**Predmet: Inteligentni sistemi za podporo odločanju**

# VAJA 4

## Nevronske mreže in učenje s PSO

**NALOGA**:

Pri tej vaji je potrebno zgraditi sistem za klasifikacijo raka dojk. V poslanih podatkih se nahajajo vrednosti značilk na podlagi katerih lahko klasificiramo rak na benigen in maligen. Sistem za klasifikacijo boste zgradili z nevronsko mrežo. Namesto vgrajenih algoritmov učenja pa boste uporabili PSO optimizacijo. Napisali boste tudi poročilo, kjer boste analizirali delovanje in sistema oziroma njegovo izgradnjo. Primerjajte dobljene rezultate z rezultati, ki jih lahko dobite z vgrajenim učenjem (ukaz `net = train(net,x,t);`). Podatki so podani v data.mat. Prvi stolpec podaja oznako tumorja (2 – maligen in 4 – benigen)
Oddajate KODO+POROČILO

Rezultate analizirajte (tako za učno kot testno množico).

Razmisli, kaj se zgodi z nevronsko mrežo, če so aktivacijske funkcije vseh nevronov v vseh nivojih linearne. Ali dobite drugačne rezultate?

POROČILO:
Poročilo naj vsebuje osnovno teoretično ozadje o uporabljenih algoritmih. Analizo rezultatov dobljenih z vgrajenim algoritmom učenja (preverite kateri algoritem uporabljate) ter rezultate dobljene z PSO algoritmom. Pri interpretaciji rezultatov poskusite upoštevati tudi ozadje problema.

**Predmet: Inteligentni sistemi za podporo odločanju**

**POMOČ**:

# PSO OPTIMIZACIJA

Metoda spada med evolucijsko računanje tako kot GA metoda. Ima tudi podobne korake implementacije kot GA: naključno generiranje začetne populacije, evalvacija kriterijske funkcije za vsak osebek in generiranje nove populacije na podlagi kriterijske funkcije ter naključnega števila. Obe metodi sta učinkoviti, ne zagotavljata pa globalnega optimuma. PSO načeloma nima definiranih operatorjev križanja in mutacije. Delci se pomikajo po problemskem prostoru s pomočjo svoje hitrosti, vsak delec pa ima svoj spomin. V njem se hranijo podatki o njegovi najboljši poziciji v prostoru. Pri PSO je populacija označena z besedo roj, posamezniki pa so delci. Algoritem temelji na deterministčnem pravilu prilagajanja, ki mu je dodana stohastika.

Pri PSO optimizaciji vsak delec skuša popraviti svoj položaj in hitrost v problemskem prostoru glede na oddaljenost njegove trenutne pozicije do najboljše pozicije, ki jo je zabeležil roj. Poznamo več izpeljank algoritma PSO, ki se razlikujejo v enačbah za prilagajanje delcev.

Pri PSO algoritmu moramo definirati faktorja *c1* in *c2*. Faktor *c1* imenujemo faktor pospeševanja oziroma samozaupanja delca (acceleration oziroma self-confidence factor). Z njim utežimo pomembnost delčeve najboljše pozicije *pbest*. Faktor *c2* imenujemo faktor pospeševanja oziroma faktor zaupanja v roj (acceleration oziroma swarm-confidence factor). Z njim utežimo pomembnost najboljše pozicije *gbest*, ki jo je zaznal celoten roj. Običajno vrednosti izbiramo iz intervala [0, 4] in *c1* je ponavadi enak *c2*.

Običajno definiramo meje prostora *Xmax* in *Xmin*. Meje preprečujejo, da bi delec ušel iz problemskega območja. V primeru, da delec uide iz področja, ga postavimo nazaj na mejo in njegovo hitrost postavimo na nič. Definiramo tudi maksimalno dopustno hitrost delca za vsako dimenzijo. V literaturi je več predlogov kako jo definiramo. En od njih je, da za maksimalno hitrost vzamemo petino obsega območja: *Vmax = (Xmax - Xmin) ./ 5*.

Na zadnje moramo definirati število delcev v roju. Običajno število je tam nekje med 20 in 50 delcev. V literaturi najdemo več različnih priporočil. Nekateri vežejo število delcev v roju tudi na dimenzijo prostora, npr: *PART_NUM = round(10+2*sqrt(Dim))*, kjer je *Dim* dimenzija prostora.

