Lints, T. (2003). Projekt MyANN, tehisnärvivõrkude õppetarkvara pakett. Bachelor's thesis, Tallinn University of Technology, Estonia. In Estonian. 136 pages.

T. Lints, "Projekt MyANN, tehisnärvivõrkude õppetarkvara pakett," 2003. Bachelor's thesis, Tallinn University of Technology, Estonia. In Estonian. 136 pages.

```
@misc{Lints03_MyANN,
  author = {Taivo Lints},
  title = {Projekt {MyANN}, Tehisn\"{a}rviv\~{o}rkude
\~{o}ppetarkvara Pakett},
  year = {2003},
  note = {Bachelor's thesis, Tallinn University of Technology,
Estonia. In Estonian. 136 pages}
}
```

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Automaatikainstituut
Reaalajasüsteemide õppetool

Taivo Lints

PROJEKT MyANN

TEHISNÄRVIVÕRKUDE ÕPPETARKVARA PAKETT

Bakalaureusetöö

Juhendajad:
prof. Leo Mõtus
prof. Ennu Rüstern

Tallinn, 2003.

Olen koostanud antud töö iseseisvalt. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

.....

Abstract

Artificial neural networks (ANNs) are computational models rooted in the studies of biological neural systems like brain. ANNs are widely used in various fields of technology because of their good function approximation capabilities. The most widely used class of ANNs is a multilayer feedforward network built of neurons with sigmoidal activation function.

In this thesis the problems of people wishing to get some elementary knowledge about ANNs are shortly discussed and then a new educational software package is presented. This package consists of three easy-to-use programs for getting acquainted with aforementioned multilayer feedforward networks. With these programs it is possible to create networks, visualize their inner workings and try out an interactive demonstration of their practical use. The whole package is open source and also contains a C++ library implementing these networks which can be used by people with C++ programming knowledge in their own projects.

Resümee

Tehisnärvivõrk on looduslikest närvivõrkudest inspireeritud vahend info töötlemiseks. Tehisnärvivõrgud kui head funktsioonide aproksimeerijad on leidnud laialdast kasutust paljudes erinevates valdkondades. Enimlevinud on mitmekihilised ilma sisemise tagasisideta võrgud, mille neuronite aktiveerimisfunktsiooniks on sigmoidfunktsioon.

Käesolevas töös vaadeldakse lühidalt probleeme, mis tekivad tehisnärvivõrkudega esmakordselt tutvuda soovijatel, ning seejärel pakutakse välja uus õppetarkvara pakett. See pakett koosneb kolmest kergesti kasutatavast programmist, mis võimaldavad tutvuda eelnimetatud mitmekihiliste tagasisideta võrkudega. Nende programmidega saab närvivõrke valmistada, visualiseerida nende sisemisi protsesse ja uurida interaktiivset näidet tehisnärvivõrkude praktilisest kasutamisest. Kogu pakett on avatud lähtekoodiga ja sisaldab ka teeki, mis neid võrke realiseerib ja mida C++ keelt valdavad programmeerijad saavad kasutada omaenda projektides.

Sisukord

1. Sissejuhatus

Efektiivsetest / kasulikest õppeprogrammidest on tõsine puudus kõikjal, ka TTÜ's. Nad ei pea olema tohutult mahukad ja universaalsed, pigem just mingi konkreetse probleemi illustreerimiseks – lihtsad, atraktiivsed, õpetlikud, väikesed ja tasuta – sellisel juhul on tudengid nende kasutamisest huvitatud. Kuna käesoleva töö autor soovis tutvuda tehisnärvivõrkudega, aga ei leidnud sobivaid eeltoodud kirjeldusele vastavaid õppeprogramme, siis tekkis mõte kirjutada need ise.

Töö eesmärk: valmistada õppeotstarbeline tarkvarapakett lihtsamate / levinumate tehisnärvivõrkude tööpõhimõtetega tutvumiseks, aga ka loodud võrkude praktiliseks kasutamiseks.

Käesolev töö jaguneb kaheks suuremaks osaks: tarkvarapakett ise ning selle loomist kommenteeriv tekst. Et pakett oleks kasutajale võimalikult arusaadav ja õpetlik, on suurem osa informatsiooni paigutatud paketti endasse – nii HTML formaadis kasutusjuhenditesse kui mahukate kommentaaridena ka otse lähtekoodi, mis mõlemad on toodud töö lisas. Lisadele eelnev tekst arutleb peamiselt paketi loomisega seotud küsimuste üle ning püüab mitte liigselt dubleerida lisades toodud põhjalikumaid tekste.

Tarkvarapakett *MyANN* on esialgu loodud ingliskeelsena, sest suurem osa potentsiaalseid kasutajaid asub väljaspool Eestit (vt. peatükk 5.3. "Potentsiaalsed kasutajad"). Vajaduse korral ka eestikeelse versiooni valmistamine on lihtne – piisab vaid tõlkimisest, sisuline töö on juba tehtud.

2. Tehisnärvivõrkude valdkonna lühitutvustus

Tehisnärvivõrk *(Artificial Neural Network, ANN)* on looduslikest närvivõrkudest inspireeritud vahend info töötlemiseks [1]. Erinevaid tehisnärvivõrkude definitsioone on välja pakutud palju. Enamus neist toetuvad järgmistele põhimõtetele:

- Närvivõrk koosneb suurest hulgast lihtsatest andmetöötlusüksustest. Igaühel neist võib olla vähesel määral isiklikku mälu.
- Võrgusõlmed (st. eelnimetatud üksused) on omavahel ühendatud suhtluskanalitega, milledes liigub enamasti numbriline informatsioon (vastandina märkidel / sümbolitel baseeruvale infole).
- Võrgusõlmed opereerivad ainult lokaalse – ühenduste kaudu saabuva ja oma mälus sisalduva – informatsiooniga. Seda piirangut võidakse mitte arvestada võrgu õpetamise ajal. [2]

Et saada paremat ülevaadet kõikvõimalikest võrkudest, mis eeltoodud kirjeldusele vastavad, on kasulik nad teatavate omaduste alusel gruppideks jaotada. Sarnaselt definitsioonide paljususele on ka klassifitseerimismeetodeid mitmeid. Üks lihtsam ja küllaltki kergestimõistetav skeem vaatleb tehisnärvivõrgu viit omadust:

- Topoloogia – mil viisil on võrgusõlmed omavahel ühendatud.
- Arhitektuur:
  - Otsesidega e. tsükliteta võrgud.
  - Välise tagasisidega võrgud, kus võrgu eelmised väljundid on uuteks sisenditeks. Ainult kõige esimene, võrgu käivitamiseks vajalik sisendvektor tuleb väliskeskkonnast.
  - Sisemise tagasisidega võrgud, millede väljundid sõltuvad mitte ainult hetkel sisenditele antud väärtustest, vaid ka eelnevatest sisenditest.
  - "Väljundita" võrgud, kus vaadeldakse kõigi võrgusõlmede olekuid, mitte paari konkreetset väljundit.
- Neuroni mudel – mida võrgusõlm täpselt teeb.
- Õpetamisalgoritm – kuidas seatakse võrgu parameetreid.

- Operatsioonide ajastus – millistel ajahetkedel / mis järjekorras võrgusõlmed omavahel suhtlevad. [3]

Tuntumatel võrgutüüpidel on välja kujunenud nimed, mis otseselt ei vasta omadustepõhisele klassifikatsioonile. Nimi võib tuleneda konkreetsele võrgutüübile alusepanijast (nt. Hopfield'i või Kohonen'i võrgud), aga ka võrgu mõnest olulisemast tunnusjoonest (nt. mitmekihilised pertseptronid või radiaalbaasvõrgud). Kõigi nende võrkude detailsem kirjeldamine ei ole siinkohal vajalik, sest käesoleva töö seisukohalt on oluline ainult enimkasutatav tüüp – mitmekihilised pertseptronid. Selle võrgu kirjeldus koos kasutatava õpialgoritmi tutvustusega on toodud projekti võrgulehel, mis asub töö lisas. NB! Kuna mõiste "pertseptron" ei ole üheselt defineeritav – erinevad uurijad nimetavad selle terminiga erinevaid asju – siis käesolevas töös on mõistele "mitmekihiline pertseptron" enamasti eelistatud väljendit "mitmekihiline ilma sisemise tagasisideta võrk", mis on veidi laiem mõiste ja peaks vähendama võimalikke valestimõistmisi.

Tehisnärvivõrkude olulisim omadus on nende võime teostada aproksimeerimist – võrku õpetatakse piiratud arvu näidisandmetega, mis võivad olla veidi ebatäpsed, ja seejärel suudab võrk mõistlikult reageerida ka sellistele sisenditele, mida ta küll õppinud ei ole, kuid mis on õpitud sisenditega mingil viisil sarnased. Selline omadus viitab närvivõrkude sobilikkusele mitmesuguste klassifitseerimise ja funktsiooni aproksimeerimisega seotud ülesannete lahendamiseks.

Kasulikke omadusi on närvivõrkudel teisigi: adaptiivsus, valmisolek paralleeltööks, veakindlus:

- Adaptiivsuse all mõistetakse seda, et keskkonna või kasutajapoolsete nõuete väikese muutumise korral on enamasti kerge õpetada närvivõrku uues olukorras adekvaatselt käituma. Võib isegi luua võrgu, mis selliseid muutusi ise jälgib ja automaatselt uute oludega kohaneb.
- Kuna närvivõrgud on juba oma olemuselt suure hulga paralleelselt töötavate andmetöötlusüksuste kogum, siis on neid lihtne realiseerida paralleelselt töötaval riistvaral (näiteks *VLSI* tehnoloogial (*Very Large-Scale Integration* –

"kõrgtihe lausintegratsioon")) ja saavutada suuremat töökiirust kui tavalisel järjestiktööl põhineval arvutil.

- Kui närvivõrk on realiseeritud riistvaraliselt, siis võib ta olla küllaltki veakindel – võrgu mõne komponendi riknemine ei muuda kogu võrku kasutuskõlbmatuks, vaid ainult vähendab töö kvaliteeti. [4]

Näiteid tehisnärvivõrkude kasutusaladest:

- **Erinevad teadusharud**. Katseandmetest süsteemi mudeli loomine, mida saab kasutada süsteemi käitumise uurimiseks ja ennustamiseks, näiteks vedelike voolamise, kliima, filtrite mudelid. Ligikaudsed arvutused, näiteks keerukate funktsioonide lähendamine.
- **Tööstus**. Protsessijuhtimine, toodete kvaliteedi ennustamine ja jälgimine, tootmistehnika rikete ennustamine.
- **Telekommunikatsioon**. Signaalitöötlus.
- **Robootika**. Hääle- ja pildituvastussüsteemid, robotite liikumise juhtimine.
- **Meditsiin**. Analüüsiandmete tõlgendamine.
- **Maavarade kaevandamine**. Kasutamata ressursside asukoha määramine sondeerimisandmetest.
- **Finantssektor**. Riskianalüüs, rahakursside ja aktsiahindade ennustamine, kahtlaste tehingute automaatne tuvastamine.
- **Arvutianimatsioon**. Tegelaste ja muude objektide juhtimine arvutimängudes ning animafilmides. [5, 6]

3. Olemasolev tarkvara

Tehisnärvivõrgud on leidnud laialdast kasutust ja seetõttu on olemas ka palju erinevaid programme nende loomiseks, uurimiseks ja kasutamiseks. Käesoleva töö huviorbiidis on eelkõige programmide kasutatavus tehisnärvivõrkudega tutvumiseks, peamiselt inimeste poolt, kes vastava temaatikaga varem üldse kokku ei ole puutunud. Sellest tulenevalt on vaadeldud peamiselt tarkvara kasutamise lihtsust ja info visualiseerimise meetodeid – graafiliselt esitatud info on tihti paremini haaratav kui tekstiline-numbriline.

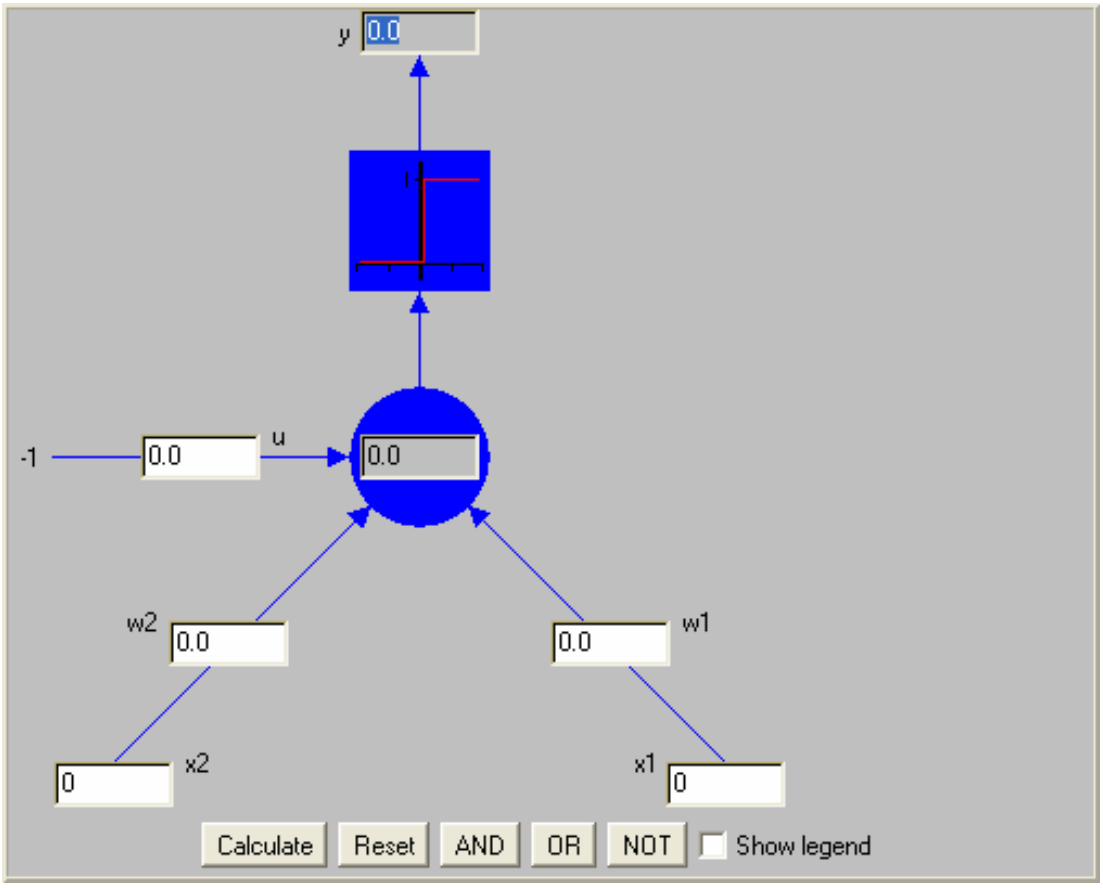
Kõikide olemasolevate programmide uurimine ei ole mõeldav, seetõttu on järgnevas lühidalt kirjeldatud ainult mõnesid neist. Üks olulisemaid valikukriteeriume oli vastava tarkvara laiem tuntus ja populaarsus.

3.1. Java apletid

Üks lihtsamini kasutatavaid õppetarkvara vorme on *Java*'s kirjutatud väikesed programmikesed (*applet*, eesti vaste "aplett"), mis on paigutatud veebilehtedele. Selliste programmide kasutamiseks piisab tavalisest veebilehitsejast. Samas võib osutuda problemaatiliseks nende kasutamine ilma internetiühenduseta arvutites.

3.1.1. Fred Corbett

Laialdaselt on levinud Fred Corbett'i kirjutatud *Java* apletid, mis analoogselt projektiga *MyANN* on valminud bakalaureusetööna (Manitoba Ülikool, Kanada). Üks neist on näitena toodud joonisel 1.

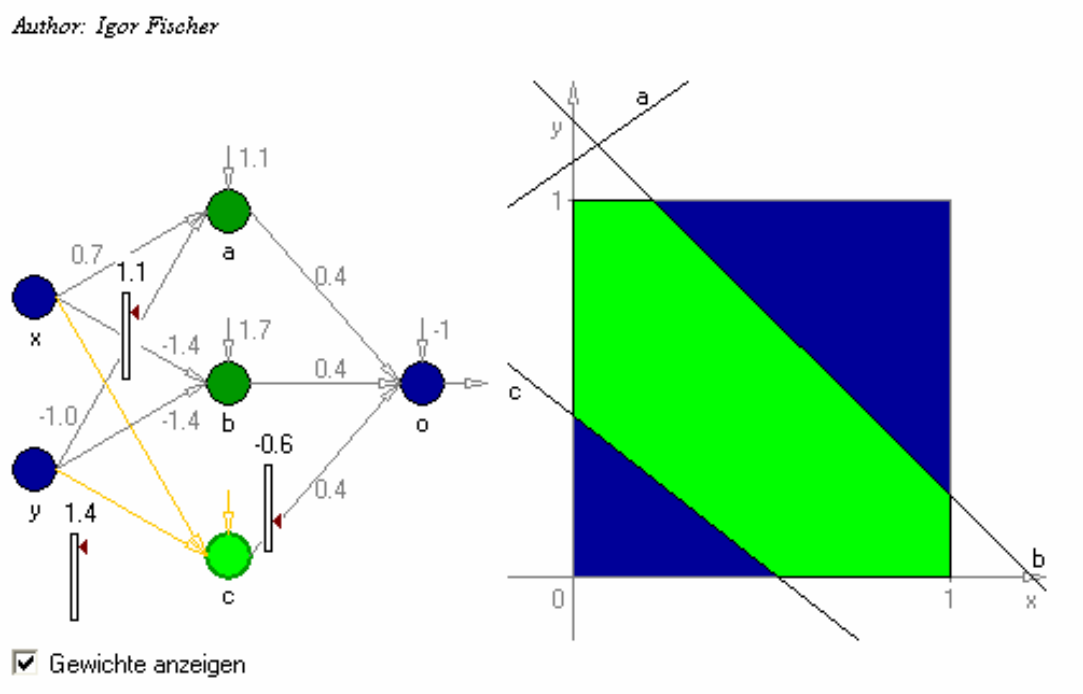


Joonis 1. Fred Corbett'i *Java* aplett *"Artificial Neuron"* [7].

Neuroni visualiseerimine sellisel viisil on küll üsna õpetlik, kuid omab ka probleeme. Esiteks on parameetrite muutmine üsna ebamugav – selleks on vaja uus väärtus vastavasse kasti trükkida. Teiseks ei ole info koheselt haaratav – kõigepealt tuleb arvulised väärtused mõttes enda jaoks tajutavaks teisendada. Nimelt ei ole arv inimaju jaoks loomulik info esitamise viis ja vajab põhjalikumat eeltöötlust, erinevalt näiteks värvuskodeeringust. Positiivse poole pealt soodustab arvude kasutamine värvikodeeringu asemel apleti mustvalget väljatrükki, samuti tema kasutamist inimeste poolt, kellel on raskusi värvide eristamisega.

3.1.2. Igor Fischer

Eeltoodust suurema kasutamismugavuse ja atraktiivsusega on Igor Fischer'i loodud apletid (Tübingen'i Ülikool, Saksamaa). Tõsi küll, sisu poolest nad Fred Corbett'i aplette ei asenda, pigem täiendavad. Joonisel 2 on toodud üks näide Fischer'i loomingust.



Joonis 2. Igor Fischer'i *Java* aplett *"2-layer binary perceptron"* [8].

Selle apleti parempoolses osas on näha, milliste võrgu sisendväärtuste korral on väljund "1" (hele piirkond) ja milliste korral "0" (tume piirkond). Vasakpoolses osas on võimalik ühenduste kaale reguleerida, mille tagajärjel nimetatud piirkonnad muutuvad.

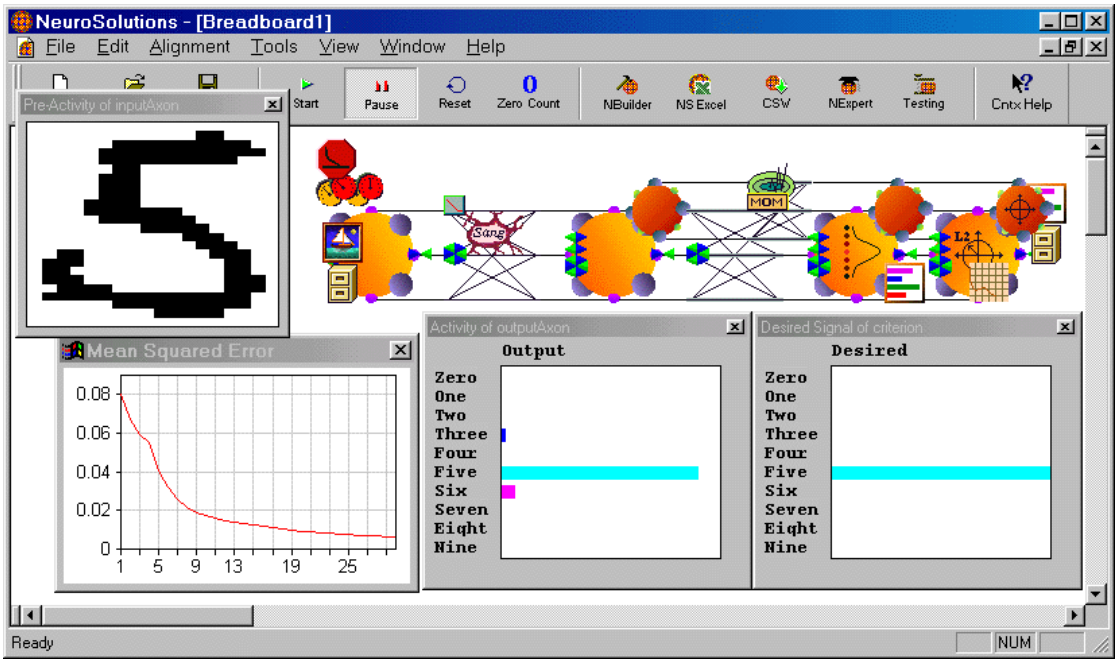
See ja mitmed analoogsed Fischer'i apletid illustreerivad suurepäraselt, mil viisil täpselt mõjuvad kaalud väikestes võrkudes, andes kaudselt ka juhtnööre võrkude struktuuri valikuks konkreetsete lihtsamate probleemide korral. Ülejäänud apletid Fischer'ilt ja tema kolleegidelt illustreerivad peamiselt võrkude õpetamisega seotud küsimusi. Võrkude tööpõhimõtte efektset visualisatsiooni loodud ei ole.

3.2. Iseseisvad programmid

Java apletid on suhteliselt lihtsalt kasutatav tarkvara vorm, kuid ka nende võimalused on oluliselt piiratud. Põhjalikum ja / või kiirem tarkvara kirjutatakse iseseisvate programmidena, mis installeeritakse kasutaja arvutisse.

3.2.1. NeuroSolutions

Küllaltki suurte võimalustega tasuline tarkvara on *NeuroSolutions (NeuroDimension, Inc.).* Koostöös *Computational Neural Engineering Lab*'iga (Florida Ülikool) toimub selle programmi pidev täiendamine vastavalt tehisnärvivõrkude tehnoloogia arengule. Näide programmi ekraanipildist on toodud joonisel 3:

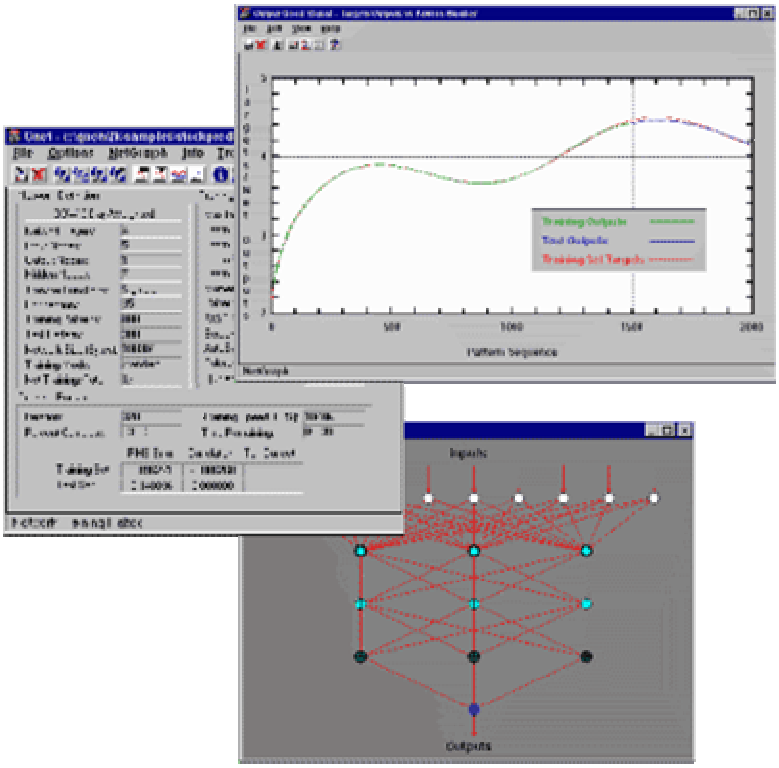


Joonis 3. *NeuroSolutions*'i ekraanipilt [9].

Nagu jooniselt näha, on võrgu visualiseerimisel mindud ikoonide kasutamise teed. Tulemus on ilmselt natukene liiga kirju. Pealegi on võrgu komponendid esitatud vaid sümboolselt ega näita eriti täpselt võrgu sisemist struktuuri. Selline lähenemine võib olla sobiv kasutajale, kellel on piisavalt eelteadmisi närvivõrkudest, aga mitte algajale huvilisele. *NeuroDimension, Inc.*'i valik on mõistetav, sest *NeuroSolutions*'i peamine sihtgrupp ongi professionaalid.

3.2.2. Qnet

Üsna populaarne tasuline programm on *Qnet (Vesta Services, Inc.)*, mis samuti on mõeldud kasutamiseks professionaalidele:



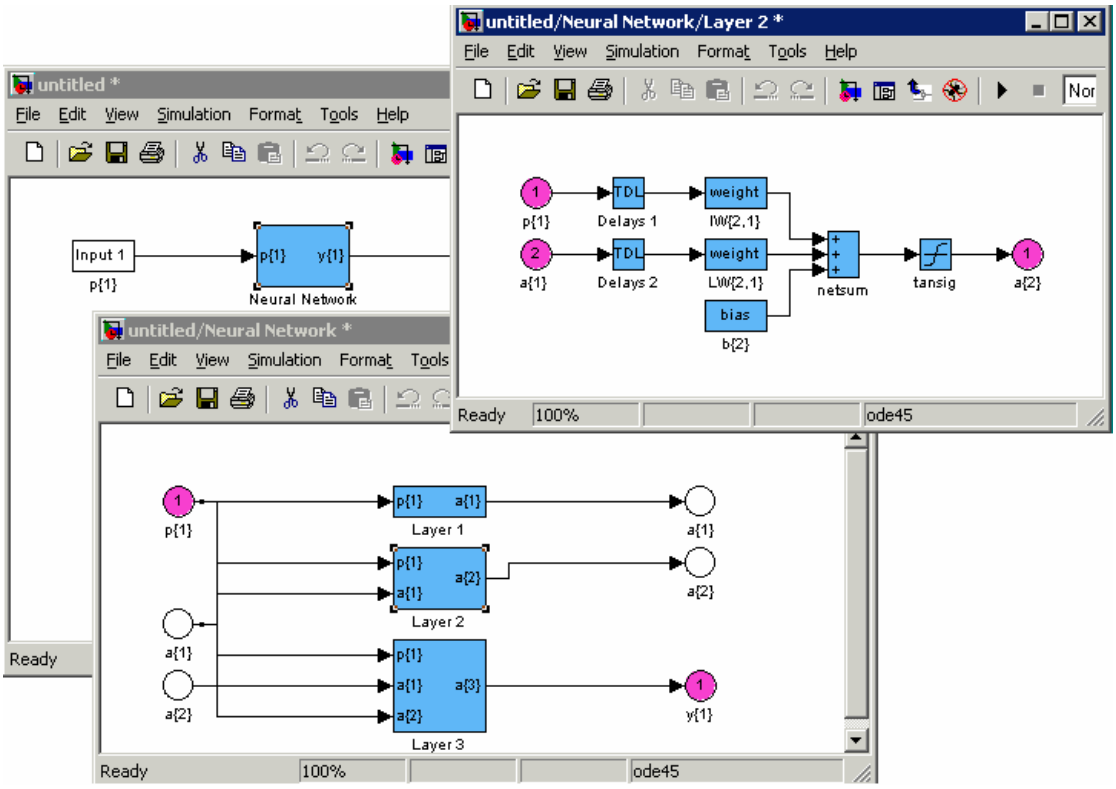
Joonis 4. Valik programmi *Qnet* ekraanipiltidest [6].

