 70%, vidēji iegūtā neironu tīkla konfigurācija spēj pietiekami precīzi paredzēt elektrības mirkļa cenu.

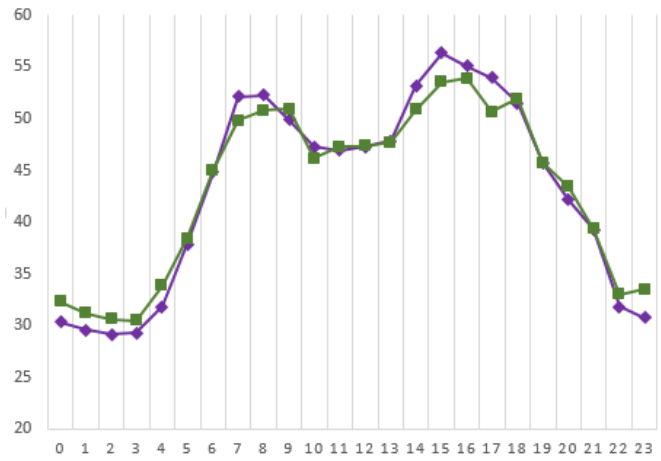
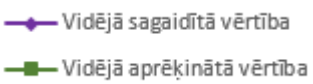
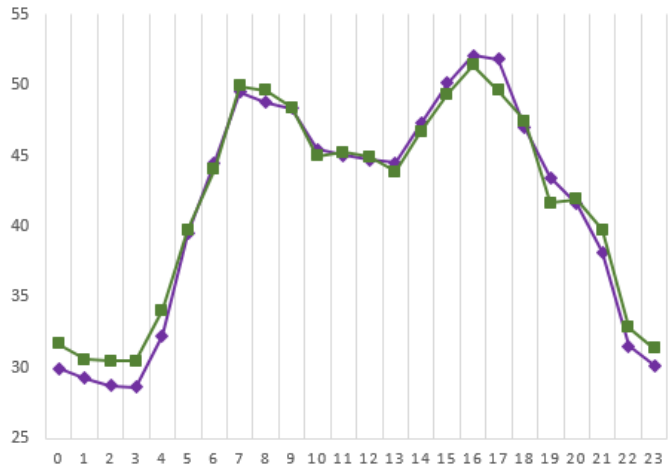
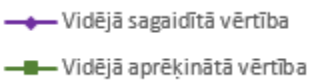
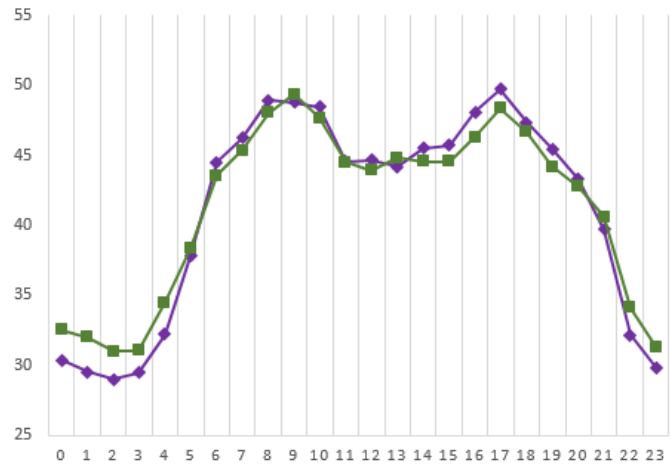
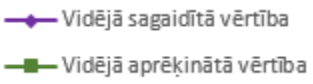
Izveidoto programmu ir iespējams papildināt un veidot padziļinātus pētījumus neironu tīklu izmantošanā reālu problēmu risināšanai, piemēram, pievienojot iespēju veikt apmācīšanu izmantojot konfigurējamu neironu tīkla slēpto slāņu skaitu, citas mākslīgo neironu aktivizācijas funkcijas, ģenētiskos algoritmus neironu tīkla optimizācijai.

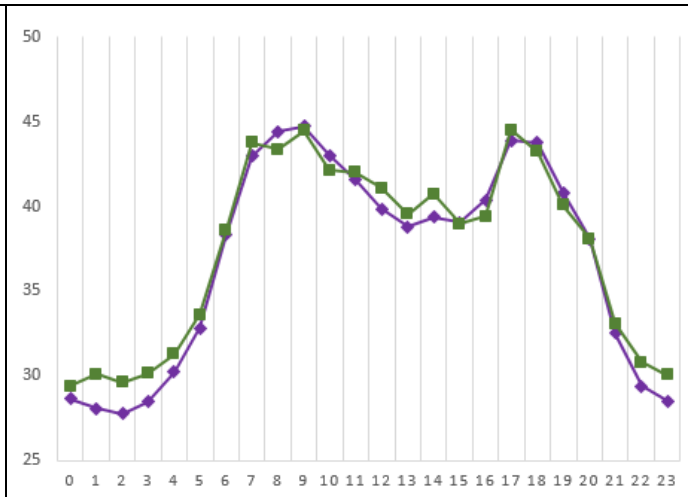
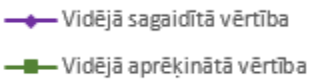
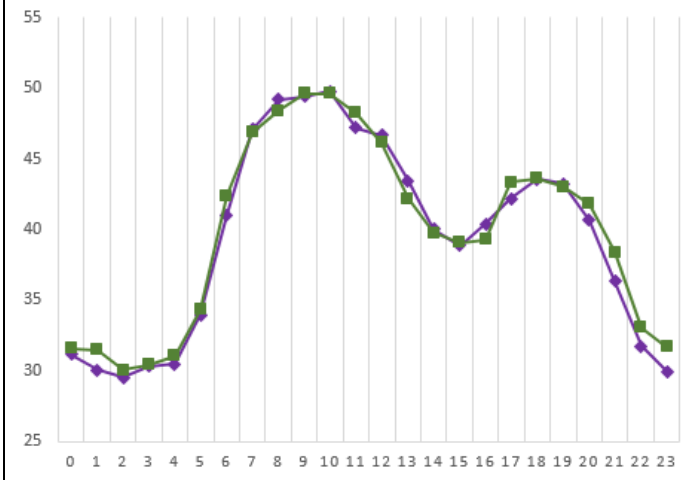
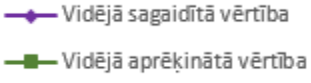
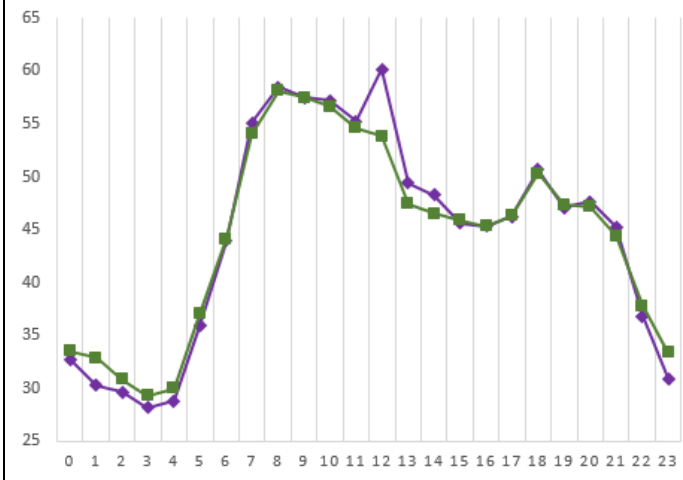
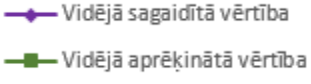
IZMANTOTĀ LITERATŪRA UN AVOTI