Naslednji korak je inicializacija začetnih vrednosti za vsak delec v roju. Tu naključno generiramo: pozicijo delca $\boldsymbol{x}_i$, ki je v dopustnem območju; hitrost delca $\boldsymbol{v}_i$, ki je v dopustnem območju hitrosti. Za vsak delec izračunamo vrednost kriterijske funkcije *f_best* in trenutno inicializirano pozicijo delca postavimo za najboljšo pozicijo delca $\boldsymbol{pbest}_i$.

V naslednjem koraku poiščemo delec z najboljšo vrednostjo kriterijske funkcije. Zapomnimo si to vrednost (*fg_best*) in pozicijo tega delca (***gbest***).

S tem smo zaključili začetne korake PSO algoritma. Od tu naprej se začne dejanska optimizacija. V zanki ponavljamo v nadaljevanju opisane korake, dokler ni izpolnjen kriterij za končanje optimizacije.

- Popravimo pozicije delcev
- Preverimo, če so pozicije znotraj dopustnega območja. Če niso, delec postavimo na mejo in njegovo hitrost postavimo na nič
- Preverimo hitrost delcev. Če hitrost delca ni v dopustnem območju hitrosti jo postavimo na nič.
- Izračunamo vrednost kriterijske funkcije za posamezni delec.

- Primerjamo novo vrednost kriterijkse funkcije z vrednostjo *f_best*. Če je vrednost boljša potem vrednost *f_best* zamenjamo s trenutno vrednostjo kriterijske funkcije in najboljšo pozicijo $\boldsymbol{pbest}_i$ zamenjamo s trenutno pozicijo delca.
- Poiščemo delec, ki ima najboljšo vrednost *f_best.* To vrednost primerjamo z vrednostjo *fg_best*. Če je *f_best* delca boljša od *fg_best*, staro *fg_best* vrednost prepišemo z *f_best* vrednostjo, pozicijo ***gbest*** pa zamenjamo s pozicijo delca $\boldsymbol{pbest}_i$.
- Izračunamo kriterije za dokončanje optimizacije. Če so kriteriji izpolnjeni je naša rešitev shranjena v spremenljivki ***gbest***. Če ne, potem nadaljujemo z zgornjimi koraki.

**NEKATERI PSO ALGORITMI**

**1. PREPROSTI - STANDARDNI PSO**:

Enačbi za adaptiranje hitrosti delcev in pozicija sta sledeči:

$$v_{ij} = v_{ij} + c_1 \cdot rand \cdot \left(pbest_{ij} - x_{ij}\right) + c_2 \cdot rand \cdot \left(gbest_j - x_{ij}\right)$$

$$x_{ij} = x_{ij} + v_{ij}$$

Indeks *i* označuje številko delca, indeks *j* pa komponento vektorja.

**2. UTEŽENI PSO (PSO-W: PSO with inertia weight):**

Enačbi za adaptiranje hitrosti delcev in pozicija sta sledeči:

$$v_{ij} = W \cdot v_{ij} + c_1 \cdot rand \cdot \left(pbest_{ij} - x_{ij}\right) + c_2 \cdot rand \cdot \left(gbest_j - x_{ij}\right)$$

$$x_{ij} = x_{ij} + v_{ij}$$

Indeks *i* označuje številko delca, indeks *j* pa komponento vektorja. Tu je osnovni enačbi dodana utež *W*. Utež predstavlja inercijo delca. Z njo spreminjamo aktivnost delca pri raziskovanju problemskega prostora. Običajno je utež nastavljena okoli 0.7. Utež lahko tudi dinamično spreminjamo.

**3. PSO S FAKTORJEM OMEJITVE (PSO-C: PSO with constriction factor):**

Enačbi za adaptiranje hitrosti delcev in pozicija sta sledeči:

$$v_{ij} = \kappa \cdot (v_{ij} + c_1 \cdot rand \cdot \left(pbest_{ij} - x_{ij}\right) + c_2 \cdot rand \cdot \left(gbest_j - x_{ij}\right))$$

$$x_{ij} = x_{ij} + v_{ij}$$

$$\kappa = \frac{2}{\left|2 - \phi - \sqrt{\phi(\phi - 4)}\right|}, \phi = c_1 + c_2 > 4$$

Indeks *i* označuje številko delca, indeks *j* pa komponento vektorja. Faktor je bil uveden z namenom boljše konvergence algoritma.

