SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 51

UČENJE NEURONSKE MREŽE PARALELNOM OPTIMIZACIJOM ROJEM ČESTICA

Goran Gradišar

Zagreb, svibanj 2010.

SADRŽAJ

1 UVOD

Optimizacija se u najjednostavnijem obliku odnosi na proučavanje problema u kojima se pokušava minimizirati ili maksimizirati neku realnu funkciju sustavnim odabirom ulaznih vrijednosti iz zadane domene [1]. Optimizacija sama po sebi ima veoma široku primjenu te postoji realna potreba za dobrim optimizacijskim algoritmima. Kao posljedica toga, razvijeno je mnogo metoda koje se koriste u optimizaciji, među kojima neke s više, a neke s manje uspjeha. U ovom radu posebno se obrađuje metoda optimizacije rojem čestica. Među optimizacijskim metodama odlikuje se jednostavnošću implementacije, brzinom te dobrim konačnim rezultatima. Kako već sam naziv metode nagovješćuje, zasniva se na socijalnom modelu roja pri optimizaciji funkcija. Problemi koji se rješavaju ovom metodom mogu biti veoma komplicirani i skupi za izvođenje na računalu, pa time i sama optimizacija treba duže vrijeme kako bi pronašla rezultat. Kao jedan od kompliciranijih i skupih problema za rješavanje logički se nameće treniranje neuronske mreže, što nas dovodi do drugog bitnog djela ovog rada.

Neuronske mreže, kao i optimizacija rojem čestica, imaju veoma široko područje primjene zbog svoje svestranosti u rješavanju različitih problema. Dijelovi iz područja neuronskih mreža, sami po sebi, bi mogli biti tema i dobra osnova za ozbiljnije radove. Ovdje ćemo pokušati barem načeti neke od tih tema i pokušati što jednostavnije objasniti neke osnovne principe na kojima djeluju neuronske mreže kako bismo pokazali s kojim problemima se susrećemo u njihovom treniranju. Definirat će se što su to neuronske mreže, pokazati neki osnovni oblici neuronskih mreža koje postoje, te objasniti neke osnovne metode koje se koriste u njihovom treniranju. Uglavnom će se usredotočiti na metode i oblike koji su korišteni i u konačnoj implementaciji.

Za ubrzavanje procesa treniranja optimizacijom rojem čestica ne treba puno razmišljati kako bi se došlo do zaključka da bi bilo dobro omogućiti da se dijelovi algoritma izvode paralelno na više različitih računala tj. fizičkih procesora. Konačni cilj ovog rada je implementacija dvaju različitih oblika paralelne optimizacije rojem čestica i korištenje u učenju neuronskih mreža te možebitno davanje odgovora na pitanje koji od ta dva oblika predstavlja bolju rješenje u učenju neuronskih mreža.

2 OPITMIZACIJA ROJEM ČESTICA

Optimizacija rojem čestica (engl. particle swarm optimization) je stohastička optimizacijska tehnika koju su razvili Eberhart i Kennedy 1995. god. inspirirani socijalnim ponašanjem jata ptica ili jata riba.

Socijalni model do kojeg dolaze se zasniva na međusobnoj udaljenosti, tj. smatraju da se ponašanje jata zasniva na održavanju optimalne međusobne udaljenosti i udaljenosti njihovih susjeda [2]. U teoriji, pojedini član jata može profitirati od saznanja i prijašnjih iskustava svih drugih članova koje su prikupili u potrazi za hranom. Ta prednost može prevagnuti u odnosu na nedostatke u modelu u kojem se međusobno natječu u potrazi za hranom [3]. To bi značilo da međusobno dijeljenje informacija daje evolucijsku prednost – napreduju na osnovu prethodnih saznanja svih članova. Ta hipoteza postaje temeljem u daljnjem razvoju metode optimizacije rojem čestica. Nadalje, evolucijski će se objasniti nastanak algoritma radi boljeg pojašnjenja.

2.1 Evolucija algoritma

Prva simulacija jata je bila vrlo brzo izvedena. Zasnivala se na izjednačavanju brzine sa najbližim susjedom. Svi članovi su na početku nasumično raspršeni u prostoru. Algoritam je pokazivao znakove jata, ali članovi su prebrzo izjednačili svoje brzine te su se sjatili u jednu točku pritom ne mijenjajući svoj smjer. Potom su u svakoj iteraciji dodali promjenu u brzini kod nasumično odabranog člana. To je uzrokovalo dovoljnu raznolikost u jatu tako da je simulacija sve više sličila na stvarnu životnu situaciju.

Kao logičan nastavak se uzima Heppnerova simulacija jata ptica koja je uvela točku koja je privlačila jato k sebi. Time se uvela dinamična sila (privlačnost točke i članova jata). Simulacija se odvijala u dvodimenzionalnom prostoru i u toj simulaciji su ptice svoju brzinu korigirale u odnosu na svoju dosadašnju najbolju poziciju te najbolju poziciju bilo kojeg člana jata. Dobrota pozicije se određuje pomoću udaljenosti te pozicije od ciljne točke – što je udaljenost manja to je pozicija bolja. Najbolja dosadašnja pozicija

nekog člana je ona pozicija u kojoj je taj član bio najbliže cilju u nekom od prethodnih koraka. Korekcija brzine po Heppneru se vrši ovako: Ako se član jata nalazi desno od svoje dosadašnje najbolje pozicije umanjuje mu se x komponenta brzine za određenu vrijednost parametra, a ako se nalazi lijevo od dosadašnje najbolje pozicije brzina se povećava za istu tu vrijednost. Isto se radi i s y komponentom brzine, samo što se u ovom slučaju brzina smanjuje ako se član nalazi iznad najbolje pozicije te povećava ako je ispod najbolje pozicije. Dodatno se taj parametar, nazovimo ga $p_{increment}$, množi nasumičnim brojem od 0 do 1. Potom vršimo na isti način korekciju brzine s obzirom na najbolju globalnu poziciju, s time da se parametar $p_{increment}$ mijenja s parametrom $g_{increment}$(global increment). Pseudokod bi izgledao ovako:

Korekcija brzine u odnosu na osobnu najbolju poziciju

$$if\ present_x[] > pbest_x[]\ then\ v_x[] = v_x[] - rand() * p_{increment}$$
$$if\ present_x[] < pbest_x[]\ then\ v_x[] = v_x[] + rand() * p_{increment}$$
$$if\ present_y[] > pbest_y[]\ then\ v_y[] = v_y[] - rand() * p_{increment}$$
$$if\ present_y[] < pbest_y[]\ then\ v_y[] = v_y[] + rand() * p_{increment}$$

Korekcija brzine u odnosu na globalno najbolju poziciju

$$if\ present_x[] > pbest_x[gbest]\ then\ v_x[] = v_x[] - rand() * g_{increment}$$
$$if\ present_x[] < pbest_x[gbest]\ then\ v_x[] = v_x[] + rand() * g_{increment}$$
$$if\ present_y[] > pbest_y[gbest]\ then\ v_y[] = v_y[] - rand() * g_{increment}$$
$$if\ present_y[] < pbest_y[gbest]\ then\ v_y[] = v_y[] + rand() * g_{increment}$$

Uglate zagrade naznačuju da se radi o više jedinki, tj. da se ovaj dio izvršava za svakog člana jata. Varijabla $present_x$ predstavlja trenutnu poziciju jedinke na x-osi , $pbest_x$ najbolju dosadašnju poziciju člana jata, *gbest* predstavlja indeks dosad najuspješnijeg člana, kombinacija $pbest_x[gbest]$ najbolju poziciciju na x-osi cijelog jata, i „v_x" brzinu člana u smjeru x-osi (jednako i za y). Parametrima $g_{increment}$ i $p_{increment}$ određujemo koliki utjecaj na korekciju brzine člana ima najbolji globalni i najbolji osobni rezultat. S velikim $g_{increment}$ te malim $p_{increment}$ simulacija brzo završava te se čini kao da je jato nasilno privučeno k cilju, dok u suprotnom slučaju, kada je $g_{increment}$ mali i $p_{increment}$ veliki, čini se kako jato polako kruži oko konačnog cilja te se realistično približava sinkroniziranim kretnjama podgrupa te u konačnici završavaju na cilju.

Nakon što se vidjelo da je ovakav algoritam u mogućnosti naći minimume za dvodimenzionalne prostore proširuje se na više dimenzija zbog čega se uvode dodatne uglate zagrade koje u ovom slučaju predstavljaju višedimenzionalnost prostora:

$$if\ present[][] > pbest[gbest][]\ then\ v[][] = v[][] - rand() * g_{increment}$$
$$if\ present[][] < pbest[gbest]\ [] then\ v[][] = v[][] + rand() * g_{increment}$$
$$if\ present[][] > pbest[][]\ then\ v[][] = v[][] - rand() * p_{increment}$$
$$if\ present[][] < pbest[][] then\ v[][] = v[][] + rand() * p_{increment}$$

Daljnjom razradom algoritma više se ne koristi usporedba trenutne i najbolje pozicije te respektivno povećanje i smanjenje brzine za određeni parametar, već se za to koristi razlika trenutne i najbolje pozicije. Dodatno se time omogućuje da član brže akcelerira ako je dalje od trenutno najboljeg i suprotno u protivnom slučaju. Pseudokod za takav oblik bi izgledao ovako:

$$v[][] = v[][] + rand() * p_{increment} * (pbest[][] - present[][])$$
$$v[][] = v[][] + rand() * g_{increment} * (pbest[gbest][] - present[][])$$

Što nas dovodi i do konačne pojednostavljene verzije:

$$v[][] = v[][] + 2 * rand() * (pbest[][] - present[][]) +$$
$$2 * rand() * (pbest[gbest][] - present[][])$$

Stohastički faktor se u ovom slučaju množi sa 2 kako bi srednja vrijednost faktora bila 1. Takva promjena omogućuje da prosječno u polovici slučajeva član preleti metu.

Kroz evoluciju i simplifikaciju algortima ponašanje jata postajalo je sve sličnije roju, te u konačnici zadovoljava 5 principa inteligencije roja po Millonasu [4]:

1. Princip bliskosti - populacija treba biti u mogućnosti izračunati jednostavne vremenske i prostorne račune
2. Princip kvalitete - populacija mora biti u mogućnosti reagirati na kvalitativne faktore okoline
3. Princip raznolikosti reakcije - populacija ne smije izrazito jednoliko reagirati
4. Princip stabilnosti - populacija ne smije mijenjati svoje ponašanje svaki put kada se promijeni okolina
5. Princip prilagodbe - populacija treba promijeniti svoje ponašanje kada je rezultat vrijedan vremenskog troška

Iz tog razloga ćemo u daljnjem tekstu koristiti pojmove čestice i roja umjesto dosadašnjih članova i jata.

2.2 Osnovni oblik

U prethodnom poglavlju smo mogli vidjeti načine na koje je evoluirao oblik i konačnu pojednostavljenu verziju računa novih brzina čestica u roju. Uz malu modifikaciju, dodavanjem inercijske težine w, koristimo ga i u konačnoj definiciji algoritma. Inercijska težina w je definirana u intervalu od 0 do 1 i njome određujemo koliko brzina čestice iz prethodnog koraka djeluje na brzinu čestice u trenutnom koraku. Smanjivanjem inercijske težine smanjujemo inerciju čestice te time dajemo prednost lokalnoj pretrazi, a njezinim povećanjem postižemo suprotni učinak tj. dajemo veću prednost globalnom pretraživanju. Lagano je zaključiti razlog tomu: povećanjem inercije čestice rade veće pomake u koraku i na taj način pretražuju globalno prostor, smanjenjem inercije se radijus kretanja u koraku smanjuje i time čestice lokalno pretražuju prostor.

Do sada smo rabili pojmove kao što su bliže, dalje, udaljenost od cilja i najbolja pozicija pa ćemo to malo pobliže objasniti i preformulirati korištenjem funkcije dobrote. Funkcija dobrote je funkcija koja kvalitativno ocjenjuje vrijednosti čestice u odnosu na cilj koji pokušavamo postići. Uspoređujući rezultate funkcije dobrote ili ,kraće, dobrotu dviju različitih čestica, možemo odrediti koja od njih je bolja ili lošija, tj. respektivno bliža ili udaljenija od konačnog cilja.