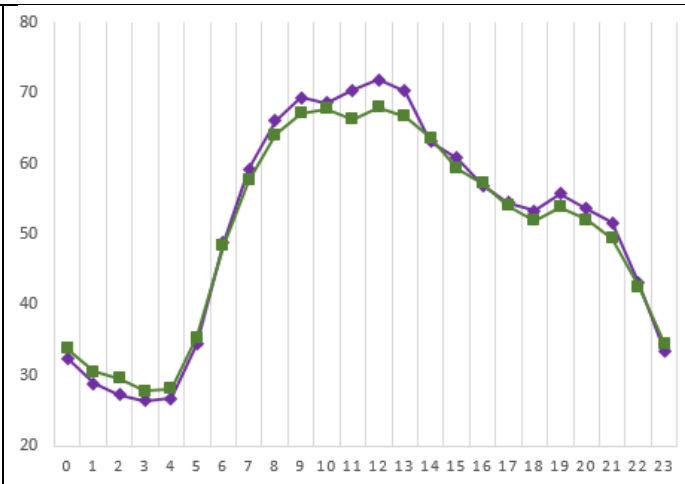
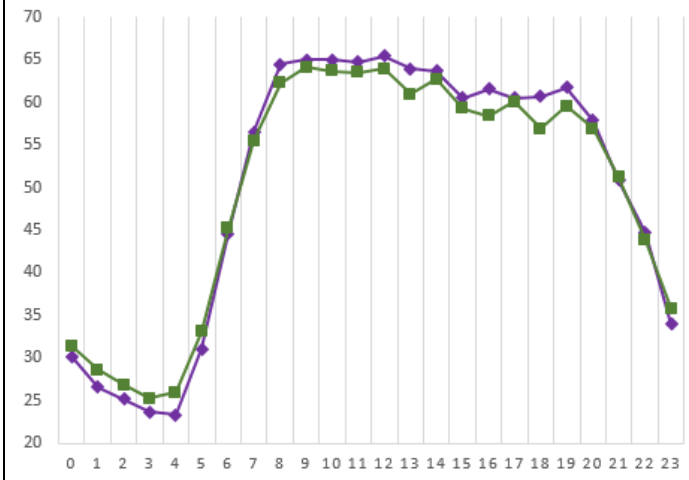
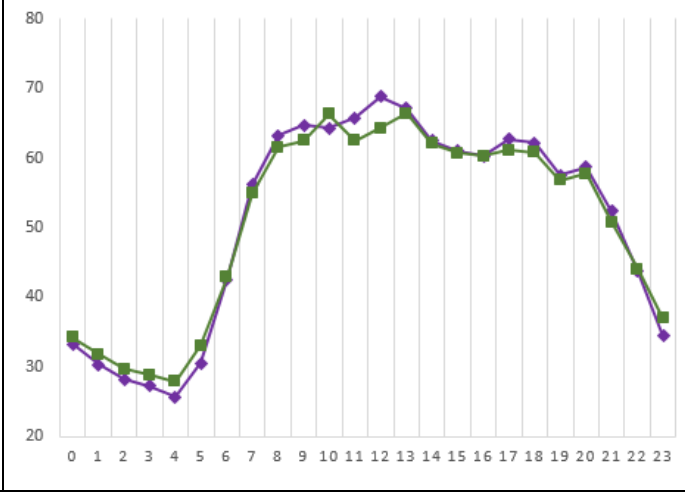
1. Artificial Intelligence A Modern Approach, Stuart J. Russell and Peter Norvig, Prentice Hall, Englewood Cliffs, New Jersey 07632, page 5
2. Machine Learning Approaches to Breast Cancer Diagnosis and Treatment Response Prediction, Katie Planey, Stanford Biomedical Informatics [tiešsaiste] – [atsauce 27.05.2016]. Piejams: http://cs229.stanford.edu/proj2011/Planey-Machine%20Learning%20Approaches%20to%20Breast%20Cancer%20Diagnosis%20and%20Treatment%20Response%20Prediction.pdf
3. Types of Machine Learning Algorithms, Taiwo Oladipupo Ayodele University of Portsmouth, United Kingdom, [tiešsaiste] – [atsauce 27.05.2016]. Pieejams: http://cdn.intechweb.org/pdfs/10694.pdf
4. Unsupervised Learning, Zoubin Ghahramani, Gatsby Computational Neuroscience Unit, 20041996, [tiešsaiste] – [atsauce 27.05.2016]. Pieejams: http://mlg.eng.cam.ac.uk/zoubin/papers/ul.pdf
5. Neural networks: a comprehensive foundation, Simon Haykin, Prentice Hall, 1999 ISBN 0-13-273350-1
6. Neural networks: A Systematic Introduction, Raul Rojas, Springer, 1996, [tiešsaiste] – [atsauce 27.05.2016]. Pieejams: http://page.mi.fu-berlin.de/rojas/neural/neuron.pdf
7. Neural Networks, Christos Stergiou and Dimitrios Siganos, [tiešsaiste] – [atsauce 27.05.2016]. Pieejams: https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html
8. A Brief Introduction to Neural Networks, David Kriesel, [tiešsaiste] – [atsauce 27.05.2016]. Pieejams: http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-1col-dkrieselcom.pdf
9. Neural Networks and Deep Learning, Michael Nielsen, 2016, [tiešsaiste] – [atsauce 27.05.2016]. Pieejams: http://neuralnetworksanddeeplearning.com/chap2.html
10. Bobinaite, V. & Zuters, J. Modelling Electricity Price Expectations in a Day-Ahead Electricity Market. Latvian Case. Economics and Business, ISSN: 1407-7337.