### 4. Hibridni PSO z mutacijo (HPSOM: hybrid PSO with mutation):

Enačbi za adaptiranje hitrosti delcev in pozicija sta sledeči:

$$v_{ij} = \kappa \cdot (v_{ij} + c_1 \cdot rand \cdot (pbest_{ij} - x_{ij}) + c_2 \cdot rand \cdot (gbest_j - x_{ij}))$$

$$x_{ij} = x_{ij} + v_{ij}$$

$$\kappa = \frac{2}{\left|2 - \phi - \sqrt{\phi(\phi - 4)}\right|}, \phi = c_1 + c_2 > 4$$

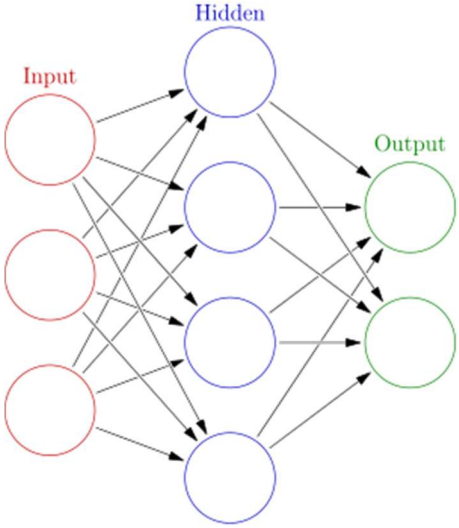
Indeks *i* označuje številko delca, indeks *j* pa komponento vektorja. Faktor je bil uveden z namenom boljše konvergence algoritma. Tu se delci adaptirajo po običajnih enačbah. Po adaptaciji pa sledi mutacija pozicije delca. Mutacijo določa faktor verjetnosti mutacije $\lambda_m$. Če je $rand < \lambda_m$ potem mutiramo pozicijo delca. Pozicijo mutiramo kot:

$$x_{ij} = x_{ij} + dx_j, \text{če je } rand < 0.5$$

$$x_{ij} = x_{ij} - dx_j, \text{če je } rand \geq 0.5$$

Odmik $dx_j$ dobimo tako, da generiramo naključno vrednost iz intervala [0, $\boldsymbol{Xmax_j}$-$\boldsymbol{Xmin_j}$]. Z mutacijo izboljšamo obnašanje metode okoli lokalnega optimuma.

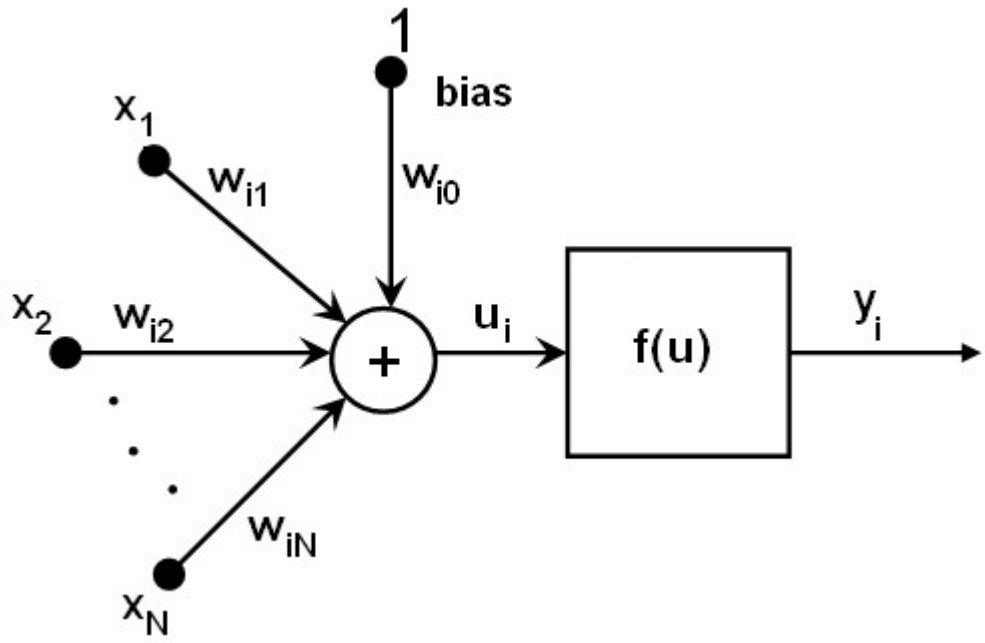
## Nevronske mreže




*Slika 1: grafična predstavitev nevronske mreže*

Za izgradnjo sistema klasifikacije lahko uporabimo usmerjeno (feedforward) nevronsko mrežo. Nevronske mreže so običajno sestavljene iz treh plasti. Prva plast – vhodna plast – skrbi za pravilno povezavo vhodov na perceptrone v drugi – skriti plasti. Iz skrite plasti pa vodijo povezave na izhodno plast. Število nevronov izhodne plasti je enako številu izhodov. Število nevronov vhodne plasti je enako številu izhodov. Število nevronov skrite plasti pa določi uporabnik sam glede na kompleksnost sistema, ki ga modelira.