Sada možemo dati konačni pseudokod algoritma koji glasi:

1. Nasumično rasporedi sve čestice unutar prostora kojega pretražujemo i spremi vrijednosti u $present[][]$
2. Postavi sve brzine čestica na nulu
3. Napravi evaluaciju funkcije dobrote za svaku česticu
4. Ako je dobrota čestice bolja od dosadašnja najbolje dobrote te čestice spremi vrijednosti čestice u $pbest[][]$
5. Među najboljim osobnim pozicijama pronađi globalno najbolju česticu i postavi gBest tako da ukazuje na dosadašnju globalno najbolju vrijednost čestice ($pBest[gBest][]$)
6. Ako globalno najbolja čestica $pbest[gbest][]$ ispunjava tražene uvjete, zaustavi algoritam i vrati vrijednosti te čestice, inače nastavi
7. Izračunaj nove brzine čestice $v[][]$ prema:

$$v[][] = w * v[][] + p_{increment} * rand() * (pbest[][] - present[][]) + g_{increment} * rand() * (pbest[gbest][] - present[][])$$

8. Ako brzine čestica izlaze iz dozvoljenog intervala postavi ih na najbliži rub intervala
9. Izračunaj nove pozicije čestica prema:

$$present[][] = present[][] + v[][]$$

10. Ako vrijednosti čestica izlaze izvan prostora kojeg pretražujemo postavimo ih na najbliži rub tog prostora
11. Vrati se na korak 3.

Često ćemo koristiti u radu pojam iteracije pa ćemo ga malo točnije definirati. Iteracijom nazivamo jedan ciklus u algoritmu optimizacije rojem čestica u kojem modificiramo brzine i vrijednosti svih čestica i izračunamo njihove nove vrijednosti dobrote.

Preporuča se korištenje maksimalnog broja iteracija u kriteriju zaustavljanja jer postoji mogućnost da se algoritam beskonačno izvodi. Ali taj kriterij nikako ne bi smio biti jedini. Preporuča se provjera konvergencije postupka. Primjerice, ako se u određenom broju iteracija ne promijeni globalno najbolja dobrota, možemo pretpostaviti da je pronađeno konačno rješenje i završiti izvođenje. Kažemo pretpostaviti jer postoji mogućnost da je algoritam zapeo u lokalnom minimumu. Drugi kriterij bi mogao biti, u

slučaju da možda znamo konačnu vrijednost funkcije dobrote koju želimo postići, korištenje te vrijednosti kao praga za zaustavljanje algoritma.

U pseudokodu algoritma možemo primijetiti da uvijek pamtimo osobne najbolje vrijednosti čestica. To nam omogućuje da iako čestice konstantno putuju po prostoru i možda se udaljavaju od svojih prethodno najboljih pozicija, ona najbolja pozicija, u kojoj se neka čestica tokom izvođenja našla, ostati će uvijek zapamćena i vraćena nakon što algoritam ispuni neki od kriterija zaustavljanja. Najveći problem u korištenju optimizacije rojem čestica je taj da postupak ne može garantirati da je dano rješenje stvarno globalni minimum ili maksimum neke funkcije, iako pokazuje jako dobre karakteristike globalnog pretraživanja prostora.

2.3 Paralelna optimizacija rojem čestica

U izradi paralelne optimizacije rojem čestica, kako i drugih paralelnih algoritama, bilo je potrebno obratiti pozornost na nekoliko sljedećih bitnih svojstava:

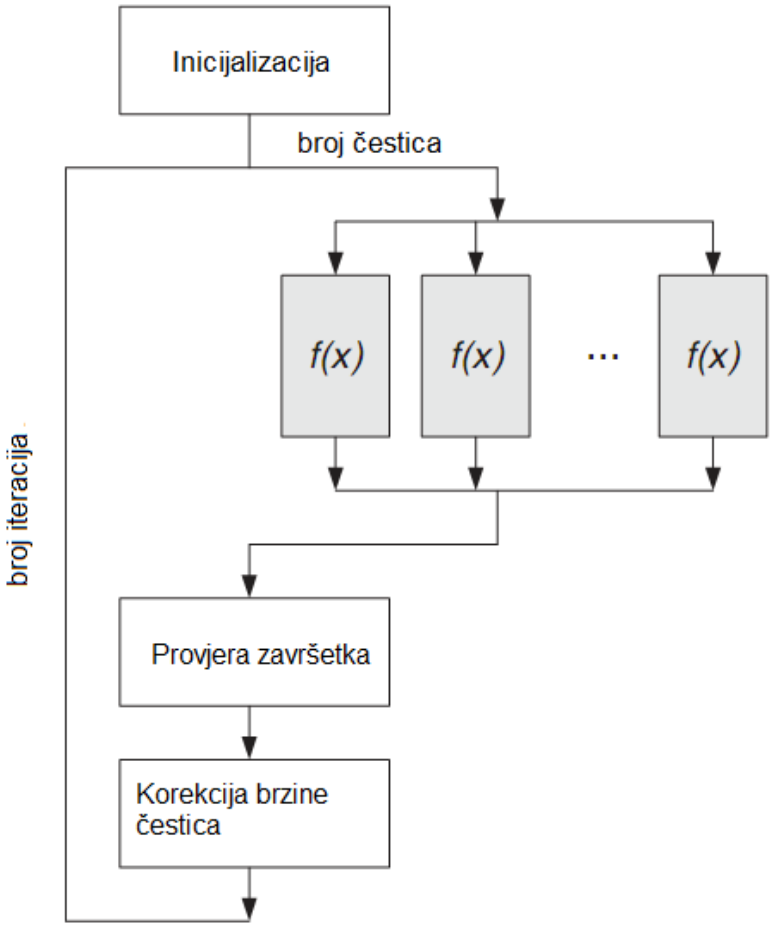
- Istodobnost - algoritam bi trebao biti u mogućnosti napraviti podjelu na više radnji koje bi se mogle izvršavati istodobno
- Skalabilnost - mogućnost izvođenja algoritma na proizvoljnom broju računala tj. da smo u mogućnosti dodati računala (fizičkih procesora) u mrežu i time dodatno ubrzati izvođenje
- Lokalnost - veći omjer lokalnog u odnosu na udaljeni pristup memoriji
- Modularnost - mogućnost uporabe dijelova algoritma unutar različitih programa

U ovom radu su izvedena dva oblika paralelne optimizacije rojem čestica, asinkrona i sinkrona implementacija. Obje implementacije koriste voditelj-radnik model kako bi se osigurala što veća skalabilnost i učinkovitost algoritma. Radnici su u ovom slučaju zaduženi samo za računanje vrijednosti funkcije koju pokušavamo minimizirati, dok se voditelj brine o preraspodijeli posla i provjeri završetka optimizacije. Algoritam optimizacije rojem čestica je pogodan za paralelnu implementaciju jer se zasniva na

populaciji jedinki koje su nezavisne jedna od druge tj. njihova funkcija dobrote koju je potrebno izračunati ovisi samo o njihovim vlastitim vrijednostima.

2.3.1 Paralelna sinkrona optimizacija rojem čestica

Sinkrona verzija algoritma se u potpunosti zasniva na osnovnom algoritmu i ako bi brojevi, koji se generiraju nasumično, bili isti u paralelnoj i osnovnoj verziji algoritmi bi davali identične rezultate. Problem koji se javlja kod sinkrone verzije je taj da nismo u mogućnosti maksimalno iskoristiti potencijal raspoloživih fizičkih procesora. Zbog osnovnog algoritma po kojemu se korekcija brzina i vrijednosti svih čestica računa tek na kraju iteracije, nakon što smo izračunali funkciju dobrote za svaku česticu iz roja, paralelna implementacija je primorana čekati rezultat zadnje čestice iako procesori radnici za to vrijeme ne obavljaju nikakav koristan rad već samo čekaju novi zadatak kao što se može vidjeti na slici 2.1.



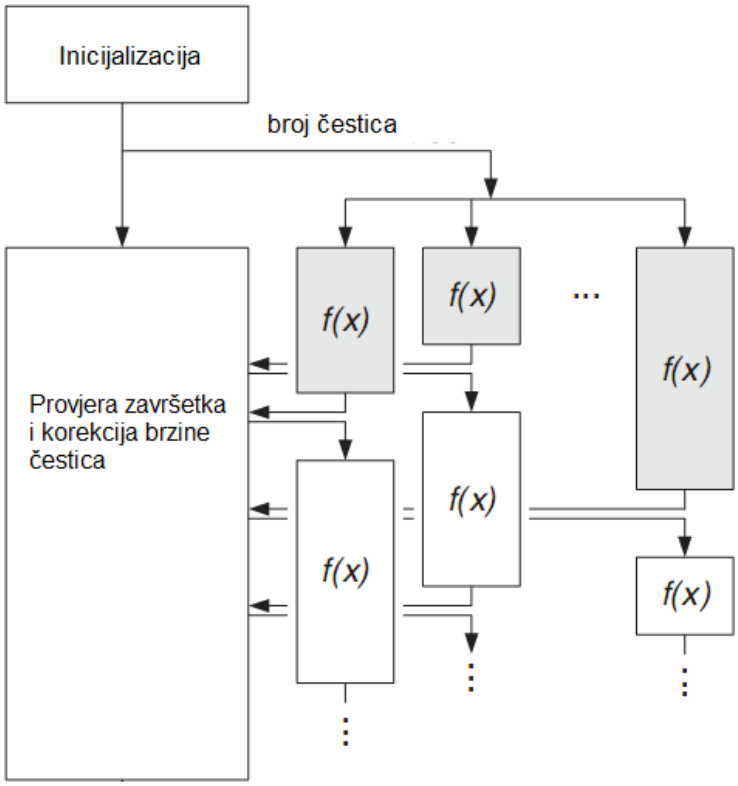
Slika 2.1. Dijagram toka paralelne sinkrone optimizacije rojem čestica

Kako bi se osigurala maksimalna iskoristivost u ovom obliku bilo bi potrebno zadovoljiti tri uvjeta, ekskluzivni pristup homogenoj mreži računala, konstantno vrijeme za izračun funkcije dobrote i broj čestica koji je djeljiv sa brojem procesora umanjen za 1 (uzimajući u obzir voditelj-radnik model). U našem slučaju drugi uvjet je zadovoljen, neuronske mreže koje učimo, učimo uvijek na istom broju primjera. Za treći uvjet morali bismo u startu prilagoditi broj čestica s kojima želimo raditi što više predstavlja neugodnost nego neko ozbiljno ograničenje jer veći broj čestica osigurava da optimizacija ne zapne u lokalnom minimumu. Prvi uvjet je najproblematičniji, budući da su mreže računala na kojima se izvršavaju paralelni programi rijetko kada sastavljene od identičnih računala i praktički nikada nismo jedini koji ih koriste. Velika prednost sinkrone implementacije je baš to što se u potpunosti podudara sa slijednim algoritmom, tako da bi sva dosadašnja iskustva i istraživanja trebala vrijediti i za ovaj oblik implementacije.

2.3.2 Paralelna asinkrona optimizacija rojem čestica

Paralelna asinkrona implementacija radi na ponešto drugačijem obliku u odnosu na osnovni sekvencijalni oblik. Dok sinkrona implementacija vrši korekciju brzine i provjeru završetka nakon svake iteracije, tj. tek nakon što izračunamo funkcije dobrote cijelog roja, asinkrona implementacija obavlja korekciju brzine za česticu onog trenutka kada saznamo njezinu dobrotu što se može dobro vidjeti iz dijagrama toka za paralelnu asinkronu implementaciju (slika 2.2.) [5].

Prema prijašnjim istraživanjima takav oblik ima poboljšanu brzinu konvergencije prema [6,7]. Nažalost autori tih djela ne spominju moguću povećanu opasnost od zapinjanja u lokalnom minimumu i istraživanja koja su rađena ne pokrivaju toliko široki spektar problema koji se pokušavaju riješiti optimizacijom rojem čestica, tako da se ne može sa sigurnošću tvrditi kako asinkroni oblik pokazuje ubrzanu konvergenciju za sve vrste problema koji bi mogli naići, iako postoji logička osnova za tako nešto. Pretpostavlja se da je razlog bržoj konvergenciji to što čestice brže reagiraju na promjene globalnih najboljih vrijednosti, ali za tako nešto nema konkretnih dokaza.