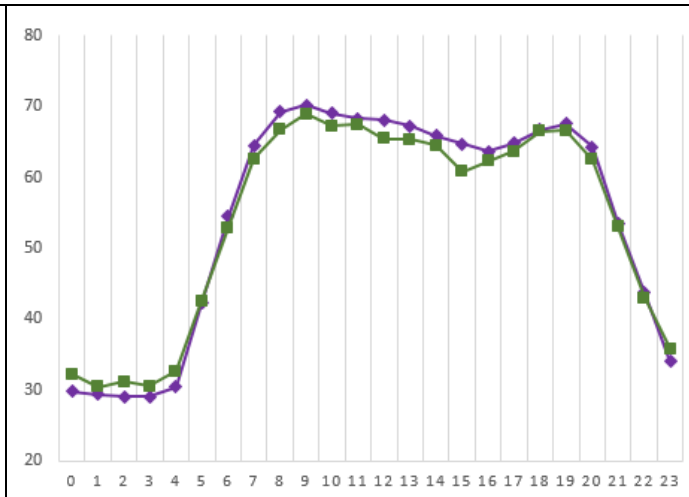
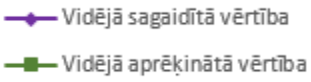
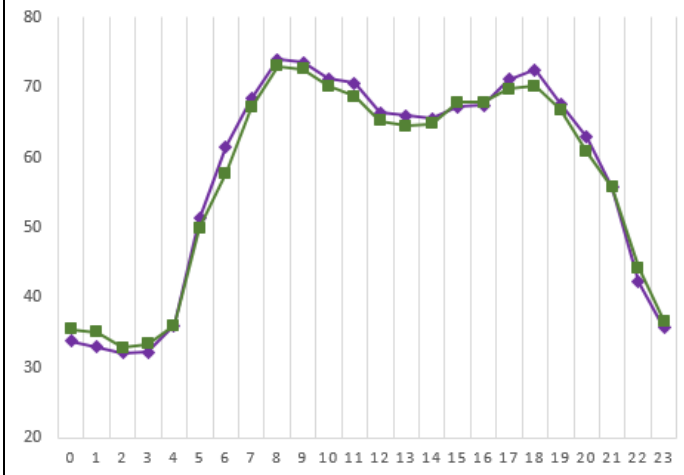
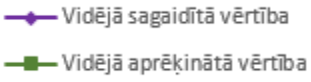
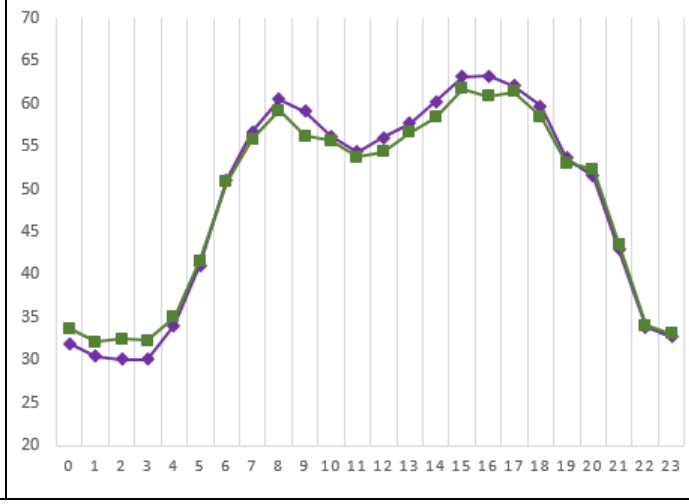
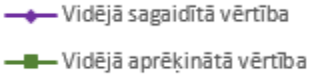
PIELIKUMI

1. Pielikums. Vidējo sagaidīto un vidējo aprēķināto vērtību salīdzinājums pa mēnešiem

Mēnesis	Grafiks	Atšifrējums
Janvāris		
Februāris		
Marts		

Aprīlis		
Maijs		
Jūnijs		

Jūlijs		Vidējā sagaidītā vērtība Vidējā aprēķinātā vērtība
Augusts		Vidējā sagaidītā vērtība Vidējā aprēķinātā vērtība
Septembris		Vidējā sagaidītā vērtība Vidējā aprēķinātā vērtība