Qnet on tõsiseltvõetav vahend tehisnärvivõrkude loomiseks, kuid võrgu enda visualiseerimisele ei ole erilist tähelepanu pööratud. Joonise 4 alumises osas toodud võrgu pilt on nähtav vaid loomisprotsessi alguses, struktuuri määramise ajal. Võrgu tööd otseselt visualiseerida ei saa, võimalik on vaid paljude muutujate olekute mitmekülgne graafiline kujutamine nagu teisteski analoogsetes programmides. *Qnet*'i kasutamine on üllatavalt lihtne, kuid seda tingimusel, et kasutaja on tehisnärvivõrkude temaatikaga kursis. Seega sobib ka *Qnet* peamiselt spetsialistidele, mitte algajatele.

3.2.3. MatLab

Insenerialadel äärmiselt laialt levinud tasuline tarkvarapakett on *MatLab*, millele saab lisada ka tööriistad närvivõrkude kasutamiseks (*Neural Network Toolbox, NNT*).

MatLab ise on suures osas tekstipõhine programm (peamine suhtlemine toimub käsurea abil) ja seetõttu väga efektseid visualisatsioone seal ei ole. Tõsi küll, *MatLab*'ile lisatav vahend *SimuLink* võimaldab süsteeme konstrueerida ka graafiliselt:



Joonis 5. Närvivõrkude konstrueerimine *Simulink*'is [10].

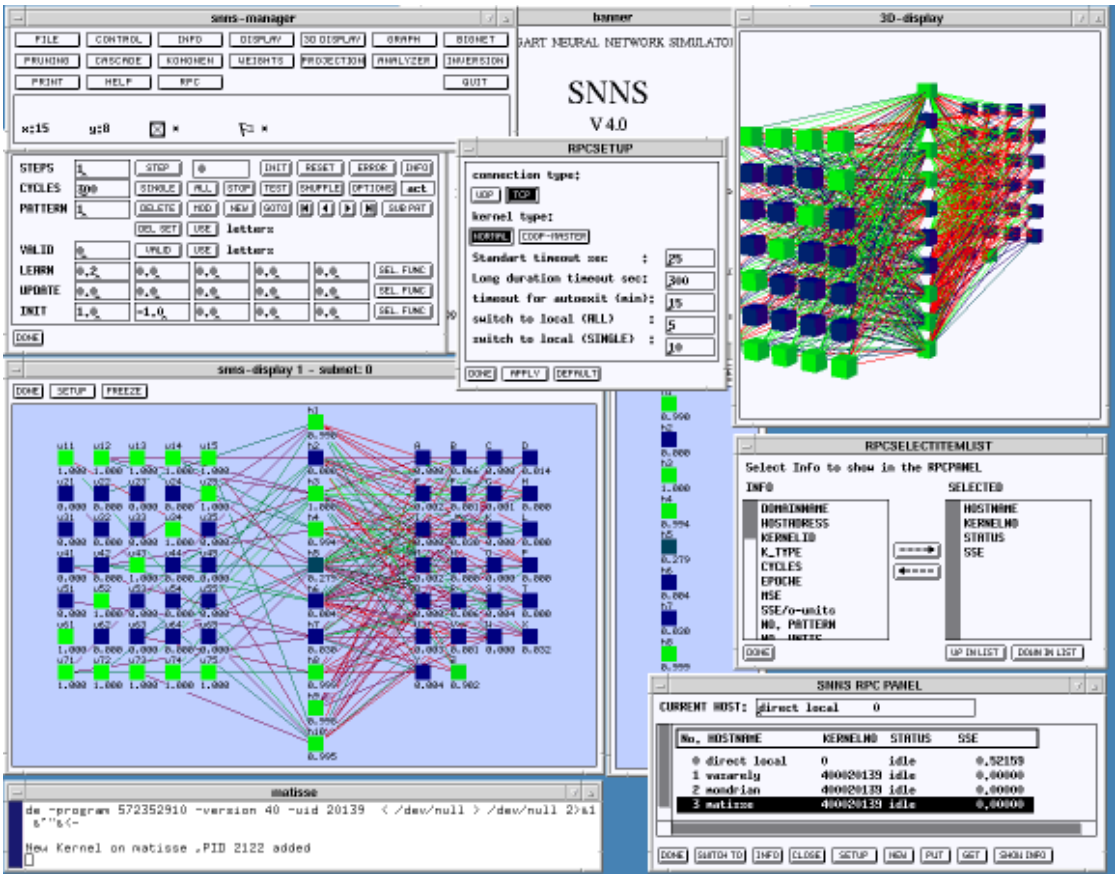
Jooniselt on näha, et võrkude struktuuri esituse detailsuse määrab *Simulink*'is kasutaja – võimalik on madalama taseme objektid grupeerida kõrgema taseme komponendiks. Võrkude ehitusest kuni tehisneuroni sisuni välja saabki suhteliselt hea ettekujutuse, dünaamikast aga mitte. Parameetrite ajaliste muutuste visualiseerimine toimub mitmesuguste graafikute joonistamise teel, aga mitte võrgu enda animeerimisega.

*NNT*s on olemas mitu spetsiaalset demonstratsiooni närvivõrkudega tutvumiseks, kuid ka nendes ei ole kasutatav esitusviis sobivaim esmase ülevaate saamiseks tehisnärvivõrkudest. Tegu on tüüpprobleemiga – võrgu parameetrite esitamine kujul, mis ei võimalda võrgu oleku kiiret tajumist kasutaja poolt (nt. kaalude esitamine liugurite abil).

Kuna *MatLab* on populaarne suurte võimalustega tarkvarapakett, kus närvivõrgud on vaid üks väike alamosa, siis kasutatakse teda laialdaselt ülikoolide õppetöös. *MatLab*'i kasutamise oskus on üliõpilasele kindlasti kasulik, aga algteadmiste omandamiseks erinevatel teemadel, näiteks süsteemiteoorias või närvivõrkude alal, oleks siiski efektiivsem kasutada spetsiaalseid lihtsaid õppeprogramme ning alles seejärel asuda tegutsema võimsamate tööriistadega nagu *MatLab*.

3.2.4. Stuttgart Neural Network Simulator

Eeltoodud eraldiseisvad programmid olid kõik tasulised, mis ei soodusta nende kasutamist õppetöös. Leidub ka mitmesugust tasuta tarkvara. Üks põhjalikumaid ja seetõttu ka laiemalt levinuid on *Stuttgart Neural Network Simulator*:



Joonis 6. *Stuttgart Neural Network Simulator*'i mitmesugused aknad [11]

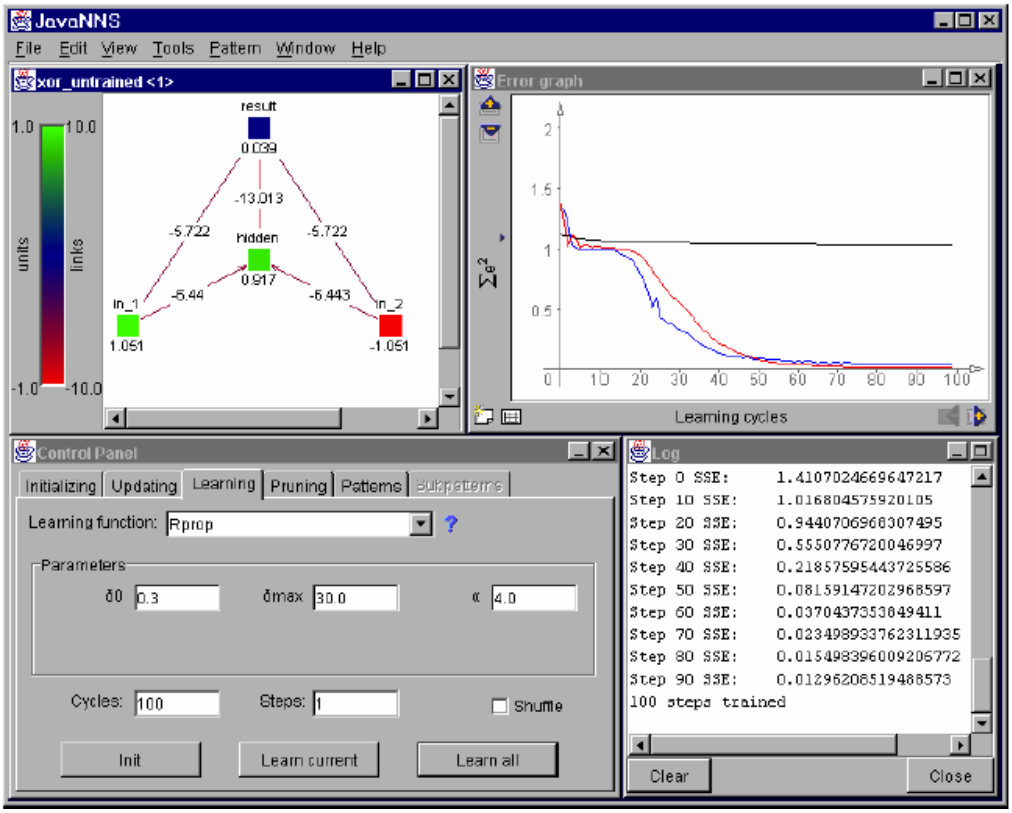
Siin võib juba näha üsna häid võtteid tehisnärvivõrkude visualiseerimiseks. Kasutatud on värvuskodeeringut ja mitmesuguseid erinevaid vaateid võrgule. Eriti efektne on

kolmemõõtmeline vaade, mille abil saab hea ettekujutuse ka üsna keerulistest võrkudest. *SNNS* on mõeldud eelkõige uurimistööks, mitte õppevahendiks, ja omab seetõttu üsna keerukat kasutajaliidest paljude erinevate võimalustega. Juba olemasolevate võrkude graafiline vaatlemine on siiski küllaltki lihtne ka algajale kasutajale.

Esmasel tutvumisel *SNNS*'iga kerkisid esile kaks tõsisemat probleemi. Esiteks puudub võimalus võrgu sisendväärtustega lihtsal viisil "mängida", eesmärgiga vaadelda võrgu tööd. Võimalik on kasutada vaid eelnevalt laaditud või enda poolt sisestatud sisendväärtusi. Teiseks probleemiks on kolmemõõtmelise vaate aeglane töö. Põhjus võib olla selles, et *SNNS* on kirjutatud *Sun* masinal *X-Windows*'i jaoks ning *MS Windows*'is jookseb vaid läbi *PC X server*'i. Kahjuks puudus võimalus katsetada programmi platvormil, mille jaoks ta valmistatud on.

3.2.5. Java Neural Network Simulator

Igor Fischer (kelle *Java* aplettidest eelnevalt juttu oli) kirjutab koos oma kolleegidega *Stuttgart Neural Network Simulator*'i tuuma kasutades uut, *Java*'l põhinevat tehisnärvivõrkude tarkvara *Java Neural Network Simulator*:

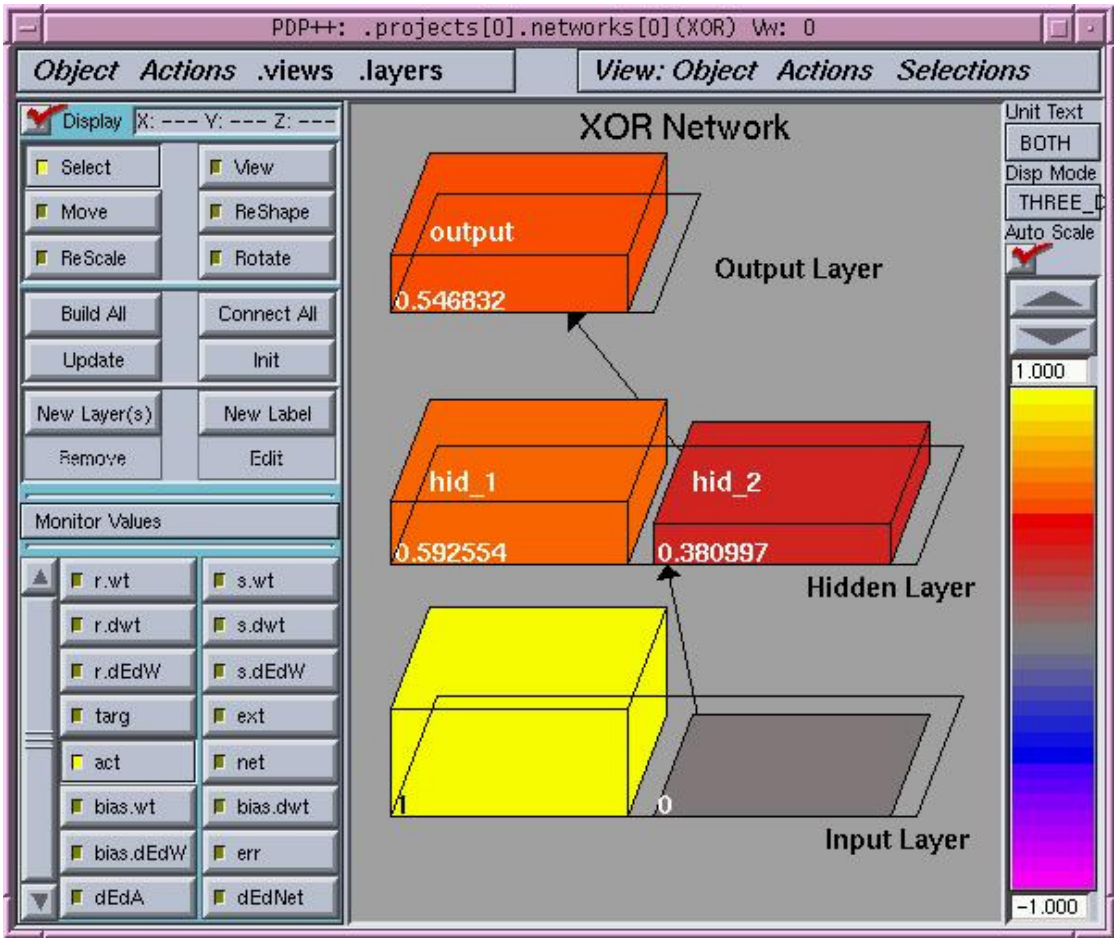


Joonis 7. *JavaNNS* ekraanipilt [12].

Hetkeseisuga on *SNNS*'iga võrreldes peamine erinevus kasutajaliideses, mis on ümber disainitud eesmärgiga muuta programmi kasutamine lihtsamaks. Kuna uus liides on kirjutatud *Java*'s, siis on programm suurema porditavusega ja töötab ka keskkonnas *MS Windows* paremini kui *SNNS*. Kahjuks ei ole *JavaNNS*'is võimalust võrkude kolmemõõtmeliseks vaatlemiseks, kuid tõenäoliselt see tulevikus lisatakse. Võrgu sisendi muutmise probleem on analoogne *SNNS*'iga.

3.2.6. PDP++

Kirjelduse järgi üsna huvitav tasuta programm on *PDP++*, mis peaks pakkuma suures valikus erinevaid võimalusi tehisnärvivõrkude visualiseerimiseks, seda ka reaalajas, näiteks võrgu õpetamise ajal.



Joonis 8. Näide *PDP++* ekraanipildist [13].

Kahjuks ei ole käesoleva töö autoril veel olnud võimalust programmi *PDP++* uurida, kuna see oli *dial-up*'i kaudu allalaadimiseks liiga suur, üle 21 MB. Arvestades olukorda, kus kiire internetiühendus ei ole veel eriti levinud, on programmi suur maht tõsiseltvõetav takistus tema kasutamiseks koduses keskkonnas, eriti kui soovitakse saada lihtsat lühikest ülevaadet vastavast temaatikast, mitte teostada põhjalikku sisulist uurimust.

4. Milleks uus pakett?

Hea õppetarkvara olulisemaid tunnuseid on kolm: atraktiivsus, lihtsus ja tasuta kättesaadavus. Järgnevas on põhjendatud, miks need tunnused olulised on. Tulenevalt käesoleva töö temaatikast on näidetena kasutatud tehisnärvivõrkude valdkonda ning ühte sellel alal enamlevinud tarkvaralist vahendit *MatLab*.

1. Üliõpilane või kes iganes EI TAHA raha kulutada mingisuguse programmi peale. *MatLab*'i või mõne muu analoogse toote ostab ta siis, kui selleks on äärmine vajadus (õppejõud käseb või lihtsalt muidu kuidagi ei saa). Kui närvivõrkudega tuleb tegeleda ainult ühe-kahe õppeaine raames (ja sealgi võib-olla vaid möödaminnes), siis ainult selle jaoks tudeng küll vabatahtlikult midagi osta ei kavatse. Analoogne on olukord siis, kui inimesel on üksnes kerge uudishimu, et "mis see tehisnärvivõrk küll olla võiks?". See ei ajenda midagi vähegi kallimat ostma.

2. Kui inimesel on huvi / vajadus närvivõrkudest lihtsalt natukene midagi teada saada, siis tal enamasti puudub motivatsioon selle nimel midagi keerukat ära õppida (näiteks *Matlab*'i keskkonna kasutamist). Pigem soovib ta ühe nupulevajutusega miskit silmale ilusat ja arusaadavat tööle panna ning sellega veidikene mängida. Kui see osutub tema jaoks huvitavaks, siis võib-olla tekib ka soov asjaga põhjalikumalt tegeleda, sh. näiteks *MatLab*'i kasutada.

3. Kui inimesel EI OLE aine vastu huvi, siis selle huvi tekitamiseks PEAB kasutatav õppevahend ilus / tore / huvitav olema. Vastasel korral tekib inimesel aine suhtes vastumeelsus (kui teda sunnitakse õppima tema jaoks ebahuvitavat asja). *MatLab* on pigem tõsine tööriist kui motiveeriv õppevahend.

Enamus eelnevas peatükis vaadeldud tarkvara kõigile nendele kolmele tunnusele – atraktiivsus, lihtsus ja tasuta kättesaadavus – ei vasta. Need, mis vastavad, näiteks Fischer'i apletid, katavad ainult mõned üksikud küsimused tehisnärvivõrkude temaatikast. See on täiesti normaalne, sest väga mahukas ja universaalne programm enamasti ei saagi kõiki kolme tingimust rahuldada, eriti lihtsuse nõuet. Et aga suurem osa problemaatikast on veel sobivate õppeprogrammide poolt katmata, on uue õppetarkvara loomine igati õigustatud.

5. Pakett MyANN

5.1. Tutvustus

MyANN on õppetarkvara pakett, mis koosneb kolmest väikesest programmist ning ühest C++ teegist. Põhjalikum informatsioon asub projekti *MyANN* koduleheküljel, mis on toodud käesoleva töö lisas. Järgnevalt ainult lühike ülevaade paketi võimalustest.

Paketi *MyANN* abil on võimalik:

- Enne programmide kasutamist tutvuda närvivõrgu üldiste tööpõhimõtetega, lugedes HTML formaadis teksti, mis on kirjutatud võimalikult kasutaja-sõbralikult, ilma erilisi eelteadmisi nõudmata.
- Valmistada mitmekihilisi ilma sisemise tagasisideta tehisnärvivõrke (programm *Trainer*). Võrgu soovitava struktuuri ja olulisemad parameetrid saab määrata konfiguratsioonifailis, mis sisaldab suurel hulgal abistavat teksti, et ei oleks vajadust samal ajal kasutusjuhendist infot otsida. Faili saab lugeda ja muuta mistahes tekstiredaktoriga, kasvõi *Notepad*'iga.
- Õpetada neid võrke näidisandmetega, kasutades tavalist vea tagasilevi algoritmi (programm *Trainer*). Õpetamiseks vajalikud lähteandmed saab jällegi määrata vastavast konfiguratsioonifailist, mis sisaldab hulgaliselt abistavat teksti.
- Visualiseerida loodud võrkude sisemisi protsesse nende töö ajal (programm *Visualizer*). Kasutajal on võimalus väga kergesti muuta võrgu sisendväärtusi ning vaadelda võrgu reaktsiooni sellele. Nii võrgu struktuur kui tema poolt töödeldud signaalide väärtused kõigis sõlmedes on esitatud võimalikult arusaadaval ja aju poolt kiiresti omastataval viisil, kasutades komponentide sobivat paigutust ning info värvuskodeeringut.
- Vaadelda simulatsiooni valgustundlike sensoritega "olendist", keda juhib tehisnärvivõrk (programm *LightChaser*). Simulatsioon on täielikult interaktiivne – kasutaja saab liigutada nii valguslaiku kui vajadusel ka olendit.

Närvivõrku, mis valgustundlikku olendit juhib, saab soovi korral asendada mõne teisega, ka omaloodud võrguga.

- Valmistada programme, mis kasutavad *MyANN* teeki (vajalik C++ keele oskus). See teek võimaldab ilma suurema vaevata kasutada oma programmides mitmekihilisi ilma sisemise tagasisideta tehisnärvivõrke. Paketi kogu lähtekood on avalik ning põhjalikult kommenteeritud.

5.2. Eelised

Projekti tegemise käigus lähtuti märksõnadest lihtne, atraktiivne, õpetlik, modifitseeritav, tasuta.

Lihtne – inimene, kes soovib kasutada *MyANN* programme, ei vaja eelnevalt pikka spetsiaalkoolitust. On proovitud luua kasutajasõbralik intuitiivne keskkond. Ainsaks eelduseks on piisav arvuti kasutamise oskus. Kui kasutajal puuduvad teadmised tehisnärvivõrkudest, siis on tal võimalus eelnevalt lugeda lühikest sissejuhatavat esseed, mis suuremas osas on püütud kirjutada "inimlikus" keeles, vältides liigseid nüansse, mis alaga esmakordselt kokkupuutuja jaoks oleks pigem segavad kui kasulikud.

Atraktiivne – keskkond on interaktiivne ning näeb ka ilus välja. See peaks vähemalt mõneks ajaks kasutaja tähelepanu köitma.

Õpetlik – kasutajal tekib ettekujutus lihtsamate närvivõrkude tööpõhimõttest. Eriti oluline on siin komponent *Visualizer*, mis teeb nähtavaks närvivõrgu sisemuse töö ajal. Info kodeerimiseks on kasutatud värve – neuronite vaheliste ühenduste värv sõltub nende kaalust, neuronite värv sõltub nende väljunditest. Selline esitusviis võimaldab närvivõrgu struktuuri ja olekut aimata juba ainsa kiire pilguheiduga, ilma vajaduseta hakata mõttes numbrilisi andmeid enda jaoks tajutavaks teisendama. Samuti saab äärmiselt lihtsalt muuta võrgu sisendväärtusi ja jälgida reaktsiooni sellele, mille tulemusena tekib kiiresti arusaamine tehisnärvivõrgu tööst. Pärast *Visualizer*'i kasutamist on lihtsam võrgu tööd ette kujutada ka ilma arvuti abita, kasutades samasugust visuaalset mälupilti.

Modifitseeritav – programmide lähtekood on avalik ning põhjalikult kommenteeritud. See võimaldab programmeerimisoskusega inimestel muuta paketi komponente enda vajadustele sobivamaks või kasutada närvivõrke ilma suurema vaevata omaenda programmides.

Tasuta – *MyANN* saab olema Internetist täiesti tasuta kättesaadav.

5.3. Potentsiaalsed kasutajad

Projekti *MyANN* peamine sihtgrupp on autori "hingesugulased" – teadmishimulised arvutihuvilised, kes täiesti vabatahtlikult soovivad tutvuda kõikvõimalike erinevate teemavaldkondadega, sealhulgas ka tehisnärvivõrkudega.

Käesolevast tööst saavad ilmselt kasu ka tudengid, kes seoses õppetööga PEAVAD tehisnärvivõrkudega tutvuma. Samuti on *MyANN* programme võimalik kasutada õppejõudude poolt demonstratsioonivahendina. E-õppe läbiviimisel on sellised programmid samuti abiks. Võib-olla saab seda projekti ära kasutada ka sellel sügisel TTÜ poolt avatavas tehnika- ja arvutiteaduste kaugõppes.

Närvivõrkude uurimine programmiga *Visualizer* võib huvi pakkuda ka neile, kes valdkonnaga põhjalikumalt kursis, näiteks inimestele, kes võrke reaalselt kusagil rakendavad ja soovivad saada nende tööst paremat ettekujutust.

MyANN sisaldab närvivõrku realiseerivat C++ teeki, mida saavad kerge vaevaga kasutada programmeerijad, eelkõige "kodukasutajad", näiteks tudengid mõne kooliga seotud projekti raames, arvutimängudes või -animatsioonis. Tööstuslikuks kasutamiseks teek ei sobi – puudub igasugune garantii töökindluse ja töö täieliku õigsuse suhtes.

Eeltoodud potentsiaalseid kasutajaid vaadates selgub tõsiasi, et Eestis on selliseid kasutajaid üsna vähe. Peamisse sihtgruppi kuuluvate inimeste osakaal ühiskonnas on küllaltki väike ilmselt igal pool maailmas. Eesti väikese rahvaarvu tõttu on siin väike ka nende absoluutarv.

Vaadeldes Eesti noorte poolt loodavate originaalprogrammide taset on alust arvata, et neid, kes suudaks ja ka ise tahaks kasutada *MyANN* teeki, on siin äärmiselt vähe.

Õppeasutusi, kus tehisnärvivõrkude temaatikat kasvõi põgusalt käsitletakse, on Eesti väiksusest tingituna samuti ainult mõned üksikud. Sellest tulenevalt on siin väga vähe nii õppejõude, kes paketti *MyANN* oma töös kasutada võiks, kui ka õpilasi, kellel seda õppetöös vaja läheks.

Peatükis 3. "Olemasolev tarkvara" vaadeldi programme kogu maailma ulatuses, mitte Eesti piires. Kuna pakett *MyANN* ühegi vaadeldud programmiga täielikult ei kattu, siis võib tema järele nõudlust olla ka väljaspool Eestit.

Seega arvestades olukorda, kus sisulise kasutusväärtuse poolest ei oma pakett *MyANN* erilisi geograafilisi piiranguid ning potentsiaalsete kasutajate arv Eestis on väga väike, valmiski paketi esialgne versioon inglise keeles. Kui aga tekib vajadus eestikeelse variandi järele, on seda luua üsna lihtne – sisuline töö on juba tehtud, piisab vaid tõlkimisest.

5.4. Kasutusviisid

Paketti *MyANN* on võimalik kasutada mitmel erineval keerukustasemel sõltuvalt kasutaja huvist ja vajadustest. Järgnevalt on toodud peamised kasutusviisid, alustades kõige lihtsamast.

5.4.1. Esmane tutvumine tehisnärvivõrkudega

Üks paketi peamisi eesmärke on anda lühikese ajaga esmane ettekujutus närvivõrkude tööpõhimõtetest. Selle saavutamiseks peaks kasutaja kõigepealt läbi lehitsema HTML formaadis sissejuhatuse tehisnärvivõrkude valdkonda, tutvuma programmi *Visualizer* kasutusjuhisega ning selle siis käivitama. Seal näeb kasutaja võrgu ehitust ning saab sisendväärtusi muutes jälgida võrgu reaktsiooni neile muutustele. Paketiga on kaasas mitu näidisvõrku, mida sel viisil uurida. Et saada aimu ka närvivõrkude kasutus-võimalustest, võiks tutvuda programmi *LightChaser* kasutusjuhisega ning seejärel vastava programmiga natukene "mängida".

5.4.2. Põhjalikum tutvumine

Kui pärast esmase ettekujutuse saamist on kasutajal soov või vajadus jätkata, siis peaks ta põhjalikumalt läbi lugema kogu paketiga kaasasoleva materjali (välja arvatud teeki ja lähtekoodi kirjeldavad osad). Seejärel saab ta asuda programmiga *Trainer* ise uusi närvivõrke valmistama, nii uurimiseks *Visualizer*'i abil kui huvi korral ka katsetamiseks *LightChaser*'is.