Slika 2.2. Dijagram toka parelelne asinkrone optimizacije rojem čestica

U paralelnoj implementaciji asinkroni oblik nam omogućava da razriješimo probleme sa kojima smo se susreli kod sinkronog oblika. Više nismo primorani čekati rezultat zadnje čestice iz roja kako bismo napravili korekciju brzina, već radniku možemo predati sljedeću česticu u nizu za koju nam je potrebna dobrota. Tu možda leži i najveći problem paralelne asinkrone implementacije, jer se gubi ekvivalentnost sa sekvencijalnim oblikom algoritma. Također postoji mogućnost da se izvršavanje zadataka na pojedinim procesorima izvršava puno duže nego na nekim drugim procesorima i u tom slučaju bi se moglo dogoditi da sporiji procesori zadržavaju tu česticu, u ekstremnim slučajevima, i po više ciklusa. To znači da bi za ostale čestice koje su se izvršavale na bržim procesorima više puta napravili korekciju smjera same čestice u odnosu na onu česticu za koju smo trebali čekati duže vrijeme kako bismo dobili rezultat. Iz tog razloga gubimo raznovrsnost kojom bi ta čestica mogla pridonijeti u istraživanju prostora rješenja za vrijeme tih ciklusa. U konačnici možemo ustvrditi kako bismo osigurali maksimalnu učinkovitost izvođenja jedini preduvjet koji moramo zadovoljiti je da broj čestica roja ne bude manji od broja procesora na kojima se izvršava algoritam, jer to je ujedno i broj zadataka koji maksimalno možemo zadati u jednom trenutku, i da, ako želimo izbjeći preskakanje ciklusa, trebamo koristiti procesore barem približne brzine izvođenja.

3 UMJETNE NEURONSKE MREŽE

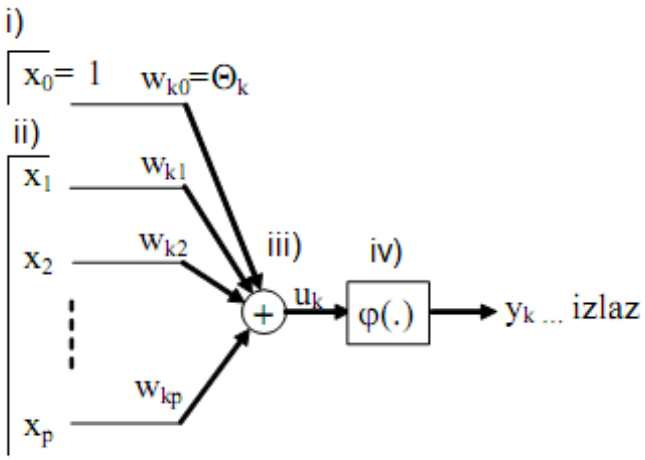
Umjetne neuronske mreže nalaze mnoge primjene u praksi zbog svoje jednostavnosti implementacije te velikih mogućnosti za učenje različitih problema. Neuronske mreže imaju sposobnost učenja na temelju primjera, kao što i npr. mala djeca uče razliku između psa i mačke. Ako treniramo neuronsku mrežu oprezno, postoji velika šansa da će pokazati mogućnost generaliziranja i izvan primjera za učenje. Baš iz tog razloga su neuronske mreže korištene u mnogim područjima, jer su u mogućnosti naučiti najrazličitije stvari jednim te istim procesom učenja što nam omogućava jednostavnu implementaciju na računalima.

Inspiracija za umjetnim neuronskim mrežama proizlazi iz promatranja načina na koji funkcionira biološka neuronska mreža. Ne treba posebno napominjati da postoji želja za stvaranje umjetnog sistema koji bi bio u mogućnosti vršiti sofisticirani račun na način na koji to čini ljudski mozak. Često se u radovima, zbog podužeg naziva, umjetne neuronske mreže nazivaju samo neuronskim mrežama tako da ćemo i u ovom radu učiniti isto.

Ne postoji opći konsenzus o definiciji neuronskih mreža, iako se većina definicija neznatno razlikuje, posebno bismo izdvojili sljedeću definiciju koja direktno rabi i usporedbu sa radom mozga [8]: Umjetna mreža je masivno paralelni raspodijeljeni procesor koji je dobar za pamćenje iskustvenog znanja i slična je mozgu u sljedeća dva aspekta:

i) Mreže stječu znanje kroz proces učenja
ii) Težine međusobnih veza između neurona se koriste za spremanje znanja

3.1 Model neurona



Slika 3.1. Model neurona

Neuron je osnovna građevna jedinica neuronske mreže, koji je zamišljen kao matematička funkcija koja predstavlja primitivan model tj. svojevrsnu apstrakciju djelovanja biološkog neurona. Neuron je sastavljen od 4 glavna dijela (slika 3.1.):

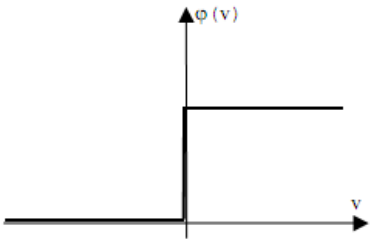
i) Niz ulaza od kojih svaki ima svoju težinu – težina je u biti, matematički gledano, množitelj kojim se množi ulaz u neuron (Notacija: Ulaz x_j na ulazu j neurona k ima težinu w_{kj}). Prag neurona može biti predstavljen kao dodatan ulaz, znači na ukupan broj ulaza u neuron dodajemo još jedan koji također ima svoju težinu ali je ulaz zato konstantan i iznosi 1.

ii) Sumator otežanih ulaza – središnji dio neurona koji obavlja linearnu kombinaciju otežanih ulaza tj. sumiranje svih ulaza pomnoženih odgovarajućim težinama

iii) Aktivacijska funkcija – koja može biti linearna ili nelinearna, a izbor najviše ovisi o problemu kojeg pokušavamo riješiti. Neke od češće korištenih funkcija možemo vidjeti na slici 3.2.

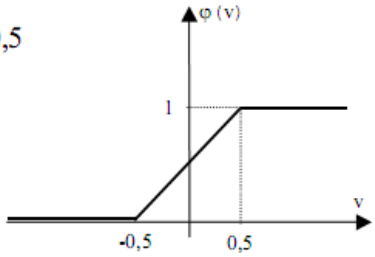
a)

$$\varphi(v)=\begin{cases}1, & v\geq 0\\ 0, & v<0\end{cases}$$



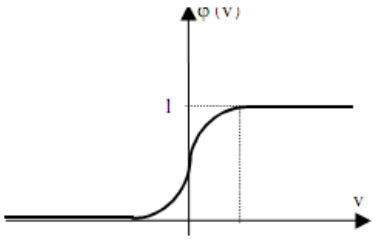
b)

$$\varphi(v)=\begin{cases}1, & v\geq 0{,}5\\ v+0{,}5, & -0{,}5<v<0{,}5\\ 0, & v\leq -0{,}5\end{cases}$$



c)

$$\varphi(v)=\frac{1}{1+\exp(-av)}$$



Slika 3.2.Različiti tipovi aktivacijske funkcije: a) prag, b) linearna po odsječcima, c) sigmoidna funkcija

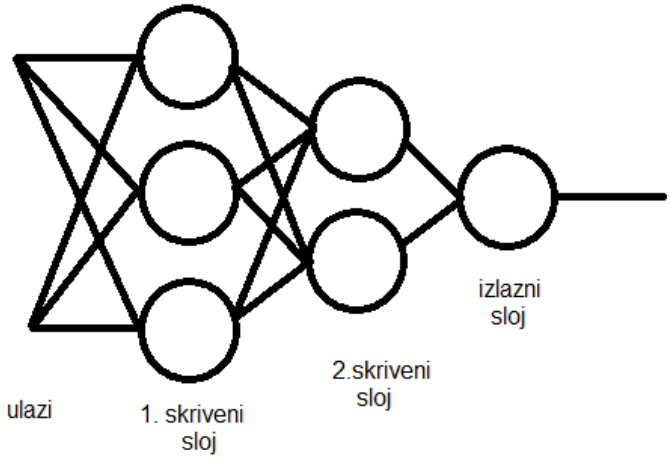
3.2 Arhitektura mreže

Postoji više različitih arhitektura neuronskih mreža, ali im je svima zajedničko to da su sastavljeni od iste osnovne građevne jedinice, neurona. Neuronske mreže su sastavljene od jednog ili više međusobno povezanih slojeva, koji su sastavljeni od jednog ili više neurona (izlazi iz neurona predstavljaju ulaz nekom drugom neuronu) .

Neuronske mreže možemo s obzirom na način djelovanja podijeliti na neuronske mreže sa i bez povratnih veza.

3.2.1 Neuronske mreže bez povratnih veza

Neuronske mreže bez povratnih veza se sastoje od jednog ili više slojeva u kojima veze između neurona ne čine cikluse tj. nakon što jedan sloj obradi podatke, izlazi neurona iz tog sloja šalju se na ulaze u sljedeći sloj. Uglavnom se koriste neuronske mreže koje su u potpunosti povezane, što znači da svaki izlaz iz jednog sloja dolazi na svaki ulaz u drugom sloju. Postoji i mogućnost da koristimo djelomičnu povezanost između nekih slojeva i u tim slučajevima to je potrebno posebno naglasiti. Često se u literaturi može naići na različite načine brojanje slojeva. Neki ne broje ulazni sloj, ali broje izlazni ili obrnuto. Zato je najsigurnije navesti broj sakrivenih slojeva od kojih je mreža načinjena, a ulazni i izlazni bi se morali podrazumijevati, znači mreža od 2 skrivena sloja predstavlja neuronsku mrežu sa jednim ulaznim slojem, 2 skrivena sloja i jednim izlaznim slojem. Primjer jedne takve mreže možemo vidjeti na slici 3.3.



Slika 3.3. Potpuno povezana neuronska mreža

Spomenut ćemo samo dva najpoznatija oblika neuronskih mreža bez povratne veze, a to su:

- Linearne neuronske mreže – mreže koje imaju linearnu aktivacijsku funkciju. Svi oblici linearnih neuronskih mreža (sa bilo kojim brojem skrivenih slojeva i neurona u njima) u pravilu se mogu svesti na osnovni oblik bez skrivenih slojeva gdje imamo samo ulazni i izlazni sloj, pošto je linearna funkcija od linearne funkcije također linearna funkcija.

- Višeslojni perceptron (MLP – *multilayer perceptron*) – ima tri glavna svojstva: model neurona koji se koristi ima nelinearnu izlaznu aktivacijsku funkciju koja je glatka, mreža sadrži jedan ili više skrivenih slojeva i dobro je povezana. Dokazano je da su takve mreže u mogućnosti izračunati bilo koju funkciju, ako obuhvaćaju dovoljan broj neurona u skrivenom sloju.

3.2.2 Neuronske mreže sa povratnim vezama

Neuronske mreže sa povratnim vezama čine cikluse u povezivanju neurona. Uobičajeno povratne veze unutar takvih mreža se kombiniraju sa elementima za kašnjenje, čime se otvara mogućnost memoriranja te tako dobivamo nelinearne dinamičke sustave i mogućnost rješavanja problema kojih inače ne bismo mogli riješiti korištenjem neuronskih mreža bez povratne veze.

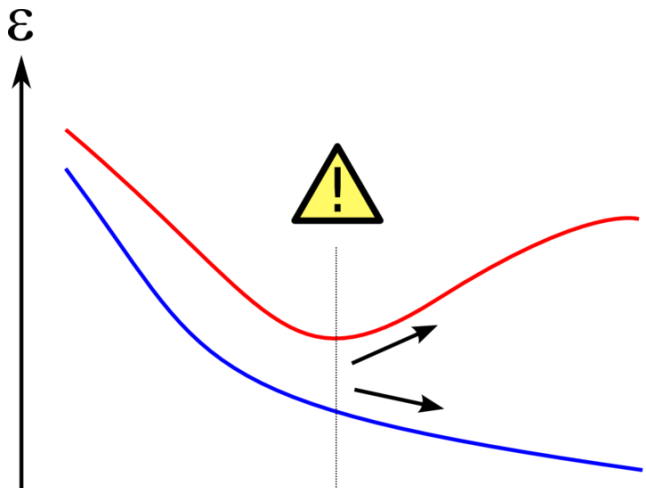
3.3 Učenje neuronske mreže

Učenje neuronske mreže može biti veoma komplicirano i stoga predstavlja najveću prepreku njihovom korištenju. Možemo u startu vidjeti da to nije lak posao gledajući samo na količinu različitih postupaka koji se redovno koriste u učenju neuronskih mreža, od kojih su neki posebno za tu svrhu i razvijeni. Samo poznatije među njima bi bile: BP (BackPropagation), BFGS (Broyden-Fletcher-Goldfarb-Shanno metoda), CG (*Conjugate Gradient*), PCG (*Preconditioned Conjugate Gradient*), Rprop (*Resilient backprogation*), Bayesovo učenje, Levenberg-Marquardt algoritam… Od svih navedenih algoritama ovdje ćemo pokazati osnovni princip rada algoritma povratne propagacije pogreške (*backpropagation*) koji je vrlo vjerojatno i najpoznatiji među njima i predstavlja dobru podlogu za shvaćanje osnovnih principa na kojima neuronska mreža uči i djeluje. Ali prije nego što opišemo taj postupak potrebno je još objasniti neke pojmove.