Oktobris		
Novembris		
Decembris		

2. Pielikums. Programmas koda galvenie fragmenti

```
#ifndef NEURALNETWORK_H
#define NEURALNETWORK_H

#include "QVector"
#include "Qstring"
#include "QFile"
#include "QTextStream"
#include "QDebug"
#include "qmath.h"
#include "dataentry.h"
#include "logger.h"

const bool DEBUG_EACH_ENTRY = false;
const double SIGMOID_BETA = 0.00001;
const double ACCURACY_THRESHOLD = 0.1;

class NeuralNetwork
{
public:
    Logger * logger;

    //izmēra parametri
    int inputCount;
    int hiddenCount;
    int outputCount;

    int neuronWidth;
    int neuronHeight;

    //array of all neuron values
    double** neuronArray;

    //array of all neuron weights
    double** neuronWeights_IH;
    double** neuronWeights_HO;

    //privātās metodes
    void initialize();
    double activationLinear(double x);
    double derivativeLinear(double x);
    double activationSigmoid(double x);
    double derivativeSigmoidFromSigmoid(double x);

    void feedForward(double* data);

public:
    NeuralNetwork(int inputCount, int hiddenCount, int outputCount, Logger
* logger);
    ~NeuralNetwork();

    void calculateSize();
    void deleteArrays();
    void createArrays();

    bool loadConfiguration(QString filename);
    bool saveConfiguration(QString filename);
    int* feedForwardData(double* data);
    double getAccuracy(QVector<DataEntry*>& set);
```

```
    double getMSE(QVector<DataEntry*>& set);
    int randInt(int low, int high);
};

#endif // NEURALNETWORK_H

#include "neuralnetwork.h"

double NeuralNetwork::activationLinear(double x)
{
    return x;
}

double NeuralNetwork::derivativeLinear(double x)
{
    return 1;
}

double NeuralNetwork::activationSigmoid(double x)
{
    return 1.0/(1.0+exp(-x*SIGMOID_BETA));
}

double NeuralNetwork::derivativeSigmoidFromSigmoid(double x)
{
    //derivative based on already calculated sigmoid value
    return SIGMOID_BETA*x*(1.0-x);
}

void NeuralNetwork::feedForward(double *data)
{
    bool debugThis = DEBUG_EACH_ENTRY;

    //if (debugThis) qDebug() << "feedForward(" << &data << ")";

    int layer = 0;
    int sizes[3];
    sizes[0] = inputCount;
    sizes[1] = hiddenCount;
    sizes[2] = outputCount;

    //set input values
    for (int i=0; i<inputCount; i++){
        neuronArray[layer][i] = data[i];
    }

    //hidden layer
    layer++;
    for (int i=0; i<hiddenCount; i++){
        neuronArray[layer][i]=0.0;
        for (int j=0; j<=sizes[layer-1]; j++){ //<= because also including
the bias neuron
            neuronArray[layer][i] += neuronWeights_IH[j][i] *
neuronArray[layer-1][j];
        }
        //if (debugThis) qDebug() << "activationSigmoid(" <<
neuronArray[layer][i] << ")=" << activationSigmoid(neuronArray[layer][i]);
        neuronArray[layer][i] = activationSigmoid(neuronArray[layer][i]);
    }

    //output
    layer++;
    for (int i=0; i<outputCount; i++){
        neuronArray[layer][i]=0.0;
```

```
        for (int j=0; j<=sizes[layer-1]; j++){ //<= because also including
the bias neuron
            neuronArray[layer][i] += neuronWeights_HO[j][i] *
neuronArray[layer-1][j];
        }
        //if (debugThis) qDebug() << "activationLinear(" <<
neuronArray[layer][i] << ")=" << activationLinear(neuronArray[layer][i]);
        neuronArray[layer][i] = activationLinear(neuronArray[layer][i]);
    }
}

NeuralNetwork::NeuralNetwork(int inputCount, int hiddenCount, int
outputCount, Logger *logger)
{
    //parameter values
    this->inputCount = inputCount;
    this->hiddenCount = hiddenCount;
    this->outputCount = outputCount;
    this->logger = logger;

    //calculates initial values
    calculateSize();

    //initializes weights and values
    createArrays();
}

NeuralNetwork::~NeuralNetwork()
{
    deleteArrays();
}

void NeuralNetwork::calculateSize()
{
    neuronWidth = 3;
    neuronHeight = qMax(qMax(inputCount, hiddenCount), outputCount) + 1;
//+1 because one extra bias neuron
}

void NeuralNetwork::deleteArrays()
{
    qDebug() << "deleting neural network";

    qDebug() << "deleting neuron array";
    for (int i=0; i<neuronWidth; i++){
        qDebug() << "deleting:" << i << "/" << neuronWidth-1;
        delete [] neuronArray[i];
    }
    delete [] neuronArray;

    qDebug() << "deleting neuron weights IH";
    for (int i=0; i<=inputCount; i++){
        qDebug() << "deleting:" << i << "/" << inputCount;
        delete [] neuronWeights_IH[i];
    }
    //delete [] neuronWeights_IH;

    qDebug() << "deleting neuron weights HO";
    for (int i=0; i<=hiddenCount; i++){
        qDebug() << "deleting:" << i << "/" << hiddenCount;
        delete [] neuronWeights_HO[i];
    }
    //delete [] neuronWeights_HO;
}
```

```
void NeuralNetwork::createArrays()
{
    int width;
    int height;

    width = neuronWidth;
    height = neuronHeight;
    neuronArray = new double*[width];
    for (int i=0; i<width; i++){
        neuronArray[i] = new double[height];
        for (int j=0; j<height; j++){
            neuronArray[i][j] = 0.5;
        }
    }

    //set bias neurons
    neuronArray[0][inputCount] = 1.0;
    neuronArray[1][hiddenCount] = 1.0;

    width = inputCount;
    height = hiddenCount;
    neuronWeights_IH = new double*[width];
    for (int i=0; i<=width; i++){ //<= because bias neuron
        neuronWeights_IH[i] = new double[height];
        for (int j=0; j<height; j++){
            neuronWeights_IH[i][j] = (0.5 -
((double)randInt(0,1000)/1000.0)); //small initial random value
        }
    }

    width = hiddenCount;
    height = outputCount;
    neuronWeights_HO = new double*[width];
    for (int i=0; i<=width; i++){   //<= because bias neuron
        neuronWeights_HO[i] = new double[height];
        for (int j=0; j<height; j++){
            neuronWeights_HO[i][j] = (0.5 -
((double)randInt(0,1000)/1000.0)); //small initial random value
        }
    }
}

bool NeuralNetwork::loadConfiguration(QString filename)
{
    QFile input(filename);

    if (input.open(QFile::ReadOnly)){
        QString line;
        int lineNumber = 0;
        int arrayLineNumber = 0;
        int width;
        int height;

        line = input.readLine().replace("\n","").replace("\r","");

        bool inputEnd = input.atEnd(); //because !input.atEnd() does not
react correctly to the last line
        bool lastLineProcessed = false;
        while (!inputEnd){
            qDebug()<<"processing line "<<lineNumber<<"; "<<line;

            switch(lineNumber){
            case 0: {//sizes
```

```
                QStringList parameters = line.split(",");
                deleteArrays();
                inputCount = parameters[0].toInt();
                hiddenCount = parameters[1].toInt();
                outputCount = parameters[2].toInt();
                calculateSize();
                createArrays();
            }   break;
            default:
                if (lineNumber<=inputCount+1){
                    //input-hidden
                    qDebug() << "processing IH:" << arrayLineNumber << ";"
<< line;
                    width = inputCount+1;
                    height = hiddenCount;
                    QStringList weightValues = line.split(",");
                    for (int i=0; i<weightValues.size(); i++){
                        double weightValueDouble =
weightValues[i].toDouble();
                        neuronWeights_IH[arrayLineNumber % width][i] =
weightValueDouble;
                    }
                    if (lineNumber==inputCount+1){
                        arrayLineNumber = -1;    //reset index
                    }
                }else
                if (lineNumber<=inputCount+1+hiddenCount+1){
                    //hidden-output
                    qDebug() << "processing HO:" << arrayLineNumber << ";"
<< line;