Krogi na sliki 1 predstavljajo perceptrone. Delovanje perceptrona je shematično predstavljeno na sliki 2. Vhodni perceptroni so brez prenosnih funkcij in uteži, so samo sredstvo za lažjo predstavitev mreže.




*Slika 2: shema perceptrona.*

Izhod perceptrona je vrednost funkcije f, ki je odvisna od linearne kombinacije vhodov: $f(x_1*w_{1i} + x_2*w_{2i}+...+x_N*w_{Ni}+w_{i0})$. Postopek učenja nevronske mreže je optimizacija uteži, da pri podanih vhodih dobimo željen izhod.

## Pomoč pri nalogi

Ker ima Matlab že pripravljeno strukturo za nevronske mreže, vam le-te ne bo potrebno programirati.

Prva stvar, ki jo morate narediti je razdeliti podatke na učno in testno množico. Pri tem morate paziti, da v učno množico ne vključite samo enega razreda. Razreda naj bosta zastopana približno v enakem številu kot v celotnem setu. Primer: Če imamo v celotnem setu 20 vzorcev iz enega razreda in 10 vzorcev iz drugega, razdelimo ta set v razmerju 2:1 (npr.: 14 vzorcev iz enega in 7 iz drugega razreda v učno ostale v testno množico). Zaradi numerike je zaželjeno, da se podatki normirajo med 0 in 1 (prva vaja b primer).

Nato generiramo nevronsko mrežo. Matlabov ukaz:

```
net = newff(x,t,n,{'tansig','tansig'});
```

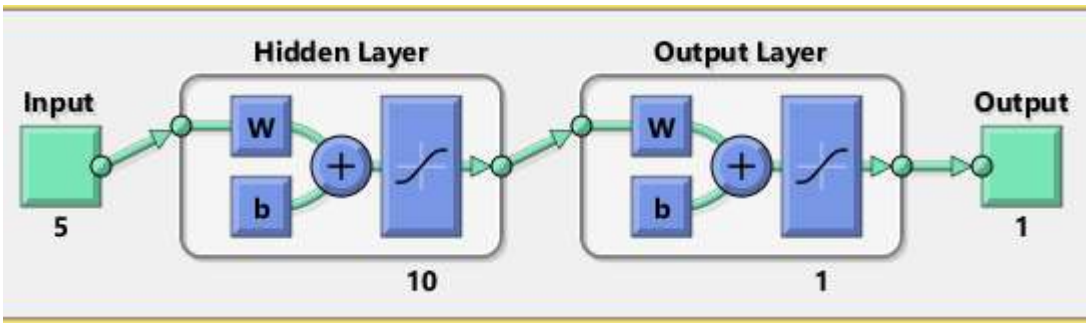
S tem kreiramo 'feedforward' mrežo z tansig funkcijo v skriti plasti in tansig funkcijo v izhodni plasti. Parameter n poda število nevronov v skriti plasti x je matrika vhodov in t je matrika željenih izhodov.

Bodite pozorni pri vektorjih!!! Vektor t mora biti vrstični, prav tako matrika x (vsaka spremenljivka ima svojo vrstico, stolpci so meritve).

Ko generiramo mrežo jo je potrebno inicializirati. To storimo tako z ukazom:

```
net = init(net);
net = configure(net,x,t);
```

S tem se postavi pravilno število uteži in uskladijo dimenzije vhodov in izhodov. Mrežo lahko pogledamo z ukazom view(net). Primer:




*Slika 3: primer nevronske mreže s petimi vhodi in enim izhodom*

Matlab tretira triplastno nevronsko mrežo kot mrežo z eno vhodno plastjo, ki je ločena od skrite in izhodne plasti. Če vpišemo v komandno okno ime mreže se nam izpišejo informacije o strukturi mreže. Izpisuje se po razdelkih: dimensions, kjer so specificirane dimenzije vhodov, število plasti, število izhodov in uteži ter druge informacije. Tisti, ki boste gledali te podatke naj vas ne zmedejo številke, saj Matlab malo drugače interpretira stvari kot literatura. Število vhodov bo v vašem primeru 1 (numInputs = 1). Parameter numInputs pomeni število vhodnih sekvenc, v večini primerov je to ena. Pomembna pa je dimenzija vhoda (net.inputs{1}.size). Ta se mora ujemati s številom vhodnih spremenljivk. Število plasti je v našem primeru dva (numLayers = 2), saj matlab vhodne plasti ne upošteva. Pomembno je še število uteži (numWeightElements), ki mora biti v našem primeru enako kot: (število vhodov + 1) * število nevronov v skriti plast + število nevronov v skriti plasti +1.