3.3.1 Populacija, primjeri za učenje, validaciju i ispitivanje

Rijetko kada je korisno korištenje neuronskih mreža da samo zapamti podatke za učenje. Mi u stvari pokušavamo naučiti neuronsku mrežu primjerima za učenje tako da daje točne rezultate i na novim, dosad neviđenim podatcima tj. da neuronska mreža dobro generalizira. Uobičajeno se skup svih mogućih slučajeva na koje želimo generalizirati naziva populacija. Primjerima nazivamo određene podskupove iz te populacije. U svakom učenju postoji mogućnost prenaučenosti tj. mogućnost da neuronsku mrežu toliko dobro naučimo na primjerima za učenje da izgubi mogućnost generaliziranja nad novim podatcima. Budući da je naš cilj u učenju neuronskih mreža imati što bolje rezultate na novim podatcima najjednostavniji pristup je evaluirati mrežu na primjerima nezavisnim od primjera za učenje. Razne mreže treniramo minimiziranjem odgovarajuće funkcije pogreške u odnosu na primjere za učenje, stoga performanse mreže možemo ocijeniti na temelju funkcije pogreške nad nezavisnim skupom, te odabiremo mrežu koja ima najmanju pogrešku nad tim skupom. Takav skup primjera nazivamo primjerima za validaciju. Pošto ovaj postupak može pak uzrokovati pretreniranost nad validacijskim skupom, performanse konačne mreže moraju biti potvrđene na trećem nezavisnom skupu iliti skupu primjera za ispitivanje [9].

Jedan od primjera korištenja validacijskog skupa je u metodi ranog zaustavljanja. Rano zaustavljanje je metoda kojom se pokušava prekinuti učenje neuronske mreže kako bismo spriječili pretreniranost neuronske mreže. Obično se koristi validacijski skup kako bismo ocijenili kvalitetu generalizacije i ako bismo primijetili da se generalizacija pogoršava prekinuli učenje. Na taj način smo validacijskim skupom utjecali na učenje i ne bismo ga smjeli koristiti u konačnoj ocjeni kvalitete generalizacije. Na slici 3.4. možemo vidjeti kako s vremenom pogreška nad primjerima za učenje opada zajedno sa pogreškom nad skupom za validaciju, ali nakon nekog vremena se pogreška na validacijskom skupu počinje povećavati dok i dalje pada na skupu za učenje.



Slika 3.4. Kretanje pogreške na skupu za učenje (plavo) i skupu za validaciju (crveno) tokom učenja neuronske mreže

3.3.2 Učenje algoritmom povratne propagacije pogreške

Učenje povratnom propagacijom pogreške zahtijeva da neuronska mreža koristi aktivacijsku funkciju koja je diferencijabilna. Izvod je preuzet iz rada [10]. Krećemo od dviju osnovnih formula za neuron j gdje je ukupna aktivacija neurona na ulazu u nelinearni blok jednaka:

$$v_j(n) = \sum_{i=0}^{p} w_{ji}(n) y_i(n) \quad (1)$$

i izlaza neurona koji je jednak:

$$y_j(n) = \varphi_j\left(v_j(n)\right) \quad (2)$$

Kod učenja neuronskih mreža pokušavamo smanjiti razliku između izlaza koje daje neuronska mreža i izlaza danih primjera za učenje. Ocjenu kvalitete neuronske mreže stoga možemo procijeniti na temelju funkcije prosječne kvadratne pogreške za sve uzorke (3).

$$E_{sr} = \frac{1}{N} \sum_{n=1}^{N} E(n) \quad (3)$$

gdje $E(n)$ predstavlja trenutnu, u koraku n, kvadratnu pogrešku i dana je izrazom:

$$E(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \quad (4)$$

u kojem C označava skup izlaznih neurona, a e_j razlika između željenog i dobivenog odziva za primjer n na izlaznom neuronu j i dana je izrazom (5).

$$e_j(n) = d_j(n) - y_j(n) = d_j(n) - \varphi_j\left(v_j(n)\right) \quad (5)$$

Znači naš osnovni cilj postaje smanjiti sumu kvadrata razlike kako bismo smanjili pogrešku neuronske mreže, što znači da bismo trebali pronaći minimum te funkcije, a da bismo to napravili u ovom slučaju koristimo metodu najbržeg spusta.

Metoda najbržeg spusta je optimizacijska metoda prvog reda, kojom se pokušava naći lokalni minimum funkcije na takav način da vrijednosti, o kojima ovisi funkcija, korigiramo u koracima jednakim negativnoj proporcionalnoj vrijednosti gradijenta dane funkcije. Kako bismo to napravili moramo pronaći gradijent funkcije u ovisnosti o težinama neuronske mreže koju pokušavamo učiti, a to je parcijalna derivacija sume kvadrata pogreške po težini, koja glasi:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)}\frac{\partial e_j(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)}\frac{\partial v_j(n)}{\partial w_{ji}(n)} \quad (6)$$

gdje za parcijalne derivacije vrijedi:

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \quad (7) \qquad \frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j{}'\left(v_j(n)\right) \quad (8)$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \quad (9) \qquad \frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \quad (10)$$

Uvrštavanjem 7, 8, 9 i 10 u (6) dobivamo formulu za gradijent:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'\left(v_j(n)\right)y_i(n) \quad (11)$$

Korekcija težina nam je definirana delta pravilom i iznosi:

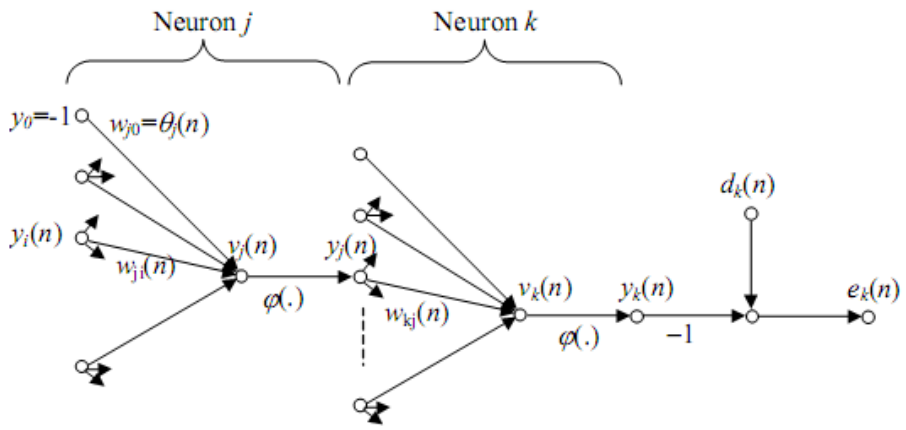
$$\Delta w_{ji}(n) = -\eta\frac{\partial E(n)}{\partial w_{ji}(n)} = \eta e_j(n)\varphi_j'\left(v_j(n)\right)y_i(n) = \eta\delta_j(n)y_i(n) \quad (12)$$

gdje je $\delta_j(n)$ tzv. lokalni gradijent:

$$\delta_j(n) = e_j(n)\varphi_j'\left(v_j(n)\right) \quad (13)$$

U slučaju kada se neuron j nalazi u skrivenom sloju (slika 3.5.) onda nije poznat željeni odziv tog neurona pa ni pogreška. Zato pokušavamo procijeniti tu pogrešku na temelju pogrešaka neurona sljedećeg sloja na koje je taj neuron spojen, zbog čega imamo drukčiju formulu za gradijent. Po prethodno izvedenim izrazima možemo pisati za lokalni gradijent:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)}\varphi_j'\left(v_j(n)\right) \quad (14)$$



Slika 3.5.

Parcijalnom derivacijom izraza (4) dobivamo da vrijedi:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_{\mathrm{k}} e_k \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_{\mathrm{k}} e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \quad (15)$$

I parcijalnim derivacijama (5) i (1) respektivno dobivamo (16) i (17):

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'\big(v_k(n)\big) \quad (16)$$

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \quad (17)$$

Konačnim uvrštavanjem izraza (15), (16) i (17) u (14) i dodatnim uređenjem po izrazu (13) dobivamo konačnu formulu za lokalni gradijent skrivenog neurona:

$$\delta_j(n) = -\varphi_j'\Big(v_j(n)\Big)\sum_{\mathrm{k}} -e_k\varphi_k'\big(v_k(n)\big)w_{kj}(n) = \varphi_j'\Big(v_j(n)\Big)\sum_{k}\delta_k w_{kj}(n) \quad (18)$$

Konačni algoritam povratne propagacije pogreške se izvodi uzastopnim prezentiranjem primjera i svako učenje jednog primjera se sastoji od dva prolaza. U prvom prolazu se krećemo mrežom prema naprijed i računamo izlaze iz svih neurona po izrazima za internu aktivnost (1) i izlaz (2) pojedinog neurona. U drugom koraku se krećemo mrežom unatrag i signali pogreške se propagiraju od izlaznog sloja prema ulazu, sloj po sloj, rekurzivno računajući lokalni gradijent za svaki neuron koristeći izraze (13) i (17) u ovisnosti o tome da li se neuron nalazi u izlaznom ili skrivenom sloju, te na temelju delta pravila iz izraza (11) računamo korekciju težine. Nakon korekcija nastavljamo sa sljedećim primjerom sve dok nije zadovoljen neki od uvjeta za zaustavljanje. Postoji i još neznatno drugačija vrsta algoritma u kojoj mreži prezentiramo sve primjere iz skupa za učenje odjednom, te lokalni gradijent u takvom slučaju računamo sumiranjem lokalnih

gradijenta za svaki primjer. Na taj način dobivamo točniju procjenu gradijentnog vektora s kojim vršimo korekciju težina. Nedostatak je u tome što je potrebno više memorije i svi uzorci za učenje moraju biti dostupni na samom početku učenja (*off-line* učenje) za razliku od učenja uzorak po uzorak gdje možemo nakon svakog koraka dodati novi uzorak (*on-line* učenje).

3.3.3 Inicijalizacija težina

Ako postoji neko *a priori* znanje o mreži koju učimo mogli bismo to znanje iskoristiti kako bismo postavili inicijalne težine na određene vrijednosti. U protivnom slučaju se predlaže da postavimo incijalne težine na male slučajne vrijednosti uniformno distribuirane na nekom intervalu. Ovdje ćemo spomenuti samo jednu od mogućih metoda koja pokazuje dosta dobre rezultate u treniranju neuronske mreže, a naziva se Nguyen-Widrow inicijalizacija sloja. Zasniva se na činjenici da optimizacijski postupak ovisi o inicijalno postavljenim težinama veza. Primjerice, u mrežama koje koriste sigmoidnu funkciju nasumično odabrane težine mogu dovesti do preranog zasićenja pojedinih neurona. Recimo, primjerice, da koristimo tangens hiperbolni kao aktivacijsku funkciju, njena domena je beskonačna, a kodomena u intervalu od -1 do 1. Tangens hiperbolni već za ulazni interval od -2 do 2 daje izlazne vrijednosti u intervalu od -0.96 do 0.96, tako da bi se bilo koji ulaz veći od tog intervala mogao smatrati suvišnim jer nam ne radi razliku na izlazu ili je minimalna. U tom slučaju kažemo da je neuron u zasićenju. Kada se neki neuron nalazi u zasićenju malim promjenama težine nećemo biti u mogućnosti vidjeti promjenu u funkciji srednje pogreške na izlazu iz mreže, pa će trebati neko vrijeme kako bi se funkcija nastavila smanjivati. Glavna ideja Nguyen-Widrow inicijalizacije je ta da odabiremo male vrijednosti za inicijalne težine i da nakon toga te vrijednosti modificiramo na takav način da su regije interesa neurona podijeljene na male intervale. Jednostavnije rečeno, generiramo težinske vrijednosti tako da distribuiramo aktivnu regiju (nezasićeni interval) svakog neurona otprilike ravnomjerno kroz cijeli ulazni prostor. Nećemo navoditi izvod za Nguyen-Widrow inicijalizaciju, ali ćemo ukratko objasniti postupak koji je dobro opisan u radu [12].