                    width = hiddenCount+1;
                    height = outputCount;
                    QStringList weightValues = line.split(",");
                    for (int i=0; i<weightValues.size(); i++){
                        double weightValueDouble =
weightValues[i].toDouble();
                        neuronWeights_HO[arrayLineNumber % width][i] =
weightValueDouble;
                    }
                }
                arrayLineNumber++;
            }

            if (input.atEnd() ){
                if (lastLineProcessed){
                   inputEnd = true;
                }else{
                    lineNumber++;
                    line =
input.readLine().replace("\n","").replace("\r","");
                    lastLineProcessed = true;
                }
            }else{
                lineNumber++;
                line = input.readLine().replace("\n","").replace("\r","");
                lastLineProcessed = false;
                inputEnd = false;
            }

        }

        input.close();
    }else{
        qDebug()<<"Could not read NN input file.";
```

```
    }
    return true;
}

bool NeuralNetwork::saveConfiguration(QString filename)
{
    int width;
    int height;

    QFile output(filename);

    if (output.open(QFile::WriteOnly | QFile::Text)){
        QTextStream out(&output);

        //parameters
        out<<QString::number(inputCount)<<",";
        out<<QString::number(hiddenCount)<<",";
        out<<QString::number(outputCount)<<"\n";

        //weights
        //input-hidden
        width = inputCount+1;
        height = hiddenCount;
        for (int i=0; i<width; i++){
            for (int j=0; j<height; j++){
                if (j!=0)
                    out<<",";
                out<<neuronWeights_IH[i][j];
            }
            out<<"\n";
        }

        //hidden-output
        width = hiddenCount+1;
        height = outputCount;
        for (int i=0; i<width; i++){
            for (int j=0; j<height; j++){
                if (j!=0)
                    out<<",";
                out<<neuronWeights_HO[i][j];
            }
            out<<"\n";
        }

        output.flush();
        output.close();
    }
    return true;
}

int* NeuralNetwork::feedForwardData(double *data)
{
    bool debugThis = DEBUG_EACH_ENTRY;

    if (debugThis) qDebug() << "feedForwardData()";

    feedForward(data);

    int* results = new int[outputCount];
    for(int i=0; i<outputCount; i++)
        results[i] = neuronArray[2][i];
        //results[i] = clampOutput(neuronArray[2][i]);

    if (debugThis) qDebug() << "EO feedForwardData()";
```

```
    return results;
}

double NeuralNetwork::getAccuracy(QVector<DataEntry*> &set)
{

    int incorrectResults = 0;
    for (int i=0; i<set.size(); i++){
        feedForward(set[i]->data);
        bool someIncorrect = false;

        for (int j=0; j<outputCount; j++){
            //if (clampOutput(neuronArray[2][j]) != set[i]->expected[j]){
            double threshold = set[i]->expected[j] * ACCURACY_THRESHOLD;
            if (qAbs(neuronArray[2][j] - set[i]->expected[j]) > threshold){
                //qDebug() << "incorrect: " <<
clampOutput(neuronArray[2][j]) << "!=" << set[i]->expected[j];
                someIncorrect = true;
            }
        }
        if (someIncorrect){
            incorrectResults++;
        }
    }

    double result = 1.0-((double)incorrectResults / (double)set.size());
    return result;
}

double NeuralNetwork::getMSE(QVector<DataEntry *> &set)
{
    double meanSquareError = 0.0;

    for (int i=0; i<set.size(); i++){
        feedForward(set[i]->data);

        for (int j=0; j<outputCount; j++){
            meanSquareError += pow((neuronArray[2][j]-set[i]->expected[j]),
2);
        }
    }

    return meanSquareError/(outputCount * set.size());
}

int NeuralNetwork::randInt(int low, int high)
{
    return qrand() % ((high + 1) - low) + low;
}

#ifndef DATATRAINER_H
#define DATATRAINER_H


#include "dataentry.h"
#include "qdebug.h"
#include "datasets.h"
#include "neuralnetwork.h"

//operācijas ar NN kopumā
class DataTrainer
{
private:
```

```
    bool neuralNetworkAssigned;

    //training parameters
    double learningRate;
    double momentum;
    long maxEpochs;
    double requiredAccuracy;
    double requiredMeanSquareError;

    double* outputErrorGradients;
    double* hiddenErrorGradients;
    double** weightChange_HO;
    double** weightChange_IH;

    long currentEpoch;

    double averageAccuracy;
    double averageMeanSquareError;

public:
    Logger * logger;
    NeuralNetwork* neuralNetwork;

    //methods
    DataTrainer(Logger * logger);
    ~DataTrainer();

    void setParameters(double requiredAccuracy, double
requiredMeanSquareError, double maxEpochs, double learningRate, double
momentum);
    void initializeNeuralNetwork(int inputCount, int hiddenCount, int
outputCount);
    void trainNetwork(DataSets* sets);
    void predictData(QString targetLocation, DataSets *dataSets);
    bool isNeuralNetworkAssigned();
private:
    //methods
    void backpropagate(double* desired);
    void updateWeights();
    void createTempArrays();
    void initializeTempArrays();
    void deleteTempArrays();
    bool endConditionsReached();
    QString entryToPrediction(DataEntry* dataEntry);
    void updateEndConditions(DataSets* sets, bool log);
};

#endif // DATATRAINER_H

#include "datatrainer.h"

DataTrainer::DataTrainer(Logger *logger=NULL)
{
    currentEpoch = 0;
    neuralNetworkAssigned = false;

    requiredAccuracy = 0;
    requiredMeanSquareError = 0;
    maxEpochs = 0;
    learningRate = 0;
    momentum = 0;

    this->logger = logger;
}
```

```
DataTrainer::~DataTrainer()
{
    if (neuralNetworkAssigned) {
        delete neuralNetwork;
    }
}

void DataTrainer::setParameters(double requiredAccuracy, double
requiredMeanSquareError, double maxEpochs, double learningRate, double
momentum)
{
    this->requiredAccuracy = requiredAccuracy;
    this->requiredMeanSquareError = requiredMeanSquareError;
    this->maxEpochs = maxEpochs;
    this->learningRate = learningRate;
    this->momentum = momentum;
}

void DataTrainer::initializeNeuralNetwork(int inputCount, int hiddenCount,
int outputCount)
{
    if (neuralNetworkAssigned){
        delete neuralNetwork;
        neuralNetworkAssigned = false;
    }

    neuralNetwork = new NeuralNetwork(inputCount, hiddenCount, outputCount,
logger);
    neuralNetworkAssigned = true;
}

void DataTrainer::trainNetwork(DataSets *sets)
{
    bool debugThis = DEBUG_EACH_ENTRY;

    logger->clear();
    logger->log("Training neural network.");

    updateEndConditions(sets, false);
    currentEpoch = 0;


    createTempArrays();

    while(!endConditionsReached()){
        //do the training, one epoch

        for (int i=0; i<sets->trainingSet.size(); i++){
            if (debugThis) qDebug() << "    Processing training
entry"<<i<<"/"<<sets->trainingSet.size();
            //for each training set element
            neuralNetwork->feedForwardData(sets->trainingSet.at(i)->data);
            initializeTempArrays();
            backpropagate(sets->trainingSet.at(i)->expected);
            updateWeights();
        }

        updateEndConditions(sets, true);
    }

    averageAccuracy = neuralNetwork->getAccuracy(sets->validationSet);//
/sets->generalizationSet.size();
```

```
    averageMeanSquareError = neuralNetwork->getMSE(sets->validationSet);//
/sets->generalizationSet.size();
    logger->log("Validation set - Accuracy:
"+QString::number((double)((int)(averageAccuracy*10000.0)/100.0))+"% MSE:
"+QString::number(averageMeanSquareError));

    deleteTempArrays();
    logger->log("Finished training.");
}

void DataTrainer::backpropagate(double *desired)
{
    bool debugThis = DEBUG_EACH_ENTRY;