Naslednji razdelek so povezave (connections). Tu dobimo informacijo o povezavah med plastmi: biasConnect: [1; 1] : definira povezavo pragov na posamezno plast (b). Vektor [1;1] pomeni, da ima tako skrita kot tudi izhodna plast priključen prag b; inputConnect: [1; 0] pove kam je priključena vhodna plast. Vektor [1;0] pomeni, da so vhodi priključeni na skrito plast in noben vhod ne gre direktno v izhodno plast; outputConnect: [0 1] definira povezanost katera plast nam izračuna končni izhod ([0 1] – pomeni, da končni izhod določa izhodna plast); layerConnect: [0 0; 1 0] nam podaja katere plasti so povezane med seboj z utežmi. Če želimo izvedeti ali potuje signal iz plasti 1 v plast 2 prek uteži podamo ukaz: net.layerConnect(2,1) (rezultat 1 pomeni DA, 0 pomeni NE).

Dostopanje do uteži newronske mreže:

V net.IW (v našem primeru net.IW{1}) so definirane uteži povezav, ki gredo iz vhodne plasti v skrito plast. Število vrstic matrike je enako število nevronov v skriti plasti, število stolpcev pa številu vhodov.

V net.b so shranjene vrednosti pragov. V net.b{1} so pragovi skrite plasti v net.b{2} pa pragovi izhodne plasti.

V net.LW (v našem primeru net.LW{2,1} ) so shranjene uteži povezav med plastmi. V našem primeru med skrito plastjo in izhodno plastjo.

Uteži lahko prepišemo v vektor z ukazom:

```
getwb(net)
```

z ukazom `setwb(net,weights)` prepišemo uteži v nevronsko mrežo. Ta ukaz nam bo prišel prav pri računanju kriterijske funkcije.

**Predmet: Inteligentni sistemi za podporo odločanju**

Ko imamo inicializirano nevronsko mrežo pričnemo optimizacijo z PSO, ki jo boste spisali sami. Uporabite lahko MSE cenilko: `mse_calc = sum((yest-targets).^2)/length(yest)`.

Izhod nevronske mreže izračunamo kot yest = net(x).

## Nekaj ostalih napotkov:

Če uporabite vse podane vhodne spremenljivke, naj bi bila meja med razredi linearna. To pomeni, da je potrebno majhno število nevronov v skriti plasti (nekje od 2 do 5 naj bi bilo dovolj).

Če ne uporabite vseh vhodnih spremenljivk vam priporočam malo večje število nevronov v skriti plasti.

Sumim, da bo PSO optimizacija trajala kar nekaj časa zato je pametno prej preveriti koliko nevronov in vhodov potrebujemo z Matlabovim neural networks toolboxom.

Ker je PSO stohastična metoda, poskusite večkrat zagnati optimizacijo in analizirajte rezultate.

Obnašanje sistema je potrebno preveriti tako na učni kot na testni množici. Pri klasifikaciji je zelo grafična predstavitev 'confusion' matrike:
PRIMER:
`plotconfusion(y,yest)`
kjer sta y in yest vrstična vektorja. Pri tem je potrebno upoštevati, da imamo mi dva razreda, delali pa smo z enim izhodom. Torej je potrebno vektor izhoda spremeniti tako, da bo imel dve vrstici.
Primer: recimo da vrednost 0 označuje prvi razred vrednost 1 pa drugi: y = [0.0 1.0 0.0 1.0 1.0 0.0]-> y =[1.0 0.0 1.0 0.0 0.0 1.0; 0.0 1.0 0.0 1.0 1.0 0.0], isto storimo z yest: yest = [0.1 0.6 0.01 0.02 0.8 0.3] -> yest = [1.0 0.0 1.0 1.0 0.0 1.0; 0.0 1.0 0.0 0.0 1.0 0.0].