Za slojeve sa m neurona i p ulaza (uključujući i prag), potrebno je prvo generirati m x (p-1) matricu nasumičnih vrijednosti unutar intervala -1 do 1, koju je potrebno

normalizirati po redovima (19). Potom možemo izračunati G po izrazu (20). Nakon toga računamo težine W za svaki ulaz u neuron po izrazu (21) i pragove po izrazu (22).

$$A(j,:) = \frac{A(j,:)}{\|A(j,:)\|} \quad (19)$$

$$G = 0.7m^{\frac{1}{p-1}} \quad (20)$$

$$W(j, 1{:}\,p-1) = G \cdot A(j,:), \text{za j} = 1, 2, \dots, \text{m} \quad (21)$$

$$W(j,p) = \text{sgn}\big(W(j,1)\big) \cdot G \cdot \beta_j, \qquad \text{za } \beta_j = -1{:}\frac{2}{m-1}{:}\,1 \quad (22)$$

Ovaj osnovni algoritam se odnosi na slučajeve kada nam se ulazi i aktivne regije neurona nalaze u intervalu od -1 do 1. Ako nam se aktivne regije neurona nalaze u nekom drugom intervalu, primjerice, od a do b potrebno je dodatno skalirati težine veza i pragove prema izrazima (23), (24), (25) i (26).

$$s = \frac{1}{2}(b-a) \quad (23)$$

$$t = \frac{1}{2}(b+a) \quad (24)$$

$$W(j, 1{:}\,p-1) = s \cdot W(j, 1{:}\,p-1) \quad (25)$$

$$\text{W}(j,p) = sW(j,p) + t \quad (26)$$

3.3.4 Pretprocesiranje ulaza

Ulazi koje neuronske mreže primaju mogu biti svakojaki i ovise najviše o problemu kojeg rješavamo. Neuronske mreže zahtijevaju numeričke kontinuirane ulaze, pa do problema dolazi kad na ulazu imamo kategoričke ulaze, primjerice spol ili rasu. Ponekad se takve varijable kodiraju tako da im se pridijele numeričke vrijednosti i da ih se tretira kontinuiranima, ali najveći problem kod takvog kodiranja je što se definira metrika koja nije razumna. Zato se kategoričke vrijednosti kodiraju tako da imamo onoliko različitih ulaza u neuronsku mrežu koliko različitih vrijednosti kategorička varijabla može poprimiti. Pripadnost određenoj vrijednosti u kategoriji onda možemo kodirati tako da na jedan od ulaza stavimo jedinicu, a na ostale nulu. Na isti način možemo kodirati i izlaz iz neuronske mreže kada ju učimo klasifikaciji. U slučaju kada imamo samo dvije kategorije moguće je koristiti samo jedan izlaz iz neuronske mreže gdje bi nula označavala pripadnost jednoj kategoriji, a jedinica drugoj.

Još jedan problem koji se javlja kod ulaznih vrijednosti u neuronsku mrežu je taj da mogu imati jako velike vrijednosti i uzrokovati zasićenje neurona na koje dolaze. Iako neuronska mreža ima mogućnost svojim težinama umanjiti vrijednost takvim ulazima, u startu usporavamo učenje neuronske mreže i povećavamo mogućnost da zaglavimo u lokalnom minimumu. Stoga je preporučljivo normalizirati sve ulaze u neuronsku mrežu kako ne bi dolazilo do takvih problema. Dodatni razlog normaliziranja ulaza je kako bismo mogli koristiti algoritam iz prethodnog poglavlja koji nam dodatno ubrzava proces učenja. Nguyen-Widrowa inicijalizacija zahtijeva ulaz u intervalu od -1 do 1, tako da je u našem slučaju potrebno skalirati ulaze na taj interval. Skaliranje se provodi tako da u primjerima za učenje prvo potražimo minimalne i maksimalne vrijednosti za sve ulaze i potom pomoću tih vrijednosti linearno skaliramo i pomičemo ulaze na zahtijevani interval. Iz tog razloga se preporuča korištenje primjera za učenje koji pokrivaju i rubne dijelove ulaznog prostora. Nakon što smo naučili neuronsku mrežu na temelju ulaznih primjera, ulazi u tu neuronsku mrežu se uvijek moraju skalirati na isti način pomoću minimalnih i maksimalnih vrijednosti izračunatih na temelju primjera za učenje iako postoji mogućnost da se pojavi vrijednost koja izlazi iz tog intervala.

4 UČENJE NEURONSKE MREŽE PARALELNOM OPTIMIZACIJOM ROJEM ČESTICA

Kao što smo već ranije spomenuli postoji puno poznatih algoritama za učenje neuronskih mreža, među kojima namjerno nismo spomenuli i optimizaciju rojem čestica. Radi se o tome da, iako zadnjih par godina postaje sve popularnija, i dalje ne postoji dovoljan broj kvalitetnih radova koji se bave ovom tematikom. Glavni problem je možda u tome što matematička analiza ovakvog tipa učenja nije toliko jednostavna zbog stohastičke naravi algoritma.

4.1 Osnovna ideja

Ranije smo već govorili o tome kako optimizacija rojem čestica ima mogućnost pronalaska minimuma različitih funkcija. Ako uzmemo u obzir princip učenja neuronskih mreža koje smo naveli u prethodnom poglavlju, možemo svesti problem učenja neuronske mreže na minimiziranje funkcije srednje kvadratne pogreške koja ovisi jedino o težinama i pragovima unutar neuronske mreže za dane uzorke iz skupa za učenje, a optimizacija rojem čestica nam baš u tome može pomoći.

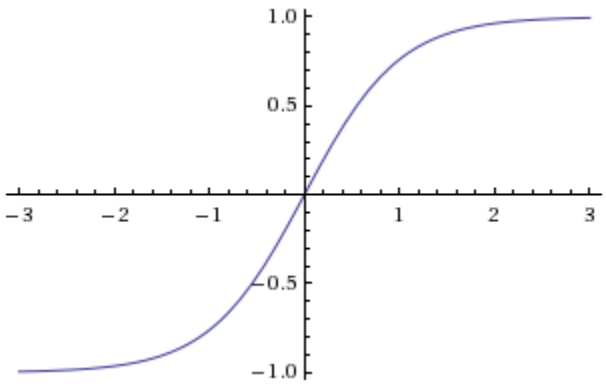
Prvo bi bilo potrebno izračunati ukupan broj težina i pragova unutar neuronske mreže jer nam taj broj ujedno označava i broj dimenzija prostora kojeg pretražujemo postupkom optimizacije rojem čestica. Svaka dimenzija tog prostora pripada jednoj težini ili pragu unutar neuronske mreže. Tada možemo kao funkciju dobrote u optimizaciji rojem čestica koristiti kvalitativnu ocjenu neuronske mreže dane funkcijom srednje kvadratne pogreške nad primjerima za učenje.

4.2 Detalji ostvarenja

Konačna implementacija je mogla otići u puno različitih smjerova jer i neuronske mreže i optimizacija rojem čestica imaju puno različitih oblika te je broj mogućih kombinacija ogroman, pa ovo poglavlje služi za pregled svih oblika koji su korišteni u konačnoj implementaciji.

Za optimizaciju rojem čestica korišten je osnovni oblik (2.2) u svojoj paralelno sinkronoj (4.2.2) i paralelno asinkronoj (4.2.3) inačici.

Neuronsku mrežu smo ponajviše htjeli iskoristiti u području klasifikacije, tako da smo shodno tome i odabrali odgovarajući oblik mreže. Korišten je oblik višeslojnog perceptrona koji ima kao izlaznu funkciju tangens hiperbolni čiji graf možemo vidjeti na slici 4.1.



Slika 4.1.

Kako bismo osigurali dobre početne uvjete i povećali vjerojatnost pravilne konvergencije u implementaciji koristimo prethodno navedenu Nguyen-Widrow metodu inicijalizacije težina i shodno tome ulaze linearno skaliramo na interval od -1 do 1.

Pošto se orijentiramo na područje klasifikacija dodatno izlaze iz neuronske mreže se također skaliraju sa intervala od -1 do 1 (izlaz funkcije tangens hiperbolni) na interval od 0 do 1.

4.3 Programsko rješenje

Za potrebe implementacije korišteno je razvojno okruženje Microsoft Visual Studio 2005. Implementacija je pisana u programskom jeziku C++ i korišten je standard MPI (*Message Passing Interface*) za potrebe paralelnog izvođenja programa. Standard MPI standard definira komunikaciju porukama koje se šalju među procesima. Konkretna implementacija korištena u razvoju paralelne aplikacije je MPICH2 verzije 1.2.1.

Konačna implementacija se sastoji od dva glavna dijela. Prvi dio se odnosi na implementaciju neuronske mreže i u cijelosti je sadržan u razredu *ANN*. U drugom djelu je implementirana paralelna optimizacija rojem čestica i sadržana je u klasi razredu *PSO*.

4.3.1 Umjetna neuronska mreža (razred *ANN*)

Ova klasa sadrži sve parametre i funkcije bitne za korištenje i učenje neuronske mreže kao što su broj slojeva (*_numOfLayers*), brojevi neurona po slojevima(*_neuronsInLayer*), vrijednosti težina (*_weights*) , primjere za učenje (*_learnInputs* i *_learnOutputs*), primjere za validaciju (*_validationInputs* i *_validationOutputs*), minimalne i maksimalne vrijednosti ulaza primjera za učenje (*_minInputs* i *_maxOutputs*).

Inicijalizacija se vrši pokretanjem konstruktora koji kao parametre prima broj slojeva i brojeve neurona po slojevima ili konstruktora koji prima ime datoteke iz koje je potrebno učitati već stvorenu neuronsku mrežu. U nastavku ćemo opisati bitne funkcije koje se koriste u klasi.

Save

Sprema neuronsku mrežu u tekstualnu datoteku u kojoj su navedeni broj slojeva, brojevi neurona po slojevima, minimalne i maksimalne vrijednosti primjera za učenje potrebni za skaliranje i vrijednosti svih težina i pragova.

Load

Prima naziv datoteke iz koje učitava neuronsku mrežu spremljenu pomoću Save funkcije.

LoadSamples

Prima naziv datoteke iz koje učitava primjere za učenje, bili oni za ispitivanje, validaciju ili učenje. Zbog toga je potrebno unaprijed podijeliti primjere u različite datoteke

u ovisnosti kako ih želimo koristiti. Primjeri moraju biti zapisani u datoteku na takav način da se u svakom retku nalazi jedan primjer te na kraju imamo onoliko redaka koliko imamo primjera. Unutar jednog primjera ili retka prvo navodimo ulazne vrijednosti, a tek onda izlazne vrijednosti. Potrebno je imati onoliko ulaza i izlaza koliko je, ranije definirano, u prvom i zadnjem sloju neuronske mreže. Vrijednosti ulaza i izlaza unutar jednog primjera se međusobno odvajaju praznim mjestom.

FindMinMax

Prima niz primjera sa odgovarajućim ulazima i vraća niz sastavljen od minimalnih i maksimalnih vrijednosti ulaza.

MapMinMax

Prima niz ulaza, niz maksimalnih i minimalnih vrijednosti te vraća linearno skalirani niz ulaza pomoću danog intervala.

Propagation

Prima niz koji predstavlja skalirane ulazne vrijednosti i vrši propagaciju kroz neuronsku mrežu. Kao izlaz nam vraća izlazne vrijednosti neuronske mreže skalirane na interval od 0 do 1.

MSE

Prima određeni broj primjera za učenje. Ulaze tih primjera propagira kroz neuronsku mrežu i dobivene izlaze uspoređuje sa izlazima iz primjera za učenje i računa srednju kvadratnu pogrešku.

FitnessFunction

Vraća srednju kvadratnu pogrešku izračunatu nad primjerima za učenje.

ValidateNetwork

Vraća srednju kvadratnu pogrešku izračunatu nad primjerima za validaciju.

4.3.2 Paralelna optimizacija rojem čestica (razred *PSO*)

Klasa PSO čuva sve parametre bitne za rad optimizacije rojem čestica i ima ih znatno više nego kod neuronske mreže, nazivi se ne razlikuju puno od naziva korištenih u poglavlju 2.2 ali ćemo svejedno spomenuti najbitnije, a to su redom broj čestica (*_numOfParticles*), broj dimenzija (*_dims*), minimalna vrijednost čestice (_minValue), maksimalna vrijednost čestice (*_maxValue*), maksimalna apsolutna brzina čestice (*_maxVelocity*), vrijednosti čestica (*_particleValues[][]*), brzine čestica *(_particleVelocity[][]*), pojedinačne najbolje dosadašnje pozicije (*_pBest[][])* , redni broj dosadašnje globalno najbolje čestice (*_gBestParticle*), vrijednosti dobrota osobno najboljih

pozicija (*_pBestFitness[]*), globalno najbolju poziciju čestice koja je prošla validaciju (_gBestValidationPassed[]).

U implementaciji optimizacije rojem čestica koristimo tri kriterija zaustavljanja od kojih bilo koji može u svakom trenutku zaustaviti optimizaciju. Prvi kriterij je maksimalni broj iteracija koji se smije izvršiti. Drugi kriterij je prag dobrote tj. u našem slučaju koliko malu vrijednost srednje kvadratne pogreške želimo. Treći kriterij provjerava kvalitetu generalizacije računanjem srednje kvadratne pogreške nad skupom za validaciju te ako određeni broj puta za redom generalizacija ne pokaže poboljšanje, a vrijednost globalno najbolje čestice se mijenjala prekidamo izvođenje algoritma i vraćamo vrijednosti globalne najbolje čestice koja je nastupila prije prvog opadanja kvalitete generalizacije. Ujedno ovaj kriterij bi se trebao izvoditi u asinkronoj implementaciji nakon svake promjene čestice što bi značilo da će se i po nekoliko puta više izvoditi nego u sinkronom obliku. Validacija je sama po sebi dosta skupa funkcija i ovisi o broju primjera nad kojima ju provodimo. Iz tog razloga je odlučeno validaciju obavljati nakon onoliko korekcija brzina i vrijednosti čestica koliko ima čestica u populaciji. Na taj način ubrzavamo optimizaciju i omogućavamo bližu usporedbu sa sinkronim oblikom koji obavlja validaciju nakon svake iteracije.

Komunikacija između različitih procesa se odvija jednako i u sinkronoj i u asinkronoj paralelnoj implementaciji. Imamo definirane samo dvije vrste poruka koje koriste osnovne funkcije definirane MPI standardom, a to su MPI_Recv i MPI_Send. To su blokirajuće nesinkronizirajuće verzije funkcija za primanje i slanje poruka. To znači da se izvođenje naredbi završava tek kada je memorijski prostor koji je korišten u primanju ili slanju siguran za korištenje, što kod primanja znači da smo dobili poruku, a kod slanja da smo ju poslali, ali ne možemo tvrditi da je još stigla.

Prvi oblik poruke se koristi kada proces voditelj želi saznati vrijednost funkcije dobrote za određenu česticu. Poruka je sastavljena od dva dijela i za „pakiranje“ takve poruke koristimo funkciju MPI_Pack. U prvom djelu se nalazi obična cjelobrojna vrijednost, kojom govorimo radnicima da li postoji još zadataka za njih, a u drugom dijelu se nalaze vrijednosti čestice za koju nam je potrebna funkcija dobrote. Znači, ako radnik primi poruku u kojoj se u prvom djelu nalazi 0, radnik prekida svoju petlju i završava sa izvođenjem, a ako je primio 1 čita drugi dio poruke i počinje računati funkciju dobrote za primljenu česticu. Po završetku radnik koristi drugi oblik poruke kako bi poslao rezultat voditelju. To je jednostavniji oblik poruke u kojem nije potrebno „pakirati“ poruku jer se

šalje samo jedna vrijednost s pomičnim zarezom koja predstavlja vrijednost funkcije dobrote za primljenu česticu.

Princip rada možemo malo bolje shvatiti ako pogledamo sljedeći pseudokod koji je podijeljen na tri dijela:

Pseudokod petlje radnika (jednak za sinkroni i asinkroni oblik):

1. Pošalji poruku sa bilo kojom vrijednošću voditelju
2. Čekaj poruku sa zadatkom (vrijednosti jedne čestice) od voditelja
3. Po primitku provjeri da li ima još zadataka, ako nema završi izvođenje, inače nastavi
4. izračunaj funkciju dobrote za dane vrijednosti čestice
5. Pošalji rezultat voditelju i vrati se na 2.

Pseudokod voditelja u paralelno asinkronoj implementaciji:

1. Popuni listu sa svim zadacima (jedan zadatak predstavlja jednu česticu za koju je potrebno izračunati funkciju dobrote)
2. Čekaj poruku od bilo kojeg radnika
3. Po primitku poruke provjeri koji zadatak smo poslali tom radniku. Ako mu nismo poslali nikakav zadatak zanemari rezultat, inače stavi taj zadatak na kraj liste zadatka.
4. Ako je kriterij zaustavljanja ispunjen skoči na 6. Inače napravi korekciju brzine i vrijednosti čestice za koju smo dobili rezultat funkcije dobrote
5. Skini sljedeći zadatak sa liste, pošalji ga radniku koji nam se javio u koraku 3. i zabilježi koji smo mu zadatak odnosno koju smo mu česticu poslali
6. Pričekaj onoliko poruka radnika koliko radnika još uvijek nije vratilo rezultat
7. Pošalji svim radnicima poruku koja označava da više nema zadataka i završi izvođenje.

Pseudokod voditelja u paralelno sinkronoj implementaciji:

1. Inicijaliziraj praznu listu slobodnih radnika i postavi brojač poslanih i primljenih zadataka na 0
2. Čekaj poruku bilo kojeg radnika
3. Po primitku poruke provjeri koji zadatak smo poslali tom radniku. Ako mu nismo poslali nikakav zadatak zanemari rezultat, inače spremi rezultat, povećaj brojač primljenih poruka i stavi radnika u listu slobodnih radnika

4. Ako je brojač primljenih poruka jednak broju čestica (završena je jedna iteracija) idi na korak 5. Inače skoči na 6.
5. Ako je ispunjen kriterij zaustavljanja skoči na 8., inače napravi korekciju brzine i vrijednosti čestica
6. Sve dok ima slobodnih radnika i broj poslanih zadataka je manji od broja čestica ukloni radnika sa liste slobodnih radnika, pošalji mu sljedeći zadatak i povećaj brojač poslanih zadataka
7. Skoči na 2
8. Svim radnicima pošalji poruku koja označava da više nema zadataka i završi izvođenje

Možemo primijetiti u pseudokodu asinkrone implementacije da voditelj na kraju izvođenja čeka rezultate zadataka čiji rezultati mu nisu potrebni. To se nikako nije moglo izbjeći jer u trenutku kada smo slali podatke nikako nismo mogli znati da će jedan od sljedećih rezultata zadovoljiti kriterij zaustavljanja.

U oba slučaja postoji mogućnost da voditelj ne pošalje poruku radniku koja označava kraj, ali samo ako nam se nekim slučajem taj radnik nikada nije ni javio. Ako ipak želimo biti sigurni i osigurati u iznimnim slučajevima pravilan završetak aplikacije možemo prije samog završetka pobrojati radnike koji nam se nisu niti jednom javili, pričekati njihove poruke i potom im poslati poruke koje označavaju kraj što je u ovoj implementaciji i učinjeno.

U nastavku slijedi kratak opis funkcija korištenih u ovoj klasi:

InitializeParticles

Nasumično razmješta čestice unutar prostora kojeg pretražujemo, u ovom konkretnom slučaju se pritom koristi Nguyen-Widrow metoda inicijalizacija težina i pragova kako bi se u samom startu omogućili bolji uvjeti za konvergenciju čitavog postupka.

RecieveTasks

Ovdje se izvodi petlja radnika koji čeka novi zadatak tj. vrijednosti neke čestice, računa funkciju dobrote i vraća je voditelju čime istodobno označava da je spremna za novi zadatak.

AdjustParticle

Kao ulazni parametar prima redni broj čestice i izvršava korekciju brzina i vrijednosti za tu česticu.

SendTask

Prima redni broj čestice, oznaku radnika i vrijednost kojom označavamo da li postoji još zadataka. Potom pakira takvu poruku i šalje je tom radniku

RecieveTask

Čeka poruku voditelja sa zadatkom i vraća vrijednosti čestice i oznaku da li postoji još zadataka.

SendFitness

Prima vrijednost funkcije dobrote i šalje poruku voditelju sa tim rezultatom.

RecieveFitness

Čeka poruku bilo kojeg radnika sa rezultatom zadatka kojeg je trebao izvršiti.

DelegateTasksSyn

Funkcija koja izvodi petlju voditelja i to za paralelni sinkroni oblik. Čeka poruke radnika, zadaje im nove zadatke te na kraju svake iteracije provjerava kriterij zaustavljana i vrši korekciju brzine i vrijednosti čestica.

DelegateTasksAsyn

Također izvršava petlju voditelja, ali u ovom slučaju za paralelni asinkroni oblik.

Optimize

Glavno čvorište programa koje kao ulaz prima sve potrebne parametre koji se koriste u optimizaciji rojem čestica. Tu se vrši i MPI inicijalizacija i postavlja procesor sa indeksom 0 kao voditelj. Za voditelja se pokreće odgovarajuća funkcija *DelegateTasksSyn* ili *DelegateTasksAsyn* ovisno o ulaznom parametru kojim odabiremo oblik paralelne optimizacije. U oba slučaja se za ostale procese tj. radnike pokreće funkcija *RecieveTasks*. Nakon što završi učenje koristi funkciju spremanja neuronske mreže i sprema rezultat u datoteku čiji smo naziv dobili preko ulaznog parametra.

4.3.3 Učitavanje konfiguracijskih parametara

Svi parametri kojima možemo utjecati na izvođenje programa nalaze se u konfiguracijskoj datoteci čiji je naziv potrebno predati kao argument komandne linije pri pokretanju programa. Parametri se zapisuju u datoteku u obliku „naziv argumenta“ = „vrijednost“, a poredak argumenata nije važan. U retcima se sve nakon znaka '#' smatra komentarima i zanemaruje pri učitavanju. Nazivi argumenata koje možemo podešavati su:

NumOfLayers – broj slojeva neuronske mreže

NeuronsInLayer - sastoji se od onoliko brojeva razdvojenih razmakom koliko postoji slojeva, a govori nam koliko neurona trebamo koristiti u svakom sloju

MaxAbsVelocity – maksimalna apsolutna brzina

MinValue* / *MaxValue – minimalna / maksimalna vrijednost čestice

Gincrement* / *Pincrement –utjecaj globalne / osobne najbolje vrijednosti na brzinu čestice

InertiaWight – težina inercije

LearningSamplesPath - puni naziv datoteke u kojem su sadržani primjeri za učenje
ValidationSamplesPath - puni naziv datoteke u kojima se nalaze primjeri za validaciju
ValidationChecks - maksimalni broj neuspjelih validacija korišten u kriteriju zaustavljana

FitnessThreshold - prag funkcije dobrote koji također koristimo u kriteriju zaustavljanja

MaxEvaluations – maksimalni broj evaluacija funkcije dobrote koji se koristi kao još jedan od kriterija za zaustavljanje

5 ISPITIVANJE I REZULTATI

U ovom poglavlju će se napraviti usporedba u brzini konvergencije sinkronog i asinkronog paralelnog modela. Dodatno će se napraviti usporedba skalabilnosti i brzine izvođenja tih dvaju oblika i za sami kraj se ostavlja ispitivanje uspješnosti klasifikacije naučene neuronske mreže za oba modela.

5.1 Podaci korišteni u ispitivanju

U ispitivanju su korišteni podaci za prepoznavanje pisanih znamenki u radu [13]. Ukupno je u ispitivanju sudjelovalo 43-oje ljudi od kojih je 30 ljudi kreiralo 3823 primjera za učenje, a preostalih 13 ljudi 1797 primjera za ispitivanje. Takva podjela osigurava da je skup primjera za ispitivanje u potpunosti neovisan o skupu za učenje i kao takvog ga možemo koristiti u ocjeni kvalitete naučene mreže. Rukom pisane znamenke pred-procesirane su programima dostupnim od strane NIST-a (*National Institute of Standards and Technology*) kako bi se uklonili uobičajeni problemi i smetnje nastale skeniranjem papira te stvorile normalizirane slike veličine 32x32 piksela. Slike su podijeljene na nepreklapajuće blokove od 4x4 piksela, što znači da imamo ukupno 8x8 takvih blokova. Prebrojani su označeni tj. uključeni pikseli unutar svakog bloka te ti brojevi predstavljaju ulazne vrijednosti primjera, a izlazni su označeni brojkom koju bi ta slika trebala predstavljati. Za potrebe ispitivanja bilo je potrebno kodirati izlaze nizom nula i jednom jedinicom na mjestu ovisno o znamenci koju primjer predstavlja, tako da odmah možemo i reći da će neuronska mreža korištena u učenju biti sastavljena od 64 ulaza i 10 izlaza.

5.2 Brzina konvergencije

Kod mjerenja brzine konvergencije je bilo potrebno postaviti novu mjeru s kojom ćemo moći ocijeniti samu brzinu konvergencije između dva različita modela. Budući da se inače u ocjeni konvergencije optimizacije rojem čestica koristi broj iteracija potreban za dostizanje određene vrijednosti funkcije dobrote, a asinkroni oblik nema klasične iteracije,

usporedbu će se raditi na osnovu potrebnog broja evaluacija funkcija dobrote. U ispitivanju se postavlja cilj srednje kvadratne pogreške u iznosu od 0.008.

Ispitivanja su izvršena za svaki oblik implementacije 30 puta i postavljeno je ograničenje u broju evaluacija funkcija na 100 000. Ako u tom razdoblju algoritam nije prešao prag pokušaj se smatra neuspjelim. U oba postupka je korišteno svih 3823 primjera za učenje, a korištena je neuronska mreža sa jednim skrivenim slojem u kojem se nalazi 20 neurona. Parametri optimizacije rojem čestica su također jednaki u obje implementacije i redom iznose:

- Broj čestica = 11
- Minimalna vrijednost čestice = -2; Maksimalna vrijednost čestice = 2
- Maksimalna apsolutna brzina = 0.1
- Težina inercije=0.7
- $g_{increment}$= 2; $p_{increment}$=1.8

Rezultate ispitivanja se mogu vidjeti u tablici 1., gdje je prikazan minimalan, maksimalan i prosječan broj evaluacija uspjelim pokušajima. Prosječan broj evaluacija u ovom slučaju ne daje dovoljno dobru informaciju, ako pogledamo i zadnji redak u kojem se nalazi prosječan broj evaluacija za dvije trećine najboljih rezultata u kojima sinkrona implementacija treba 16% manje evaluacija kako bi došla do traženog rezultata.

Tablica 1. Broj potrebnih evaluacija

	Paralelna sinkrona implementacija	**Paralelna asinkrona implementacija**
Minimalni broj evaluacija	17629	20870
Maksimalni broj evaluacija	89 169	34196
Neuspjelih pokušaja	6	7
Prosječno evaluacija	**26 602,50**	**25 763,05**
Prosjek 2/3 najboljih evaluacija	**20 722,08**	**24 934,38**

Uz to se može primijetiti kako asinkrona implementacija ima jedan neuspjeli pokušaj više od paralelne sinkrone implementacije. Rezultati nam govore kako u učenju neuronske mreže sinkrona implementacija ipak pokazuje bolju konvergenciju u većini slučajeva u odnosu na asinkronu implementaciju.

5.3 Brzina izvođenja

Ispitivanja brzine i skalabilnosti su vršena na 12 računala sa Intel Pentium IV 2.66 GHz procesorima i 512 MB radne memorije koja su međusobno povezana lokalnom mrežom brzine 100Mbit/s.

Brzinu izvođenja se mjerila na način da se ograniči izvođenje algoritma na 5000 evaluacija funkcije dobrote bez obzira na njenu vrijednost. Na taj način omogućena je direktna usporedba sa ispitivanjem brzine konvergencije. Parametri optimizacije čestice u ovom slučaju ne igraju nikakvu ulogu niti ne utječu na rezultat, ali ćemo napomenuti kako su isti kao i u prethodnom ispitivanju.

Za svaku kombinaciju procesora i oblik implementacije mjeri se vrijeme izvođenja 5 puta. Prosječno vrijeme izvođenja kroz tih 5 pokušaja možemo vidjeti u tablici 2. Na dnu tablice možemo u postocima vidjeti koliko je kraće vrijeme izvođenja asinkrone implementacije u odnosu na sinkronu implementaciju optimizacije rojem čestica. Također treba napomenuti da stupac kojim se označava izvođenje na jednom računalu ustvari predstavlja vrijeme izvođenja slijednog algoritma.

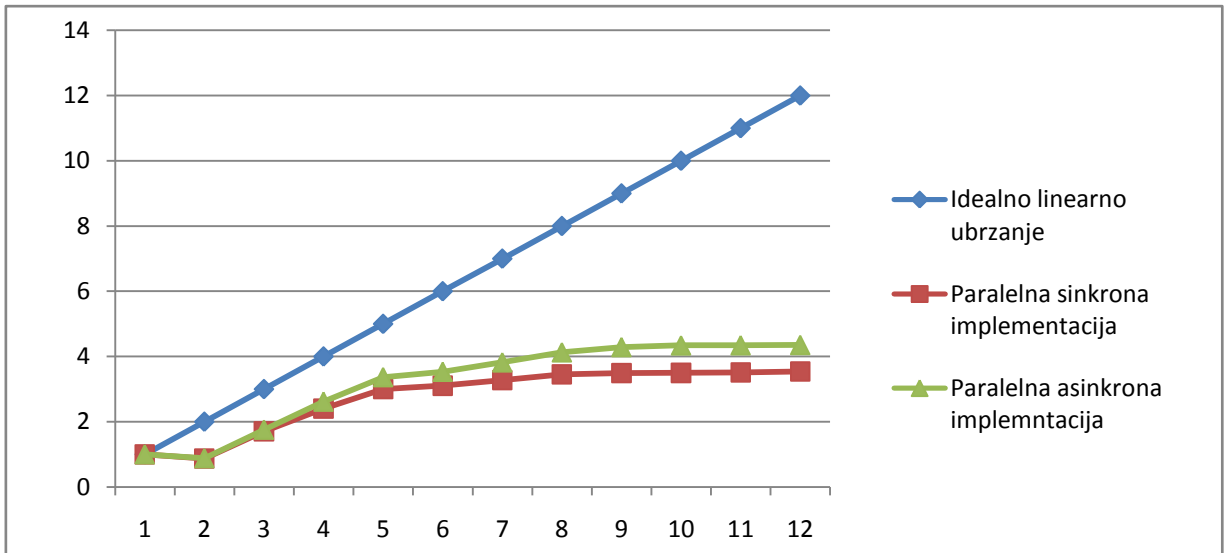
Tablica 2. Vrijeme izvođenja u ovisnosti o broju računala

Broj računala	1	2	3	4	5	6	7	8	9	10	11	12
Paralelna sinkrona implementacija	235,45	269,08	137,90	98,00	78,34	75,83	71,85	68,16	67,34	67,30	67,05	66,50
Paralelna asinkrona implemntacija	234,95	267,40	133,86	89,86	69,83	66,55	61,53	56,95	54,82	54,16	54,10	54,00
Ubrzanje u postotcima	0,01	0,62%	2,93%	8,31%	10,86%	12,24%	14,36%	16,45%	18,59%	19,52%	19,31%	18,80%

U obje implementacije može se primijetiti kako se dodavanjem procesora u početku znatno smanjuje vrijeme izvođenja kako bi kasnije to smanjenje počelo usporavati s obzirom na broj dodanih procesora. To se može bolje vidjeti u tablici 3. i na grafu (slika 5.1.) u kojima su prikazana relativna ubrzanja za oba paralelna algoritma.

Tablica 3. Relativno ubrzanje

Broj računala	1	2	3	4	5	6	7	8	9	10	11	12
Paralelna sinkrona implementacija	1,00	0,88	1,71	2,40	3,01	3,10	3,28	3,45	3,50	3,50	3,51	3,54
Paralelna asinkrona implemntacija	1,00	0,88	1,76	2,61	3,36	3,53	3,82	4,13	4,29	4,34	4,34	4,35

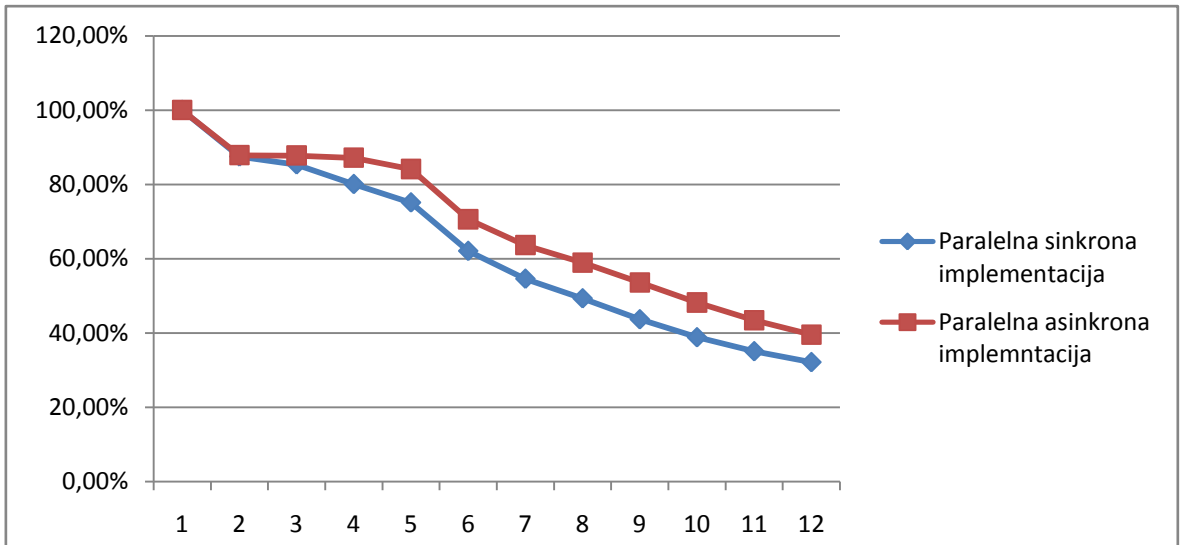


Slika 5.1. Grafički prikaz relativnog ubrzanja

Razlog tomu treba tražiti u preopterećenju voditelja porukama i zagušenju komunikacijskog kanala. Ispostavlja se da vrijeme potrebno za računanje srednje kvadratne pogreške za neuronsku mrežu sa preko 1500 težina i preko 3000 primjera i nije tako veliko. Štoviše toliko se brzo izvodi da dodavanjem 10-og, 11-og i 12-og procesora više se ne mogu vidjeti nikakva poboljšanja na vremenu izvođenja. Također treba napomenuti da na grafovima jedan procesor označava slijedni algoritam, pa shodno tome ubrzanje izvođenja za dva procesora (jedan voditelj i jedan radnik) u biti pokazuje usporenje u odnosu na slijedni algoritam zbog troškova komunikacije. Dodatnim promatranjem komunikacije između procesa u slučaju sa 12 računala može se vidjeti kako dva do tri radnika jedva uspiju dobiti pokoji zadatak tokom cijelog izvođenja. Sa povećanjem broja računala radnici sve manje vremena provode obavljajući koristan rad, a sve više vremena provode čekajući poruku sa novim zadatkom, što znači da ne koristimo njihov puni potencijal što se može primijetiti u tablici 4. i grafu (slika 5.2.) koji prikazuju relativnu učinkovitost za oba paralelna algoritma.

Tablica 4. Relativna učinkovitost

Broj računala	1	2	3	4	5	6	7	8	9	10	11	12
Paralelna sinkrona implementacija	100,00%	87,50%	85,37%	80,09%	75,14%	62,10%	54,62%	49,35%	43,71%	38,87%	35,12%	32,19%
Paralelna asinkrona implemntacija	100,00	87,86%	87,76%	87,15%	84,11%	70,61%	63,64%	58,94%	53,57%	48,20%	43,43%	39,55%



Slika 5.2. Grafički prikaz relativne učinkovitosti

U stvari to je problem na kojeg se često nailazi pri razvoju paralelnih programa i direktno je povezan sa zrnatošću algoritma. Zrnatost paralelnog algoritma opisuje omjer između količine računanja i količine komunikacije. U ovom slučaju je relativno sitna zrnatost tj. vrijeme potrebno za izračun zadatka nije puno veće od vremena potrebnog za slanje zadatka. Problem sitnozrnatosti mogao bi se riješiti jedino povećanjem obujma zadatka koji se zadaje radnicima. Primjerice, umjesto da se radnicima šalje samo jedna čestica, moglo bi se istovremeno poslati više čestica kako bi se smanjio broj poruka koji se šalje i tako rasteretio voditelj, ali time se automatski smanjuje broj mogućih zadataka, a posljedično se smanjuje i maksimalan broj računala koji bi se mogli koristiti.

U konačnici treba napomenuti, kako je ranije bilo i nagoviješteno, da paralelna asinkrona implementacija, iako i sama nailazi na ograničenja, pokazuje bolju skalabilnost u odnosu na sinkronu implementaciju.

5.4 Uspješnost klasifikacije

Za učenje neuronske mreže korišteni su isti parametri optimizacije rojem čestica kao i u prethodnim ispitivanjima. Učena je neuronska mreža sa 64 ulazna neurona, 20 skrivenih neurona (u jednom sloju) i 10 izlaznih neurona. Primjeri su klasificirani u određenu klasu u ovisnosti najvećeg izlaza iz neuronske mreže. Primjerice, u slučaju kada dva izlaza daju vrijednosti 0.8 i 0.7, a ostali izlazi daju kao rezultat 0, odabire se klasa na koju pokazuje izlaz s većom vrijednosti tj. u ovom slučaju 0.8. Učenje je vršeno sa

maksimalnim brojem evaluacija funkcija dobrote u iznosu od 200 000, ali je, naravno, korišten i kriterij zaustavljanja koji provjerava generalizaciju na skupu za validaciju. Skup za validaciju je kreiran nasumičnim odabirom 15% primjera iz skupa za učenje (to je nešto više od 500 primjera). Skup za ispitivanje nije diran i koristi se samo u konačnoj procjeni generalizacije.

Učenje mreže traje otprilike 20-ak minuta i staje negdje oko 110 000 evaluacije funkcije dobrote zbog zadovoljavanja kriterija zaustavljanja po provjeri generalizacije. Konačna srednja kvadratna pogreška na skupu za treniranje u ovom slučaju iznosi 0.002894, te ostaje samo provjera uspješnosti klasifikacije. Jedan od boljih načina za to je korištenje matrice konfuzije. Matrica konfuzije je sastavljena od redaka koji predstavljaju predviđene klase i stupaca koji predstavljaju stvarne klase primjera. Na taj način se mogu vrlo lako primijetiti postoje li neki specifični problemi kao što bi moglo biti konstantno klasificiranje 7-ice kao 1-ice. Matrice konfuzije za primjere za učenje, validaciju i ispitivanje su prikazani u tablicama 5., 6. i 7. U tablici 8. prikazana je matrica konfuzije za sve primjere zajedno.

Tablica 5. Matrica konfuzije za primjere za učenje

		Stvarna klasa										
		0	1	2	3	4	5	6	7	8	9	preciznost
Predviđena klasa	0	309	0	0	0	1	0	0	0	0	0	99,68%
	1	0	334	0	1	0	0	2	1	3	0	97,95%
	2	0	0	316	0	0	1	0	0	0	0	99,68%
	3	0	0	0	318	0	1	0	0	1	2	98,76%
	4	1	0	0	0	333	0	0	0	2	1	98,81%
	5	0	0	0	2	0	328	0	0	3	1	98,20%
	6	2	1	0	0	0	0	315	0	0	0	99,06%
	7	0	1	1	0	0	0	0	331	0	1	99,10%
	8	0	2	4	1	0	0	0	0	303	0	97,74%
	9	0	1	0	2	0	3	0	0	2	320	97,56%
odziv		99,04%	98,53%	98,44%	98,15%	99,70%	98,50%	99,37%	99,70%	96,50%	98,46%	98,65%

Tablica 6. Matrica konfuzije za primjere za validaciju

		Stvarna klasa										
		0	1	2	3	4	5	6	7	8	9	preciznost
Predviđena klasa	0	64	0	0	0	0	1	1	0	0	1	95,52%
	1	0	48	0	1	0	0	0	0	2	1	92,31%
	2	0	0	57	0	0	0	0	0	0	2	96,61%
	3	0	0	1	63	0	0	0	0	1	2	94,03%
	4	0	0	0	0	52	0	0	0	0	1	98,11%
	5	0	1	0	1	0	42	0	0	1	0	93,33%
	6	0	0	1	0	1	0	58	0	0	0	96,67%
	7	0	0	0	0	0	0	0	53	0	0	100,00%
	8	0	0	0	0	0	0	1	1	62	0	96,88%
	9	0	1	0	0	0	0	0	1	0	50	96,15%
odziv		100,00%	96,00%	96,61%	96,92%	98,11%	97,67%	96,67%	96,36%	93,94%	87,72%	95,98%

Tablica 7. Matrica konfuzije za primjere za ispitivanje

		Stvarna klasa										
		0	1	2	3	4	5	6	7	8	9	preciznost
Predviđena klasa	0	176	0	0	0	0	1	0	0	0	0	99,44%
	1	0	174	4	0	2	0	1	2	8	1	90,63%
	2	0	0	168	3	0	0	0	0	1	0	97,67%
	3	0	0	1	169	0	0	0	0	0	2	98,26%
	4	1	0	0	0	178	0	1	1	0	3	96,74%
	5	1	0	0	4	0	178	0	3	5	4	91,28%
	6	0	0	2	0	0	0	179	0	0	0	98,90%
	7	0	0	0	3	0	0	0	163	0	0	98,19%
	8	0	1	2	0	1	1	0	1	153	6	92,73%
	9	0	7	0	4	0	2	0	9	7	164	84,97%
odziv		98,88%	95,60%	94,92%	92,35%	98,34%	97,80%	98,90%	91,06%	87,93%	91,11%	94,71%

Tablica 8. Matrica konfuzije za sve primjere

		Stvarna klasa										
		0	1	2	3	4	5	6	7	8	9	preciznost
Predviđena klasa	0	549	0	0	0	1	2	1	0	0	1	99,10%
	1	0	556	4	2	2	0	3	3	13	2	95,04%
	2	0	0	541	3	0	1	0	0	1	2	98,72%
	3	0	0	2	550	0	1	0	0	2	6	98,04%
	4	2	0	0	0	563	0	1	1	2	5	98,08%
	5	1	1	0	7	0	548	0	3	9	5	95,47%
	6	2	1	3	0	1	0	552	0	0	0	98,75%
	7	0	1	1	3	0	0	0	547	0	1	98,92%
	8	0	3	6	1	1	1	1	2	518	6	96,10%
	9	0	9	0	6	0	5	0	10	9	534	93,19%
odziv		99,10%	97,37%	97,13%	96,15%	99,12%	98,21%	98,92%	96,64%	93,50%	95,02%	97,12%

U tablicama je izračunata preciznost i odziv neuronske mreže za svaku klasu. U samom desnom uglu tablice prikazana je točnost. To su mjere koje se uvode radi preciznijeg vrednovanja klasifikatora. Točnost predstavlja udio točno klasificiranih primjera među svim primjerima. Preciznost predstavlja udio točno klasificiranih primjera u skupu svih jednako klasificiranih primjera, a odziv pokazuje udio točno klasificiranih primjera unutar primjera svake klase. Nijedna od tih mjera nije dovoljna sama za sebe i uvelike ovisi o problemu koji se rješava. Primjerice za skup od 10000 primjera od kojih su samo 100 pozitivno klasificirani, a ostali negativno, neuronska mreža koja bi davala na izlazu samo negativnu vrijednosti pokazivala bi 99%-tnu točnost iako od takve mreže nemamo nikakve realne koristi.

Može se odmah primijetiti, što je i bilo za očekivati, da su najbolji rezultati postignuti na skupu za učenje. Rezultati nad skupom za validaciju su naravno nešto slabiji, a na skupu za ispitivanje su najlošiji gdje točnost iznosi 94,71%. U usporedbi sa nekim postojećim algoritmima koji su implementirani u Matlabu, pritom se poglavito misli na *scaled conjugate gradient* metodu, koja uči neuronsku mrežu istog oblika, pokazuje se kako rezultati dobiveni u ovom radi možda i nisu toliko dobri koliko se čini na prvi pogled Primjerice, za zadnje ispitivanje iz prethodnog poglavlja, koristeći Matlabovu implementaciju, izvođenje završava u roku od jedne do dvije minute i uz to, povremeno, dosežući sumu kvadratne pogreške reda 10^{-4} i boljim rezultatima na konačnom skupu za ispitivanje koji se kreću negdje u iznosu od 97% do 99% točno klasificiranih primjera. Razlog treba tražiti u tome da optimizacija rojem čestica ne koristi nikakvo posebno znanje o neuronskim mrežama kako bi pronašla minimum srednje kvadratne pogreške već se u istom obliku koristi i za rješavanje potpuno drukčijih problema što je samo po sebi pothvat.

6 ZAKLJUČAK

Cilj ovog rada je ostvariti paralelnu implementaciju optimizacije rojem čestica. U konačnici su implementirana dva paralelna oblika, a to su sinkroni i asinkroni oblik. Ispitivanja su pokazala kako sinkroni oblik pokazuje bolju i stabilniju konvergenciju pri treniranju neuronskih mreža iako se takva paralelna implementacija izvodi sporije nego asinkroni oblik. Kada se obje činjenice uzmu u obzir moglo bi se zaključiti da će im u konačnici biti potrebno jednako vrijeme kako bi dosegle sličan rezultat. Primjerice korištenjem sedam procesora u optimizaciji, veća brzina izvođenja asinkronog oblika nadoknađuje zaostatak za sinkronom zbog sporije konvergencije. Takvim razmišljanjem bismo se mogli prevariti, jer nestabilna konvergencija može uzrokovati puno problema i to pogotovo u kasnijim dijelovima algoritma kada je potrebno jako oprezno vršiti korekcije pravaca.

Možemo primijetiti kako se u postojećim radovima učenje neuronske mreže većinom izvodi nad jednostavnijim problemima za učenje i na taj način autori možda i svjesno stvaraju krivu predodžbu o uspješnosti optimizacije rojem čestica u treniranju neuronskih mreža.

Gledajući konačni oblik algoritma u odnosu na algoritam kojim smo se koristili na samom začetku, mora se primijetiti kako je to uistinu bio dugotrajan i mukotrpan proces u kojem se konstantno pokušavaju poboljšati rezultati optimizacije i naučene neuronske mreže, što ponekad dovodi i do degradacije performansi zbog osjetljivosti algoritma, ali na sreću broj takvih „poboljšanja“ predstavlja manjinu.

U konačnici je pokazano da je moguće učiti neuronsku mrežu optimizacijom rojem čestica i da takvo učenje daje pristojne rezultate, ali isto tako da se u tom obliku trenutno ne može nositi sa najboljim metodama korištenim na tom području

LITERATURA

[1] Optimization, http://en.wikipedia.org/wiki/Optimization_(mathematics), 10.6.2010

[2] Kennedy, J., Eberhart, R. Particle Swarm Optimization. From Proc. IEEE Int'l. Conf. on Neural Networks, Perth, 1995, Australia, IV:1942-1948

[3] Wilson, E.O., Sociobiology: The new synthesis. Cambridge, MA: Belknap Press, 1975

[4] Millonas, M. M., Swarms, phase transitions, and collective intelligence, in : Langton C.(Ede), Artificial Life III,1994

[5] Byung-IlKoh, Alan D.George, Raphael T., Haftka and Benjamin J. Fregly, Parallel asynchronous particle swarm optimization, 2006

[6] Schutte, J.F.,Particle swarms in sizing and global optimization. Master's thesis, University of Pretoria, Department of Mechanical Engineering, 2001

[7] Carlisleand, A., Dozier, G. An off-the-shelf pso. In Proceedings of the Workshop on Particle Swarm Optimization, Indianapolis, 2001

[8] Alexander, I., Morton, H., An Introduction to Neural Computing. London: Chapman and Hall, 1990

[9] Bishop, C.M. Neural Networks for Pattern Recognition, Oxford: Oxford University Press, 1995

[10] Lončarić S. Predavanje iz predmeta Neuronske mreže: Proces učenja. FER, Zagreb

[11] Derrick Nguyen and Bernard Widrow, Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights, Proceedings of the International Joint Conference on Neural Networks,1990, 3:21–26

[12] A.Pavelka and A.Procházka, Algorithms for initialization of neural network weights, 2004

[13] C. Kaynak, Methods of Combining Multiple Classifiers and Their Applications to Handwritten Digit Recognition, MSc Thesis, Institute of Graduate Studies in Science and Engineering, Bogazici University, 1995

SAŽETAK / ABSTRACT

Učenje neuronske mreže paralelnom optimizacijom rojem čestica

U radu su proučeni model neuronskih mreža i modeli paralelne optimizacije rojem čestica. Ostvareno je programsko rješenje paralelnog i asinkronog oblika optimizacije rojem čestica sa više kriterija zaustavljanja i pokazano je na koji način paralelni procesi komuniciraju međusobno u ostvarenju paralelne implementacije. Ispitivanje je izvršeno učenjem neuronske mreže klasifikaciji rukom pisanih znamenki. Pokazano je veće ubrzanje i učinkovitost asinkrone implementacije u odnosu na sinkronu implementaciju. Također je pokazano da asinkroni oblik sporije konvergira od asinkronog. Na samom kraju je prikazana i uspješnost klasifikacije neuronske mreže dobivene ovim postupkom i napravljena usporedba sa *scaled conjugate gradient metodom*.

Learning Neural Network with Particle Swarm Optimization

In this work basic principles of neural networks and parallel particle swarm optimizations were studied. Algorithm was implemented for both the synchronous and asynchronous model of particle swarm optimization and was shown the way in which parallel processes communicate with each other. Testing was done by learning artificial neural network in classification of handwritten digits. It was shown that parallel speedup and efficiency of asynchronous implementation is better than the synchronous implementation. It was also shown that synchronous implementation has better convergence rate. In the end we showed classification results for a neural network trained in this manner and comparison was made with scaled conjugate gradient method.

KLJUČNE RIJEČI

- Optimizacija rojem čestica
- Učenje neuronske mreže
- Paralelna sinkrona optimizacija rojem čestica
- Paralelna asinkrona optimizacija rojem čestica

KEYWORDS

- Particle swarm optimization
- Learning neural network
- Parallel synchronous particle swarm optimization
- Parallel asynchronous particle swarm optimization