    if (debugThis) qDebug() << "    backpropagating errors...";

    if (debugThis) {
        for (int i=0; i<neuralNetwork->outputCount; i++){
            qDebug() << "   neural network output["<< i <<"] =" <<
neuralNetwork->neuronArray[2][i] << "; expected value " << desired[i];
        }
    }

    //calculate changes in weights between hidden and output layers
    if (debugThis) qDebug() << "    calculate changes in weights between
hidden and output layers";
    for (int i=0; i<neuralNetwork->outputCount; i++){
        double currentOutput = neuralNetwork->neuronArray[2][i];
        //outputErrorGradients[i] = currentOutput * (1.0 - currentOutput) *
(desired[i] - currentOutput);
        outputErrorGradients[i] = neuralNetwork-
>derivativeLinear(currentOutput) * (desired[i] - currentOutput);
        if (debugThis) qDebug() << "    outputErrorGradients[" << i << "]
=" << outputErrorGradients[i];

        for (int j=0; j<neuralNetwork->hiddenCount+1; j++){
            if (debugThis) qDebug() << "    learning rate =" <<
learningRate << "; neuralNetwork[1]["<<j<<"] =" << neuralNetwork-
>neuronArray[1][j] << "; outputErrorGradients["<<i<<"] =" <<
outputErrorGradients[i];
            weightChange_HO[j][i] += learningRate * neuralNetwork-
>neuronArray[1][j] * outputErrorGradients[i];
        }
    }

    if (debugThis) {
        for (int i=0; i<neuralNetwork->outputCount; i++){
            for (int j=0; j<neuralNetwork->hiddenCount+1; j++){
                qDebug() << "   weightChange_HO[" << j << "][" << i << "] =
" << weightChange_HO[j][i];
            }
        }
    }

    if (debugThis) qDebug() << "    calculate changes in weights between
input and hidden layers";
    //calculate changes in weights between input and hidden layers
    for (int i=0; i<neuralNetwork->hiddenCount; i++){
        //hidden neuron gradient
        double weightedSum = 0.0;
        for (int j=0; j<neuralNetwork->outputCount; j++){
            weightedSum += neuralNetwork->neuronWeights_HO[i][j] *
outputErrorGradients[j];
        }
```

```
        double currentHiddenValue = neuralNetwork->neuronArray[1][i];
        if (debugThis) qDebug() << "    currentHiddenValue =" <<
currentHiddenValue;
        if (debugThis) qDebug() << "    weightedSum =" << weightedSum;

        //hiddenErrorGradients[i] = currentHiddenValue * (1.0 -
currentHiddenValue) * weightedSum;
        hiddenErrorGradients[i] = neuralNetwork-
>derivativeSigmoidFromSigmoid(currentHiddenValue) * weightedSum;
        if (debugThis) qDebug() << "    hiddenErrorGradients["<<i<<"] =" <<
hiddenErrorGradients[i];

        for (int j=0; j<neuralNetwork->inputCount+1; j++){
            weightChange_IH[j][i] += learningRate * neuralNetwork-
>neuronArray[0][j] * hiddenErrorGradients[i];
        }
    }

    if (debugThis) {
        for (int i=0; i<neuralNetwork->hiddenCount; i++){
            for (int j=0; j<neuralNetwork->inputCount+1; j++){
                qDebug() << "   weightChange_IH[" << j << "][" << i << "] =
" << weightChange_IH[j][i];
            }
        }
    }



}

void DataTrainer::updateWeights()
{
    bool debugThis = DEBUG_EACH_ENTRY;

    if (debugThis) qDebug() << "change weights";
    //change weights
    float average_IH = 0.0;
    for (int i=0; i<neuralNetwork->hiddenCount; i++){
        for (int j=0; j<neuralNetwork->inputCount+1; j++){
            neuralNetwork->neuronWeights_IH[j][i] += weightChange_IH[j][i];
            average_IH += weightChange_IH[j][i];
        }
    }
    average_IH = average_IH/neuralNetwork->inputCount+1;
    //qDebug() << "average_IH =" << average_IH;

    for (int i=0; i<neuralNetwork->outputCount; i++){
        for (int j=0; j<neuralNetwork->hiddenCount+1; j++){
            neuralNetwork->neuronWeights_HO[j][i] += weightChange_HO[j][i];
        }
    }
}

void DataTrainer::createTempArrays()
{
    bool debugThis = DEBUG_EACH_ENTRY;

    //initialize arrays
    int width;
    int height;

    if (debugThis) qDebug() << "    initializing error output gradients";
```

```
    outputErrorGradients = new double[neuralNetwork->outputCount];
    for (int i=0; i<neuralNetwork->outputCount; i++){
        outputErrorGradients[i] = 0.0;
    }

    if (debugThis) qDebug() << "    initializing error hidden gradients";
    hiddenErrorGradients = new double[neuralNetwork->hiddenCount];
    for (int i=0; i<neuralNetwork->hiddenCount; i++){
        hiddenErrorGradients[i] = 0.0;
    }

    if (debugThis) qDebug() << "    initializing weight changes hidden-
>output";
    width = neuralNetwork->hiddenCount+1;
    height = neuralNetwork->outputCount;
    if (debugThis) qDebug() << "    width:" << width << "height:" <<
height;
    weightChange_HO = new double*[width];
    for (int i=0; i<width; i++){
        weightChange_HO[i] = new double[height];
    }
    for (int i=0; i<width; i++){
        for(int j=0; j<height; j++){
            weightChange_HO[i][j] = 0.0;
        }
    }

    if (debugThis) qDebug() << "    initializing weight changes input-
>hidden";
    width = neuralNetwork->inputCount+1;
    height = neuralNetwork->hiddenCount;
    if (debugThis) qDebug() << "    width:" << width << "height:" <<
height;
    weightChange_IH = new double*[width];
    for (int i=0; i<width; i++){
        weightChange_IH[i] = new double[height];
    }
    for (int i=0; i<width; i++){
        for(int j=0; j<height; j++){
            weightChange_IH[i][j] = 0.0;
        }
    }
}

void DataTrainer::initializeTempArrays()
{
    bool debugThis = DEBUG_EACH_ENTRY;

    //initialize arrays
    int width;
    int height;

    if (debugThis) qDebug() << "    initializing error output gradients";
    for (int i=0; i<neuralNetwork->outputCount; i++){
        outputErrorGradients[i] = 0.0;
    }

    if (debugThis) qDebug() << "    initializing error hidden gradients";
    for (int i=0; i<neuralNetwork->hiddenCount; i++){
        hiddenErrorGradients[i] = 0.0;
    }

    if (debugThis) qDebug() << "    initializing weight changes hidden-
>output";
```

```
    width = neuralNetwork->hiddenCount+1;
    height = neuralNetwork->outputCount;
    if (debugThis) qDebug() << "    width:" << width << "height:" <<
height;
    for (int i=0; i<width; i++){
        for(int j=0; j<height; j++){
            weightChange_HO[i][j] = weightChange_HO[i][j] * momentum;
        }
    }


    if (debugThis) qDebug() << "    initializing weight changes input-
>hidden";
    width = neuralNetwork->inputCount+1;
    height = neuralNetwork->hiddenCount;
    if (debugThis) qDebug() << "    width:" << width << "height:" <<
height;
    for (int i=0; i<width; i++){
        for(int j=0; j<height; j++){
            weightChange_IH[i][j] = weightChange_IH[i][j] * momentum;
        }
    }
}

void DataTrainer::deleteTempArrays()
{
    bool debugThis = DEBUG_EACH_ENTRY;

    if (debugThis) qDebug() << "    delete arrays";
    //delete arrays
    if (debugThis) qDebug() << "    d1";
    delete [] outputErrorGradients;
    if (debugThis) qDebug() << "    d2";
    delete [] hiddenErrorGradients;

    //for (int i=0; i<neuralNetwork->hiddenCount; i++){
        if (debugThis) qDebug() << "    d3";
        delete [] *weightChange_HO;
        //delete [] weightChange_HO[i];
    //}
    if (debugThis) qDebug() << "    d4";
    delete [] weightChange_HO;

    //for (int i=0; i<neuralNetwork->inputCount; i++){
        //if (debugThis) qDebug() << "  d5" << i;
    delete [] *weightChange_HO;
        //delete [] weightChange_IH[i];
    //}
    if (debugThis) qDebug() << "    d6";
    delete [] weightChange_IH;
}

bool DataTrainer::endConditionsReached()
{
    //all end conditions that can happen
    if (averageAccuracy>=requiredAccuracy) return true;
    if (averageMeanSquareError<=requiredMeanSquareError) return true;