5.4.3. Uurimine

Inimesed, kes on tehisnärvivõrkude valdkonnaga põhjalikumalt kursis, aga siiski soovivad saada nende tööst veelgi paremat ettekujutust, saavad põhimõtteliselt visualiseerida ja uurida ka enda poolt mõne muu tarkvara abil loodud võrke, kui need vastavad paketi *MyANN* poolt toetatud võrgutüübile – mitmekihilised ilma sisemise tagasisideta võrgud, mille neuronite aktiveerimisfunktsiooniks on sigmoidfunktsioon. Selleks tuleb need võrgud kirjutada *MyANN* konfiguratsioonifaili, mida saab teha kas käsitsi või võimalusel lasta kasutataval tarkvaral genereerida sobivas formaadis faile. Juhul, kui *MyANN* huvipakkuvat võrgutüüpi ei toeta, siis on võimalik paketti modifitseerida (vt. ülejärgmine punkt).

5.4.4. Tehisnärvivõrgu kasutamine isiklikes programmides

Paketis sisalduvat C++ teeki on küllaltki lihtne kasutada isiklikes programmides. Alustuseks peaks läbi lugema vastava lehekülje *MyANN* veebilehel, kus on toodud teegi kasutusnäide ning viidatud põhjalikuma informatsiooni asukohale.

5.4.5. Paketi modifitseerimine

Kui pakett ei vasta täielikult kasutaja nõuetele, kas siis tehnilistel või sisulistel põhjustel, on C++ keelt valdavatel kasutajatel võimalik programme vastavalt vajadusele modifitseerida, sest kogu pakett on avatud lähtekoodiga. Sel juhul peakski kõigepealt tutvuma lähtekoodiga, mis suhteliselt kergesti loetaval kujul on esitatud projekti veebilehel. Vajalikuks võib osutuda ka teegi *Allegro* (autor Shawn Hargreaves) dokumentatsiooniga tutvumine, sest seda kasutavad nii *Visualizer* kui *LightChaser*.

5.5. Kuhu edasi?

Hetkeseisuga ei ole *MyANN* veel täielikult valmis – viimistlemist vajavad nii kasutusjuhendid kui programmikood.

5.5.1. Kasutusjuhendid

Praeguse seisuga annab projekti võrgulehekülg programmide kasutamiseks küllaltki konkreetsed tehnilised juhised stiilis "see nupp teeb seda ja konfiguratsioonifailid peavad olema sellised", aga sisulist õpetust ja näiteid on veel liiga vähe. Lisada tuleks esmased käitumisjuhised analoogselt peatükis 5.4. "Kasutusviisid" tooduga. Samuti on vaja koostada põhjalikke näiteid võrkude valmistamisest, kus alustatakse eesmärgi püstitamisega ning seejärel see samm sammult täidetakse. Üks sellistest näidetest võiks olla programmi *LightChaser* jaoks uue juhtvõrgu loomine. Ideid kasutus-juhendite täiendamiseks hakkab ilmselt andma ka kasutajate tagasiside.

5.5.2. Programmikood

Lisas esitatu on paketi teine versioon, mis esimesega võrreldes on radikaalselt ümber disainitud (koodi disaini all on siinkohal mõeldud koodi struktuuri, jaotust klassideks, jaotust failideks jms.), kusjuures uus versioon sai kirjutatud täiesti "puhtalt lehelt", mitte olemasolevat koodi modifitseerides (välja arvatud mõned üksikud kohandatud lõigud). Kolmanda versiooni tegemine tõenäoliselt sellist suurt muutust enam kaasa ei too, küll aga on vaja parandada suur hulk väiksemaid probleeme. Näiteks tuleb tagada graafilist kasutajaliidest omavate programmide töö erinevate arvutite peal – praegune kood võib nii mõnegi masina kokku jooksutada, isegi kui kasutada ettenähtud operat-sioonisüsteemi (projekti tööversioon toetab *DOS*'i, töötab ka *Win95* ja *Win98* peal). Kuna peale C++ on kasutatud vaid Allegro teeki, mis on hea porditavusega, siis ilmselt hakkab *MyANN* toetama ka teisi operatsioonisüsteeme, esmajärjekorras uuemaid *Windows*'i versioone ning *Linux*'it. Kõikvõimalikke pisiprobleeme, mis kõrvaldamist ootavad, on veel palju: töö ühtluse tagamine erineva kiirusega arvutitel; arvestamine ka sülearvutite kasutamisega, mille klaviatuuril puudub *NumPad*; jms. Paljud vead selguvad arvatavasti alles siis, kui programmi laiemalt kasutama hakatakse.

Sisulise külje täiendamiseks on võimalusi palju. Näiteks visualiseerida võrgu õppimisprotsess. *Visualizer*'i eelnev versioon seda isegi võimaldas, kuid mitmetel põhjustel praegune versioon enam mitte. Muuhulgas olid probleemideks: kaalude muutumise äärmiselt aeglane, silmale peaaegu märkamatu tempo; näidisandmete pidevast kiirest vahetamisest (õppeprotsessi käigus) tulenevalt kas võrgu virvendamine või ainult ühe näidisandmete eksemplari visualiseerimine; samuti mõningad koodi disainiga seotud aspektid. Praeguses projektis on käsitletud ainult ühte tüüpi närvivõrke, seetõttu tuleks kaaluda ka teistsuguste neuronite ning võrguarhitektuuride realiseerimist. Sisuliste täienduste tegemine sõltub peamiselt autori motivatsioonist projekti jätkata. Võimatu pole ka arendustöö jätkamine teiste isikute poolt.

5.5.3. Õppeprogrammide valmistamine

Kui vaadelda antud projekti laiemalt kui ainult tehisnärvivõrkude seisukohast, siis võib jõuda järeldusele, et tudengite poolt õppeprogrammide kirjutamist peaks igati soodustama – sellest võidavad nii tegija ise kui teised õppurid. Et õppeprogrammide kirjutamine leviks laiemalt, oleks esiteks vajalik õppejõudude poolne toetus analoogsetele ideedele. Teiseks peaks tudengite jaoks valmistama vastavateemalisi abimaterjale, nii programmeerimise kui ka psühholoogiliste ja pedagoogiliste aspektide kohta, näiteks kuidas infot efektiivselt esitada. See oleks kindlasti paljude õppurite jaoks huvitav temaatika – psühholoogia ja disain on üsna populaarsed ained. Tekiks ka koostöövõimalusi humanitaarsemate ülikoolidega, näiteks loodava Tallinna Ülikooliga, kus psühholoogia, disain ja pedagoogika on esindatud oluliselt suuremas mahus kui TTÜ õppekavades.

6. Kokkuvõte

Seoses käesoleva töö veidi eripärase formaadiga tuleb projekti põhiosa alles pärast seda kokkuvõtet. Aga kui siinkohal vaadelda juba tööd tervikuna, siis võib nentida, et töö eesmärk – valmistada õppeotstarbeline tarkvarapakett lihtsamate tehisnärvivõrkudega tutvumiseks ja ka nende kasutamiseks väljaspool paketti, teistes programmides – sai edukalt täidetud. Valminud pakett on atraktiivne ja omanäoline ning loodetavasti leiab rakendust nii välismaal kui ka Eestis. Lisaks leidub palju soodsaid võimalusi edasiliikumiseks, seda nii projekti *MyANN* tehnilise ja sisulise täiendamise kui õppeprogrammide kirjutamise propageerimise ning sel teel üldise õpikeskkonna parandamise näol.

7. Kasutatud kirjandus

1. Neural Networks.
Christos Stergiou, Dimitrios Siganos.
http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html

2. CSE5301 Neural Networks (Neuro-Fuzzy Computing), Lecture Notes.
A. P. Papliński.
http://www.csse.monash.edu.au/~app/CSC437nn/

3. Neural Network Applications in Control.
Edited by G. W. Irwin, K. Warwick, K. J. Hunt.
IEE Control Engineering series 53, 1995.

4. Neural Networks – a Comprehensive Foundation.
Simon Haykin.
Prentice Hall International.

5. Neural Network Toolbox For Use with MATLAB, User's Guide, Version 4.
Howard Demuth, Mark Beale.
http://www.mathworks.com/access/helpdesk/help/pdf_doc/nnet/nnet.pdf

6. Qnet 2000.
Vesta Services, Inc.
http://www.qnetv2k.com/

7. Web Applets for Interactive Tutorials on Artificial Neural Learning.
Fred Corbett.
http://home.cc.umanitoba.ca/~umcorbe9/anns.html

8. Neural networks visualization applets.
Igor Fischer.
http://www-ra.informatik.uni-tuebingen.de/mitarb/fischer/applets.html

9. NeuroSolutions.
NeuroDimension, Inc.
http://www.neurosolutions.com/products/ns/

10. Neural Network Toolbox – Demos.
The MathWorks, Inc.
http://www.mathworks.com/products/neuralnet/demos.jsp

11. Stuttgart Neural Network Simulator.
University of Stuttgart & University of Tübingen.
http://www-ra.informatik.uni-tuebingen.de/SNNS/

12. Java Neural Network Simulator.
University of Tübingen.
http://www-ra.informatik.uni-tuebingen.de/software/JavaNNS/welcome.html

13. The PDP++ Software.
Randall C. O'Reilly, Chadley K. Dawson, James L. McClelland.
http://www.cnbc.cmu.edu/Resources/PDP++//PDP++.html

Järgnevatele allikatele ei olnud otsest põhjust töö tekstist viidata, aga kuna ka nendest oli projekti valmimisel palju abi, siis on nad siinkohal ära märgitud.

14. The Backprop Algorithm.
Donald R. Tveter.
http://www.dontveter.com/bpr/public2.html

15. An introduction to neural nets.
André LaMothe.
http://www.xgames3d.com/articles/netware/areadnet.htm

16. Essays on AI.
http://www.generation5.org/essays.shtml

17. Thinking in C++, 2nd ed. Volume 1.
Thinking in C++, 2nd ed., Volume 2, Revision 2.
Bruce Eckel.
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html

18. Allegro manual.
Shawn Hargreaves.
http://alleg.sourceforge.net/onlinedocs/en/index.html

8. Lisa: õppetarkvara pakett MyANN

Töö lisana on toodud käesoleva töö peamine osa – pakett *MyANN*. Lisa on parema loetavuse huvides jaotatud kaheks: projekti võrgulehekülg ning programmide lähtekood. Tegelikkuses on ka lähtekoodi sisaldavad HTML failid üks osa veebilehest.

Kuna enamus HTML formaadis faile (eriti need, mis sisaldavad lähtekoodi), olid *MS Word*'i jaoks liiga keerulised, siis on nad välja trükitud veebilehitseja abil. Seetõttu ei jätku lisas töö lehekülgede ühtne numeratsioon.

Projekti võrgulehekülg

NB! Järgnev materjal on spetsiaalselt disainitud lugemiseks arvutist, seetõttu ei näe ta paberi peal eriti hea välja!

NB! Võrgulehekülg ei ole veel täielikult valmis ja saab oma lõpliku viimistluse hiljem.

Project MyANN

General information

About this project
Introduction to artificial neural networks

MyANN software

Applications
Library
Source code

Download

More about ANNs

A few links

Project information

Objective:

To create an educative software package about artificial neural networks (ANNs).

Keywords:

attractive, enjoyable, simple, educative, open source, free

With MyANN software you can:

- Create multilayered feedforward ANNs.
- Train these ANNs with training patterns using backpropagation.
- Visualize their inner activities during work.
- Play with a light-sensitive "creature" controlled by ANN.
- Create your own software using MyANN library (C++ knowledge required).

Potential users:

- Everybody, who wants to learn more about artificial neural networks.
- Students looking for a quick introduction to ANNs (because tomorrow is an examination :)
- Teachers, for demonstrations in lectures.
- Software writers, who want to use neural networks in their products (e.g. in student projects, games, computer animations, automatics. NB! This product has no guarantee whatsoever, do NOT use it in any critical application!).

Motivation:

I'm a student in Tallinn Technical University and I constantly feel the lack of educational software products that could be described with keywords like those a few paragraphs above. It's not only about our university 'cause it's very difficult to find anything cool AND educative AND free from Internet as well. So I just tried to make something myself. Why artificial neural networks? Because one of my interests is Artificial Life and I wanted to learn a bit more about ANNs (but no, I

don't expect much of a liveliness from them :) and what could be a better way to learn about something than creating it yourself!

Taivo Lints, Estonia
taivo@vkg.werro.ee

Introduction to artificial neural networks (ANNs)

The world today

Maybe you have heard here and there about some technological wonders based on artificial neural networks -- about intelligent robots, smart weapons or even stock trading advisory systems? What's going on? Are we really living at the turning point of history where machines are taking over and mankind will be doomed? To be honest, it is not completely impossible, but artificial neural networks in their current state of evolution are hardly the right target for accusations of that kind. The most complex neural networks of our time are not working in the dark corners of high-tech labs. No, they are in our heads.

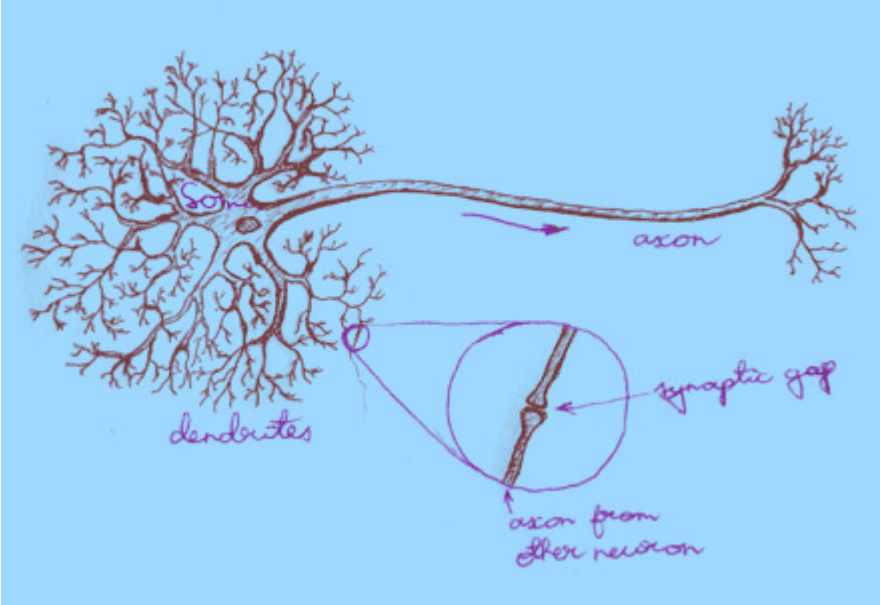
Start thinking!

How much do we know about the inner mechanisms of our brain? How often do we think about how exactly we think about the things we think about? The fact that this sentence looks a bit weird should be considered alarming as it shows the lack of discussions on the topic. Think about it!

Biology

Human brain is an extremely complex structure. Its workings can be discussed on a huge number of different levels -- starting from the low end of atoms and molecules and going up to the social interactions of global scale. All these levels are remarkably interesting, but as our current concern is artificial neural networks, we will go to the cellular level.

The main component of our biological neural net is a cell called neuron. Many different types of neurons are present in brain, but their general structure is mostly the same:



On the drawing you can see the main parts of a neural cell:

- *Dendrites* collect incoming signals. They have a lot of branches and their surface is quite irregular. *Dendrites* got their name because of a tree-like structure.

- *Soma* is the body of the neural cell. It processes incoming signals (and, of course, performs all the activities necessary for a cell to survive).

- *Axon* transmits signals to the *dendrites* of other neurons. *Axon* has less branches, smoother surface and greater length than *dendrites*.

- *Synapses* are connections between neurons. They are actually tiny gaps that allow signals to jump from an *axon* to a *dendrite*.

The main principle of neuron's work is already visible from the description of its structure -- neuron collects signals, processes them and feeds into other neurons. In reality, however, everything is very complex. First of all, signals are far from simplicity. They are passed via electrochemical processes based on different ions (mainly sodium (Na), potassium (K) and chloride (Cl)). Even worse, the information is coded not into the amplitude of signals, but instead into frequency of these electrochemical pulsations (although that doesn't automatically mean we could just forget about the amplitude). Secondly, although the scales are microscopic, the numbers are macroscopic. Each neuron has up to

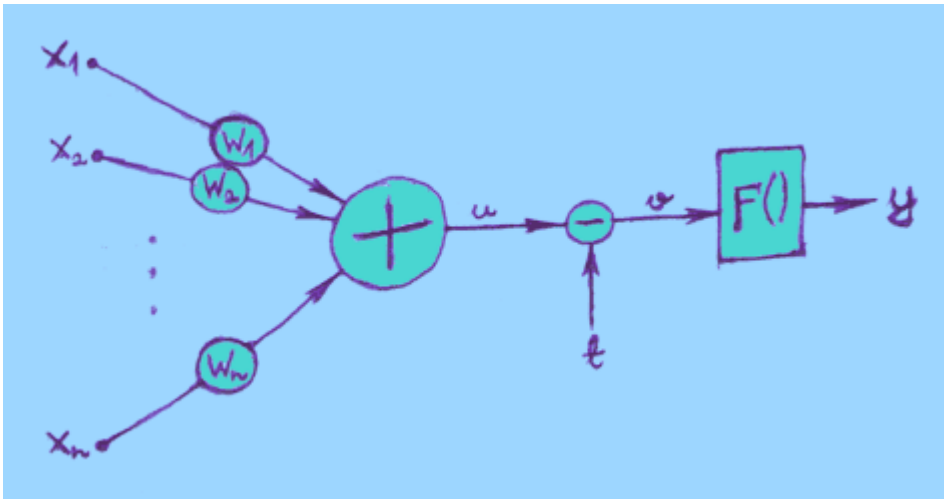
10,000 input connections via *dendrites*, all passing signals into the *soma*. And the number of such neurons in a brain is estimated to be somewhere between 10 and 100 billion (thats 10,000,000,000 and 100,000,000,000)! Finally, all this unimaginably huge system is in constant change -- new connections are being formed, some cells stop functioning due to aging, etc.

Technology

Todays technology is nowhere near the complexity described in previous paragraph, neither in numbers nor in principles. Although artificial neural networks DO have some resemblance to their biological counterparts, they are still tiny and simple structures. Their simplicity, however, does NOT mean they are fully understood and easily described by mathematical tools.

Artificial neuron

There are a great many different models of neurons and neural networks, lots of them being successfully used in various applications. Still, most of them have a lot in common and differ only in details. The most widespread structure of an artificial neuron is:



If it looks a bit messy and scary first, then don't worry, actually it is very simple.

- On the left there are inputs x_1, x_2, ..., x_n.
- Each input goes through a connection, wich has a certain weight w, and is multiplied by that weight: x * w.
- All these multiplied values are then added together:
  $u = x_1*w_1 + x_2*w_2 + ... + x_n*w_n$

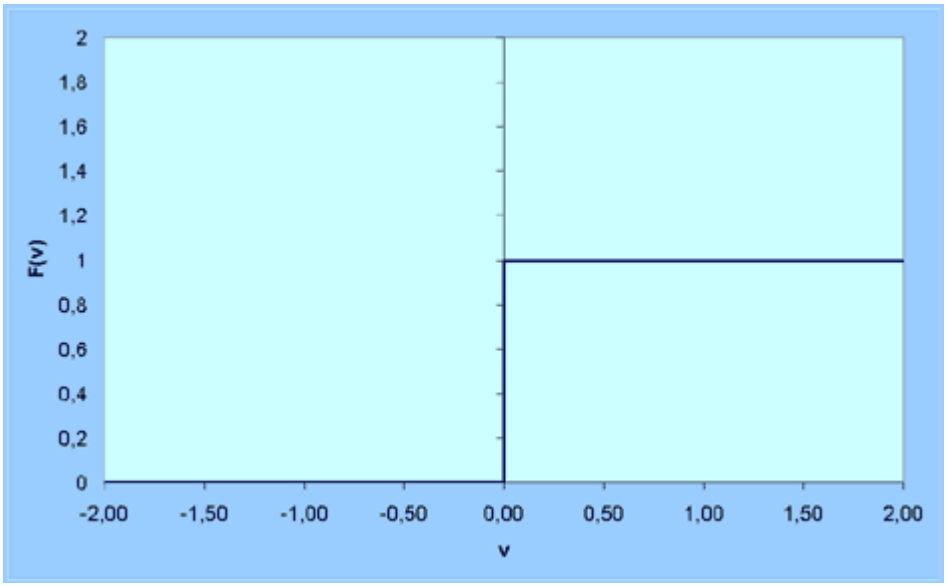
- This sum u is then adjusted with treshold t:
  v = u - t

- And finally a so called "activation function" F() is applied to the adjusted sum, giving us the output of neuron:
  y = F(v)

In general all these parameters and variables (inputs, weights, treshold) can be both positive or negative (and zero, too). Negative treshold is often called with a different name: bias.

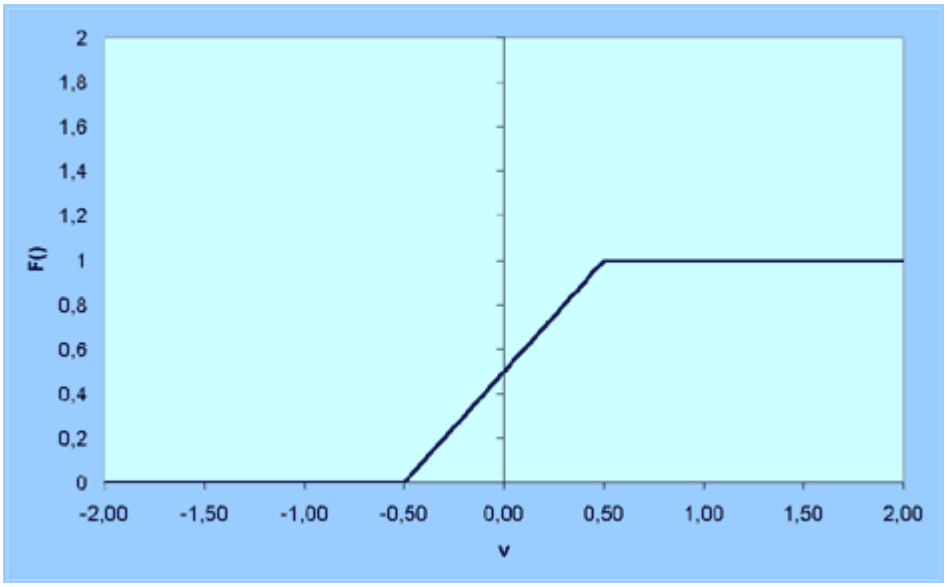
There are three basic types of activation function F():

1. Treshold function (not to confuse with treshold parameter t):



$F(v) = 0$, if $v < 0$
$F(v) = 1$, if $v = 0$ or $v > 0$

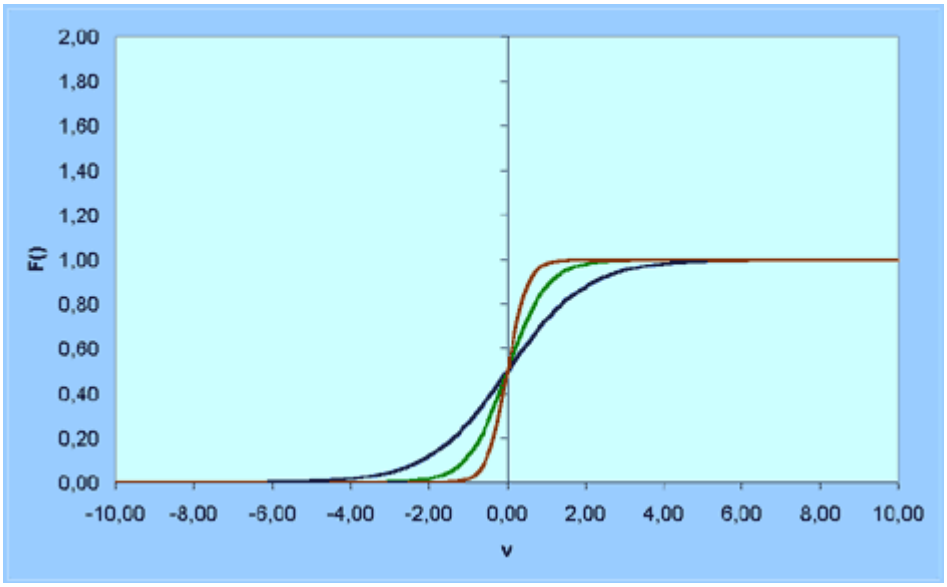
2. Piecewise-Linear function:



F(v) = 0, if v < -0.5 or v = -0.5
F(v) = v + 0.5, if -0.5 < v < 0.5
F(v) = 1, if v = 0.5 or v > 0.5

This is just an example, it may have other values as well.

3. Sigmoid function:



F(v) = 1 / (1 + exp(-av)), where a is the slope parameter. Three examples with different a are given on a chart.

The sigmoid function above is known as "logistic function", but there are other forms of sigmoid function, too. For example:
F(v) = tanh(v / 2) = (1 - exp(-v)) / (1 + exp(-v))

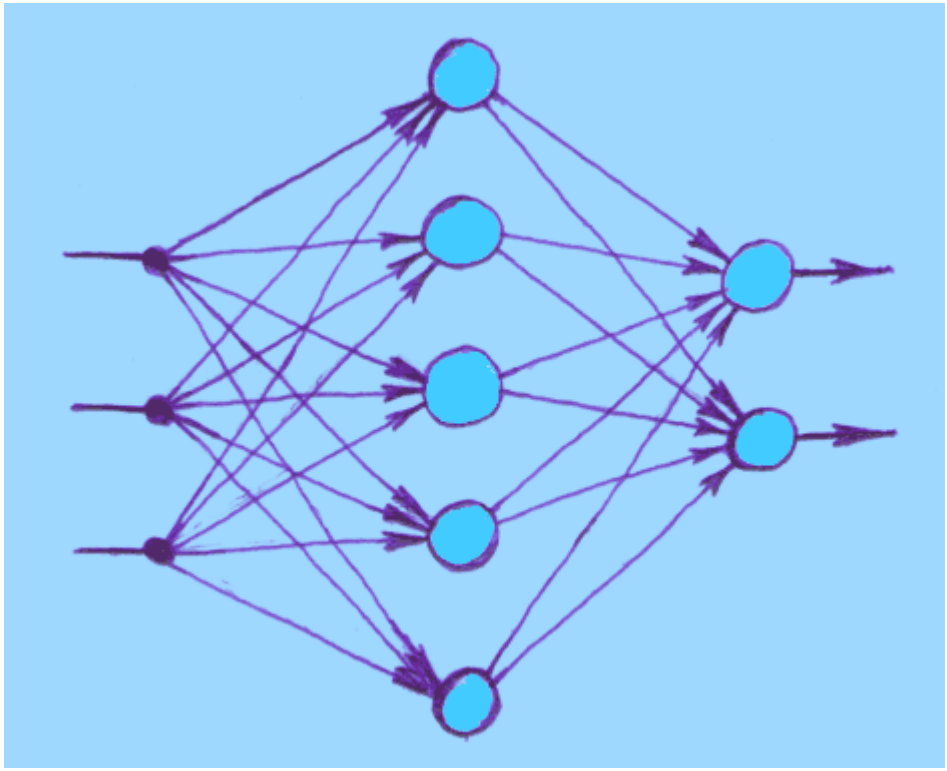
Sigmoid functions are the most common activation

functions used in artificial neural networks.

Artificial neural network

As already mentioned before, there is a plethora of different network models, and the level of similarity is considerably lower than it was in artificial neurons. We can compare the neuron with a building block -- by using standard blocks it is possible to build a great number of houses in different styles. Start customizing your blocks and the possibilities jump to infinity.

The most common class of ANNs is multilayer feedforward networks:



It has a layered structure. Input nodes, whose only purpose is to distribute the input values into network, form an input layer (on the left on picture). All other nodes are artificial neurons, which also form layers: the output layer (on the right), and between input and output layers is one or more "hidden" layers. "Feedforward" implies that signals move only towards outputs -- no feedback loops are present. Network on the picture is "fully connected" in the sense that every neuron in each layer is connected to every other node in its neighbouring layers. If some connections were missing, it would be a "partially connected" network.

Training a network

If you have carefully followed the story so far, you probably already have some ideas about how ANN will work: it is given a bunch of input values, it processes them and gives back some output values. Clean and simple :) Except one problem -- we would like the ANN to have a REASONABLE reaction to given inputs, not just any random reaction! That's were training comes in.

As there are different architectures of networks, there are also appropriate training algorithms for each of them. From network's point of view it would make more sense to call them learning algorithms, especially because some of them do not require any teacher (unsupervised learning). However, in case of multilayer feedforward networks a supervised learning procedure is used most often, usually a "backpropagation" algorithm. Its principle is:

- Input values are fed into network.
- Network's reaction (output values) is read out and compared to the desired reaction.
- Error values are calculated out of the difference between desired and actual outputs using some error function.
- The weights of connections owned by output layer are adjusted according to the error values.
- Error values are then BACKpropagated to the hidden layer next to output layer, and its weights are adjusted.
- Backpropagation and weight adjustment is repeated until input layer is reached.
- The whole procedure is repeated until error is acceptably small.

The point of using ANNs is that although we train them only with a few training patterns (which consist of input values and desired output values for these inputs), the ANN can also correspond reasonably to the inputs that are similar to the learned ones, but not exactly the same. This is called "function approximation" and is quite useful in many applications.

The details of training procedure usually contain heavy math and are not easily grasped. Following is just a short overview about HOW the backpropagation can be done. Answering the question "WHY it can be done that way" would definitely be more reasonable and enlightening, but I'm currently not up to the task of doing it, especially in a user friendly way :) Anyway, if you get an intense feeling of repulsion when looking at the following section, then feel free to skip it and scroll down to the next paragraph.

First of all, the calculation of output values is done using the

neuron model descibed above (input values are set into the input layer; then all neurons in first hidden layer are updated according to the neuron model; then neurons in second hidden layer; ... ; until output layer is reached).

Now we have output values for this given set of input values, but most likely they are not what we want them to be. So we can calculate error value for each neuron in the output layer.

error_value = desired_output - actual_output

Then an error signal is calculated for each of these neurons:

error_signal = error_value * F ' (v)

where v is the adjusted sum in neuron (see the neuron model above) and F ' (v) is the derivative of activation function F() in point v.

And then all weights owned by these neurons are updated:

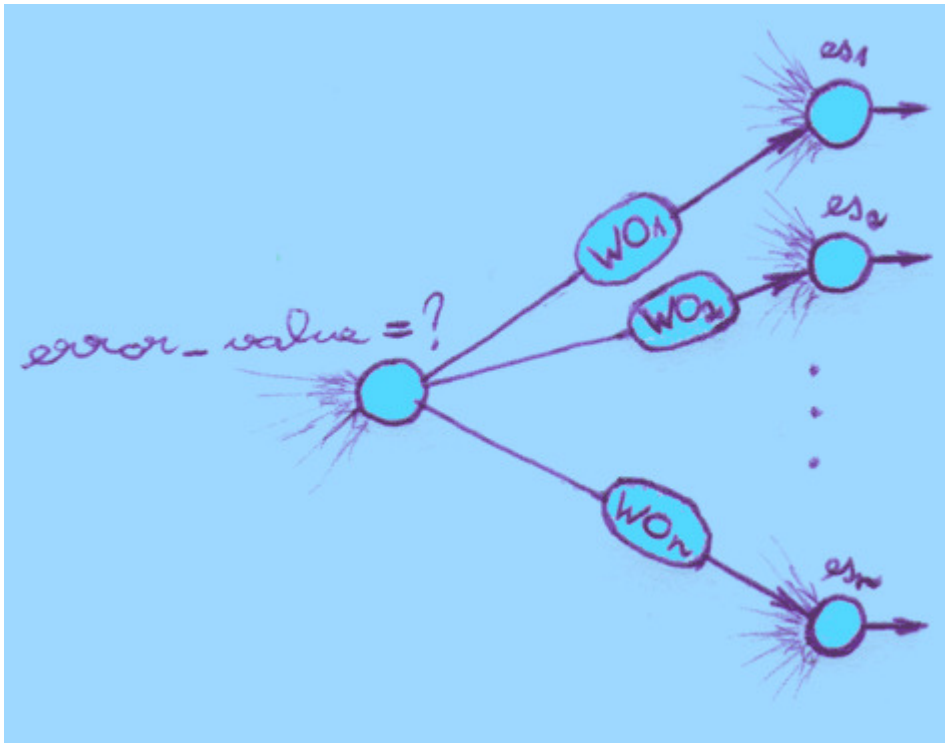
$w_1 = w_1 + \text{learning\_rate} * \text{error\_signal} * x_1$

$w_2 = w_2 + \text{learning\_rate} * \text{error\_signal} * x_2$

...

where x_1, x_2, ... are inputs into this neuron from neurons located in previous layer (once again, see the neuron model above) and learning_rate is a parameter affecting the speed of learning (it should be noted that it is also a "forgetting speed" in some sense -- if learning_rate is very high, then network quickly learns new training patterns, but on the cost of forgetting others; plus additional danger that ANN becomes so instable it can't learn any single pattern at all, like a hyperactive child.).

Now the weights of output layer are updated, but what about other layers? We can't repeat exactly the same procedure because we don't have desired output values for hidden layers. The solution is BACKpropagation of the error from output layer into hidden layers.



To get the error_value for a neuron that is NOT in output layer, the following formula can be used:

$$error\_value = wo_1 * es_1 + wo_2 * es_2 + ... + wo_n * es_n$$

where "wo" is the weight of a connection between current neuron and a neuron in following layer (see the picture above), and "es" is the error_signal of that other neuron.

Formulas for calculating the error_signal and new weights are here exactly the same as they were for output neurons.

This backpropagation of error and weights updating is done all the way through network till input layer is reached. Then the whole process is repeated until error is small enough. If there is more than one training pattern, then they are used by turns: inputs of the first pattern are fed into network, error is calculated, weights are adjusted; inputs of the second pattern are fed into network, error is calculated, weights are adjusted; ... ; then again the first pattern, second pattern, ...

It is quite likely that you have at least one question after reading previous algorithm: what exactly is the derivative of F(v) and why is it used anyway? It should be clear that F ' (v) is not the same for all activation functions F(v). One of the most common activation functions is F(v) = 1 / (1 + exp(-av)) and it's derivative is mostly given in a form F ' (v) = a * y * (1 - y), where y = F(v) (y is the output of the neuron, see neuron model above). It may be somewhat

surprising first, but if you take a few minutes and try it on a paper then it makes more sense. First you get a big messy derivative, but some replacements should reduce it down to given short form which is much easier to use than the big messy one. Why is derivative needed, anyway? Because backpropagation is a gradient descent algorithm, meaning it tries to move the system towards the deepest descent on error surface for getting the fastest reduction of error. But finding the deepest descent of a surface is, naturally, done by using gradient, which in turn is found by using derivatives. As simple as that :) Feel free to search for additional information from Internet or books.

What can be done with ANNs?

As we have now seen, artificial neural networks are very far from the complexity of their biological counterparts. There is also an additional problem -- in brain all neurons are working all the time (in parallel), but our computers work in sequential fashion, meaning they have to update artificial neurons one by one which is an awfully time-consuming process. Special hardware can be built for ANNs, but often it would be too expensive. So a question rises: do we need ANNs at all? Where?

The answer is: although ANNs probably are NOT the hoped breakthrough in Artificial Life research, they do have turned out to be very useful in many fields of technology. ANNs have been successfully used in various signal processing applications from sonar and radar classificators to word recognizers. They have a lot of use in industry: process control, quality prediction and inspection systems, process and machine diagnosis, etc. In robotics ANNs are used for vision systems and for controlling the robot's movements. Business applications include price prediction programs for stocks and currencies. ANNs can also be used in computer games and animations for various purposes like controlling the character movement or doing some other necessary decisions.

Summary

This was just a short introduction to the field of ANNs. Only one neuron model, one network architecture and one training algorithm were described and even this was done without much details. Currently I do NOT have any plans to write more essays on standard ANNs, so this page is unlikely to be filled with any further information. However, there is a great variety of materials available on Internet and in books, for beginners and for professionals. If you DO have interest in the topic, please take some time and look for more information. A few links are

provided in Links section, accessible from the front page. But before clicking away towards the shining mountains of wisdom, please check out some educational software based on the same models described in this essay: see the Software section on the front page.

Back | *Home*

MyANN applications

MyANN applications are free simple educational programs about multilayer feedforward neural networks. If you don't know anything about artificial neural networks (ANNs), then it would be a good idea to read Introduction to ANNs first.

MyANN applications are:

> **Trainer**: creates a multilayer feedforward network and trains it. Saves the result into a file that is human readable and can also be used as a configuration file for the other two applications. Read Trainer help for more information.
>
> **Visualizer**: visualizes created network. Allows user to change network inputs and see what happens. Read Visualizer help for more information.
>
> **LightChaser**: an example about practical use of ANN. Simulates a light-sensitive "creature" whose movement is controlled by ANN. Read LightChaser help for more information.

If you have downloaded the package and don't know where to start, then start Visualizer first.

Trainer help

Trainer is a simple tool for generating a multilayer feedforward network and training it with backpropagation algorithm. It uses two configuration files and produces one output file.

Using Trainer is very easy:

If you don't know much about computers and you are currently using Windows, then:

- Edit configuration files with Notepad (just double-click on *configuration.txt* or *trainingfile.txt*). Don't forget to save your changes before moving on.
- Start *trainer.exe* by double-clicking on it.
- Read results with Notepad from *config_generated.txt*.

Otherwise:

If you call *trainer.exe* without any command line parameters, it will search for *configuration.txt* and *trainingfile.txt*, and it will produce *config_generated.txt*. If you want to use other filenames, then call *trainer.exe configfile trainingfile outputfile*, where you replace each filename with the names of your own files (outputfile parameter is optional).

Note that training time is different every time you start the trainer. That's because connection weights are initialized randomly every time. So when training was not successful in the first attempt, it may be successful next time.

Currently it is not possible to stop the training procedure at your will. Well, at least not in a clean way -- Ctrl+Break or "close the window" button (in Windows) will usually work.

The configuration files are pretty much self-explanatory. File *configuration.txt* looks like this:

```
In this file you can describe your network.

*********************
* Network structure *
*********************
```

Without this section your NeuralNetwork object will have
NO neurons (unless you modify source code) and is therefore
quite useless!

Currently it is possible to create only a fully connected
multilayer feedforward network. To do that, just answer
some easy questions below.

First layer in network is input layer. It consists of simple
buffer nodes that do NOT compute anything, but only hold
input values. How many input nodes your network must have?

```
  input_layer: 2
```

Other layers consist of computing neurons. How many nodes
should be in computational layers starting from the input
layer side of the network? For adding a layer just add a
line with the keyword "comp_layer:". The last computational
layer will also act as the output layer. Layers without
any neurons will be ignored.

```
  comp_layer: 4
  comp_layer: 4
  comp_layer: 1
```

****
* Neuron parameters *
****

Work of an artificial neural network depends on neuron parameters.
If you have no idea what they should be, then you may set them
to default values (which are given in help texts before
each parameter). Or just leave them empty (delete the
numbers after each parameter identifier), then they will be
automatically set to default values. Actually it is even allowed
to delete this section "Neuron parameters" completely from this
file and leave only the "Network structure" section. Then
everything will be set to default as well. But make a backup
of this file first ;)

Currently it is possible to use only one type of neuron.
Its activation function is sigmoidal (logistic function
1 / (1 + exp(-av)) where "a" is slope parameter and "v"
is internal activity level).

- Parameter: treshold. Type: double. Default value: 0 - -
Treshold (or bias) value is used for lowering or increasing
the sum (internal activity level) before applying the activation
function.

```
  treshold: 0
```

- Parameter: slope_parameter. Type: double. Default value: 0.5 - -
It is the slope parameter of the sigmoid function. When slope
parameter approaches infinity, sigmoid function becomes simply a
treshold function (do not confuse with treshold parameter, which was
described above).

```
  slope_parameter: 0.5
```

****
* Additional info *
****

It is also possible in this file to set max_weight parameter
(normally you should do it in training configuration file) and
all weights of connections. However, in normal circumstances
you shouldn't do it here. If you want to set all weights manually,
then first generate a network with needed structure, then edit this
generated file according to your needs. You can then use the
generated and edited file as a config file for creating

```
a new network object.

Note: you will see that generated configuration file contains
max_weight parameter. In most cases it is not used during the
normal work of neural network, but some rare applications may
use it (like my neural network visualizer). Therefore when you
change the weights manually, consider if it is also necessary
to change max_weight value.
```

Do not forget to add space between any keywords and numbers after them! For example `comp_layer: 5` is correct, but `comp_layer:5` is not.

File *trainingfile.txt* looks like this:

```
Typical backpropagation algorithm is used for training.
In this file you can configure its parameters.

**********************
* Training patterns *
**********************

If you are going to train your network, you MUST first
define all training patterns. The easiest way is to describe
them is to do it in this file.

Give as much training patterns as necessary, using format:
"pattern: inputs -> desired outputs". Note that number of
inputs and outputs in patterns must match your network
structure given in configuration file.

  pattern: 0 0 -> 0
  pattern: 0 1 -> 1
  pattern: 1 0 -> 1
  pattern: 1 1 -> 0


*******************************
* Training related parameters *
*******************************

Process of training depends on several parameters. If you
have no idea what they should be, then you may set them
to default values (which are given in help texts before
each parameter). Or just leave them empty (delete the
numbers after each parameter identifier), then they will be
automatically set to default values. Actually it is even allowed
to delete this section "Training related parameters" completely
from this file and leave only the "Training patterns" section.
Then everything will be set to default as well. But make a
backup of this file first ;)


- Parameter: error_limit. Type: double. Default value: 0.1 - -
Error limit is used for evaluating the success of training.
If all outputs are in a range from (desired_output - error_limit)
to (desired_output + error_limit), then training is completed.

  error_limit: 0.1

- Parameter: max_iterations. Type: int. Default value: 20000 - -
When this number of iterations is reached, training will be
stopped and is failed (i.e. outputs were still outside the
given error limits).

  max_iterations: 20000

- Parameter: learning_rate. Type: double. Default value: 5 - -
```

```
How fast should network learn? Too small values make
learning too slow. Too big may halt the learning process.

  learning_rate: 5

- Parameter: max_weight. Type: double. Default value: 10 - -
Maximal allowed weight for connections. Weights will be
in range -max_weight..+max_weight.

  max_weight: 10
```

Be reasonable with your patterns. Not everything is possible to achieve. For example, if you require: `0 0 -> 0` and `0 0 -> 1` at the same time, it is just impossible.

Patterns can also contain fractions, like `0.5 0.8 -> 0.7`.

The result of training, *config_generated.txt*, looks like this:

```
input_layer: 2

comp_layer: 4
  weights:  -9.9055592274468722  -8.6426648049884633
  weights:   4.2954430717673731 -10.0000000000000000
  weights:  -5.2333839447251931   8.6645811142879534
  weights: -10.0000000000000000   5.4667244897148892

comp_layer: 4
  weights:   7.3365444984805759   7.4310744304231662  -0.2002578717261061   7.9767345222931514
  weights:   9.9999903345722743  -8.4489966768275337  10.0000000000000000   6.1203844957474400
  weights:   9.9999109055190250   6.5702854017823036   4.1924238186179448  -9.9997423320067149
  weights:  -5.0713789883692932   5.2303987739264022   3.8133885383571613   1.0846559577464676

comp_layer: 1
  weights:   9.9944199304957611  -9.9860383437970999  -9.9937227905149673   6.2474381765918432

treshold: 0.0000000000000000
slope_parameter: 0.5000000000000000
max_weight: 10.0000000000000000
```

It follows the same format as *configuration.txt*, so it can be used as a configuration file as well. There is one `weights:` keyword for each neuron. First number after keyword is the weight of a connection between this neuron and the first neuron in previous layer; second number is the weight of a connection between this neuron and the second neuron in previous layer; etc.

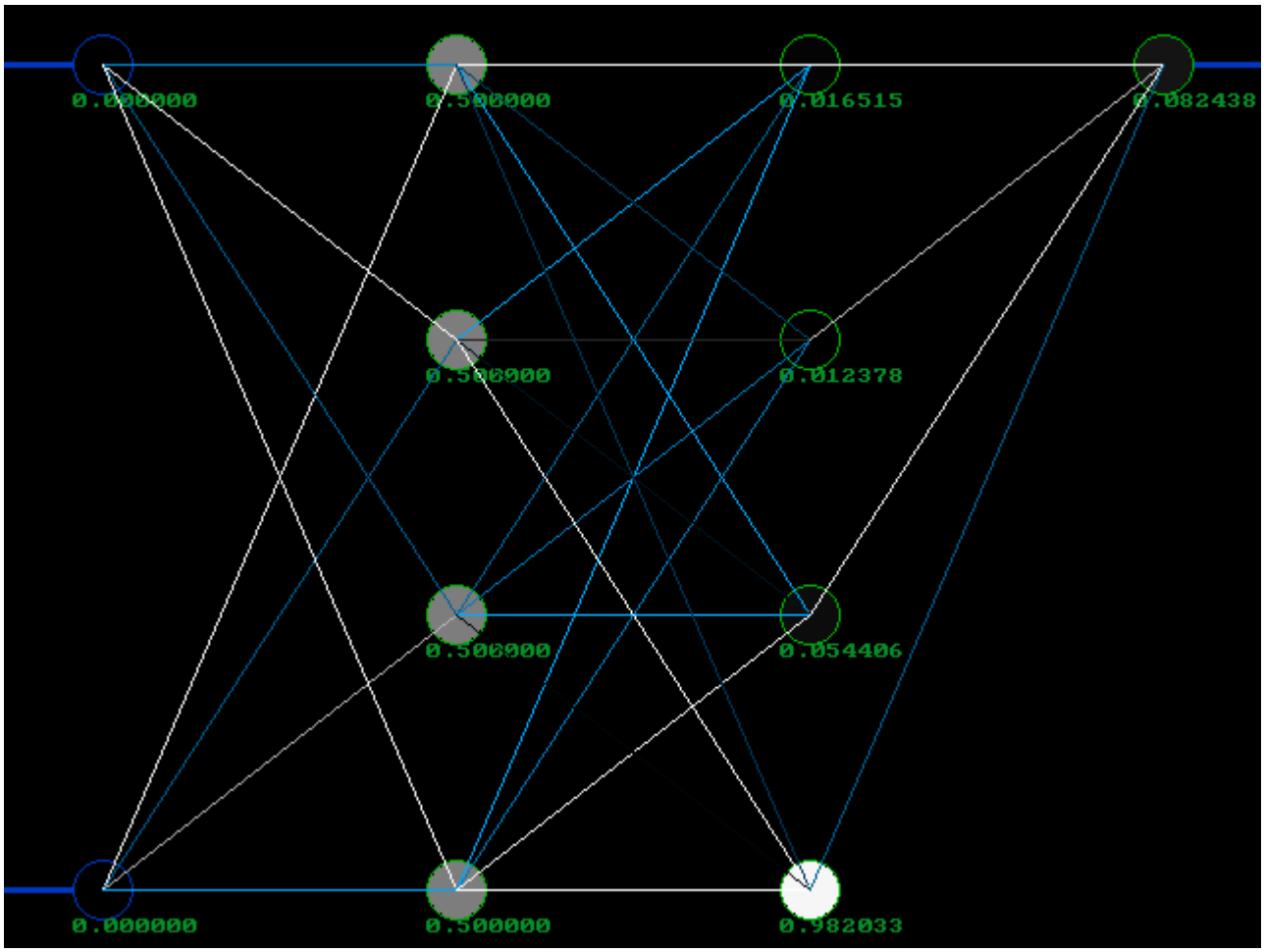
Back | *Home*

Visualizer help

Visualizer allows you to see what happens inside ANN when it is working.

When you start Visualizer, it tries to read network information from *config_generated.txt*. If this file is missing or incorrect, it asks to load some other configuration file. You don't need to exit Visualizer to do that, just press "L" to open loading dialog.

After successful start you should see a screen like that (although network may be different, depending on your configuration file):



- Blue bold lines on the left edge of picture are input pins of your network, similar lines on the right edge are output pins.
- Blue circles on the input pins are input nodes, which form an input layer.

Their purpose is just to distribute input values into network.

- Green circles are neurons. They work exactly as described in Introduction to ANNs (their activation function is "logistic function"). You can see that neurons form layers, too. The first one after input layer is called the first "hidden" layer, second is the second "hidden" layer and so on, except the last one, which is called output layer (on this picture it has only one neuron).

  Circles are filled with a color that is either black or white or some sort of grey. It indicates the output value of that node. Black is 0, white is 1. This output value is also written below each neuron.

- Nodes are connected with lines. They represent connections between neurons. Color of the line indicates the weight of connection. Black (not visible) is 0, white is the maximum allowed positive weight (default is +10) and brightest blue is the most negative allowed value (default is -10). Greys are, naturally, weights between 0 and +max_weight, and dark blues between 0 and -max_weight.

Using Visualizer is simple:

- To change input value on the first input pin, hold down key "1" on your keyboard (but not on NumPad section) and move your mouse up and down (pressing mouse buttons is not necessary). To change the second input value, hold down key "2" and move mouse up and down. This goes up to the key "0", which corresponds to the input number 10 (in case your network has so many inputs). If you have more than 10 inputs, then currently you can't modify the last ones (but if it is really necessary, then it can be easily fixed in source code if you have C++ programming experience and Allegro library, or if you write me and ask to fix it).

  If you want to change all inputs to 0, then move your mouse down and then press "1", then "2" and so on. Same applies to any other input value.

  You can also change several inputs simultaneously. Just hold down all the number keys you want and move mouse up and down. However, due to the keyboard hardware not all key combinations work.

- To load another ANN, press "L" and choose another configuration file. There are several examples already included with this software package (text files that start with *conf*), and you can make new ones with Trainer, or by editing files manually. All example files include a short description (you can read it with Notepad, or whatever text file viewer).

If the loaded configuration file does not describe weights of connections, then they are assigned randomly between values -max_weight and +max_weight.

If the network in configuration file is very big, then you have to wait a little bit while it is being loaded.

- To get a short help about keys in Visualizer, press "F1".

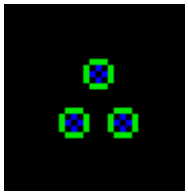
- To save a screenshot into bitmap file *screenshot.bmp*, press "F12". However, that does not work in menus (in help and in loading menu).

- And finally, to exit the program, press "Esc". If your network is very big and computer has a hard time working on it, you have to hold down Escape until you see a text "ESC pressed, will QUIT".

Back | *Home*

LightChaser help

LightChaser is a program that simulates a light-sensitive "creature" whose movement is controlled by ANN.



This is what our creature looks like (after being magnified). It has three light sensors. Each sensor reads the light intensity exatly from the center of sensor (the black dot surrounded by a blue circle). So the creature doesn't actually "see" like we do, it just senses the intensity of light on it. It is similar to the situation when you close your eyes: you don't see anything, but you sense the intensity of light on your eyes. For example when you are in a dark room with your eyes closed and somebody points a flashlight at your face, you can sense it. The creature has three sensors, so it can figure out if the light intesity changes in some direction. For example if the upper sensor gives higher light intensity readings than the other two sensors, then most probably the light intensity rises towards "up" direction.

Creature's movement is controlled by ANN. This ANN reads in sensor readings (which should be in range from 0 to 1) and returns movement information. Currently it works this way:

- The first input to ANN comes from the lower left sensor.
- The second input comes from the upper sensor.
- The third input comes from the lower right sensor.

- The first output of ANN says how fast the creature should move to the left direction.
- The second output says how fast to move right.
- The third output says how fast to move up.
- The fourth output says how fast to move down.

The description of outputs may look a bit weird at first. The catch is that our neurons can only output a value from 0 to 1, but not any negative values. So if we want the creature to be able to move both to the left and to the right, we can calculate the horizontal speed this way:

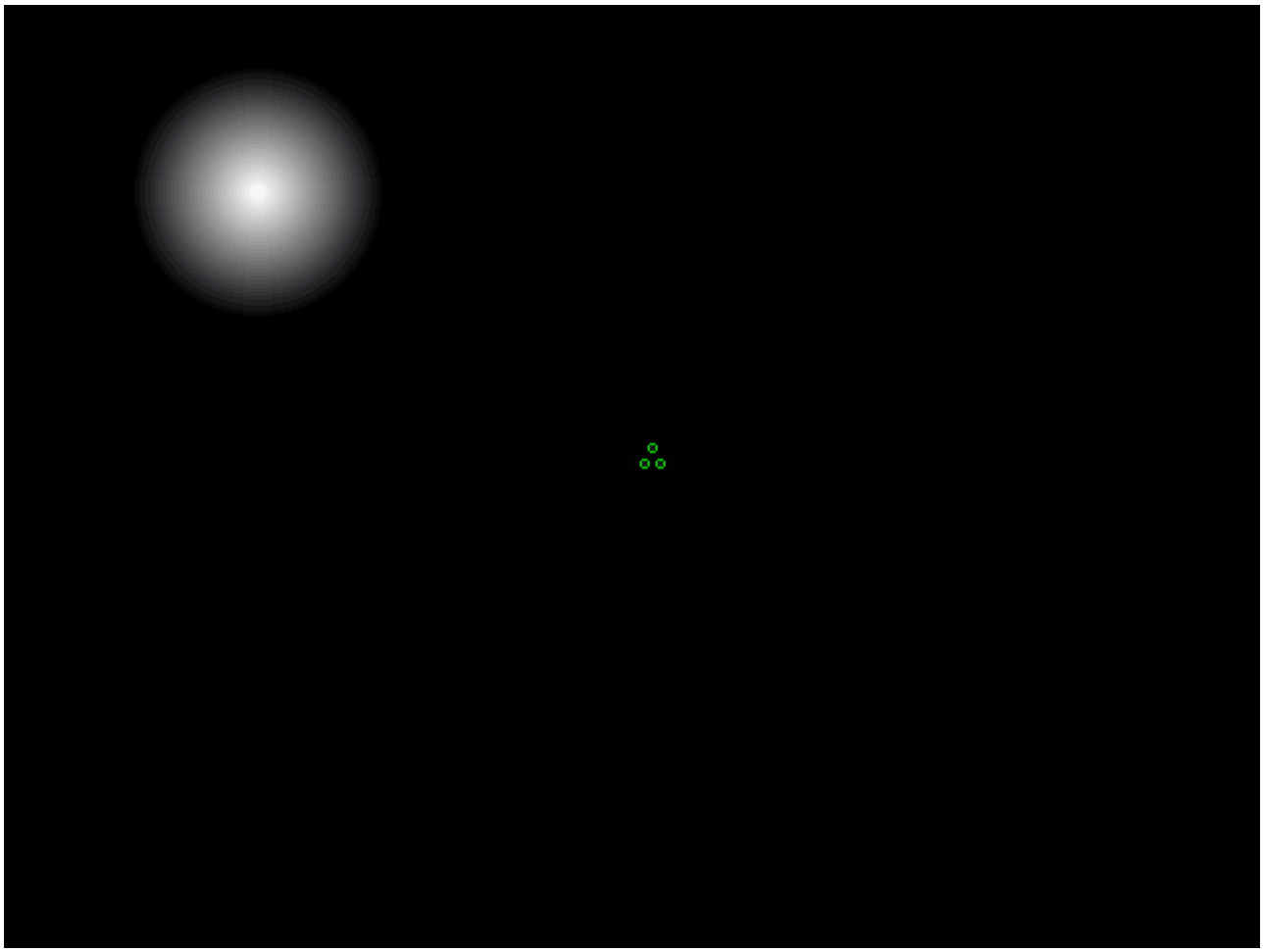
```
speed_x = speed_right - speed_left
```

Similar principle applies to the vertical speed.

It is apparent that speed_x and speed_y can only be in a range from -1 to +1. If this is too slow, then it is possible to speed up the creature by multiplying both speeds with a speedup_factor. You can do it by pressing some keys, described later on this page.

So the creature needs ANN with exactly 3 inputs and 4 outputs. If these values are different, LightChaser will say so and ask to load correct ANN.

OK, now we have enough information to start the LightChaser program. It should look something like this:



However, it is very likely that you immediately get an error message about incorrect configuration file. This is because LightChaser tries to read *config_generated.txt*, which is the same file that Trainer usually modifies, and which often doesn't contain what LightChaser needs. To load a correct ANN, press "L" and choose some other configuration file. Examples for LightChaser usually begin with *config_lc_* and also contain a short description (read it with Notepad or any other text editor).

On the screen you will see the same creature described before and a moving spotlight. When creature is under the spotlight, it will usually do something (depending on the ANN it contains).

The help about keys in LightChaser:

- To move the spotlight, move your mouse.

- To move the creature, hold down "Alt" (the left one) and move your mouse. It can be useful when creature wanders somewhere on the edge of screen and you want to get it back to the center.

- To change the size of spotlight, press mouse buttons (left button increases and right button decreases).

- To change the horizontal speed of spotlight, press "8", "9" or "6" on NumPad (which is usually located on the right side of keyboard. If you don't have any, then... too bad :) It may be a problem with portable computers. You can fix it easily in source code if you have C++ programming knowledge and Allegro library, or you can write me and ask to fix it.). Pressing "8" decreases the speed_x, "9" increases and "6" zeroes. However, speed_x is not the absolute value of horizontal speed, but the actual value, meaning that when speed_x < 0, then spotlight moves left, and when speed_x > 0, it will move right. Hence the interesting behaviour of buttons "8" and "9" (try it and you will know what I mean). This approach to changing the speed allows, for example, to make the spotlight bounce on the edge of screen like a basketball (hold down "9", for example).

- To change the vertical speed of spotlight, press "4", "7" or "5" on NumPad. Pressing "4" decreases the speed_y, "7" increases and "5" zeroes. Behaviour is similar as described in previous point for speed_x.

- To change the speedup_factor of the creature (described somewhere above), press "/", "*" or "-" on NumPad. Pressing "/" decreases the speedup_factor, "*" increases and "-" sets it back to default (which should be 3).

- To load another ANN into the creature, press "L". There are several examples included with this software package, they usually start with *conf_lc_* and also contain a short description (read it with Notepad or any other text editor). Feel free to create new ones with Trainer (read the

description of creature again to make sure you create correct ANNs).

If the loaded configuration file does not describe weights of connections, then they are assigned randomly between values -max_weight and +max_weight.

If the network in configuration file is very big, then you have to wait a little bit while it is being loaded. Very big ANN will also slow down the work of program.

- To get a short help about keys in LightChaser, press "F1".
- To save a screenshot into bitmap file *screenshot.bmp*, press "F12". However, that does not work in menus (in help and in loading menu).
- And finally, to exit the program, press "Esc". If your network is very big and computer has a hard time working on it, you have to hold down Escape until you see a text "ESC pressed, will QUIT".

Back | *Home*

MyANN library for C++

All MyANN applications use the same C++ library that is described here. This library implements the same feedforward multilayer ANN that was described in Introduction to ANN's and used in applications. With this library you can easily use ANNs in your own programs like games, computer animations, automatics, school projects, etc. Knowledge of C++ is required, though.

Warning: this library has no guarantee whatsoever, do NOT use it in any critical application!

In the next section is an example how to use MyANN library in one of the easiest ways.

First think about the network you need: how many inputs, how many outputs, which training patterns. You don't need to figure out every single detail, but knowing the main characteristics would be good.

Create this network using Trainer. Test it in Visualizer (not necessary, but useful).

Now you can use it in your code. The header file *neuralnetwork.h* has to be included in your C++ file. Source files of MyANN library can be either included in your project (check the manual of your programming environment) or compiled into a library file (something like *neuralnetwork.a*) and linked with your project.

```
#include "ann/neuralnetwork.h"
#include <vector>
```

Including <vector> may not be necessary as it is already done in *neuralnetwork.h*, but it is a good style to include things explicitly in every file they are used.

```
NeuralNetwork myNet;
```

This creates a NeuralNetwork object with the name myNet.

```
vector<double> v_dbl_inputs,
               v_dbl_outputs;
```

These are the vectors for feeding inputs into the ANN and for getting back the outputs.

```
v_dbl_inputs.assign(3,0);
v_dbl_outputs.assign(2,0);
```

Let's assume your network has 3 inputs and 2 outputs. You MUST make sure that the input vector is of the right size. You can do it with .resize() function, but using .assign() is even better, because this way you initialize all the elements of vector, too (currently they are all assigned 0). Size of the output vector is not so important, because if it is incorrect, then ANN will resize it itself.

```
v_dbl_inputs[0]= something;
v_dbl_inputs[1]= something;
v_dbl_inputs[2]= something;
```

Here you put into the input vector the values your program wants to feed into the ANN.

```
myNet.set_inputs(&v_dbl_inputs);
myNet.update();
myNet.get_outputs(&v_dbl_outputs);
```

And that's how you interact with the ANN. First you give it the input vector (actually the address of the input vector, using &). Then you command ANN to update itself, and finally read out the outputs. Now you will probably use the outputs somewhere in your program...

```
something = v_dbl_outputs[0];
something = v_dbl_outputs[1];
```

...then set new values into v_dbl_inputs and use ANN again. Et cetera...

It was one of the easiest ways to use MyANN library, but there are many more possibilities. For example it is not necessary to use Trainer, which actually doesn't do anything but only calls library functions. For more information, go to the Source code section and read source files. They are well commented (at least I tried to make them readable). Good luck with coding ;)

Source code

If you prefer to browse it offline, then please download the whole software package from download section, all source files are included there (both HTML and original versions). Otherwise read on.

Following is the source code in HTML with syntax highlight, designed for easy browsing.

ANN library and Trainer contain pure C++ code. Visualizer and LightChaser use free Allegro game programming library by Shawn Hargreaves, which is a very useful library indeed. You can get it from: http://alleg.sourceforge.net/.

/ann

- connection.h
- neuron.h
- neuralnetwork.h
- connection.cpp
- neuron.cpp
- neuralnetwork.cpp
- configuration.txt
- trainingfile.txt

/trainer

/ann

(see above)

- main.cpp
- configuration.txt
- trainingfile.txt

/visualizer

/visnet

/ann

(see above)

- mypalette.h
- visual_ann.h
- mypalette.cpp
- visual_ann.cpp

main.cpp
config_generated.txt

/lightchaser

/creature

/ann

(see above)

creature.h
creature.cpp

/light

light.h
light.cpp

/palette

mypalette.h
mypalette.cpp

main.cpp
config_generated.txt

copyright.txt

Source files are converted to HTML using Colorer Library take5 from Cail Lomecb (Igor Ruskih), http://colorer.sf.net/.

Back | *Home*

Download

Choose the file according to your
operating system:

MyANNforDOS.zip (... k)
MyANNforWIN.zip (... k)
MyANNforLINUX.gz (... k)

For other platforms you have to compile it yourself
(programming knowledge required)

MyANNforOTHER.zip (... k)
MyANNforOTHER.gz (... k)

ANN library and Trainer contain pure C++ code. Visualizer and LightChaser use
free Allegro game programming library by Shawn Hargreaves, which currently supports
DOS, Unix (Linux, FreeBSD, Irix, Solaris), Windows, QNX, and BeOS (MacOS port is in alpha stage).

ANN information

http://directory.google.com/Top/Computers/Artificial_Intelligence/Neural_Networks/	A lot of links to various pages about neural networks.
http://www.csse.monash.edu.au/~app/CSC437nn/	Neural Networks, Lecture Notes. *A. P. Paplinski.*
http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html	Neural Networks. *C. Stergiou, D. Siganos.*
http://www.mathworks.com/access/helpdesk/help/pdf_doc/nnet/nnet.pdf (BIG, over 5MB)	Neural Network Toolbox For Use with MATLAB, User's Guide, Version 4. *H. Demuth, M. Beale.*
http://neil.fraser.name/writing/tank/	A funny story showing that ANNs should be used carefully.

Java Applets

http://www-ra.informatik.uni-tuebingen.de/mitarb/fischer/applets.html	Author: *Igor Fischer.*
http://home.cc.umanitoba.ca/~umcorbe9/anns.html	Author: *Fred Corbett.*

Free software

http://www-ra.informatik.uni-tuebingen.de/SNNS/	Stuttgart Neural Network Simulator.
http://www-ra.informatik.uni-tuebingen.de/software/JavaNNS/welcome.html	Java Neural Network Simulator.
http://www.cnbc.cmu.edu/Resources/PDP++//PDP++.html	PDP++

Programming

http://mindview.net/Books/TICPP/ThinkingInCPP2e.html	Thinking in C++. *Bruce Eckel.*
http://alleg.sourceforge.net/	Allegro. *Shawn Hargreaves*.

Back | *Home*

Programmide lähtekood

Copyright information for NeuralNetwork programs and
libraries by Taivo Lints, Estonia.

This version is not ment for distributing as it is
still in development phase. However, it is allowed to
copy it for evaluation purposes. Final copyright status
for this project will be decided later and will most
probably be GNU GPL or GNU LGPL (for ANN library) or
giftware. Distribution versions of this project will
(hopefully) be available on Internet. Release date
and URL are not specified at the moment.

Notice is written in May, 2003.

- - - - - - -

Informatsioon käesoleva projekti (programmid ja teegid,
mis realiseerivad ja demonstreerivad tehisnärvivõrke)
autoriõiguste kohta.

Projekt on hetkel veel arendusfaasis ning seetõttu ei
ole need programmid ja teegid mõeldud levitamiseks.
Koopiaid on lubatud teha projektiga tutvumise ja
selle hindamise eesmärgil. Lõplik autoriõiguste kaitse
tase määratakse hiljem ning saab tõenäoliselt olema
kas GNU GPL või GNU LGPL (teekidele) või giftware
(st. piiranguteta). Levitamiseks mõeldud versioonid
on (loodetavasti) mõne aja pärast kättesaadavad
Internetis. Kuupäevad ja veebiaadressid ei ole
veel kindlaks määratud.

Taivo Lints
Mai, 2003.

/ann

```
// - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: connection.h                                 |
// Purpose: provides interface for class Connection   |
// Author: Taivo Lints, Estonia                       |
// Date: May, 2003                                    |
// Copyright: see copyright.txt                       |
// - - - - - - - - - - - - - - - - - - - - - - - - - -

#ifndef CONNECTION_H
#define CONNECTION_H

class Connection {    // Represents connection between two neurons.

  public:

    Connection(int source, double weight);  // Connection constructor.
                    // You must give all the required parameters
                    // when creating a new connection.

    int source;     // Starting point of the connection (number of the
                    // OTHER neuron; NOT the number of neuron, who owns
                    // this connection! "number" means vpNeurons index.).

    double weight;  // Weight of the connection.

};


#endif // CONNECTION_H
```

```
// - - - - - - - - - - - - - - - - - - - - - - - -
// File: neuron.h                                 |
// Purpose: provides interface for class Neuron   |
// Author: Taivo Lints, Estonia                   |
// Date: May, 2003                                |
// Copyright: see copyright.txt                   |
// - - - - - - - - - - - - - - - - - - - - - - - -

#ifndef NEURON_H
#define NEURON_H

#include "connection.h"
#include <vector>
using namespace std;

class Neuron {       // Represents a typical neuron for artificial neural
                     // networks. Activation function is sigmoidal (logistic
                     // function 1 / (1 + exp(-av)) where "a" is slope parameter
                     // and "v" is internal activity level). Neuron supports
                     // backpropagation learning.


  public:

    // ******************************
    // * Construction & Destruction *
    // ******************************

    Neuron(vector<Neuron*>* pvpNeurons, bool flag_comp_neuron);
                     // Neuron constructor. Needs a pointer to the vpNeurons
                     // vector where this neuron will belong (because it
                     // needs to communicate with other neurons). Also needs
                     // to know the type of this neuron (if flag_comp_neuron
                     // is "true", then it will be computational neuron;
                     // if "false", then input node).

    Neuron(const Neuron& rN);  // Neuron copy-constructor.

    Neuron& operator=(const Neuron& rN);  // Operator overloading.

    ~Neuron();       // Neuron destructor.


    // *************************
    // * Parameters & variables *
    // *************************

    vector<Neuron*>* pvpNeurons; // Pointer to the vpNeurons vector where this
                                 // neuron belongs.

    double treshold;          // Treshold (or bias) value for lowering or
                              // increasing the sum (internal activity level)
                              // before applying the activation function.

    double slope_parameter;   // Slope parameter of the sigmoid function
                              // ("a" in the equation 1 / (1 + exp(-av)) ).
                              // When the slope parameter approaches infinity,
                              // the sigmoid function becomes simply a
                              // treshold function.

    double learning_rate;     // How fast should neuron learn.
                              // Too small makes learning too slow.
                              // Too big rate may give no learning results
                              // at all (i.e. no convergence).

    double max_weight;        // Maximal allowed weight for connections. Weights
                              // will be in range -max_weight..+max_weight.

    double output;            // Output value of this neuron.

    vector<Connection*> vpConnections;  // All connections owned by this neuron.
                              // If you decide to rearrange something in
                              // vpNeurons in your NeuralNetwork object, then
```

```
                                // do NOT forget to update this vector here!

    double error;               // Error value for backpropagation learning.

    // *************
    // * Functions *
    // *************

    void add_connection(int source, double weight); // Adds a connection
                                // to vpConnections vector. Does NOT verify
                                // it's correctness! "source" is source neuron's
                                // index in vpNeurons vector in class
                                // NeuralNetwork.

    void update();              // Reads inputs through connections and
                                // calculates the output.

    void learn();               // Modifies weights of the connections
                                // according to the error. Also backpropagates
                                // error. Should be applied starting from the
                                // last neuron and moving towards the input
                                // layer (to BACKpropagate the error).


  // ******************
  // * Protected stuff *
  // ******************

  // You can't use that stuff from outside code, except from child classes.

  protected:

    bool flag_comp_neuron;      // If true, then this neuron is computational
                                // neuron; if false, then it is input node.
                                // This flag is set by constructor.

};

#endif // NEURON_H
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: neuralnetwork.h                                      |
// Purpose: provides interface for class NeuralNetwork        |
// Author: Taivo Lints, Estonia                               |
// Date: May, 2003                                            |
// Copyright: see copyright.txt                               |
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

#ifndef NEURALNETWORK_H
#define NEURALNETWORK_H

#include "neuron.h"
#include <vector>
using namespace std;

class NeuralNetwork {  // Represents a typical multilayered feedforward
                       // artificial neural network. Uses backpropagation
                       // for training.

                       // I am currently not supporting modifying the
                       // structure of already generated network. It would
                       // be useful when you want to create/edit the network
                       // node by node with mouse in a graphical environment,
                       // so it may be supported in the future...

                       // By the way, it would probably be a brilliant idea
                       // to create a network that modifies structure ITSELF
                       // at runtime, but then it shouldn't be such a simple
                       // layered feedforward network :)
   public:

     // ******************************
     // * Construction & Destruction *
     // ******************************

     NeuralNetwork(char* config_file);
                  // NeuralNetwork constructor. Creates a network from your
                  // configuration file. If file not found, then you will get
                  // an empty network. You can check it by calling
                  // get_number_of_layers(). It will be zero when network is
                  // empty.

     NeuralNetwork(const NeuralNetwork& rN);  // NeuralNetwork copy-constructor.

     NeuralNetwork& operator=(const NeuralNetwork& rN); // Operator overloading.

     ~NeuralNetwork(); // NeuralNetwork destructor.


     // ********************************************
     // * Functions for getting general information *
     // ********************************************

     int get_number_of_inputs();   // Returns the number of input nodes.
     int get_number_of_outputs();  // Returns the number of output nodes.
     int get_number_of_layers();   // Returns the number of layers, including
                                   // the input layer (which is made of buffer
                                   // nodes, not real neurons).

     int get_number_of_nodes();    // Returns the total number of nodes,
                                   // including input nodes.


     // ************************************************
     // * Functions for setting some neuron parameters *
     // ************************************************

     // Normally you don't need to use them, just give
     // these parameters in training configuration file,
     // or don't give them at all, then default values are used.

     void set_treshold(double treshold);
                                   // Sets treshold value in all neurons
```

```
                                // (see "neuron.h" for more information).

void set_slope_parameter(double slope_parameter);
                                // Sets slope_parameter value in all neurons.
                                // (see "neuron.h" for more information).


// ****************************************
// * Functions for basic use of the network *
// ****************************************

bool set_inputs(vector<double>* v_dbl_inputvalues); // Sets network inputs.
                                // Returns false if size of given input
                                // vector is incorrect.

void update();                  // Updates the network (calls update() for
                                // all neurons).

void get_outputs(vector<double>* vp_outputs); // Writes network outputs
                                // into given vector. If vector size
                                // is incorrect, will fix it.


// ************************************************
// * A useful class for storing a training pattern *
// ************************************************

// Normally you don't need to use it, just give
// your patterns in training configuration file.

class Pattern {  // A useful class for storing one training pattern.
                 // As you see, this is just a class, not an object.
                 // There is a private vector vPatterns containing
                 // Pattern objects (see somewhere below). That vector
                 // is normally filled by train() function, but if you
                 // want to do it manually with add_training_pattern()
                 // from outside code, then you can create a pattern
                 // object this way:

                 // NeuralNetwork::Pattern myPattern;

  public:

    vector<double> vInputs;    // Input values for all input nodes (starting
                               // from the first in vpNeurons).

    vector<double> vOutputs;   // Desired output values (in the same order
                               // as output nodes in vpNeurons).
};


// ****************************
// * Some training parameters *
// ****************************

// Normally you don't need to set them, just give
// these parameters in training configuration file,
// or don't give them at all, then default values are used.

double error_limit;          // For evaluating the success of training.
                             // If all outputs are in a range from
                             // (desired_output - error_limit) to
                             // (desired_output + error_limit), then
                             // training is completed.

int max_iterations;          // Maximum allowed number of training
                             // iterations (epochs). When this number
                             // of iterations is reached, training
                             // will be stopped and is failed (i.e.
                             // outputs were still outside the given
                             // error limits).


// ************************************************************
```

```
// * Functions for setting other training parameters & patterns *
// *************************************************************

// Normally you don't need to use them, just give these parameters
// in training configuration file, or don't give them at all,
// then default values are used. Patterns, unlike parameters,
// do NOT have default values, so they should be present in training
// configuration file if you want to train the network (or if you
// really want to set them manually, then use these functions).

void set_learning_rate(double learning_rate);
                              // Sets learning rate in all neurons.
                              // (see "neuron.h" for more information).

void set_max_weight(double max_weight);
                              // Sets max_weight value in all neurons
                              // (see "neuron.h" for more information).

void add_training_pattern(Pattern myPattern);
                              // Adds a training pattern to vPatterns
                              // (see somewhere below).

bool verify_all_patterns();   // Verifies that the number of input and
                              // output values in all patterns is
                              // correct (e.g. if you have two input
                              // neurons in your network, then every
                              // pattern must hold two input values,
                              // i.e. the size of vInputs vector must be
                              // two). Returns "true" if all OK.

                              // When you add training patterns manually,
                              // please USE this function before calling
                              // train(NULL) and do NOT call train(NULL)
                              // if something is wrong, otherwise the
                              // program may crash.

void erase_all_patterns();    // Deletes all training patterns.

int get_number_of_patterns(); // Returns the number of training patterns
                              // in vPatterns vector.


// *************************
// * Functions for training *
// *************************

int train(char* training_file);
            // Trains network using backpropagation algorithm. Returns 0,
            // if training successful; returns 1, if goal not reached;
            // returns 2, if problems with patterns. Also sets iterations
            // parameter (see the next section below). Gets information
            // from a file. Example: train("trainingfile.txt")

            // If you have fiddled something manually and don't want to
            // use training file, then call train(NULL), because
            // otherwise all your patterns will be deleted.


// ***********************************
// * Some information about training *
// ***********************************

int iterations;  // Train() function writes here the number of training
                 // iterations (epochs) that it had to go through before
                 // reaching the goal (if goal was not reached, then
                 // iterations = max_iterations).

// ***********************************
// * Functions for saving the network *
// ***********************************

void save_network(char* savefile);  // Saves your network into a file.
            // Example: save_network("configuration_generated.txt")
            // You can use this generated file for initializing a
```

```
                // new network (instead of the usual configuration file,
                // where weights are not described).


    // ***********************************************
    // * Functions for getting some neuron parameters *
    // ***********************************************

    // All these functions assume, that all neurons have similar
    // parameters. So they just use values taken from the first
    // neuron in vpNeurons vector (which is actually an input node,
    // but still should have all these parameters set correctly).
    // If network is empty, then all these functions return 0.

    double get_treshold();
    double get_slope_parameter();
    double get_learning_rate();
    double get_max_weight();   // Returns the value of parameter max_weight,
                               // NOT the maximum weight found in network!


  // ******************
  // * Protected stuff *
  // ******************

  // You can't use that stuff from outside code, except from child classes.

  protected:

    vector<Neuron*> vpNeurons;     // Vector of pointers to neurons. This is
                                   // the actual "implementation" of network.
                                   // Vector allows array style indexing
                                   // and thus makes it easier to describe
                                   // connections between neurons (I can say:
                                   // "This neuron is connected to a neuron
                                   // number 7," meaning vpNeurons[7]).

    vector<int> v_int_layers;      // Contains information about how many
                                   // nodes there are in each layer, starting
                                   // from the input layer.

    vector<Pattern> vPatterns;     // Vector of training patterns.

    void load_config_file(char* config_file);
                                   // Loads info from configuration file.
                                   // Is used by constructor. Does not
                                   // guarantee anything and should be used
                                   // carefully.

    void create_unset_connections();  // Creates missing connections so that
                                   // network will be a fully connected multi-
                                   // layer feedforward network. Is used by
                                   // constructor.

    int load_training_file(char* training_file);
                                   // Loads info from training file. Is used
                                   // by train() function. Returns the number
                                   // of loading errors.

};

#endif // NEURALNETWORK_H
```

```cpp
// - - - - - - - - - - - - - - - - - - - - -
// File: connection.cpp                      |
// Purpose: implements class Connection      |
// Author: Taivo Lints, Estonia              |
// Date: May, 2003                           |
// Copyright: see copyright.txt              |
// - - - - - - - - - - - - - - - - - - - - - -

#include "connection.h"
using namespace std;

// - - - - - Class Connection - - - - -

// - Connection constructor - -
// You must give all the required parameters when creating a new connection
// (see connection.h).
Connection::Connection(int init_source, double init_weight) {

   source = init_source;  // Sets Connection parameters.
   weight = init_weight;

}
```

```
// - - - - - - - - - - - - - - - - - - - - -
// File: neuron.cpp                         |
// Purpose: implements class Neuron         |
// Author: Taivo Lints, Estonia             |
// Date: May, 2003                          |
// Copyright: see copyright.txt             |
// - - - - - - - - - - - - - - - - - - - - -

#include "neuron.h"
#include "connection.h"
#include "math.h"
#include <vector>
using namespace std;

// - - - - - Class Neuron - - - - -


// ******************************
// * Construction & Destruction *
// ******************************

// - Neuron constructor - -
// Sets Neuron parameters to default values. Needs a pointer to the
// vpNeurons vector where this neuron will belong (because it needs
// to communicate with other neurons).
Neuron::Neuron(vector<Neuron*>* init_pvpNeurons, bool inp_flag_comp_neuron) {

  pvpNeurons = init_pvpNeurons;

  treshold = 0;
  slope_parameter = 0.5;

  output = 0;

  error = 0;
  learning_rate = 5;
  max_weight = 10;

  flag_comp_neuron = inp_flag_comp_neuron;

}

// - Neuron copy-constructor - -
Neuron::Neuron(const Neuron& rN) {

  pvpNeurons = rN.pvpNeurons;
  treshold = rN.treshold;
  slope_parameter = rN.slope_parameter;
  learning_rate = rN.learning_rate;
  max_weight = rN.max_weight;
  output = rN.output;
  error = rN.error;
  flag_comp_neuron = rN.flag_comp_neuron;

  Neuron& rN2 = const_cast<Neuron&>(rN);
              // A suspicious thing to do, but compiler will throw warnings
              // about invalid conversions in stl_container when I don't
              // cast away the const-ness. At the moment I don't have time
              // to trace the source of this problem. At least it SEEMS to
              // be a harmless problem...

  // Must NOT point to the connections of the OTHER neuron,
  // must create its own connections.
  for(vector<Connection*>::iterator i = rN2.vpConnections.begin();
      i != rN2.vpConnections.end(); i++) {
    vpConnections.push_back(new Connection(0, 0));
    *(*(--vpConnections.end())) = *(*i);
  }

}

// - Operator= overloading - -
Neuron& Neuron::operator=(const Neuron& rN) {
```

```
  if(&rN != this) {   // Check for self-assignment.

    pvpNeurons = rN.pvpNeurons;
    treshold = rN.treshold;
    slope_parameter = rN.slope_parameter;
    learning_rate = rN.learning_rate;
    max_weight = rN.max_weight;
    output = rN.output;
    error = rN.error;
    flag_comp_neuron = rN.flag_comp_neuron;

    Neuron& rN2 = const_cast<Neuron&>(rN);
                // A suspicious thing to do, but compiler will throw warnings
                // about invalid conversions in stl_container when I don't
                // cast away the const-ness. At the moment I don't have time
                // to trace the source of this problem. At least it SEEMS to
                // be a harmless problem...

    // Must NOT point to the connections of the OTHER neuron,
    // must create its own connections.
    for(vector<Connection*>::iterator i = rN2.vpConnections.begin();
        i != rN2.vpConnections.end(); i++) {
      vpConnections.push_back(new Connection(0, 0));
      *(*(--vpConnections.end())) = *(*i);
    }
  }

  return *this;

}

// - Neuron destructor - -
Neuron::~Neuron() {

  // Deletes all Connections (to prevent memory leak).
  for(vector<Connection*>::iterator i = vpConnections.begin();
      i != vpConnections.end(); i++)
    delete *i;

}


// *************
// * Functions *
// *************

// - Function: add_connection() - -
// Adds a connection to vpConnections vector. Does NOT verify
// it's correctness!
void Neuron::add_connection(int source, double weight) {

  // Input nodes can't have connections.
  if(not flag_comp_neuron)
    return;

  vpConnections.push_back(new Connection(source, weight));

}

// - Function: update() - -
// Reads inputs through connections and calculates the output.
void Neuron::update() {

  // Input nodes don't update themselves.
  if(not flag_comp_neuron)
    return;

  double sum = 0;  // The sum of all inputs (source neuron outputs
                   // multiplied by corresponding connection weights).

  // Calculating the sum.
  for(vector<Connection*>::iterator i = vpConnections.begin();
```

```
      i != vpConnections.end(); i++)
    sum += (*pvpNeurons)[(*i)->source]->output * (*i)->weight;

  sum -= treshold;  // Applying treshold value.

  // Calculating new output (i.e. applying the activation function).
  output = 1 / (1 + exp(-slope_parameter * sum));

}

// - Function: learn() - -
// Updates error value and modifies weights of the connections.
// Also backpropagates error.
void Neuron::learn() {

  // Input nodes don't learn.
  if(not flag_comp_neuron) {
    error = 0;
    return;
  }

  // Calculating error signal (multiplying the error value with
  // the derivative of activation function).
  double error_signal = error * slope_parameter * output * (1 - output);

  // Backpropagating error signal and THEN updating weights.
  for(vector<Connection*>::iterator i = vpConnections.begin();
      i != vpConnections.end(); i++) {
    (*pvpNeurons)[(*i)->source]->error += error_signal * (*i)->weight;
    (*i)->weight += learning_rate * error_signal *
                    (*pvpNeurons)[(*i)->source]->output;

    if((*i)->weight > max_weight)   // Weights should NOT be allowed to go
      (*i)->weight = max_weight;    // to infinity.

    if((*i)->weight < -max_weight)
      (*i)->weight = -max_weight;
  }

  error = 0;  // Must be done, otherwise error will accumulate (look at
              // the "+=" operation a few rows up from here).

}
```

```
// - - - - - - - - - - - - - - - - - - - - -
// File: neuralnetwork.cpp                  |
// Purpose: implements class NeuralNetwork  |
// Author: Taivo Lints, Estonia             |
// Date: May, 2003                          |
// Copyright: see copyright.txt             |
// - - - - - - - - - - - - - - - - - - - - -

#include "neuralnetwork.h"
#include "neuron.h"
#include <vector>
#include <fstream>
#include <string>
#include <sstream>
using namespace std;

// - - - - - Class NeuralNetwork - - - - -

// ****************************
// * Construction & Destruction *
// ****************************

// - NeuralNetwork constructor - -
// Creates a network from your configuration file. If file not found,
// then you will get an empty network. You can check it by calling
// get_number_of_layers(). It will be zero when network is empty.
NeuralNetwork::NeuralNetwork(char* config_file) {

  load_config_file(config_file);
  create_unset_connections();

  error_limit = 0.1;
  max_iterations = 20000;

  iterations = 0;

}

// - NeuralNetwork copy-constructor - -
NeuralNetwork::NeuralNetwork(const NeuralNetwork& rN) {

  error_limit = rN.error_limit;
  max_iterations = rN.max_iterations;
  iterations = rN.iterations;
  v_int_layers = rN.v_int_layers;
  vPatterns = rN.vPatterns;

  NeuralNetwork& rN2 = const_cast<NeuralNetwork&>(rN);
              // A suspicious thing to do, but compiler will throw warnings
              // about invalid conversions in stl_container when I don't
              // cast away the const-ness. At the moment I don't have time
              // to trace the source of this problem. At least it SEEMS to
              // be a harmless problem...


  // Must NOT point to the neurons of the OTHER NeuralNetwork,
  // must create its own neurons.
  for(vector<Neuron*>::iterator i = rN2.vpNeurons.begin();
      i != rN2.vpNeurons.end(); i++) {

    vpNeurons.push_back(new Neuron(NULL, false));
    *(*(--vpNeurons.end())) = *(*i);
    (*(--vpNeurons.end()))->pvpNeurons = &vpNeurons;
                                  // This neuron now belongs to the vpNeurons
                                  // vector of our new NeuralNetwork.
  }

}

// - Operator= overloading - -
NeuralNetwork& NeuralNetwork::operator=(const NeuralNetwork& rN) {

  if(&rN != this) {    // Check for self-assignment.
```

```
    error_limit = rN.error_limit;
    max_iterations = rN.max_iterations;
    iterations = rN.iterations;
    v_int_layers = rN.v_int_layers;
    vPatterns = rN.vPatterns;

    NeuralNetwork& rN2 = const_cast<NeuralNetwork&>(rN);
                // A suspicious thing to do, but compiler will throw warnings
                // about invalid conversions in stl_container when I don't
                // cast away the const-ness. At the moment I don't have time
                // to trace the source of this problem. At least it SEEMS to
                // be a harmless problem...


    // Must NOT point to the neurons of the OTHER NeuralNetwork,
    // must create its own neurons.
    for(vector<Neuron*>::iterator i = rN2.vpNeurons.begin();
        i != rN2.vpNeurons.end(); i++) {

      vpNeurons.push_back(new Neuron(NULL, false));
      *(*(--vpNeurons.end())) = *(*i);
      (*(--vpNeurons.end()))->pvpNeurons = &vpNeurons;
                                    // This neuron now belongs to the vpNeurons
                                    // vector of our new NeuralNetwork.
    }
  }

  return *this;

}


// - NeuralNetwork destructor - -
// Performs clean-up.
NeuralNetwork::~NeuralNetwork() {

  // Deletes all Neurons (to prevent memory leak).
  for(vector<Neuron*>::iterator i = vpNeurons.begin();
      i != vpNeurons.end(); i++)
    delete *i;

}

// *******************************************
// * Functions for getting general information *
// *******************************************

// - Function: get_number_of_inputs() - -
// Returns the number of input nodes.
int NeuralNetwork::get_number_of_inputs() {

  if(get_number_of_layers() == 0)
    return(0);

  return(*v_int_layers.begin());

}

// - Function: get_number_of_outputs() - -
// Returns the number of output nodes.
int NeuralNetwork::get_number_of_outputs() {

  if(get_number_of_layers() == 0)
    return(0);

  return(*(--v_int_layers.end()));

}

// - Function: get_number_of_layers() - -
// Returns the number of layers, including the input layer (which is made of
// buffer nodes, not real neurons).
```

```
int NeuralNetwork::get_number_of_layers() {

  return(static_cast<int>(v_int_layers.size()));

}

// - Function: get_number_of_nodes() - -
// Returns the total number of nodes.
int NeuralNetwork::get_number_of_nodes() {

  return(static_cast<int>(vpNeurons.size()));

}

// ************************************************
// * Functions for setting some neuron parameters *
// ************************************************

// - Function: set_treshold - -
// Sets treshold value in all neurons.
void NeuralNetwork::set_treshold(double treshold) {

  for(vector<Neuron*>::iterator i = vpNeurons.begin();
      i != vpNeurons.end(); i++)
    (*i)->treshold = treshold;

}

// - Function: set_slope_parameter - -
// Sets slope_parameter value in all neurons.
void NeuralNetwork::set_slope_parameter(double slope_parameter) {

  for(vector<Neuron*>::iterator i = vpNeurons.begin();
      i != vpNeurons.end(); i++)
    (*i)->slope_parameter = slope_parameter;

}


// ****************************************
// * Functions for basic use of the network *
// ****************************************

// - Function: set_inputs - -
// Sets network inputs. Returns false if size of given input vector is incorrect.
bool NeuralNetwork::set_inputs(vector<double>* pv_dbl_inputvalues) {

  // How many inputs does our network have?
  int num_of_inputs = get_number_of_inputs();

  // Check, if input vector size is correct or not.
  if(static_cast<int>(pv_dbl_inputvalues->size()) != num_of_inputs)
    return(false);

  // Assign input values to buffer neurons in first layer.
  for(int i = 0; i < num_of_inputs; i++)
    vpNeurons[i]->output = (*pv_dbl_inputvalues)[i];

  return(true);

}

// - Function: update() - -
// Updates the network (calls update() for all neurons).
void NeuralNetwork::update() {

  // Update starts from the beginning of vpNeurons and goes linearly through
  // this vector. So when (imaginary) layers are correctly ordered in this
  // vector of neurons, then the information from inputs is propagated through
  // all the network during this "for" cycle.
  for(vector<Neuron*>::iterator i = vpNeurons.begin();
      i != vpNeurons.end(); i++)
    (*i)->update();
```

```
}

// - Function: get_outputs - -
// Writes network outputs into given vector. If vector size is incorrect,
// then will fix it.
void NeuralNetwork::get_outputs(vector<double>* pv_outputs) {

  // How many outputs and neurons do we have?
  int num_of_outputs = get_number_of_outputs(),
      num_of_neurons = get_number_of_nodes();

  // If given vector is of wrong size, then will fix it.
  if(static_cast<int>(pv_outputs->size()) != num_of_outputs)
    pv_outputs->resize(num_of_outputs);

  // What is the number of first output neuron in vpNeurons vector?
  int first_output_node = num_of_neurons - num_of_outputs;

  // Writing neuron output values into given vector.
  for(int i = first_output_node; i < num_of_neurons; i++)
    (*pv_outputs)[i - first_output_node] = vpNeurons[i]->output;

}

// ************************************************
// * Functions for setting some training parameters *
// ************************************************

// - Function: set_learning_rate - -
// Sets learning rate in all neurons.
void NeuralNetwork::set_learning_rate(double learning_rate) {

  for(vector<Neuron*>::iterator i = vpNeurons.begin();
      i != vpNeurons.end(); i++)
    (*i)->learning_rate = learning_rate;

}

// - Function: set_max_weight - -
// Sets max_weight value in all neurons.
void NeuralNetwork::set_max_weight(double max_weight) {

  for(vector<Neuron*>::iterator i = vpNeurons.begin();
      i != vpNeurons.end(); i++)
    (*i)->max_weight = max_weight;

}

// - Function: add_training_pattern - -
// Adds a training pattern to vPatterns.
void NeuralNetwork::add_training_pattern(Pattern myPattern) {

  vPatterns.push_back(myPattern);

}

// - Function: verify_all_patterns - -
// Verifies that the number of input and output values in
// all patterns is correct. Returns "true" if all OK.
bool NeuralNetwork::verify_all_patterns() {

  // How many inputs and outputs do we have?
  int num_of_inputs = get_number_of_inputs(),
      num_of_outputs = get_number_of_outputs();

  // Verify all patterns
  for(vector<Pattern>::iterator i = vPatterns.begin();
      i != vPatterns.end(); i++)
    if( (static_cast<int>((*i).vInputs.size()) != num_of_inputs) or
        (static_cast<int>((*i).vOutputs.size()) != num_of_outputs) )
      return(false);
```

```
  return(true);

}

// - Function: erase_all_patterns - -
// Deletes all training patterns.
void NeuralNetwork::erase_all_patterns() {

  vPatterns.clear();

}

// - Function: get_number_of_patterns - -
// Returns the number of training patterns in vPatterns vector.
int NeuralNetwork::get_number_of_patterns() {

  return(static_cast<int>(vPatterns.size()));

}


// *************************
// * Functions for training *
// *************************

// - Function: train - -
// Trains network using backpropagation algorithm. Returns 0,
// if training successful; returns 1, if goal not reached;
// returns 2, if problems with patterns. Also sets iterations
// parameter (see "neuralnetwork.h" for more information).

// If you don't want to use training file (a rare case, though),
// then call train(NULL).

int NeuralNetwork::train(char* training_file) {

  // If training file IS given, then load it.
  if(training_file != NULL) {
    erase_all_patterns();   // Deletes old data.
    if(load_training_file(training_file) > 0)  // Reads new data from file.
      return(2);
  }

  // If wrong patterns, then exit function.
  if(not verify_all_patterns())
    return(2);

  int i;  // We want to preserve the value of "for" cycle iterator beyond
          // the scope of "for".

  // Repeats training until ready, or until max_iterations limit reached.
  for(i = 0; i < max_iterations; i++) {

    // First we have to make sure if training is necessary (maybe network
    // already behaves like we want?).

    bool goal_reached = true;

    // Goes through all patterns.
    for(vector<Pattern>::iterator iter = vPatterns.begin();
        iter != vPatterns.end(); iter++) {

      set_inputs( &((*iter).vInputs) );
      update();

      vector<double> v_outputs;
      get_outputs(&v_outputs);

      // Checks, if outputs are inside or outside allowed limits.
      for(int output_number = 0;
          output_number < static_cast<int>(v_outputs.size());
          output_number++) {
```

```
        double target_output = (*iter).vOutputs[output_number],
               output = v_outputs[output_number],
               error = abs(target_output - output);

        if(error > error_limit) {
          goal_reached = false;
          goto learn;  // Using "goto" is generally considered as a poor
                       // programming style. However, this is one of the
                       // rare cases where using "goto" is not strongly
                       // deprecated (i.e. jumping out of multiple loops,
                       // alternatively done by writing several flag
                       // testings and breaks. Of course, the situation
                       // must still be analyzed carefully before using "goto"
                       // to avoid breaking your code (e.g. nonlocal goto
                       // may cause skipping some destructors -- a very bad
                       // thing to do!)).
        }
      }
    }

    // If goal is reached, then exits this function.
    if(goal_reached) {
      iterations = i;
      return(0);
    }

    // If goal is NOT reached, then performs a training procedure.
    learn:  // A label for goto.

    // What is the number of first output neuron in vpNeurons vector?
    int first_output_node = get_number_of_nodes() - get_number_of_outputs();

    // Goes through all patterns.
    for(vector<Pattern>::iterator iter = vPatterns.begin();
        iter != vPatterns.end(); iter++) {

      set_inputs( &((*iter).vInputs) );
      update();

      vector<double> v_outputs;
      get_outputs(&v_outputs);

      // Sets error value in output neurons.
      for(int output_number = 0;
          output_number < static_cast<int>(v_outputs.size());
          output_number++) {

        double target_output = (*iter).vOutputs[output_number],
               output = v_outputs[output_number];

        vpNeurons[first_output_node + output_number]->
          error = target_output - output;
      }

      // Commands each neuron to learn (and to backpropagate the error).
      // Goes through vpNeurons in reverse direction because of the
      // BACKpropagation algorithm.
      for(vector<Neuron*>::reverse_iterator r = vpNeurons.rbegin();
          r != vpNeurons.rend(); r++)
        (*r)->learn();
    }
  }

  // Apparently the goal was not reached.
  iterations = i;
  return(1);

}


// ***********************************
// * Functions for saving the network *
// ***********************************
```

```
// - Function: save_network - -
// Saves your network into a file. You can use this generated file
// for initializing a new network (instead of the usual configuration
// file, where weights are not described).
void NeuralNetwork::save_network(char* savefile) {

  // Creates an output file (if already exists, then overwrites it).
  ofstream out_file(savefile);

  // Sets some output parameters.
  out_file.setf(ios::fixed, ios::floatfield);
  out_file.precision(30);  // Won't ever be so big, but it automatically cuts
                           // down to the max. possible precision. I hope ;)

  // If no neurons, then nothing to save.
  if(get_number_of_nodes() == 0) {
    out_file << "Network is empty, nothing to save here." << endl;
    return;
  }

  // Saves input layer information.
  out_file << "input_layer: " << v_int_layers[0] << endl;

  // For counting nodes (to detect the end of current layer).
  int node_counter = 0;

  int layer_number = 0;

  // Goes through all neurons.
  for(int neuron_number = 0;
      neuron_number < get_number_of_nodes(); neuron_number++) {

    // Checking the end of a layer.
    if(node_counter == v_int_layers[layer_number]) {
      layer_number++;
      out_file << endl << "comp_layer: " << v_int_layers[layer_number] << endl;
      node_counter = 0;
    }

    if(layer_number > 0) {

      out_file << "  weights:";

      // Saves all weights of this neuron.
      for(int connection_number = 0;
          connection_number <
            static_cast<int>(vpNeurons[neuron_number]->vpConnections.size());
          connection_number++) {

        double this_weight = vpNeurons[neuron_number]->
                             vpConnections[connection_number]->weight;

        // For keeping columns a bit more straight.
        if(abs(this_weight) < 10)
          out_file << " ";

        if(this_weight < 0)
          out_file << " " << this_weight;
        else
          out_file << "  " << this_weight;

      }
    out_file << endl;
    }

    node_counter++;

  }

  // Saves some other parameters, too (assuming that all neurons
  // have same parameters!).
```

```
  out_file << endl;
  out_file << "treshold: " << get_treshold() << endl;
  out_file << "slope_parameter: " << get_slope_parameter() << endl;
  out_file << "max_weight: " << get_max_weight() << endl;

}


// ***********************************************
// * Functions for getting some neuron parameters *
// ***********************************************

// All these functions assume, that all neurons have similar
// parameters. So they just use values taken from the first
// neuron in vpNeurons vector (which is actually an input node,
// but still should have all these parameters set correctly).
// If network is empty, then all these functions return 0.

// - Function: get_treshold - -
double NeuralNetwork::get_treshold() {

  if(get_number_of_nodes() == 0)
    return(0);

  return(vpNeurons[0]->treshold);

}

// - Function: get_slope_parameter - -
double NeuralNetwork::get_slope_parameter() {

  if(get_number_of_nodes() == 0)
    return(0);

  return(vpNeurons[0]->slope_parameter);

}

// - Function: get_learning_rate - -
double NeuralNetwork::get_learning_rate() {

  if(get_number_of_nodes() == 0)
    return(0);

  return(vpNeurons[0]->learning_rate);

}

// - Function: get_max_weight - -
// Returns the value of parameter max_weight, NOT the maximum
// weight found in network.
double NeuralNetwork::get_max_weight() {

  if(get_number_of_nodes() == 0)
    return(0);

  return(vpNeurons[0]->max_weight);

}


// *****************
// * Private stuff *
// *****************

// You can't use that stuff from outside code.
// That's why the "density" of comments is lower here...

// - Function: load_config_file - -
// Loads info from configuration file, if this file exists.
// It loads as much as it can without raising any alarms,
// so make sure your configuration file is correct.
void NeuralNetwork::load_config_file(char* config_file) {
```

```
// Opens the configuration file
ifstream inp_file(config_file);

// If no file found, then exits this function.
if(!inp_file)
  return;

// Some variables.
string inp_line_string;  // For storing a line read from file

int neuron_number = 0;  // Neuron counter for weight adding.

// Now we will go through the file, reading it line by line.
while(getline(inp_file, inp_line_string)) {

  // Converts string to stringstream for easier use.
  stringstream inp_line_stream(inp_line_string);

  // Some variables.
  string param_label = "";
  int num_of_neurons = 0;
  double inp_treshold,
         inp_slope_parameter;

  // Extracts the first non-white-space word.
  inp_line_stream >> param_label;

  // Creates input layer.
  if(param_label == "input_layer:") {
    inp_line_stream >> num_of_neurons;

    for(int i = 0; i < num_of_neurons; i++)  // Adds neurons.
      vpNeurons.push_back(new Neuron(&vpNeurons, false));

    v_int_layers.push_back(num_of_neurons);  // Adds layer size.
  }

  // Creates computational layer.
  if(param_label == "comp_layer:") {
    inp_line_stream >> num_of_neurons;

    for(int i = 0; i < num_of_neurons; i++)  // Adds neurons.
      vpNeurons.push_back(new Neuron(&vpNeurons, true));

    if(num_of_neurons > 0)      // Adds layer only if layer exists.
      v_int_layers.push_back(num_of_neurons);

    neuron_number = 0;  // Resets neuron counter for weight adding.
  }

  // Adjusts weights, if they are given (no max_weight checking!)
  if(param_label == "weights:") {

    int size_this_lr = *(v_int_layers.end() - 1);

    if( (get_number_of_layers() < 2) or // Input layer can't have connections!
        (neuron_number >= size_this_lr) )  // Too much "weights:" are given!
      break;

    int size_prev_lr = *(v_int_layers.end() - 2),
        start_of_this_lr = get_number_of_nodes() - size_this_lr,
        start_of_prev_lr = start_of_this_lr - size_prev_lr,
        connections_added = 0;

    double weight;

    while( (inp_line_stream >> weight) and
           (connections_added < size_prev_lr) ) {
      vpNeurons[start_of_this_lr + neuron_number]->
        add_connection(start_of_prev_lr + connections_added, weight);

      connections_added++;
```

```
      }

      neuron_number++;
    }

    // Sets treshold values.
    if(param_label == "treshold:")
      if(inp_line_stream >> inp_treshold)
        set_treshold(inp_treshold);

    // Sets slope_parameter values.
    if(param_label == "slope_parameter:")
      if(inp_line_stream >> inp_slope_parameter)
        set_slope_parameter(inp_slope_parameter);

  }

}

// - Function: create_unset_connections - -
// Creates missing connections so that network will be a
// fully connected multilayer network.
void NeuralNetwork::create_unset_connections() {

  // If less than two layers then no connections to create.
  if(get_number_of_layers() < 2)
    return;

  // Initializes random number generator.
  srand(time(0));

  // Pointer to the first neuron in first comp_layer.
  vector<Neuron*>::iterator i = vpNeurons.begin() + v_int_layers[0];

  int current_layer = 1,
      size_prev_lr = v_int_layers[0],
      size_this_lr = v_int_layers[1],
      start_of_prev_lr = 0,
      start_of_this_lr = size_prev_lr,
      neuron_number = 0;

  // Will go through the vpNeurons one neuron at a time.
  while(i != vpNeurons.end()) {

    // First node from previous layer without a connection to this neuron.
    int source = start_of_prev_lr +
      static_cast<int>((*i)->vpConnections.size());

    // Adds all missing connections.
    while(static_cast<int>((*i)->vpConnections.size()) < size_prev_lr) {

      // Finds a random weight in range -max_weight..+max_weight.
      double help_value = (1 - static_cast<double>(rand()) * 2 / RAND_MAX),
             rnd_weight = (*i)->max_weight * help_value;

      (*i)->add_connection(source, rnd_weight);

      source++;
    }

    neuron_number++;

    // Update the layer and neuron counting.
    if( (neuron_number == size_this_lr) and
        (current_layer < get_number_of_layers() - 1) ) {

      start_of_prev_lr += size_prev_lr;
      start_of_this_lr += size_this_lr;

      current_layer++;

      size_prev_lr = v_int_layers[current_layer - 1];
      size_this_lr = v_int_layers[current_layer];
```

```
      neuron_number = 0;

    }

    i++;

  }

}

// - Function: load_training_file - -
// Loads info from training file. Returns the number of loading errors.
// But be aware that zero errors does not guarantee that everything is
// OK. Use verify_all_patterns(), too.
int NeuralNetwork::load_training_file(char* training_file) {

  // Opens the training file.
  ifstream inp_file(training_file);

  // If no file found, then exits this function.
  if(!inp_file)
    return(1);

  // Some variables.
  int loading_errors = 0;  // How many errors found in training file.
  string inp_line_string;  // For storing a line read from file

  // Now we will go through the file, reading it line by line.
  while(getline(inp_file, inp_line_string)) {

    // Converts string to stringstream for easier use.
    stringstream inp_line_stream(inp_line_string);

    // Some variables.
    string param_label = "";
    int inp_max_iterations;
    double inp_error_limit,
           inp_learning_rate,
           inp_max_weight;

    // Extracts the first non-white-space word.
    inp_line_stream >> param_label;

    // Creates a pattern.
    if(param_label == "pattern:") {
      Pattern ConstrPattern;
      bool inputs = true;  // First numbers represent inputs.
      string word;

      // Goes through the line word by word.
      while(inp_line_stream >> word) {

        stringstream helper_stream(word),
                     helper_stream_copy(word);

        double number;

        if(helper_stream >> number) {
          if(inputs)
            ConstrPattern.vInputs.push_back(number);
          else
            ConstrPattern.vOutputs.push_back(number);
        }
        else {
          helper_stream_copy >> word;

          if(word == "->")
            inputs = false;  // After "->" will come outputs.
          else
            loading_errors++;
        }
      }
```

```
      add_training_pattern(ConstrPattern);
    }

    // Sets error_limit value.
    if(param_label == "error_limit:")
      if(inp_line_stream >> inp_error_limit)
        error_limit = inp_error_limit;

    // Sets max_iterations value.
    if(param_label == "max_iterations:")
      if(inp_line_stream >> inp_max_iterations)
        max_iterations = inp_max_iterations;

    // Sets learning_rate values.
    if(param_label == "learning_rate:")
      if(inp_line_stream >> inp_learning_rate)
        set_learning_rate(inp_learning_rate);

    // Sets max_weight values.
    if(param_label == "max_weight:")
      if(inp_line_stream >> inp_max_weight)
        set_max_weight(inp_max_weight);

  }

  return(loading_errors);

}
```

In this file you can describe your network.

****
* Network structure *
****

Without this section your NeuralNetwork object will have
NO neurons (unless you modify source code) and is therefore
quite useless!

Currently it is possible to create only a fully connected
multilayer feedforward network. To do that, just answer
some easy questions below.

First layer in network is input layer. It consists of simple
buffer nodes that do NOT compute anything, but only hold
input values. How many input nodes your network must have?

```
  input_layer: 2
```

Other layers consist of computing neurons. How many nodes
should be in computational layers starting from the input
layer side of the network? For adding a layer just add a
line with the keyword "comp_layer:". The last computational
layer will also act as the output layer. Layers without
any neurons will be ignored.

```
  comp_layer: 4
  comp_layer: 4
  comp_layer: 1
```

****
* Neuron parameters *
****

Work of an artificial neural network depends on neuron parameters.
If you have no idea what they should be, then you may set them
to default values (which are given in help texts before
each parameter). Or just leave them empty (delete the
numbers after each parameter identifier), then they will be
automatically set to default values. Actually it is even allowed
to delete this section "Neuron parameters" completely from this
file and leave only the "Network structure" section. Then
everything will be set to default as well. But make a backup
of this file first ;)

Currently it is possible to use only one type of neuron.
Its activation function is sigmoidal (logistic function
1 / (1 + exp(-av)) where "a" is slope parameter and "v"
is internal activity level).

- Parameter: treshold. Type: double. Default value: 0 - -
Treshold (or bias) value is used for lowering or increasing
the sum (internal activity level) before applying the activation
function.

```
  treshold: 0
```

- Parameter: slope_parameter. Type: double. Default value: 0.5 - -
It is the slope parameter of the sigmoid function. When slope
parameter approaches infinity, sigmoid function becomes simply a
treshold function (do not confuse with treshold parameter, which was
described above).

```
  slope_parameter: 0.5
```

****
* Additional info *
****

It is also possible in this file to set max_weight parameter
(normally you should do it in training configuration file) and
all weights of connections. However, in normal circumstances
you shouldn't do it here. If you want to set all weights manually,
then first generate a network with needed structure, then edit this
generated file according to your needs. You can then use the
generated and edited file as a config file for creating
a new network object.

Note: you will see that generated configuration file contains
max_weight parameter. In most cases it is not used during the
normal work of neural network, but some rare applications may
use it (like my neural network visualizer). Therefore when you
change the weights manually, consider if it is also necessary
to change max_weight value.

Typical backpropagation algorithm is used for training.
In this file you can configure its parameters.

****
* Training patterns *
****

If you are going to train your network, you MUST first
define all training patterns. The easiest way is to describe
them is to do it in this file.

Give as much training patterns as necessary, using format:
"pattern: inputs -> desired outputs". Note that number of
inputs and outputs in patterns must match your network
structure given in configuration file.

```
  pattern: 0 0 -> 0
  pattern: 0 1 -> 1
  pattern: 1 0 -> 1
  pattern: 1 1 -> 0
```

****
* Training related parameters *
****

Process of training depends on several parameters. If you
have no idea what they should be, then you may set them
to default values (which are given in help texts before
each parameter). Or just leave them empty (delete the
numbers after each parameter identifier), then they will be
automatically set to default values. Actually it is even allowed
to delete this section "Training related parameters" completely
from this file and leave only the "Training patterns" section.
Then everything will be set to default as well. But make a
backup of this file first ;)

- Parameter: error_limit. Type: double. Default value: 0.1 - -
Error limit is used for evaluating the success of training.
If all outputs are in a range from (desired_output - error_limit)
to (desired_output + error_limit), then training is completed.

```
  error_limit: 0.1
```

- Parameter: max_iterations. Type: int. Default value: 20000 - -
When this number of iterations is reached, training will be
stopped and is failed (i.e. outputs were still outside the
given error limits).

```
  max_iterations: 20000
```

- Parameter: learning_rate. Type: double. Default value: 5 - -
How fast should network learn? Too small values make
learning too slow. Too big may halt the learning process.

```
  learning_rate: 5
```

- Parameter: max_weight. Type: double. Default value: 10 - -
Maximal allowed weight for connections. Weights will be
in range -max_weight..+max_weight.

```
  max_weight: 10
```

/trainer

```cpp
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: main.cpp                                             |
// Purpose: Program for creating and training neural nets     |
// Author: Taivo Lints, Estonia                               |
// Date: May, 2003                                            |
// Copyright: see copyright.txt                               |
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

#include "ann/neuralnetwork.h"
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {

  // Variables for storing filenames.
  char* config_file = "configuration.txt";
  char* training_file = "trainingfile.txt";
  char* output_file = "config_generated.txt";

  // Gives some information.
  cout << endl;
  cout << " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -" << endl;
  cout << " This program can be called without any arguments," << endl;
  cout << " or you can call it this way:" << endl << endl;
  cout << "  trainer.exe configuration.txt trainingfile.txt [output.txt]";
  cout << endl << endl << " where [output.txt] is optional" << endl;
  cout << " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -";
  cout << endl << endl;

  // Check command line arguments for configuration and training files.
  if(argc < 3) {
    cout << "Program called with less than two arguments." << endl;
    cout << "Will use default files " << config_file << " and " << training_file;
    cout << endl << endl;
  }
  else {
    config_file = argv[1];
    training_file = argv[2];
    cout << "Configuration file: " << config_file << endl;
    cout << "Training_file: " << training_file << endl << endl;
  }

  // Checks command line arguments for output file name.
  if(argc < 4) {
    cout << "Output file name not given in command line, will use default:" << endl;
    cout << " " << output_file << endl << endl;
  }
  else {
    output_file = argv[3];
    cout << "Output file: " << output_file << endl << endl;
  }

  // Creates NeuralNetwork object.
  NeuralNetwork myNet(config_file);

  // If network empty, then nothing to do.
  if(myNet.get_number_of_inputs() == 0) {
    cout << "Empty network, nothing to do." << endl;
    cout << "Make sure that configuration file exists and is correct!" << endl;
    exit(1);
  }

  // Trains network.
  cout << endl << "Training the network, PLEASE WAIT..." << endl << endl;

  switch(myNet.train(training_file)) {
    case 0: cout << "Training completed after " <<
              myNet.iterations << " iterations." << endl <<
              "Will save the network..." << endl; break;
    case 1: cout << "Training not completed even after " <<
              myNet.iterations << " iterations!" << endl <<
              "Will save the UNFINISHED network..." << endl; break;
    case 2: cout << "Patterns file missing or patterns incorrect!" << endl; break;
```

```
    default: cout << "Something is wrong with training function." << endl;
  }

  // Saves network.
  myNet.save_network(output_file);
  cout << "Network saved." << endl;

}
```

In this file you can describe your network.

****
* Network structure *
****

Without this section your NeuralNetwork object will have
NO neurons (unless you modify source code) and is therefore
quite useless!

Currently it is possible to create only a fully connected
multilayer feedforward network. To do that, just answer
some easy questions below.

First layer in network is input layer. It consists of simple
buffer nodes that do NOT compute anything, but only hold
input values. How many input nodes your network must have?

```
  input_layer: 2
```

Other layers consist of computing neurons. How many nodes
should be in computational layers starting from the input
layer side of the network? For adding a layer just add a
line with the keyword "comp_layer:". The last computational
layer will also act as the output layer. Layers without
any neurons will be ignored.

```
  comp_layer: 4
  comp_layer: 4
  comp_layer: 1
```

****
* Neuron parameters *
****

Work of an artificial neural network depends on neuron parameters.
If you have no idea what they should be, then you may set them
to default values (which are given in help texts before
each parameter). Or just leave them empty (delete the
numbers after each parameter identifier), then they will be
automatically set to default values. Actually it is even allowed
to delete this section "Neuron parameters" completely from this
file and leave only the "Network structure" section. Then
everything will be set to default as well. But make a backup
of this file first ;)

Currently it is possible to use only one type of neuron.
Its activation function is sigmoidal (logistic function
1 / (1 + exp(-av)) where "a" is slope parameter and "v"
is internal activity level).

- Parameter: treshold. Type: double. Default value: 0 - -
Treshold (or bias) value is used for lowering or increasing
the sum (internal activity level) before applying the activation
function.

```
  treshold: 0
```

- Parameter: slope_parameter. Type: double. Default value: 0.5 - -
It is the slope parameter of the sigmoid function. When slope
parameter approaches infinity, sigmoid function becomes simply a
treshold function (do not confuse with treshold parameter, which was
described above).

```
  slope_parameter: 0.5
```

****
* Additional info *
****

It is also possible in this file to set max_weight parameter
(normally you should do it in training configuration file) and
all weights of connections. However, in normal circumstances
you shouldn't do it here. If you want to set all weights manually,
then first generate a network with needed structure, then edit this
generated file according to your needs. You can then use the
generated and edited file as a config file for creating
a new network object.

Note: you will see that generated configuration file contains
max_weight parameter. In most cases it is not used during the
normal work of neural network, but some rare applications may
use it (like my neural network visualizer). Therefore when you
change the weights manually, consider if it is also necessary
to change max_weight value.

Typical backpropagation algorithm is used for training.
In this file you can configure its parameters.

****
* Training patterns *
****

If you are going to train your network, you MUST first
define all training patterns. The easiest way is to describe
them is to do it in this file.

Give as much training patterns as necessary, using format:
"pattern: inputs -> desired outputs". Note that number of
inputs and outputs in patterns must match your network
structure given in configuration file.

```
  pattern: 0 0 -> 0
  pattern: 0 1 -> 1
  pattern: 1 0 -> 1
  pattern: 1 1 -> 0
```

****
* Training related parameters *
****

Process of training depends on several parameters. If you
have no idea what they should be, then you may set them
to default values (which are given in help texts before
each parameter). Or just leave them empty (delete the
numbers after each parameter identifier), then they will be
automatically set to default values. Actually it is even allowed
to delete this section "Training related parameters" completely
from this file and leave only the "Training patterns" section.
Then everything will be set to default as well. But make a
backup of this file first ;)

- Parameter: error_limit. Type: double. Default value: 0.1 - -
Error limit is used for evaluating the success of training.
If all outputs are in a range from (desired_output - error_limit)
to (desired_output + error_limit), then training is completed.

```
  error_limit: 0.1
```

- Parameter: max_iterations. Type: int. Default value: 20000 - -
When this number of iterations is reached, training will be
stopped and is failed (i.e. outputs were still outside the
given error limits).

```
  max_iterations: 20000
```

- Parameter: learning_rate. Type: double. Default value: 5 - -
How fast should network learn? Too small values make
learning too slow. Too big may halt the learning process.

```
  learning_rate: 5
```

- Parameter: max_weight. Type: double. Default value: 10 - -
Maximal allowed weight for connections. Weights will be
in range -max_weight..+max_weight.

```
  max_weight: 10
```

/visualizer

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: main.cpp                                                        |
// Purpose: program for visualizing artificial neural networks           |
// Author: Taivo Lints, Estonia                                          |
// Date: May, 2003                                                       |
// Copyright: see copyright.txt                                          |
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

#include "allegro.h"
#include "visnet/visual_ann.h"
#include "visnet/mypalette.h"
#include "stdio.h"  // For sprintf() and strcat() (Allegro needs char* -s)
#include <vector>
using namespace std;

int main() {

  // ******************************
  // * Setting up the environment *
  // ******************************

  allegro_init();
  install_keyboard();
  install_mouse();
  install_timer();

  set_color_depth(8);
  set_gfx_mode(GFX_AUTODETECT, 800, 600, 0, 0);

  // Configures the text to be drawn with transparent background.
  text_mode(-1);

  // Sets the color of warning texts.
  int color_of_wtxt = 90;
  change_color(color_of_wtxt, 63, 5, 0);

  // Sets the color of notice texts.
  int color_of_ntxt = 91;
  change_color(color_of_ntxt, 55, 45, 10);

  // Sets colors for file loading menu.
  int color_of_gui_bg = 92,
      color_of_gui_fg = 93;

  change_color(color_of_gui_bg, 0, 0, 20);
  change_color(color_of_gui_fg, 0, 50, 0);

  gui_bg_color = color_of_gui_bg;
  gui_fg_color = color_of_gui_fg;

  // Sets colors for help background.
  int color_of_help_bg = 94;
  change_color(color_of_help_bg, 0, 0, 10);

  // Creates a bitmap object for double buffering (to avoid screen flickering).
  BITMAP* dblbuf = create_bitmap(SCREEN_W, SCREEN_H);
  clear_bitmap(dblbuf);


  // ********************************
  // * Initializing other variables *
  // ********************************

  // Initializes variables for loading the configuration file.
  char config_file[500];
  sprintf(config_file, "config_generated.txt");

  char message[] = "Load new configuration file";  // Message for load menu.
  char* ext = "TXT";   // File filter for load menu.

  char help_msg1[150];  // Creating some help messages.
  sprintf(help_msg1, "File ");
  strcat(help_msg1, config_file);
```

```
strcat(help_msg1, " is missing");

char help_msg2[150];
sprintf(help_msg2, "Quit, fix ");
strcat(help_msg2, config_file);
strcat(help_msg2, " and restart program");

// Creates a visual network object.
VisualANN* pVisNet = new VisualANN(config_file);

// Gets the number of input nodes in network. This value is needed
// for processing the user input.
int num_of_inps = pVisNet->get_number_of_inputs();

// How many of these inputs are accessible by user? If you change this
// number, you MUST also change the code in "Processing user interaction"
// (in the "switch" statement).
int user_accessible_inps = 10;

// Creates a vector for feeding inputs into the network (vector size
// is num_of_inps and vector is filled with zeros).
vector<double> v_dbl_inpvalues(num_of_inps, 0);

// Now cuts down the num_of_inps value because of the way this
// value is used in a "switch" statement later on.
if(num_of_inps > user_accessible_inps)
  num_of_inps = user_accessible_inps;

// Some flags noticing when to print additional help for user.
bool flag_no_inputs = false,
     flag_just_started = true;

if(num_of_inps == 0)
  flag_no_inputs = true;

// Program loops while this flag is "true".
bool keepgoing = true;


// *****************
// * The main loop *
// *****************

while(keepgoing) {

  //  ----------------------------
  // | Updating & drawing network |
  //  ----------------------------

  pVisNet->update();
  pVisNet->draw(dblbuf);


  //  -------------------------------
  // | Printing some additional help |
  //  -------------------------------

  if(flag_no_inputs) {
    textout(dblbuf, font, "Network has no inputs!", 2, 10, color_of_wtxt);

    if(flag_just_started) {
      textout(dblbuf, font, "Possible reasons:", 15, 30, color_of_wtxt);
      textout(dblbuf, font, help_msg1, 35, 42, color_of_wtxt);
      textout(dblbuf, font, "or is incorrect", 35, 54, color_of_wtxt);
      textout(dblbuf, font, "Solutions:", 15, 74, color_of_wtxt);
      textout(dblbuf, font, help_msg2, 35, 86, color_of_wtxt);
      textout(dblbuf, font,
        "or press L and load another configuration file", 35, 98, color_of_wtxt);
    }
    else {
      textout(dblbuf, font, "Possible reasons:", 15, 30, color_of_wtxt);
      textout(dblbuf, font, "Loaded file is incorrect", 35, 42, color_of_wtxt);
      textout(dblbuf, font,
```

```
      "or path contains non-english characters", 35, 54, color_of_wtxt);
    textout(dblbuf, font, "Solutions:", 15, 74, color_of_wtxt);
    textout(dblbuf, font, "Fix your file and reload it", 35, 86, color_of_wtxt);
    textout(dblbuf, font,
      "or press L and load another configuration file", 35, 98, color_of_wtxt);
    textout(dblbuf, font,
      "or in case of incorrect path move your file into", 35, 110, color_of_wtxt);
    textout(dblbuf, font, "correct path and load again", 35, 122, color_of_wtxt);
    textout(dblbuf, font, "Your selected path was:", 15, 140, color_of_ntxt);
    textout(dblbuf, font, config_file, 35, 152, color_of_ntxt);
  }
}


//  ----------------------------
// | Processing user interaction |
//  ----------------------------

// - - Setting network inputs - - -

// Gets mouse y value and normalizes it.
double vertical_input = static_cast<double>(mouse_y);
vertical_input = 1 - vertical_input / (SCREEN_H - 1);

// Now here's an interesting statement. It sets values for feeding
// into network. To avoid crashing when there are less than 10 inputs
// in network (by indexing out of the vector), it jumps / switches
// to the right place. But as there are no "break"-s at the ends of
// lines, then it will go all the way down through "case"-s following
// the one it jumped to.

switch(num_of_inps) {
  case 10 : if(key[KEY_0]) v_dbl_inpvalues[9] = vertical_input;
  case  9 : if(key[KEY_9]) v_dbl_inpvalues[8] = vertical_input;
  case  8 : if(key[KEY_8]) v_dbl_inpvalues[7] = vertical_input;
  case  7 : if(key[KEY_7]) v_dbl_inpvalues[6] = vertical_input;
  case  6 : if(key[KEY_6]) v_dbl_inpvalues[5] = vertical_input;
  case  5 : if(key[KEY_5]) v_dbl_inpvalues[4] = vertical_input;
  case  4 : if(key[KEY_4]) v_dbl_inpvalues[3] = vertical_input;
  case  3 : if(key[KEY_3]) v_dbl_inpvalues[2] = vertical_input;
  case  2 : if(key[KEY_2]) v_dbl_inpvalues[1] = vertical_input;
  case  1 : if(key[KEY_1]) v_dbl_inpvalues[0] = vertical_input;
  default:;
}

// And finally feeds the vector into network.
pVisNet->set_inputs(&v_dbl_inpvalues);


// - - Loading new network - - -

if(key[KEY_L]) {

  // Calls file selection menu.
  int i = file_select_ex(message, config_file, ext, 490, 450, 500);
  rest(250);  // In case menu was exited by pressing Esc, this avoids
              // quitting the program.

  // If user chose something, then makes a new network of it.
  if(i != 0) {
    delete pVisNet;
    pVisNet = new VisualANN(config_file);

    num_of_inps = pVisNet->get_number_of_inputs();

    v_dbl_inpvalues.assign(num_of_inps, 0);  // Resize & write zeros.

    if(num_of_inps > user_accessible_inps)
      num_of_inps = user_accessible_inps;

    flag_just_started = false;

    if(num_of_inps == 0)
```

```
          flag_no_inputs = true;
        else
          flag_no_inputs = false;
      }
    }

    // - - Other user interactions - - -

    // Press Escape to exit.
    if(key[KEY_ESC]) {
      text_mode(0);
      textout(screen, font, "ESC pressed, will QUIT", 2, 2, color_of_wtxt);
      rest(2000);
      keepgoing = false;
    }

    // Press F1 for help.
    if(key[KEY_F1]) {
      rectfill(screen, 0, 0, 385, 164, color_of_help_bg);

      textout(screen, font, "Keys:", 5, 8, color_of_ntxt);
      textout(screen, font, "1 + mouse up/down  - change first input", 15, 30, color_
      textout(screen, font, "2 + mouse up/down  - change second input", 15, 42, colo
      textout(screen, font, ".. + mouse up/down - change .. input", 15, 54, color_of_
      textout(screen, font, "L    - load network", 15, 78, color_of_ntxt);
      textout(screen, font, "F1   - this help", 15, 102, color_of_ntxt);
      textout(screen, font,
        "F12 - save screenshot (does NOT work in menus)", 15, 114, color_of_ntxt);
      textout(screen, font, "Esc - quit program", 15, 126, color_of_ntxt);
      textout(screen, font, "Press any key to leave help", 15, 150, color_of_ntxt);

      rest(100);
      clear_keybuf();
      while(!keypressed());
      rest(250);
    }

    // Press F12 to save screenshot.
    if(key[KEY_F12]) {
      PALETTE pal;
      get_palette(pal);
      save_bitmap("screenshot.bmp", dblbuf, pal);
      textout(screen, font, "Screenshot saved", 2, 2, color_of_ntxt);
      rest(2000);
    }


    //  ----------
    // | Blitting |
    //  ----------

    // Copies the dblbuf into video memory at the right moment
    // (when vertical retrace is beginning in monitor).
    vsync();
    blit(dblbuf, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
    clear_bitmap(dblbuf);

  }


  // ***************
  // * Cleaning up *
  // ***************

  destroy_bitmap(dblbuf);

}

END_OF_MAIN();
```

```
input_layer: 2

comp_layer: 4
  weights:  -8.1807158268899407  10.0000000000000000
  weights:  10.0000000000000000  -6.1846947176681555
  weights:  -5.8802124208199507   7.2573985050033283
  weights:  10.0000000000000000  -9.8721825535036967

comp_layer: 4
  weights:   9.9998795745896309 -10.0000000000000000  -6.3472089931307050 -10.000000(
  weights:  -4.1915042285789372   1.6162414096714324  -7.4165880865987779  -7.525808(
  weights: -10.0000000000000000  -1.4211829303745751 -10.0000000000000000   9.999807(
  weights:  -3.5083865720635061  10.0000000000000000  -0.4871684440449007  10.000000(

comp_layer: 1
  weights:   9.9999660751028241   7.6672206522919426   9.9999413773041130  -5.726346(

treshold: 0.0000000000000000
slope_parameter: 0.5000000000000000
max_weight: 10.0000000000000000
```

/visualizer
/visnet

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: mypalette.h                                                     |
// Purpose: provides interface for palette modification functions        |
// Author: Taivo Lints, Estonia                                          |
// Date: March, 2003                                                     |
// Copyright: see copyright.txt                                          |
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

#ifndef MYPALETTE_H
#define MYPALETTE_H

// Changes colors start..start + 63 to a gradient (from black
// to white), others remain the same.
void make_grayscale_palette_64(int start);

// Changes colors start..start + 63 to a gradient (from black
// to some kind of blue-green), others remain the same.
void make_blueish_palette_64(int start);

// Changes the color of a palette entry (with index color_nr).
void change_color(int color_nr, int r, int g, int b);

#endif // MYPALETTE_H
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: visual_ann.h                                 |
// Purpose: provides interface for class VisualANN    |
// Author: Taivo Lints, Estonia                       |
// Date: May, 2003                                    |
// Copyright: see copyright.txt                       |
// - - - - - - - - - - - - - - - - - - - - - - - - - -

#ifndef VISUAL_ANN_H
#define VISUAL_ANN_H

#include "ann/neuralnetwork.h"
#include "allegro.h"
#include <vector>
using namespace std;

class VisualANN : public NeuralNetwork {
                      // Adds some functionality to NeuralNetwork class,
                      // namely the ability to visualize itsself.
  public:

    // ******************************
    // * Construction & Destruction *
    // ******************************

    VisualANN(char* config_file);
                 // VisualANN constructor. Creates a network from your
                 // configuration file (using NeuralNetwork constructor).


    // ********************
    // * Some parameters *
    // ********************

    int neuron_radius;             // Size of the circle representing a neuron.
                                   // Default is 15 and it is reduced
                                   // automatically in constructor when neurons
                                   // are too close to each other.

                                   // After construction there is no automatical
                                   // reduction anymore (i.e. when you change it
                                   // manually).

    int edge_x;                    // How far should neurons be from the left
                                   // and right edges of bitmap.

    // *************************
    // * Functions for drawing *
    // *************************

    void draw(BITMAP* pBitmap);    // Draws network on given bitmap (which
                                   // should be the right size, i.e. the
                                   // size of screen when VisualANN object
                                   // was created).


  // *****************
  // * Private stuff *
  // *****************

  // You can't use that stuff from outside code.

  private:

    class Position {               // Just a useful class for storing the
      public:                      // position of something (e.g. a neuron).
        int x,
            y;
    };

    vector<Position> vPositions;   // Contains the positions of neurons,
                                   // exactly mirroring vpNeurons vector
                                   // (i.e. position of the first neuron in
```

```
                                    // vpNeurons is stored first in vPositions,
                                    // second is second, etc.)

    // Some colors (on the palette). // NB! Visualizer needs 8 bit color depth.
    int color_of_pin;                // Palette colors 100 and higher are used
    int color_of_txt;                // here, so please don't change them in
    int color_of_node_circle;        // your own program.
    int start_of_blue;
    int start_of_white;

    // Some network parameters duplicated for quicker / easier access.
    int num_of_nodes;
    int second_layer_start;  // If second layer doesn't exist, then it will be
                             // set to num_of_nodes value.

    int output_layer_start;  // If input layer is the only one, then it will
                             // be set to 0, because then input layer is also
                             // output layer.
    double max_weight;

};

#endif // VISUAL_ANN_H
```

```cpp
// - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: mypalette.cpp                                |
// Purpose: implements palette modification functions |
// Author: Taivo Lints, Estonia                       |
// Date: March, 2003                                  |
// Copyright: see copyright.txt                       |
// - - - - - - - - - - - - - - - - - - - - - - - - - -

#include "mypalette.h"
#include "allegro.h"
using namespace std;

// - Function: make_grayscale_palette_64 - -
// Changes colors start..start + 63 to a gradient (from black
// to white), others remain the same.
void make_grayscale_palette_64(int start) {
  RGB rgb;

  for(int i = start; i < start + 64; i++) {
    if(i > 255) break;
    rgb.r = rgb.g = rgb.b = i - start;
    set_color(i, &rgb);
  }
}

// - Function: make_blueish_palette_64 - -
// Changes colors start..start + 63 to a gradient (from black
// to some kind of blue-green), others remain the same.
void make_blueish_palette_64(int start) {
  RGB rgb;

  for(int i = start; i < start + 64; i++) {
    if(i > 255) break;

    rgb.r = 0;
    rgb.g = static_cast<int>(static_cast<float>(i - start) / 1.5);
    rgb.b = i - start;
    set_color(i, &rgb);
  }
}

// - Function: change_color - -
// Changes the color of a palette entry (with index color_nr).
void change_color(int color_nr, int r, int g, int b) {
  RGB rgb;

  rgb.r = r;
  rgb.g = g;
  rgb.b = b;
  set_color(color_nr, &rgb);
}
```

```
// - - - - - - - - - - - - - - - - - - - -
// File: visual_ann.cpp                   |
// Purpose: implements class VisualANN    |
// Author: Taivo Lints, Estonia           |
// Date: May, 2003                        |
// Copyright: see copyright.txt           |
// - - - - - - - - - - - - - - - - - - - -

#include "visual_ann.h"
#include "ann/neuralnetwork.h"
#include "ann/connection.h"
#include "mypalette.h"
#include "allegro.h"
#include "stdio.h"  // For sprintf()
#include <vector>
using namespace std;

// - - - - - Class VisualANN - - - - -

// ******************************
// * Construction & Destruction *
// ******************************

// - VisualANN constructor - -
// VisualANN constructor. Creates a network from your configuration
// file (using NeuralNetwork constructor).
VisualANN::VisualANN(char* config_file) : NeuralNetwork(config_file) {

  // Setting colors on palette.
  color_of_pin = 100;
  color_of_txt = 101;
  color_of_node_circle = 102;
  start_of_blue = 120;
  start_of_white = 190;

  change_color(color_of_pin, 0, 14, 48);
  change_color(color_of_node_circle, 0, 45, 0);
  change_color(color_of_txt, 2, 35, 10);
  make_blueish_palette_64(start_of_blue);
  make_grayscale_palette_64(start_of_white);

  // Some network parameters are duplicated for quicker / easier access.
  num_of_nodes = get_number_of_nodes();
  second_layer_start = get_number_of_inputs();
  output_layer_start = num_of_nodes - get_number_of_outputs();
  max_weight = get_max_weight();

  // Drawing related parameters.
  neuron_radius = 15;
  edge_x = 50;

  // If network is empty, then nothing to do.
  if(get_number_of_layers() == 0)
    return;

  // How far should neurons be from lower and upper edges of bitmap?
  int edge_y = 2 * neuron_radius;

  // How far should they be from each other in x direction?
  double gap_x;
  if(get_number_of_layers() > 1) {
    gap_x = static_cast<double>(SCREEN_W - 2 * edge_x) /
                                (get_number_of_layers() - 1);

    // If neuron circles are horizontally too close, then reduces neuron_radius.
    if(gap_x < 3 * neuron_radius)
    neuron_radius = static_cast<int>(gap_x / 3 + 4);
  }

  // Variable for storing the appropriate vertical gap between neurons.
  double gap_y = 0;

  // This will remember the smallest vertical cap between neurons and is
```

```
  // used to keep vertical gap between neurons reasonable (by reducing
  // neuron radius).
  int min_gap_y = SCREEN_H;

  // For storing the position of a neuron.
  Position Pos;

  // Goes through all layers.
  for(int i = 0; i < get_number_of_layers(); i++) {

    // How many nodes in this layer?
    int nodes_in_lr = v_int_layers[i];

    // For gap_y there must be at least two nodes.
    if(nodes_in_lr > 1)
      gap_y = static_cast<double>(SCREEN_H - 2 * edge_y) / (nodes_in_lr - 1);

    // Updates min_gap_y.
    if( (gap_y < min_gap_y) and (gap_y != 0) )
      min_gap_y = static_cast<int>(gap_y);

    // x is same for all neurons in one layer.
    Pos.x = static_cast<int>(edge_x + i * gap_x);

    // Stores positions for all neurons in this layer.
    for(int j = 0; j < nodes_in_lr; j++) {
      Pos.y = static_cast<int>(edge_y + j * gap_y);
      vPositions.push_back(Pos);
    }
  }

  // If neuron circles are vertically too close, then reduces neuron_radius.
  if(min_gap_y < 3 * neuron_radius)
    neuron_radius = min_gap_y / 3 + 4;

}

// *************
// * Functions *
// *************

// - Function: draw - -
// Draws network on given bitmap.
void VisualANN::draw(BITMAP* pBitmap) {

  // A buffer for storing some text (output numbers).
  char buf[80] = "";

  for(int i = 0; i < num_of_nodes; i++) {

    // Position of this neuron.
    int x = vPositions[i].x,
        y = vPositions[i].y;

    // Output of this neuron.
    double output = vpNeurons[i]->output;

    // Calculates color of node.
    int color_of_node;
    if(output < 0)
      color_of_node = static_cast<int>(start_of_blue - output * 63);
    else
      color_of_node = static_cast<int>(start_of_white + output * 63);

    // If input layer, then will draw the input pins,
    // and circle color is also different.
    if(i < second_layer_start) {
      rectfill(pBitmap, 0, y - 1, x, y + 1, color_of_pin);

      // If we have ONLY input layer, then must also draw ouput pins.
      if(output_layer_start == 0)
        rectfill(pBitmap, x, y - 1, SCREEN_W - 1, y + 1, color_of_pin);
```

```
      circlefill(pBitmap, x, y, neuron_radius - 1, color_of_node);
      circle(pBitmap, x, y, neuron_radius, color_of_pin);
    }
    else {
      if(i >= output_layer_start) {  // Output layer needs output pins.
        rectfill(pBitmap, x, y - 1, SCREEN_W - 1, y + 1, color_of_pin);
      }

      // All non-input layer nodes are drawn here.
      circlefill(pBitmap, x, y, neuron_radius - 1, color_of_node);
      circle(pBitmap, x, y, neuron_radius, color_of_node_circle);
    }

    // Prints output value under node.
    sprintf(buf, "%f", output);
    textout(pBitmap, font, buf, x - 15, y + neuron_radius, color_of_txt);

    // Draws connections (except for input layer).
    if(i >= second_layer_start) {

      for(vector<Connection*>::iterator iter = vpNeurons[i]->
          vpConnections.begin() ; iter != vpNeurons[i]->vpConnections.end();
          iter++) {

        //Calculates color of connection.
        int col_con;
        double weight = (*iter)->weight;
        if(weight < 0)
          col_con= static_cast<int>(start_of_blue - weight / max_weight * 63);
        else
          col_con = static_cast<int>(start_of_white + weight / max_weight * 63);

        // Source coordinates.
        int sx = vPositions[(*iter)->source].x;
        int sy = vPositions[(*iter)->source].y;

        // Draws the connection.
        line(pBitmap, sx, sy, x, y, col_con);
      }
    }

  }
}
```

/lightchaser

```
// - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: main.cpp                                     |
// Purpose: Program for testing neural nets           |
//          in a light-sensitive moving creature      |
// Author: Taivo Lints, Estonia                       |
// Date: May, 2003                                    |
// Copyright: see copyright.txt                       |
// - - - - - - - - - - - - - - - - - - - - - - - - - -

#include "creature/creature.h"
#include "light/light.h"
#include "palette/mypalette.h"
#include "allegro.h"
#include "stdio.h"  // For sprintf() and strcat() (Allegro needs char* -s)
#include <vector>
using namespace std;

int main() {

  // ******************************
  // * Setting up the environment *
  // ******************************

  allegro_init();
  install_keyboard();
  install_mouse();
  install_timer();

  set_color_depth(8);
  set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0);

  // Configures the text to be drawn with transparent background.
  text_mode(-1);

  // Sets the color of warning texts.
  int color_of_wtxt = 90;
  change_color(color_of_wtxt, 63, 5, 0);

  // Sets the color of notice texts.
  int color_of_ntxt = 91;
  change_color(color_of_ntxt, 55, 45, 10);

  // Sets colors for file loading menu.
  int color_of_gui_bg = 92,
      color_of_gui_fg = 93;

  change_color(color_of_gui_bg, 0, 0, 20);
  change_color(color_of_gui_fg, 0, 50, 0);

  gui_bg_color = color_of_gui_bg;
  gui_fg_color = color_of_gui_fg;

  // Sets colors for help background.
  int color_of_help_bg = 94;
  change_color(color_of_help_bg, 0, 0, 10);

  // Sets the colors of creature.
  int color_of_icircle = 95,
      color_of_ocircle = 96;
  change_color(color_of_icircle, 0, 0, 63);
  change_color(color_of_ocircle, 0, 58, 0);

  // Creates gradient for light.
  int start_of_gradient = 100;
  make_grayscale_palette_64(start_of_gradient);

  // Creates a bitmap object for double buffering (to avoid screen flickering).
  BITMAP* dblbuf = create_bitmap(SCREEN_W, SCREEN_H);
  clear_bitmap(dblbuf);


  // ********************************
  // * Initializing other variables *
```

```
// ********************************

// Initializes variables for loading the configuration file.
char config_file[500];
sprintf(config_file, "config_generated.txt");

char message[] = "Load new configuration file";  // Message for load menu.
char* ext = "TXT";   // File filter for load menu.

char help_msg1[150];  // Creating some help messages.
sprintf(help_msg1, "File ");
strcat(help_msg1, config_file);
strcat(help_msg1, " is missing");

char help_msg2[150];
sprintf(help_msg2, "Quit, fix ");
strcat(help_msg2, config_file);
strcat(help_msg2, " and restart program");


// Creates a Creature object.
Creature* pCreature= new Creature(dblbuf, color_of_icircle, color_of_ocircle,
                                  start_of_gradient, config_file);

// Variables for tuning creature's speed.
const double default_speedup_factor = pCreature->speedup_factor;
double change_of_speedup_factor = 0.01;

// Creates a Light object.
Light* pLight = new Light(dblbuf, start_of_gradient);

// Variable for changing spotlight's speed and radius.
double change_of_lightspeed = 0.1;
int change_of_radius = 5;

// Variables for storing mouse information;
int mickey_x = 0,
    mickey_y = 0;

// A flag noticing what kind of additional help to print for user.
bool flag_just_started = true;

// Program loops while this flag is "true".
bool keepgoing = true;


// *****************
// * The main loop *
// *****************

while(keepgoing) {

  //  ----------------------------
  // | Updating & drawing objects |
  //  ----------------------------

  // Done first because these objects should be behind any texts
  // that may be printed later.

  pLight->update();
  pLight->draw();

  pCreature->update();
  pCreature->draw();


  //  ------------------------------
  // | Printing some additional help |
  //  ------------------------------

  if(pCreature->flag_incor_net) {
    textout(dblbuf, font, "Network is incorrect!", 2, 10, color_of_wtxt);
```

```
  if(flag_just_started) {
    textout(dblbuf, font, "Possible reasons:", 15, 30, color_of_wtxt);
    textout(dblbuf, font, help_msg1, 35, 42, color_of_wtxt);
    textout(dblbuf, font,
     "or is incorrect (wrong number of inputs or outputs)", 35, 54, color_of_wtxt
    textout(dblbuf, font, "Solutions:", 15, 74, color_of_wtxt);
    textout(dblbuf, font, help_msg2, 35, 86, color_of_wtxt);
    textout(dblbuf, font,
     "or press L and load another configuration file", 35, 98, color_of_wtxt);
  }
  else {
    textout(dblbuf, font, "Possible reasons:", 15, 30, color_of_wtxt);
    textout(dblbuf, font,
      "Loaded file is incorrect (wrong number of inputs or outputs)",
       35, 42, color_of_wtxt);
    textout(dblbuf, font,
      "or path contains non-english characters", 35, 54, color_of_wtxt);
    textout(dblbuf, font, "Solutions:", 15, 74, color_of_wtxt);
    textout(dblbuf, font, "Fix your file and reload it", 35, 86, color_of_wtxt);
    textout(dblbuf, font,
      "or press L and load another configuration file", 35, 98, color_of_wtxt);
    textout(dblbuf, font,
      "or in case of incorrect path move your file into", 35, 110, color_of_wtxt
    textout(dblbuf, font, "correct path and load again", 35, 122, color_of_wtxt),
    textout(dblbuf, font, "Your selected path was:", 15, 140, color_of_ntxt);
    textout(dblbuf, font, config_file, 35, 152, color_of_ntxt);
  }
}


//  ----------------------------
// | Processing user interaction |
//  ----------------------------

// Gets mouse movement information.
get_mouse_mickeys(&mickey_x, &mickey_y);


// - - Moving objects - - -

if(key[KEY_ALT]) {           // creature
  pCreature->x += mickey_x;
  pCreature->y += mickey_y;
}
else {                       // or light
  pLight->x += mickey_x;
  pLight->y += mickey_y;
}

// - - Changing spotlight's speed - - -

if(key[KEY_9_PAD])
  pLight->speed_x += change_of_lightspeed;
if(key[KEY_8_PAD])
  pLight->speed_x -= change_of_lightspeed;
if(key[KEY_4_PAD])
  pLight->speed_y += change_of_lightspeed;
if(key[KEY_7_PAD])
  pLight->speed_y -= change_of_lightspeed;

if(key[KEY_6_PAD])
  pLight->speed_x = 0;
if(key[KEY_5_PAD])
  pLight->speed_y = 0;

// - - Tuning creature's speed - - -

if(key[KEY_ASTERISK]) {
  pCreature->speedup_factor += change_of_speedup_factor;

  char buf[80];
  sprintf(buf, "speedup_factor = %f", pCreature->speedup_factor);
  textout_right(dblbuf, font, buf, 625, 8, color_of_ntxt);
```

```
}

if(key[KEY_SLASH_PAD]) {
  pCreature->speedup_factor -= change_of_speedup_factor;
  if(pCreature->speedup_factor < 0)
    pCreature->speedup_factor = 0;

  char buf[80];
  sprintf(buf, "speedup_factor = %f", pCreature->speedup_factor);
  textout_right(dblbuf, font, buf, 625, 8, color_of_ntxt);
}

if(key[KEY_MINUS_PAD]) {
  pCreature->speedup_factor = default_speedup_factor;

  char buf[80];
  sprintf(buf, "speedup_factor = %f", pCreature->speedup_factor);
  textout_right(dblbuf, font, buf, 625, 8, color_of_ntxt);
}

// - - Changing light's radius - - -

if(mouse_b bitand 1) pLight->radius += change_of_radius;
if( (mouse_b bitand 2) and (pLight->radius > 5) )
  pLight->radius -= change_of_radius;

// - - Loading new network - - -

if(key[KEY_L]) {

  // Calls file selection menu.
  int i = file_select_ex(message, config_file, ext, 490, 400, 450);
  rest(250);  // In case menu was exited by pressing Esc, this avoids
              // quitting the program.

  // If user chose something, then loads it into creature.
  if(i != 0) {
    pCreature->load_network(config_file);
    flag_just_started = false;
  }

  // After using menu the mickeys are screwed (moving mouse while in menu
  // shouldn't move light or creature). For correction:
  get_mouse_mickeys(&mickey_x, &mickey_y);
}

// - - Other user interactions - - -

// Press Escape to exit.
if(key[KEY_ESC]) {
  text_mode(0);
  textout(screen, font, "ESC pressed, will QUIT", 2, 2, color_of_wtxt);
  rest(2000);
  keepgoing = false;
}

// Press F1 for help.
if(key[KEY_F1]) {
  rectfill(screen, 0, 0, 445, 200, color_of_help_bg);

  textout(screen, font, "Keys:", 4, 10, color_of_ntxt);
  textout(screen, font, "mouse          - move spotlight", 15, 30, color_of_ntxt);
  textout(screen, font, "Alt + mouse    - move creature", 15, 42, color_of_ntxt);
  textout(screen, font,
   "mouse buttons - change size of spotlight", 15, 54, color_of_ntxt);
  textout(screen, font,
   "Numpad: 8,9,6  - change horizontal speed of spotlight", 15, 76, color_of_ntxt
  textout(screen, font,
   "Numpad: 4,7,5  - change vertical speed of spotlight", 15, 88, color_of_ntxt);
  textout(screen, font,
   "Numpad: /,*,-  - tune creature's speed", 15, 100, color_of_ntxt);
  textout(screen, font, "L    - load network", 15, 122, color_of_ntxt);
  textout(screen, font, "F1   - this help", 15, 144, color_of_ntxt);
```

```
      textout(screen, font,
       "F12 - save screenshot (does NOT work in menus)", 15, 156, color_of_ntxt);
      textout(screen, font, "Esc - quit program", 15, 168, color_of_ntxt);
      textout(screen, font, "Press any key to leave help", 15, 190, color_of_ntxt);

      rest(100);
      clear_keybuf();
      while(!keypressed());
      rest(250);

      // After using menu the mickeys are screwed (moving mouse while in menu
      // shouldn't move light or creature). For correction:
      get_mouse_mickeys(&mickey_x, &mickey_y);
    }

    // Press F12 to save screenshot.
    if(key[KEY_F12]) {
      PALETTE pal;
      get_palette(pal);
      save_bitmap("screenshot.bmp", dblbuf, pal);
      textout(screen, font, "Screenshot saved", 2, 2, color_of_ntxt);
      rest(2000);

      // After using menu the mickeys are screwed (moving mouse while in menu
      // shouldn't move light or creature). For correction:
      get_mouse_mickeys(&mickey_x, &mickey_y);

    }


    //  ----------
    // | Blitting |
    //  ----------

    // Copies the dblbuf into video memory at the right moment
    // (when vertical retrace is beginning in monitor).
    vsync();
    blit(dblbuf, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
    clear_bitmap(dblbuf);

  }


  // ***************
  // * Cleaning up *
  // ***************

  destroy_bitmap(dblbuf);
  delete pCreature;
  delete pLight;

}

END_OF_MAIN();
```

```
input_layer: 3

comp_layer: 15
  weights:   4.2036283822082856  -9.6491726110698792   5.7932909176514622
  weights:  -9.7846942346500754   9.3862715243711090  -4.6717189235602685
  weights:  -9.6052005530597704   8.9666956029458706  -5.0989005940879171
  weights:  -0.8111561919813262   8.4284323756765218  -9.7550249019698843
  weights:   0.9677298687262701   7.4651979151840351  -9.7023110867314646
  weights:  -9.0817161782930516  -5.5777777396784192   9.7874123576060050
  weights:   9.3859171141981470  -3.2289263879400019  -9.3305651339552309
  weights:  -9.4484862572132133   7.8178986003578066  -1.3814442320292726
  weights:  -8.0832181278852708  -6.4732327865837282   9.8313767236935199
  weights:  -9.2405418694411967   9.6750706708949838   0.9748273741406155
  weights:   7.0491655676909799  -9.9318399177388823   4.6098811982619896
  weights:  -9.7306664068161197   6.4547372057130827   6.2463592449401810
  weights:  -9.9932419998764193  -6.5232142875350503  -6.0321372594023872
  weights:   4.9606258069159095   3.5967703245558251   4.8867386886369282
  weights:   9.2667707125659806   4.0253499456188893  -9.5548596352202839

comp_layer: 15
  weights:   7.2202949415070101  -3.9885129248711286  -8.3045512914564803  -3.311758
  weights:  -6.9584703659134606   3.8735569939669161   7.8756937790661601  -6.661015
  weights:  -2.1486173592001840  -5.2732634955081110  -9.9390562826516913   2.307936
  weights:   3.7534043269045552  -4.0495994741455972   1.0474473782348537  -6.829728
  weights:  -9.6343817280124640   7.9150772327319574   9.6851738913805363   9.837117
  weights:   0.0726618321184945   9.9610788349739821   9.9336211625821491  -4.658685
  weights:   4.5362286327035894  -0.9711626177109082  -9.7738382465359344  -9.611359
  weights:  -3.7921278590173366  -3.0657081393473518   3.4516184636155165   8.133884
  weights:   5.0600420077707859  -6.2128935673934889  -7.0322814817593002  -2.446040
  weights:  -5.7100844229408869  -0.7864245347856874  -4.3769858499857550  -1.888876
  weights:  -9.8398912322376013   5.2075016735672639   9.4488393272129425   8.918907
  weights:   1.8207017147716573  -6.9299300586615029  -9.8216516446600046   8.272193
  weights:  -9.2591714858706755   6.5323971104995886  -9.9304762242074052   1.920417
  weights:  -0.4201325242451531   1.9785340963390707   4.8777896452794236   9.446939
  weights:  -6.1320544267773753  -9.0671319270089228  -7.5886831908211354  -6.781593

comp_layer: 4
  weights:   9.6799648035843973  -8.5192860992309978   4.3248473356987791  -5.264443
  weights:  -6.6571436372976533  -9.7523694664627119  -2.5108309051741755   0.993375
  weights:  -9.9999963190868861  -2.1512554345569339  -4.9143400107328548  -4.688464
  weights:   9.2252348945484819  -9.8142430254767046   6.1968947789455502  -5.274276

treshold: 0.0000000000000000
slope_parameter: 0.5000000000000000
max_weight: 10.0000000000000000
```

/lightchaser
/creature

```
// - - - - - - - - - - - - - - - - - - - - - - - - -
// File: creature.h                                 |
// Purpose: provides interface for class Creature   |
// Author: Taivo Lints, Estonia                     |
// Date: May, 2003                                  |
// Copyright: see copyright.txt                     |
// - - - - - - - - - - - - - - - - - - - - - - - - -

#ifndef CREATURE_H
#define CREATURE_H

#include "ann/neuralnetwork.h"
#include "allegro.h"
#include <vector>
using namespace std;

class Creature {       // Represents a moving creature with
                       // light intensity sensors.


  public:

    // ******************************
    // * Construction & Destruction *
    // ******************************

    Creature(BITMAP* pArena, int color_of_icircle, int color_of_ocircle,
             int start_of_gradient, char* conf_file);
                  // Creature constructor. Needs a pointer to the bitmap this
                  // creature will "live" on. Also needs the palette colors
                  // for inner and outer circles of sensors, start of light's
                  // color gradient on palette, and a configuration file
                  // for neural network.

    Creature(const Creature& rC);  // Creature copy-constructor.

    Creature& operator=(const Creature& rC);  // Operator overloading.

    ~Creature();       // Creature destructor.


    // *************************
    // * Parameters & variables *
    // *************************

    double x,                 // Position of creature.
           y;

    double speed_x,           // Speed of creature.
           speed_y;

    double speedup_factor;    // Speeds up creatures movement.

    int color_of_icircle,     // Palette colors for inner and outer circles
        color_of_ocircle;     // of sensors.

    bool flag_incor_net;      // Indicates that neural network is incorrect
                              // (configuration file is missing or the numbers
                              // of inputs and outputs are wrong).

    // *************
    // * Functions *
    // *************

    void update();            // Updates creature's position (except when
                              // flag_incor_net is true).

    void draw();              // Draws creature on the bitmap.

    void load_network(char* config_file); // Loads a new network into creature.


  // ****************
```

```
  // * Private stuff *
  // *****************

  // You can't use that stuff from outside code.

  private:

    BITMAP* pArena;           // A pointer to the bitmap this creature
                              // "lives" on.

    NeuralNetwork* pNet;      // A pointer to the neural network that controls
                              // creature's movement.

    vector<double> v_dbl_inps;  // A vector for feeding sensor information
                                // into neural network.

    vector<double> v_dbl_outputs; // A vector for getting control values from
                                  // neural network.

    class Position {          // Just a useful class for storing the
      public:                 // position of something (e.g. sensors).
        int x,
            y;
    };

    vector<Position> vPositions;  // Relative positions of sensors.

    int start_of_gradient;    // Start of light's color gradient on palette.

};

#endif // CREATURE_H
```

```
// - - - - - - - - - - - - - - - - - - -
// File: creature.cpp                   |
// Purpose: implements class Creature   |
// Author: Taivo Lints, Estonia         |
// Date: May, 2003                      |
// Copyright: see copyright.txt         |
// - - - - - - - - - - - - - - - - - - -

#include "creature.h"
#include "ann/neuralnetwork.h"
#include "allegro.h"
using namespace std;

// - - - - - Class Creature - - - - -

// ******************************
// * Construction & Destruction *
// ******************************

// - Creature constructor - -
// Needs a pointer to the bitmap this creature will "live" on.
// Also needs the palette colors for inner and outer circles of sensors,
// start of light's color gradient on palette, and a configuration file
// for neural network.
Creature::Creature(BITMAP* inp_pArena, int c_of_icircle, int c_of_ocircle,
                   int s_of_gradient, char* conf_file) {

  pArena = inp_pArena;

  x = pArena->w / 2;
  y = pArena->h / 2;

  speed_x = 0;
  speed_y = 0;

  speedup_factor = 3;

  color_of_icircle = c_of_icircle;
  color_of_ocircle = c_of_ocircle;

  start_of_gradient = s_of_gradient;

  // Currently the number of inputs (sensors) MUST be 3:
  // lower left, upper and lower right sensor (in that order).
  v_dbl_inps.assign(3, 0);

  // Sets the relative positions of these three sensors.
  Position* pPos = new Position;
  pPos->x = -4;
  pPos->y = 3;
  vPositions.push_back(*pPos);
  delete pPos;

  pPos = new Position;
  pPos->x = 0;
  pPos->y = -5;
  vPositions.push_back(*pPos);
  delete pPos;

  pPos = new Position;
  pPos->x = 4;
  pPos->y = 3;
  vPositions.push_back(*pPos);
  delete pPos;

  // Outputs are: speed_left, speed_right, speed_up, speed_down (in this order).
  // Speed_x is calculated as speed_right - speed_left. Similar formula for y.
  v_dbl_outputs.assign(4, 0);

  // Creates network and checks its correctness.
  pNet = new NeuralNetwork(conf_file);

  if( (pNet->get_number_of_inputs() != 3) or
```

```
      (pNet->get_number_of_outputs() != 4) )
    flag_incor_net = true;
  else
    flag_incor_net = false;

}

// - Creature copy-constructor - -
Creature::Creature(const Creature& rC) {

  // Copy of creature can currently live only on the same bitmap as original.
  pArena = rC.pArena;

  x = rC.x;
  y = rC.y;

  speed_x = rC.speed_x;
  speed_y = rC.speed_y;

  speedup_factor = rC.speedup_factor;

  color_of_icircle = rC.color_of_icircle;
  color_of_ocircle = rC.color_of_ocircle;

  start_of_gradient = rC.start_of_gradient;

  v_dbl_inps = rC.v_dbl_inps;

  vPositions = rC.vPositions;

  v_dbl_outputs = rC.v_dbl_outputs;

  // Must have its own network.
  pNet = new NeuralNetwork(NULL);
  *pNet = *(rC.pNet);  // Fortunately I wrote operator= overloading for
                       // NeuralNetwork class :) (otherwise it wouldn't
                       // work correctly).

  // It's safer to check the correctness again than just copy flag_incor_net.
  if( (pNet->get_number_of_inputs() != 3) or
      (pNet->get_number_of_outputs() != 4) )
    flag_incor_net = true;
  else
    flag_incor_net = false;

}

// - Operator= overloading - -
Creature& Creature::operator=(const Creature& rC) {

  if(&rC != this) {   // Check for self-assignment.

    pArena = rC.pArena;

    x = rC.x;
    y = rC.y;

    speed_x = rC.speed_x;
    speed_y = rC.speed_y;

    speedup_factor = rC.speedup_factor;

    color_of_icircle = rC.color_of_icircle;
    color_of_ocircle = rC.color_of_ocircle;

    start_of_gradient = rC.start_of_gradient;

    v_dbl_inps = rC.v_dbl_inps;

    vPositions = rC.vPositions;

    v_dbl_outputs = rC.v_dbl_outputs;

    // Must have its own network.
```

```
    pNet = new NeuralNetwork(NULL);
    *pNet = *(rC.pNet);  // Fortunately I wrote operator= overloading for
                         // NeuralNetwork class :) (otherwise it wouldn't
                         // work correctly).

    // It's safer to check the correctness again than just copy flag_incor_net.
    if( (pNet->get_number_of_inputs() != 3) or
        (pNet->get_number_of_outputs() != 4) )
      flag_incor_net = true;
    else
      flag_incor_net = false;

  }

  return *this;

}

// - Creature destructor - -
Creature::~Creature() {

  // Deletes network (to prevent memory leak).
  delete pNet;

}


// *************
// * Functions *
// *************

// - Function: update - -
// Updates creature position.
void Creature::update() {

  // Updates only when neural network is correct.
  if(not flag_incor_net) {

    int ix = static_cast<int>(x);
    int iy = static_cast<int>(y);

    // Reads and normalizes sensor information.
    for(int i = 0; i < 3; i++) {
      int light_intensity = getpixel(pArena,
                ix + vPositions[i].x, iy + vPositions[i].y) - start_of_gradient;

      if(light_intensity < 0)
        light_intensity = 0;

      if(light_intensity > 64)
        light_intensity = 64;

      v_dbl_inps[i] = static_cast<double>(light_intensity) / 64;
    }

    // Uses network.
    pNet->set_inputs(&v_dbl_inps);
    pNet->update();
    pNet->get_outputs(&v_dbl_outputs);


    // Updates speed and position.
    speed_x = (v_dbl_outputs[1] - v_dbl_outputs[0]) * speedup_factor;
    speed_y = (v_dbl_outputs[3] - v_dbl_outputs[2]) * speedup_factor;

    x += speed_x;
    y += speed_y;

     // Keep creature on screen
    if(x + vPositions[0].x < 0)
      x = 0 - vPositions[0].x;

    if(x + vPositions[2].x > pArena->w - 1)
```

```
      x = pArena->w - 1 - vPositions[2].x;

    if(y + vPositions[1].y < 0)
      y = 0 - vPositions[1].y;

    if(y + vPositions[0].y > pArena->h - 1)
      y = pArena->h - 1 - vPositions[0].y;
  }

}

// - Function: draw - -
// Draws creature on the bitmap.
void Creature::draw() {

  int ix = static_cast<int>(x);
  int iy = static_cast<int>(y);

  circle(pArena, ix + vPositions[0].x, iy + vPositions[0].y, 1, color_of_icircle);
  circle(pArena, ix + vPositions[0].x, iy + vPositions[0].y, 2, color_of_ocircle);

  circle(pArena, ix + vPositions[1].x, iy + vPositions[1].y, 1, color_of_icircle);
  circle(pArena, ix + vPositions[1].x, iy + vPositions[1].y, 2, color_of_ocircle);

  circle(pArena, ix + vPositions[2].x, iy + vPositions[2].y, 1, color_of_icircle);
  circle(pArena, ix + vPositions[2].x, iy + vPositions[2].y, 2, color_of_ocircle);

}

// - Function: load_network - -
// Loads a new network into creature.
void Creature::load_network(char* config_file) {

  delete pNet;

  pNet = new NeuralNetwork(config_file);

  // Checks correctness.
  if( (pNet->get_number_of_inputs() != 3) or
      (pNet->get_number_of_outputs() != 4) )
    flag_incor_net = true;
  else
    flag_incor_net = false;

}
```

/lightchaser
/light

```
// - - - - - - - - - - - - - - - - - - - - - - - -
// File: light.h                                  |
// Purpose: provides interface for class Light    |
// Author: Taivo Lints, Estonia                   |
// Date: May, 2003                                |
// Copyright: see copyright.txt                   |
// - - - - - - - - - - - - - - - - - - - - - - - -

#ifndef LIGHT_H
#define LIGHT_H

#include "allegro.h"
using namespace std;

class Light {       // Represents a round moving spotlight.


  public:

    // ******************************
    // * Construction & Destruction *
    // ******************************

    Light(BITMAP* pArena, int start_of_gradient);
                    // Light constructor. Needs a pointer to the bitmap this
                    // light will "live" on. Also needs the start of 64 color
                    // gradient on palette.

    Light(const Light& rL);  // Light copy-constructor.

    Light& operator=(const Light& rL);  // Operator overloading.

    ~Light();       // Light destructor.


    // ***************************
    // * Parameters & variables *
    // ***************************

    int radius;               // Radius of the spotlight.

    double x,                 // Position of light.
           y;

    double speed_x,           // Speed of light (hahaha ;).
           speed_y;


    // *************
    // * Functions *
    // *************

    void update();            // Updates light position and size.

    void draw();              // Draws light on the bitmap.


  // *****************
  // * Private stuff *
  // *****************

  // You can't use that stuff from outside code.

  private:

    BITMAP* pArena;           // A pointer to the bitmap this light "lives" on.
    BITMAP* pSprite;          // A pointer to the bitmap with the image of light.

    int start_of_gradient;    // Start of 64 color gradient on palette.

    int left_edge,            // x value of lights left edge.
        top_edge;             // y value of lights top edge.
                              // (for drawing the sprite)
```

```
    int previous_radius;      // Radius of the spotlight when update() was
                              // called the last time.

    BITMAP* create_light_sprite();
                              // Creates an image of light with current radius
                              // and returns a pointer to it.

};

#endif // LIGHT_H
```

```cpp
// - - - - - - - - - - - - - - - - - -
// File: light.cpp                    |
// Purpose: implements class Light    |
// Author: Taivo Lints, Estonia       |
// Date: May, 2003                    |
// Copyright: see copyright.txt       |
// - - - - - - - - - - - - - - - - - -

#include "light.h"
#include "allegro.h"
using namespace std;

// - - - - - Class Light - - - - -

// ******************************
// * Construction & Destruction *
// ******************************

// - Light constructor - -
// Needs a pointer to the bitmap this light will "live" on.
// Also needs the start of 64 color gradient on palette.
Light::Light(BITMAP* inp_pArena, int inp_start_of_gradient) {

  radius = 64;

  x = 100;
  y = 100;

  speed_x = 1;
  speed_y = 1;

  start_of_gradient = inp_start_of_gradient;

  pArena = inp_pArena;
  pSprite = create_light_sprite();

  left_edge = static_cast<int>(x - radius);
  top_edge = static_cast<int>(y - radius);

  previous_radius = radius;

}

// - Light copy-constructor - -
Light::Light(const Light& rL) {

  radius = rL.radius;

  x = rL.x;
  y = rL.y;

  speed_x = rL.speed_x;
  speed_y = rL.speed_y;

  // Copy of light can currently live only on the same bitmap as original.
  pArena = rL.pArena;

  start_of_gradient = rL.start_of_gradient;

  // Must have its own light image.
  pSprite = create_light_sprite();

  left_edge = rL.left_edge;
  top_edge = rL.top_edge;

  previous_radius = rL.previous_radius;

}

// - Operator= overloading - -
Light& Light::operator=(const Light& rL) {

  if(&rL != this) {    // Check for self-assignment.
```

```
    radius = rL.radius;

    x = rL.x;
    y = rL.y;

    speed_x = rL.speed_x;
    speed_y = rL.speed_y;

    // Copy of light can currently live only on the same bitmap as original.
    pArena = rL.pArena;

    // Must have its own light image.
    pSprite = create_light_sprite();

    start_of_gradient = rL.start_of_gradient;

    left_edge = rL.left_edge;
    top_edge = rL.top_edge;

    previous_radius = rL.previous_radius;
  }

  return *this;

}

// - Light destructor - -
Light::~Light() {

  // Deletes light image (to prevent memory leak).
  destroy_bitmap(pSprite);

}


// *************
// * Functions *
// *************

// - Function: update - -
// Updates light position and size.
void Light::update() {

  // If radius has changed, then creates new image of light.
  if(radius != previous_radius) {
    destroy_bitmap(pSprite);
    pSprite = create_light_sprite();
    previous_radius = radius;
  }

  // Updates the position.
  x += speed_x;
  y += speed_y;


  // Bouncing against walls.
  if(x < 0) {
    x = 0;
    speed_x = -speed_x;
  }

  if(x > pArena->w) {
    x = pArena->w;
    speed_x = -speed_x;
  }

  if(y < 0) {
    y = 0;
    speed_y = -speed_y;
  }

  if(y > pArena->h) {
    y = pArena->h;
```

```
    speed_y = -speed_y;
  }

  // Updating some drawing related variables.
  left_edge = static_cast<int>(x - radius);
  top_edge = static_cast<int>(y - radius);

}

// - Function: draw - -
// Draws light on the bitmap.
void Light::draw() {

  draw_sprite(pArena, pSprite, left_edge, top_edge);

}


// *****************
// * Private stuff *
// *****************

// - Function: create_light_sprite - -
// Creates image of light with current radius and returns a pointer to it.
BITMAP* Light::create_light_sprite() {

  BITMAP* pLight_Sprite;

  // Creates a bitmap and clears it.
  pLight_Sprite = create_bitmap(2 * radius, 2 * radius);
  clear_bitmap(pLight_Sprite);

  // Fills it with light.
  for(int i = radius; i > 0; i--)
    circlefill(pLight_Sprite, radius - 1, radius - 1, i,
                start_of_gradient + 64 * (radius - i) / radius);

  return pLight_Sprite;

}
```

/lightchaser
/palette

mypalette.cpp ja mypalette.h
vt. /visualizer/visnet

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: mypalette.h                                                     |
// Purpose: provides interface for palette modification functions        |
// Author: Taivo Lints, Estonia                                          |
// Date: March, 2003                                                     |
// Copyright: see copyright.txt                                          |
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

#ifndef MYPALETTE_H
#define MYPALETTE_H

// Changes colors start..start + 63 to a gradient (from black
// to white), others remain the same.
void make_grayscale_palette_64(int start);

// Changes colors start..start + 63 to a gradient (from black
// to some kind of blue-green), others remain the same.
void make_blueish_palette_64(int start);

// Changes the color of a palette entry (with index color_nr).
void change_color(int color_nr, int r, int g, int b);

#endif // MYPALETTE_H
```

```cpp
// - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: mypalette.cpp                                |
// Purpose: implements palette modification functions |
// Author: Taivo Lints, Estonia                       |
// Date: March, 2003                                  |
// Copyright: see copyright.txt                       |
// - - - - - - - - - - - - - - - - - - - - - - - - - -

#include "mypalette.h"
#include "allegro.h"
using namespace std;

// - Function: make_grayscale_palette_64 - -
// Changes colors start..start + 63 to a gradient (from black
// to white), others remain the same.
void make_grayscale_palette_64(int start) {
  RGB rgb;

  for(int i = start; i < start + 64; i++) {
    if(i > 255) break;
    rgb.r = rgb.g = rgb.b = i - start;
    set_color(i, &rgb);
  }
}

// - Function: make_blueish_palette_64 - -
// Changes colors start..start + 63 to a gradient (from black
// to some kind of blue-green), others remain the same.
void make_blueish_palette_64(int start) {
  RGB rgb;

  for(int i = start; i < start + 64; i++) {
    if(i > 255) break;

    rgb.r = 0;
    rgb.g = static_cast<int>(static_cast<float>(i - start) / 1.5);
    rgb.b = i - start;
    set_color(i, &rgb);
  }
}

// - Function: change_color - -
// Changes the color of a palette entry (with index color_nr).
void change_color(int color_nr, int r, int g, int b) {
  RGB rgb;

  rgb.r = r;
  rgb.g = g;
  rgb.b = b;
  set_color(color_nr, &rgb);
}
```