    if (currentEpoch>=maxEpochs) return true;
    return false;
}

void DataTrainer::predictData(QString targetLocation, DataSets *dataSets)
{
    QFile outputFile(targetLocation);
```

```
    if (outputFile.open(QFile::WriteOnly|
                        QFile::Text)){
        logger->log("Output file opened: " + targetLocation);
        QTextStream out(&outputFile);

        //output sizes used for calculation
        out << neuralNetwork->inputCount << ", " << neuralNetwork-
>outputCount << "\n";

        //for each entry in every set, generate output line
        for (int i=0; i<dataSets->trainingSet.size(); i++){
            QString line = entryToPrediction(dataSets->trainingSet[i]);
            out << line << "\n";
        }
        for (int i=0; i<dataSets->generalizationSet.size(); i++){
            QString line = entryToPrediction(dataSets-
>generalizationSet[i]);
            out << line << "\n";
        }
        for (int i=0; i<dataSets->validationSet.size(); i++){
            QString line = entryToPrediction(dataSets->validationSet[i]);
            out << line << "\n";
        }

        logger->log("Output complete.");
        outputFile.flush();
        outputFile.close();
    }
}

QString DataTrainer::entryToPrediction(DataEntry *dataEntry)
{
    QString result = "";
    QString sep = ", ";
    QString currentSep = "";

    //format of line: input1, input2, ..., inputN, output1, output2, ...
outputN, expected1, expected2, ..., expectedN

    //process
    neuralNetwork->feedForwardData(dataEntry->data);

    //convert to text
    for (int i=0; i<neuralNetwork->inputCount; i++){
        if (i!=0) currentSep = sep;
        result += currentSep + QString::number(neuralNetwork-
>neuronArray[0][i]);
    }
    int max = neuralNetwork->outputCount;
    for (int i=0; i<max; i++){
        result +=  currentSep + QString::number(neuralNetwork-
>neuronArray[2][i]);
    }
    for (int i=0; i<max; i++){
        result += currentSep + QString::number(dataEntry->expected[i]);
    }

    return result;
}



void DataTrainer::updateEndConditions(DataSets* sets, bool log)
```

```
{
    averageAccuracy = neuralNetwork->getAccuracy(sets-
>generalizationSet);// /sets->generalizationSet.size();
    averageMeanSquareError = neuralNetwork->getMSE(sets-
>generalizationSet);// /sets->generalizationSet.size();

    //epochs
    currentEpoch++;
    if (log)
    logger->log("Generalization set - Epoch:
"+QString::number(currentEpoch)+" Accuracy:
"+QString::number((double)((int)(averageAccuracy*10000.0)/100.0))+"% MSE:
"+QString::number(averageMeanSquareError));
}


bool DataTrainer::isNeuralNetworkAssigned()
{
    return neuralNetworkAssigned;
}

#ifndef DATASTORAGE_H
#define DATASTORAGE_H

#include "commonfunctions.h"
#include "rowdata.h"
#include "columnprocess.h"
#include "QVector"
#include "QComboBox"
#include "QObject"
#include "QFile"
#include "QTextStream"
#include "logger.h"

class DataStorage : public QObject
{
    Q_OBJECT

public slots:
    void setProcessColumnType(int i);
    void setProcessColumnAmount(int i, int newAmount);

public:
    Logger * logger;

    DataStorage(Logger * logger);
    ~DataStorage();

    //rows
    void addRow(QVector<QString> values, QVector<DataTypes> dataTypes);
    void setRows(QVector<RowData> newRows);
    RowData getRow(int i);
    int rowsCount();
    //EO rows

    //columns
    QString getColumn(int i);
    int getColumnCount();
    void setColumns(QVector<QString> columnNames);
    void addColumn(QString name);
    //EO columns

    //for connections
    ColumnProcess *getColumnProcess(int i);
```

```
    //EO for connections

    //export data
    int predictColumns;
    int dataColumns;
    void exportData(QString location);
    int calculateNewRowLength();
    void processColumnRule(int column, QVector<QString> *newColumns,
QVector<RowData> *newRows, QVector<ProcessOptions> *newProcessOptions);
    void processExclude();
    void processInclude(int column, QVector<QString> *newColumns,
QVector<RowData> *newRows, QVector<ProcessOptions> *newProcessOptions);
    void processIncludePrevious(int column, QVector<QString> *newColumns,
QVector<RowData> *newRows, QVector<ProcessOptions> *newProcessOptions);
    void processPredict(int column, QVector<QString> *newColumns,
QVector<RowData> *newRows, QVector<ProcessOptions> *newProcessOptions);
    //EO export data

    //debug
    void printAll();
    void printRow();
    void printRow(int i);
    //EO debug

private:
    //data storage
    QVector<QString> columnNames;
    QVector<ColumnProcess*> columnProcess;
    QVector<RowData> rows;
    //EO data storage
};

#endif // DATASTORAGE_H

#include "datastorage.h"
//data

void DataStorage::setProcessColumnType(int i)
{
    qDebug() << i;
    //qDebug() << cb->currentText();
}

void DataStorage::setProcessColumnAmount(int i, int newAmount)
{
    qDebug() << i;
    qDebug() << newAmount;
}

DataStorage::DataStorage(Logger * logger = NULL)
{
    this->logger = logger;
}

void DataStorage::addRow(QVector<QString> values, QVector<DataTypes>
dataTypes)
{
    RowData newRow(values, dataTypes);
    rows.append(newRow);
}

void DataStorage::setRows(QVector<RowData> newRows)
{
    rows.resize(0);
```

```
    RowData RD;
    foreach(RD, newRows){
        addRow(RD.getValues(), RD.getDataTypes());
    }
}

RowData DataStorage::getRow(int i)
{
    return rows.at(i);
}

QString DataStorage::getColumn(int i)
{
    return columnNames.at(i);
}

int DataStorage::rowsCount()
{
    return rows.count();
}

int DataStorage::getColumnCount()
{
    return columnNames.count();
}

void DataStorage::printAll()
{
    for (int i=0; i<rowsCount(); i++){
        printRow(i);
    }
}

void DataStorage::setColumns(QVector<QString> columnNames)
{
    this->columnNames.resize(0);
    columnProcess.resize(0);

    for (int i=0; i<columnNames.size(); i++){
        addColumn(columnNames.at(i));
    }
}

void DataStorage::addColumn(QString name)
{
    columnNames.append(name);
    ColumnProcess *newCProc = new ColumnProcess;
    columnProcess.append(newCProc);
}

ColumnProcess *DataStorage::getColumnProcess(int i)
{
    return columnProcess[i];
}

void DataStorage::exportData(QString location)
{
    predictColumns = 0;
    dataColumns = 0;

    QVector<QString> *newColumns = new QVector<QString>;
    QVector<RowData> *newRows = new QVector<RowData>;
```

```
    QVector<ProcessOptions> *newProcessOptions = new
QVector<ProcessOptions>;

    //fill with dummy rows
    int newRowLength = calculateNewRowLength();
    for(int i=0; i<rows.length(); i++){
        QVector<QString> newValues;
        QVector<DataTypes> newDataTypes;

        for(int j=0; j<newRowLength; j++){
            //newValues.append("");
            //newDataTypes.append(DT_QSTRING);
        }

        RowData *newRowData = new RowData(newValues, newDataTypes);
        newRows->append(*newRowData);
    }

    QFile file(location + "/output.csv");
    if (!file.open(QFile::WriteOnly|
                QFile::Text)){
        qDebug() << "Could not open file for writing";
        return;
    }
    QTextStream out(&file);

    int skipRows=0;
    for (int i=0; i<columnProcess.length(); i++){
        //calculate skippable rows
        if (columnProcess[i]->getProcessOption()==PO_INCLUDE_PREVIOUS){
            skipRows = (skipRows<columnProcess[i]-
>getProcessAmount())?columnProcess[i]->getProcessAmount():skipRows;
        }
    }

    for(int i=0; i<getColumnCount(); i++) {
        qDebug() << "processing column: " + QString::number(i);
        processColumnRule(i, newColumns, newRows, newProcessOptions);
    }

    qDebug() << "total rows: " + QString::number(rowsCount()-skipRows);
    qDebug() << "newRows: " + QString::number(newRows->length());
    qDebug() << "newColumns: " + QString::number(newColumns->length());

    out << rowsCount()-skipRows << " " << predictColumns << " " <<
dataColumns << "\n";
    for (int j=0; j<newRows->length(); j++){
        int addedCount = 0;
        if (j>=skipRows){
            //first print columns for prediction
            for (int i=0; i<newColumns->length(); i++){
                if (newProcessOptions->at(i) == PO_PREDICT){
                    out << "\"" + (*newRows)[j].getValue(i) + "\"";
                    if (addedCount!=newColumns->length()-1){
                        out << ",";
                    }
                    addedCount++;
                }
            }

            //second, print columns for data
            for (int i=0; i<newColumns->length(); i++){
                if (newProcessOptions->at(i) != PO_PREDICT){
                    out << "\"" + (*newRows)[j].getValue(i) + "\"";
```

```
                    if (addedCount!=newColumns->length()-1){
                        out << ",";
                    }
                    addedCount++;
                }
            }

            //qDebug() << QString::number(addedCount);
            out << "\n";
        }
    }

    file.flush();
    file.close();

    delete newColumns;
    delete newProcessOptions;
    delete newRows;
}

int DataStorage::calculateNewRowLength()
{
    int result = 0;
    for (int i=0; i<columnProcess.length(); i++){
        result += columnProcess[i]->getLengthEffect();
    }
    return result;
}

void DataStorage::processColumnRule(int column, QVector<QString>
*newColumns, QVector<RowData> *newRows, QVector<ProcessOptions>
*newProcessOptions)
{
    //generates csv file with
    ColumnProcess *CP = columnProcess[column];

    //process columns
    qDebug() << "process option: "+QString::number(CP->getProcessOption());
    qDebug() << QString::number(PO_EXCLUDE) + ", " +
QString::number(PO_INCLUDE) + ", " + QString::number(PO_INCLUDE_PREVIOUS) +
", " + QString::number(PO_PREDICT);
    switch (CP->getProcessOption()){
        case PO_INCLUDE: processInclude(column, newColumns, newRows,
newProcessOptions); break;
        case PO_INCLUDE_PREVIOUS: processIncludePrevious(column,
newColumns, newRows, newProcessOptions); break;
        case PO_EXCLUDE: processExclude(); break;
        case PO_PREDICT: processPredict(column, newColumns, newRows,
newProcessOptions); break;
        default: ;
    }
}

void DataStorage::processExclude()
{
    return;
}

void DataStorage::processInclude(int column, QVector<QString> *newColumns,
QVector<RowData> *newRows, QVector<ProcessOptions> *newProcessOptions)
{
    dataColumns++;
    QString name = columnNames[column];
```

```
    qDebug() << "adding row: " + name;

    qDebug() << "rows.length(): " + QString::number(rows.length());
    for (int i=0; i<rows.length(); i++){
        (*newRows)[i].appendValue(rows[i].getValue(column));
        (*newRows)[i].appendDataType(rows[i].getDataType(column));
    }

    newProcessOptions->append(columnProcess[column]->getProcessOption());
    newColumns->append(name);
}

void DataStorage::processIncludePrevious(int column, QVector<QString>
*newColumns, QVector<RowData> *newRows, QVector<ProcessOptions>
*newProcessOptions)
{
    ColumnProcess *CP = columnProcess[column];
    QString name = columnNames[column];

    int amount = CP->getProcessAmount();
    for(int n = 0; n < amount; n++){
        QString nameFull = columnNames[column];

        qDebug() << "adding row: " + name;

        qDebug() << "rows.length(): " + QString::number(rows.length());
        for (int i=amount; i<rows.length(); i++){   //total number of rows
gets reduced by
            (*newRows)[i].appendValue(rows[i-n].getValue(column));
            (*newRows)[i].appendDataType(rows[i-n].getDataType(column));
        }

        dataColumns++;
        newProcessOptions->append(columnProcess[column]-
>getProcessOption());
        newColumns->append(nameFull);
    }
}

void DataStorage::processPredict(int column, QVector<QString> *newColumns,
QVector<RowData> *newRows, QVector<ProcessOptions> *newProcessOptions)
{
    predictColumns++;
    QString name = columnNames[column];

    qDebug() << "adding row: " + name;

    qDebug() << "rows.length(): " + QString::number(rows.length());
    for (int i=0; i<rows.length(); i++){
        (*newRows)[i].appendValue(rows[i].getValue(column));
        (*newRows)[i].appendDataType(rows[i].getDataType(column));
    }

    newColumns->append(name);
    newProcessOptions->append(columnProcess[column]->getProcessOption());
}


void DataStorage::printRow(int i)
{
    getRow(i).PrintRow();
}

DataStorage::~DataStorage()
```

```
{
    ColumnProcess *CP;
    foreach (CP, columnProcess) {
        delete CP;
    }
}


#ifndef DATASETS_H
#define DATASETS_H

#include "QVector"
#include "QEventLoop"
#include "datastorage.h"
#include "dataentry.h"
#include "logger.h"

class DataSets
{
    //this deals with data splitting into sets
public:
    Logger * logger;
    double proportionTrainingSet;
    double proportionGeneralizationSet;
    double proportionValidationSet;

    int inputSize;
    int outputSize;

    QVector<DataEntry*> trainingSet;
    QVector<DataEntry*> generalizationSet;
    QVector<DataEntry*> validationSet;
    DataSets(Logger * logger);
    ~DataSets();
    void generateRandomSets(QString preprocessedFileLocation, double
proportionTrainingSet, double proportionGeneralizationSet, double
proportionValidationSet);
    void readAndMarkRandomLine(QVector<QString> &lines, double*
newExpected, double* newData, int totalSize, int newExpectedSize, int
newDataSize, QVector<int> &unsedEntries);
    int randInt(int low, int high);
    bool setsExist();
    void emptySets();
};

#endif // DATASETS_H

#ifndef DATASETS_H
#define DATASETS_H

#include "QVector"
#include "QEventLoop"
#include "datastorage.h"
#include "dataentry.h"
#include "logger.h"

class DataSets
{
    //this deals with data splitting into sets
public:
    Logger * logger;
    double proportionTrainingSet;
    double proportionGeneralizationSet;
    double proportionValidationSet;
```

```
    int inputSize;
    int outputSize;

    QVector<DataEntry*> trainingSet;
    QVector<DataEntry*> generalizationSet;
    QVector<DataEntry*> validationSet;
    DataSets(Logger * logger);
    ~DataSets();
    void generateRandomSets(QString preprocessedFileLocation, double
proportionTrainingSet, double proportionGeneralizationSet, double
proportionValidationSet);
    void readAndMarkRandomLine(QVector<QString> &lines, double*
newExpected, double* newData, int totalSize, int newExpectedSize, int
newDataSize, QVector<int> &unsedEntries);
    int randInt(int low, int high);
    bool setsExist();
    void emptySets();
};

#endif // DATASETS_H
```

```
#ifndef LOGGER_H
#define LOGGER_H

#include "QTextEdit"
#include "QString"
#include <QDebug>

class Logger
{
public:
    QTextEdit *target;
    Logger(QTextEdit *target);
    void log(QString line);
    void clear();
};

#endif // LOGGER_H

#include "logger.h"


Logger::Logger(QTextEdit *target)
{
    this->target = target;
}

void Logger::log(QString line)
{
    qDebug() << line;
    if (target==NULL){
        qDebug() << "uninitialized target!";
    }else{
        target->append(line);
    }
}

void Logger::clear()
{
    target->clear();
}
```

Bakalaura darbs “Mašīnmācīšanās pielietojums elektrības mirkļa cenas paredzēšanai” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: Sigurds Eglītis

Rekomendēju/nerekomendēju darbu aizstāvēšanai (nevajadzīgo izsvītrot)

Vadītājs: doc., Dr.dat. Jānis Zuters

Recenzents:

Darbs iesniegts Datorikas fakultātē 30.05.2016.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

prot. Nr.

Komisijas sekretār__: