

UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Laura Guzelj Blatnik

**Nevronske mreže z vzratnim razširjanjem napak v
funkcijskem programskem jeziku**

Delo diplomskega seminarja

Mentor: prof. dr. Ljupčo Todorovski
Somentor: asist. dr. Aljaž Osojnik

Ljubljana, 2020

KAZALO

1. Uvod	4
2. Umetne nevronske mreže	4
2.1. Biološko ozadje	4
2.2. Zgradba nevronskih mrež	4
2.3. Delovanje perceptrona	5
2.4. Lastnosti nevronskih mrež	8
2.5. Uporaba nevronskih mrež	9
3. Učenje nevronskih mrež	9
3.1. Nadzorovano učenje	10
3.2. Pravilo delta	10
3.3. Vzvratno razširjanje napake	11
3.4. Lastnosti vzratnega razširjanja napake	16
3.5. Pristranskost in varianca	16
3.6. Normalizacija podatkov	18
4. Funkcijski programski jezik OCaml	19
4.1. Lastnosti funkcijskih programskih jezikov	19
4.2. Lastnosti OCaml	19
5. Implementacija nevronske mreže	20
5.1. Primer nevronske mreže	20
5.2. Evalvacija na naboru podatkov	20
6. Priloga	21
Slovar strokovnih izrazov	33
Literatura	33

**Nevronske mreže z vzratnim razširjanjem napak v funkcijskem
programskem jeziku**

POVZETEK

Angleški prevod slovenskega naslova dela

ABSTRACT

Math. Subj. Class. (2010):

Ključne besede:

Keywords:

1. UVOD

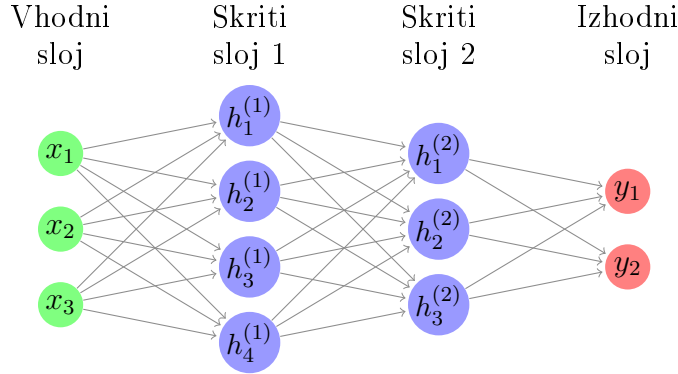
Človeški možgani so kompleksen organ predvsem zaradi nešteti funkcij, ki jih opravljajo. In zgolj vprašanje časa je bilo, kdaj bodo znanstveniki skoraj neskončne zmožnosti možganov prenesli v računalništvo. Ko so poskušali idejo uresničiti, so si predvsem želeli strukture, ki se bo – podobno kot možgani – sposobna učiti, odzivati na spremembe in prepoznavati neznane situacije. Tako je 1949 Donald Hebb v svojem delu prvič predstavil idejo o umetnih nevronske mrežah, kjer se je zgledoval predvsem po možganih. Dvoslojni perceptorn se je rodil nekoliko kasneje, leta 1962 je ta pojem prvi omenil Rosenblatt. Z razvojem strojne opreme se je zanimanje za nevronske mreže zopet povečalo v osemdesetih letih prejšnjega stoletja. Leta 1986 je več raziskovalcev neodvisno drug od drugega vpeljalo teorijo o večslojnih nevronske mrežah in vzvratnem razširjanju napake. Kljub širokemu spektru uporabe, ki jih nudijo nevronske mreže, je interes zanje kasneje nekoliko upadel. V zadnjih letih pa so nevronske mreže zopet vroča tema, predvsem zaradi njihove zmožnosti prepoznavanja slik. Do danes so umetne nevronske mreže močno napredovale, namesto običajnih mrež se vedno bolj uporabljajo globoke nevronske mreže, ki v nekaterih lastnostih celo prekašajo možgane. Kljub vsemu pa so možgani sposobni masrčičesa, česar računalniki ne bodo nikoli.

Diplomska naloga se poglobi v umetne nevronske mreže z vzvratnim razširjanjem napake. Primer take mreže sem tudi implementirala v funkcijskem programskem jeziku OCaml.

2. UMETNE NEVRONSKE MREŽE

2.1. Biološko ozadje. Kot namiguje že samo ime, se umetne nevronske mreže v marsičem zgledujejo po človeških možganih. Osnovni gradniki možganov so nevroni, ki so med seboj povezani s sinapsami. Prav te goste povezave so ključne za delovanje živčevja. Skozi naše življenje se sinapse spreminjajo po jakosti in številu, ko se učimo se ustvarjajo nove povezave med nevroni, že obstoječe pa postajajo močnejše. Nevron se aktivira le, ko po sinapsi do njega pride signal s točno določeno frekvenco, ta impulz nato nevron posreduje sosednjim nevronom. Človeške možgane sestavlja 10^{10} nevronov, vsak ima približno 10^4 sinaps. Umetne nevronske mreže načeloma sestavlja veliko manj gradnikov, posledično so pri izvajanju operacij veliko hitrejša od možganov. Kot bomo videli v nadaljevanju, je osnovna ideja umetnih nevronske mrež zelo podobna živčevju v možganih. Lahko bi rekli, da so umetne nevronske mreže zgolj abstraktna poenosatvitev delovanja možganov.

2.2. Zgradba nevronske mreže. Nevronske mreže so sestavljene iz nevronov, ki matematično gledano niso nič drugega kot funkcije. Nevroni so med seboj povezani, vsaka povezava ima svojo težo oziroma utež. Nevrone nadalje razdelimo v sloje. Pomembno je poudariti, da nevroni v istem sloju med seboj niso povezani, vsak nevron je povezan z vsemi nevroni v naslednjem (oziroma prejšnjem sloju).



SLIKA 1. Perceptron z dvema skritima slojema nevronov

Nevronska mreža lahko vsebuje le vhodni in izhodni sloj – takrat govorimo o dvoslojni nevronske mreži. Če se med tema slojema skriva še kakšen skriti sloj, potem mreža postane večslojna. Število nevronov v posameznem sloju kot tudi število skritih slojev nevronske mreže je odvisno od problema, ki ga nevronska mreža rešuje. Pri mrežah, ki jih bom obravnavala v diplomski nalogi, so vse povezave usmerjene, čeprav je pomembno omeniti, da obstajajo tudi mreže z dvosmernimi povezavami. Usmerjeni nevronske mreže rečemo perceptron, glede na število slojev ločimo dvoslojni in večslojni perceptron.

2.3. Delovanje perceptrona. Vzemimo dvoslojni perceptron z m nevroni v vhodnem sloju, njihova stanja označimo z X_1, X_2, \dots, X_m in n nevroni v izhodnem sloju, označimo jih Y_1, Y_2, \dots, Y_n . Običajno sloj nevronov zapišemo v obliki vektorja:

$$X = (X_1, X_2, \dots, X_m)^T \quad \text{in} \quad Y = (Y_1, Y_2, \dots, Y_n)^T.$$

Uteži med nevroni predstavimo z matriko:

$$W = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1m} \\ W_{21} & W_{22} & \cdots & W_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ W_{n1} & W_{n2} & \cdots & W_{nm} \end{bmatrix}$$

kjer element matrike W_{ij} predstavlja utež med j -tim nevronom v vhodnem sloju in i -tim nevronom v izhodnem sloju.

2.3.1. Funkcija kombinacije: Stanje i -tega nevrona po uporabljeni funkciji aktivacije je sledeče:

$$A_i = \sum_j W_{ij} X_j + C_i,$$

kjer je C_i pa konstanta aktivacije za i -ti nevron (njen pomen je pojasnjen v poglavju 2.3.3). Za cel sloj izhodnih nevronov pa lahko funkcijo aktivacije zapišemo

z linearno preslikavo:

$$A = WX,$$

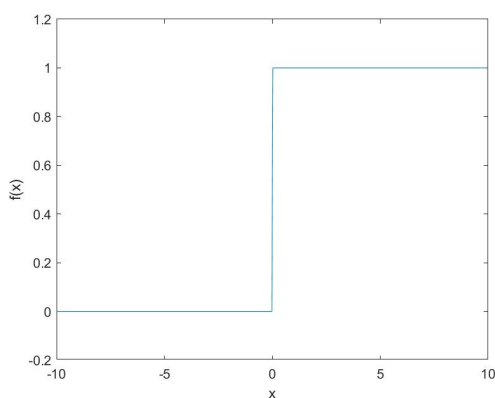
kjer je W matrika uteži, X pa vhodni vektor. Če upoštevamo še konstante aktivacije, je potrebno matriko uteži W in vektor X malo preoblikovati. Vektorju na konec dodamo še eno komponento z vrednostjo 1 in dobimo $X = (X_1, X_2, \dots, X_m, 1)^\top$ matriki uteži pa dodamo še stolpec s konstantami aktivacije za vsak nevron v izhodnem sloju, torej:

$$W = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1m} & C_1 \\ W_{21} & W_{22} & \cdots & W_{2m} & C_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ W_{n1} & W_{n2} & \cdots & W_{nm} & C_n \end{bmatrix}$$

2.3.2. Funkcija aktivacije: Cilj funkcije aktivacije je normalizirati vrednost, ki jo dobimo s funkcijo kombinacije. Glede na namen uporabe nevronske meže lahko uporabimo različne funkcije aktivacije. Najpogostejše se uporabljata stopničasta pragovna funkcija ali pa sigmoidna funkcija.

Stopničasta pragovna funkcija nevron aktivira le, če je vrednost dobljena z funkcijo aktivacije nad oziroma pod določeno mejo. Uporaba te funkcije je dokaj omejena. Ker vhodnim podatkom pripiše le dve različni vrednosti, jo lahko uporabimo le pri klasifikaciji v dve kategoriji oziroma za reševanje linearnih problemov. Prav tako jo lahko uporabimo samo pri dvoslojnih perceptronih. Kot bomo videli v nadaljevanju, za večslojne perceptrone ni primerna, saj njen odvod ni gladek, kjer je definiran, pa je enak 0, kar pa pomeni, da za učenje ne moremo uporabiti posplošenega pravila delta. Primer stopničaste pragovne funkcije z izhodnima vrednostima 0 in 1 je:

$$f(x) = \begin{cases} 0, & \text{če } x \leq 0 \\ 1, & \text{sicer.} \end{cases}$$



SLIKA 2. Stopničasta pragovna funkcija aktivacije

Sigmoidne funkcije so nelinearne, zato se uporabljajo pri večslojnih perceptornih in reševanju nelinearnih problemov. So zvezne in neskončnokrat zvezno odvedljive, tako jih lahko uporabimo pri posplošenem pravilu delta. Odvisno na kateri interval želimo normalizirati izhodne vrednosti, lahko uporabimo različne sigmoidne funkcije. Primera sigmoidne funkcije sta:

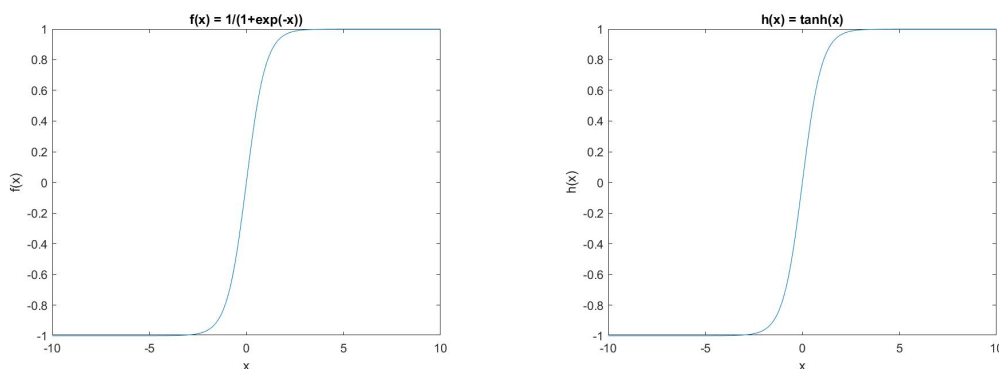
$$(1) \quad f(x) = \frac{1}{1 + e^{-x}} \quad \text{in} \quad h(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Kot je razvidno z grafov (slika 3) je na intervalu $(-2, 2)$ naklon obeh funkcij velik, zato da na omenjenem intervalu mreža pri napovedovanju jasne rezultate in učenje je hitro. Po drugi strani pa je zunaj tega intervala odvod majhen, tako je učenje mreže in posledično napovedovanje zelo počasno in nenatančno.

Kot bomo videli v nadaljevanju, pri učenju mreže s posplošenim pravilom delta potrebujemo tudi odvod funkcije aktivacije, ki ga pri zgornjih dveh funkcijah lahko izrazimo z $f(x)$ oziroma $h(x)$ na naslednji način:

$$(2) \quad f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x)(1 - f(x))$$

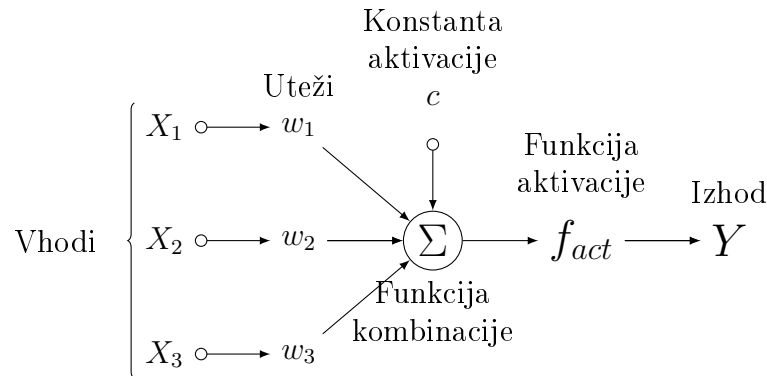
$$(3) \quad h'(x) = \left(\frac{\sinh(x)}{\cosh(x)} \right)' = \frac{\cosh^2(x) - \sinh^2(x)}{\cosh^2(x)} = 1 - \tanh^2(x) = 1 - h(x)^2$$



SLIKA 3. Sigmoidni funkciji aktivacije

Katero funkcijo aktivacije bomo izbarli je odvisno od problema, ki ga želimo z mrežo obravnavati.

2.3.3. Konstanta aktivacije. Sloju nevronov pogosto dodamo še en nevron - konstanto aktivacije. To je nevron z vrednostjo 1, njegova naloga pa je, da funkcijo aktivacije primerno premakne levo oziroma desno po grafu. Brez dodatnega nevrona bi mreža pri vhodnem podatku $X = 0$ vedno vračala vrednost 0, posledično bi bila aproksimacija dane funkcije slaba in rezultati netočni.



SLIKA 4. Funkcija nevrona

Pomen konstante aktivacije najlažje predstavimo s pomočjo linearne funkcije. Z enačbo oblike $y = kx$ lahko zapišemo le premice, ki prečkajo koordinatno izhodišče. Če pa funkciji dodamo še začetno vrednost, ki ustreza pomenu konstante aktivacije, potem lahko z enačbo $y = kx + n$ zapišemo poljubno premico. Z uporabo konstante aktivacije postane mreža bolj fleksibilna, problem bolje aproksimira in vrača bolj točne rezultate.

2.3.4. Funkcija nevrona. Izhodni sloj nevronov Y dobimo tako, da na vhodnem nevronu najprej uporabimo funkcijo kombinacije in nato še funkcijo aktivacije – obema funkcijama skupaj pravimo funkcija nevrona, tj. izračun izhodnih vrednosti nevrona. Za posamezen nevron velja:

$$Y_i = f\left(\sum_j W_{ij}X_j + C_i\right)$$

oziroma v vektorski obliki za cel sloj izhodnih nevronov:

$$Y = f(WX),$$

kjer velja, da funkcijo aktivacije uporabimo na vsaki komponenti vektorja WX .

2.4. Lastnosti nevronske mreže. Nevronske mreže odlikuje kar nekaj pozitivnih lastnosti. Najpomembnejše med njimi so sledeče:

Paralelizabilnost: Posamičen nevron deluje relativno neodvisno od ostalih nevronov. To posledično omogoča paralelno računanje, v praksi pa to pomeni zelo hitre operacije. Zato lahko nevronske mreže uporabimo pri reševanju kompleksnih problemov.

Stabilnost: Izgubljen ali nedelujoč nevron le malo pokvari natančnost delovanja nevronske mreže (več kot je takih nevronov, manj natančna je nevronska mreža). Nevronske mreže so tako stabilne v smislu uničenih nevronov in njihovih povezav. Poleg tega pa so stabilne tudi v smislu nepopolnih podatkov. To pomeni, da mrežo lahko naučimo pravilnega delovanja, tudi če v učnih priemerih del podatkov manjka.

Velja pa, da bolj nepopolni kot so podatki, slabša je aproksimacija nevronske mreže.

Matematično ozadje: Cilj nevronske mreže je aproksimirati poljubno funkcijo in kasneje na podlagi te aproksimacije napovedati vrednosti pri poljubnih vhodnih podatkih. Pri računanju se nevronske mreže močno opirajo na linearno algebro, linearnime preslikave, lastne vrednosti in vektorje ter negibne točke.

Glavna pomanjkljivost nevronskih mrež je težavno določanje topologije mreže. Medtem ko je število nevronov v vhodnem in izhodnem sloju določeno s problemom, je število nevronov v posameznem skritem sloju in število skritih slojev odvisno od posameznega primera. Idealno topologijo nevronske mreže je tako težko doseči, do nje lahko pridemo le s poskušanjem. Še ena slaba lastnost pa je nezmožnost razlaganja odločitev (ang. black box). Zaradi kompleksne strukture je vsak rezultat mreže produkt večih neodvisnih operacij nevronov, ki jih je težko razložiti.

2.5. Uporaba nevronskih mrež. Z razvojem strojne opreme v zadnjih letih so močno napredovale tudi nevronske mreže. Razvite so bile hitrejše implementacije algoritmov, posledično je zaradi njihove učinkovitosti uporaba nevronskih mrež danes zelo pogosta v industriji. Ko nevronske mreže naučimo pravilnega delovanja, se ta zna odzvati na nepoznane situacije. Tako lahko, zaradi njihove splošnosti in zmožnosti sprejetja velikega števila vhodnih podatkov, nevronske mreže uporabimo za reševanje najrazličnejših problemov. Nevronske mreže se dandanes uporabljajo na vseh mogočih področjih. Dva najpomembnejša namena uporabe sta napovedovanje in klasifikacija. Primeri uporabe v industriji so:

- prepoznavanje in klasifikacija (delov) slik,
- prepoznavanje ročno napisanih besedil,
- prepoznavanje govora,
- kontrola kvalitete v proizvodnji,
- določanje diagnoz v zdravstvu,
- napovedovanje različnih vrednosti glede na trenutne trende.

3. UČENJE NEVRONSKIH MREŽ

Najprej je smiselno definirati, kaj pojem učenje sploh pomeni. Po [1, str. 37] za učenje potrebujemo sistem, ki strmi k izpolnitvi določene naloge oziroma cilja. Pred učenjem sistem ni sposoben zadostno opraviti naloge. S ponavljanjem določenih opravil poskušamo sistem pripraviti do tega, da bo deloval bolje, kjer je bolje lahko hitreje, ceneje, bolj pravilno... Učenje lahko torej definiramo kot zaporedje ponovitev, kjer pri vsaki ponovitvi skušamo zmanjšati napako tako, da bi se zastavljenemu cilju čimbolj približali.

Obstaja kar nekaj različnih pravil, kako umetno nevronske mreže naučiti pravilnega delovanja. V svoji diplomski nalogi sem se osredotočila na posplošeno pravilo

delta oziroma vzvratno razširjanje napake, ki je najpogosteje uporabljan algoritem za učenje pri večslojnih perceptronih.

3.1. Nadzorovano učenje. Vzvratno razširjanje napake je primer nadzorovanega učenja. Za tako učenje potrebujemo učne primere – torej vhodne podatke in vrednosti, ki želimo, da jih mreža pri danih vhodnih podatkih vrača. Cilj učenja je najti funkcijo, ki bo najbolje aproksimirala relacijo med vhodnimi in izhodnimi podatki. Pri vzvratnem razširjanju napake do take funkcije pridemo tako, da spreminjamo vrednosti na utežeh, pri čemer poskušamo minimalizirati napako med želenim in dejanskim izhodom mreže.

Potrebno je omeniti še, da poleg nadzorovanega učenja poznamo tudi nenadzorovano učenje. Tu izhodnih podatkov nimamo. Mreža vhodne podatke obdela po svojih kriterijih in poskuša najti strukturo, ki povezuje vhodne podatke med seboj. Nevroske mreže se za nenadzorovano učenje uporabljajo redkeje.

3.2. Pravilo delta. Pravilo delta je eno izmed pravil, ki ga lahko uporabimo pri učenju nevronske mreže, ponavadi se uporablja pri dvoslojnih perceptronih. Če je mreža večslojna, potem govorimo o vzvratnem razširjanju napake oziroma posplošenem pravilu delta. Za pravilom delta stoji povsem preprosta ideja. Najprej uteži med nevroni poljubno nastavimo (pomembno je le, da niso vse uteži nastavljene na 0), nato na izbranem učnem primeru izračunamo, za koliko se je naša mreža zmotila glede na pričakovan izhod. Dobimo napako s pomočjo katere lahko vrednosti na utežeh popravimo tako, da omenjeno napako minimaliziramo. Postopek ponavljamo na ostalih učnih primerih, dokler se uteži ne ustalijo. Takrat se učenje konča.

Za začetek vzemimo preprost dvoslojni perceptron z enim izhodnim nevronom Y , število vhodnih nevronov je poljubno. Pravilo delta pri spreminjanju vrednosti uteži upošteva razliko med želenim in dobljenim izhodom mreže. Tako definiramo napako mreže $E(l)$ kot kvadrat razlike omenjenih vrednosti:

$$(4) \quad \begin{aligned} E(l) &= (d(l) - Y(l))^2 \\ &= (d(l) - WX(l))^2 \end{aligned}$$

kjer je $d(l)$ želen izhod in $Y(l) = WX(l)$ z mrežo izračunana vrednost pri l -tem učnem primeru, vhodnem vektorju $X(l)$ in matriki uteži W .

Pravilo delta temelji na gradientnem iskanju, vektor uteži popravljamo v negativni smeri odvoda napake po utežeh. Le ta se glasi:

$$\frac{dE(l)}{dW} = -2(d(l) - WX(l))X(l)$$

Sedaj lahko zapišemo enačbo za pravilo delta in nove vrednosti uteži W^N :

$$(5) \quad \begin{aligned} W^N &= W - \eta \frac{dE(l)}{dW} \\ &= W + \eta(d(l) - WX(l))X(l) \end{aligned}$$

V zgornji enačbi η predstavlja hitrost učenja in vpliva na to kako hitro bodo uteži konvergirale k optimalnim.

3.2.1. *Gradientno iskanje.* Za pravilom delta stoji gradientno iskanje, ki ni nič drugega kot iskanje lokalnega minimuma. Iskanje poteka tako, da začnemo na naključni točki na grafu (zato utežem na začetku pripišemo naključne vrednosti) in se nato v vsakem koraku premaknemo po krivulji v smeri negativnega odvoda, to je smer, kjer funkcija najhitreje pada. Dolžino vsakega premika določa stopnja učenja η , zato je pomembno kakšno vrednost zanjo izberemo.

3.3. **Vzratno razširjanje napake.** Vzratno razširjanje napake je zgolj posplošitev pravila delta za večslojne perceptrone. Problem pri računanju se pojavi, ker uteži med skritimi sloji niso direktno odvisne od napake med izhodnim stanjem in želeno vrednostjo, zato moramo pravilo malo prilagoditi. Ko popravljamo uteži, to počnemo od izhodnega sloja nevronov proti vhodnemu (od tod tudi ime – vzratno razširjanje napake oz. *backpropagation* v angleščini).

3.3.1. *Notacija.* Definirajmo notacijo, ki jo bomo uporabili pri izpeljavi vzratnega razširjanja napake.

- X_i označuje i -ti nevron v vhodnem sloju.
- H_{ki} označuje i -ti nevron v k -tem skritem sloju.
- Y_i označuje i -ti nevron v izhodnem sloju.
- $W_{ij}^{(k)}$ označuje utež med j -tim nevronom v $k - 1$ plasti in i -tim nevronom v k -ti plasti.
- A_{ki} označuje stanje i -tega nevrona v k -ti skriti plasti po uporabljeni funkciji kombinacije: $A_{ki} = \sum_{j=1}^{N_{k-1}} W_{ij}^{(k)} H_{k-1,j}$, poseben primer je stanje i -tega nevrona v izhodni plasti, ki ga označimo z A_i .

3.3.2. *Izpeljava vzratnega razširjanja napake.* Vzemimo splošen večslojen perceptron z N_X nevroni v vhodnem sloju in N_Y nevroni v izhodnem sloju. Nevronska mreža naj sestoji iz m skritih slojev, $m > 0$, vsak skriti sloj pa naj vsahuje N_k nevronov, kjer velja $1 \leq k \leq m$.

Preden se lotimo popravljanja uteži, moramo za dani l -ti učni primer s pomočjo funkcije nevrona izračunati izhodne vrednosti. Za i -ti nevron v prvem skitem sloju lahko zapišemo:

$$H_{1i}(l) = f \left(\sum_{j=1}^{N_X} W_{ij}^{(1)} X_j(l) \right)$$

in

$$A_{1i}(l) = \sum_{j=1}^{N_X} W_{ij}^{(1)} X_j(l).$$

Za poljubni k -ti skriti sloj, kjer je $1 \leq k \leq m$, tako zapišemo:

$$H_{ki}(l) = f(A_{ki}(l)),$$

kjer je

$$A_{ki}(l) = \sum_{j=1}^{N_{k-1}} W_{ij}^{(k)} H_{k-1,j}(l).$$

V zgornji enačbi $H_{k-1,j}$ označuje že izračunane vrednosti nevronov v predhodnem sloju.

V izhodnem sloju dobimo vrednosti, ki nas najbolj zanimajo. Velja:

$$Y_i(l) = f(A_i(l)) \quad \text{in} \quad A_i(l) = \sum_{j=1}^{N_m} W_{ij}^{(m+1)} H_{m,j}(l).$$

Tako smo dobili izhodne vrednosti, ki jih je izračunala mreža. Vrednosti primerjamo s tistimi, ki bi si jih želeli dobiti za dani l -ti učni primer. Za posamezen nevron ozančimo napako sledeče:

$$(6) \quad e_i(l) = d_i(l) - Y_i(l),$$

oziroma v vektoski obliki:

$$e(l) = d(l) - Y(l).$$

Podobno kot pri pravilu delta, napako celotne nevronske mreže za l -ti učni primer definiramo:

$$(7) \quad E(l) = \frac{1}{2} \sum_{i=1}^{N_Y} e_i^2(l).$$

Napako želimo minimalizirati in podobno kor prej s pomočjo gradientnega iskanja uteži popraviti v smeri njenega negativnega odvoda. Nove uteži izračunamo s pomočjo enake formule kot prej (enačba (5)):

$$W^N = W - \eta \frac{dE}{dW},$$

kjer η označuje stopnjo učenja.

Pri izračunu zgornjega odvoda si bomo pomagali s verižnim pravilom za odvajanje. Za izhodni sloj je napako enostavno izračunati. Problem se pojavi pri skritih slojih, saj uteži niso direktno odvisne od napake. Kot bomo videli v nadaljevanju, si bomo tudi v tem primeru pomagali z verižnim pravilom.

Najprej razpišimo formulo (7), kjer upoštevamo formuli za e_i in Y_i :

$$\begin{aligned} E(l) &= \frac{1}{2} \sum_{i=1}^{N_Y} e_i^2(l) \\ &= \frac{1}{2} \sum_{i=1}^{N_Y} (d_i(l) - Y_i(l))^2(l) \\ &= \frac{1}{2} \sum_{i=1}^{N_Y} \left(d_i(l) - f\left(\sum_{j=1}^{N_m} W_{ij}^{(m+1)} H_{m,j}(l)\right) \right)^2 \end{aligned}$$

Sedaj za izračun odvoda uporabimo verižno pravilo, formulo razpišemo za posamezno utež:

$$(8) \quad \frac{\partial E(l)}{\partial W_{ij}^{(m+1)}} = \frac{\partial E(l)}{\partial e_i(l)} \frac{\partial e_i(l)}{\partial Y_i(l)} \frac{\partial Y_i(l)}{\partial A_i(l)} \frac{\partial A_i(l)}{\partial W_{ij}^{(m+1)}}$$

Zgornje parcialne odvode pa lahko ob pomoči definicij posameznih izrazov zapišemo sledeče:

$$(9) \quad \frac{\partial E(l)}{\partial e_i(l)} = \frac{\partial(\frac{1}{2} \sum_{i=1}^{N_Y} e_i^2(l))}{\partial e_i(l)} = e_i(l)$$

$$(10) \quad \frac{\partial e_i(l)}{\partial Y_i(l)} = \frac{\partial(d_i(l) - Y_i(l))}{\partial Y_i(l)} = -1$$

$$(11) \quad \frac{\partial Y_i(l)}{\partial A_i(l)} = \frac{\partial(f(A_i(l)))}{\partial A_i(l)} = f'(A_i(l))$$

$$(12) \quad \frac{\partial A_i(l)}{\partial W_{ij}^{(m+1)}} = \frac{\partial \sum_{j=1}^{N_m} W_{ij}^{(m+1)} H_{m,j}(l)}{\partial W_{ij}^{(m+1)}} = H_{m,j}(l)$$

Pri tem velja, da je f' odvod funkcije aktivacije in je odvisen od tega, katero funkcijo aktivacije uporabimo.

Zaradi boljše preglednosti parcialni odvod zapišemo še malo drugače:

$$\frac{\partial E(l)}{\partial W_{ij}^{(m+1)}} = \frac{\partial E(l)}{\partial A_i(l)} \frac{\partial A_i(l)}{\partial W_{ij}^{(m+1)}}$$

Če upoštevamo zgoraj izračunani odvod dobimo:

$$\frac{\partial E(l)}{\partial W_{ij}^{(m+1)}} = \frac{\partial E(l)}{\partial A_i(l)} H_{m,j}(l)$$

Označimo:

$$(13) \quad \frac{\partial E(l)}{\partial A_i(l)} = \Delta_i^{(m+1)}(l)$$

Združimo vse zgornje rezultate in zapišemo:

$$\begin{aligned}
(14) \quad W_{ij}^{N(m+1)} &= W_{ij}^{(m+1)} - \eta \frac{dE(l)}{dW_{ij}^{(m+1)}} \\
&= W_{ij}^{(m+1)} - \eta \Delta_i^{(m+1)}(l) H_{mj}(l) \\
&= W_{ij}^{(m+1)} - \eta e_i(l) f'(A_i(l)) H_{mj}(l)
\end{aligned}$$

Tako smo izpeljali pravilo za spremembno uteži, ki povezujejo izhodni sloj s predhodnikom.

Vzvratno razširjanje napake pride do izraza šele pri skritih slojih. Ker ne moremo direktno izračunati napake e_i , si pri računanju pomagamo z že spremenjenimi utežmi v izhodnem sloju in pa $\Delta^{(m+1)}$, ki smo ga spotoma izračunali. Za uteži $W_{ij}^{(m)}$, ki povezujejo predzadnji in zadnji skriti sloj lahko ob upoštevanju verižnega pravila zapišemo:

$$\frac{\partial E(l)}{\partial W_{ij}^{(m)}} = \frac{\partial E(l)}{\partial A_{mi}(l)} \frac{\partial A_{mi}(l)}{\partial W_{ij}^{(m)}}$$

Drugi člen odvoda izračunamo podobno kot prej

$$\frac{\partial A_{mi}(l)}{\partial W_{ij}^{(m)}} = H_{m-1,j}(l)$$

in označimo

$$\frac{\partial E(l)}{\partial A_{mi}(l)} = \Delta_i^{(m)}(l)$$

Zopet uporabimo verižno pravilo za odvajanje in zapišemo:

$$\Delta_i^{(m)}(l) = \frac{\partial E(l)}{\partial H_{mi}(l)} \frac{\partial H_{mi}(l)}{\partial A_{mi}(l)}$$

Drugi člen predstavlja odvod funkcije aktivacije:

$$\frac{\partial H_{mi}(l)}{\partial A_{mi}(l)} = f'(A_{mi}(l)),$$

prvi člen pa ponovno razpišemo s pomočjo verižnega pravila sledeče:

$$\begin{aligned}
\frac{\partial E(l)}{\partial H_{mi}(l)} &= \sum_{j=1}^{N_Y} \frac{\partial E(l)}{\partial A_j(l)} \frac{\partial A_j(l)}{\partial H_{mi}(l)} \\
&= \sum_{j=1}^{N_Y} \frac{\partial E(l)}{\partial A_j(l)} \frac{\partial (\sum_{p=1}^{N_m} W_{jp}^{N(m+1)} H_{mp}(l))}{\partial H_{mi}(l)} \\
&= \sum_{j=1}^{N_Y} \frac{\partial E(l)}{\partial A_j(l)} W_{ji}^{N(m+1)}
\end{aligned}$$

Pri tem velja, da so elementi matrike uteži $W^{N(m+1)}$ vrednosti, ki smo jih že popravili v prejšnjem koraku. Uporabimo enačbo (13) in izrazimo vrednost $\Delta_i^{(m)}(l)$ sledeče:

$$\begin{aligned}\Delta_i^{(m)}(l) &= f'(A_{mi}(l)) \sum_{j=1}^{N_Y} \frac{\partial E(l)}{\partial A_j(l)} W_{ji}^{N(m+1)} \\ &= f'(A_{mi}(l)) \sum_{j=1}^{N_Y} \Delta_j^{(m+1)}(l) W_{ji}^{N(m+1)}\end{aligned}$$

Vse združimo in vstavimo v enačbo za izračun spremenjenih uteži $W^{N(m)}$ ter dobimo:

$$\begin{aligned}(15) \quad W_{ij}^{N(m)} &= W_{ij}^{(m)} - \eta \frac{dE(l)}{dW_{ij}^{(m)}} \\ &= W_{ij}^{(m)} - \eta \Delta_i^{(m)}(l) H_{m-1,j}(l) \\ &= W_{ij}^{(m)} - \eta f'(A_{mi}(l)) \left(\sum_{p=1}^{N_Y} \Delta_p^{(m+1)}(l) W_{pi}^{N(m+1)} \right) H_{m-1,j}(l)\end{aligned}$$

Za uteži $W^{(k)}$, ki povezujejo poljuben $k-1$ in k -ti skriti sloj, kjer je $1 < k < m$, podobno zapišemo:

$$\frac{\partial A_{ki}(l)}{\partial W_{ij}^{(k)}} = H_{k-1,j}(l)$$

in

$$\Delta_i^{(k)}(l) = f'(A_{ki}(l)) \sum_{j=1}^{N_{k+1}} \Delta_j^{(k+1)}(l) W_{ji}^{N(k+1)}$$

Spremenjene uteži nato izračunamo:

$$\begin{aligned}(16) \quad W_{ij}^{N(k)} &= W_{ij}^{(k)} - \eta \Delta_i^{(k)}(l) H_{k-1,j}(l) \\ &= W_{ij}^{(k)} - \eta f'(A_{ki}(l)) \left(\sum_{p=1}^{N_{k+1}} \Delta_p^{(k+1)}(l) W_{pi}^{N(k+1)} \right) H_{k-1,j}(l)\end{aligned}$$

Ostane nam le še matrika uteži $W^{(1)}$, ki povezuje vhodni sloj s prvim skritim slojem. Tu velja:

$$\frac{\partial A_{1i}(l)}{\partial W_{ij}^{(1)}} = X_j(l)$$

in

$$\Delta_i^{(1)}(l) = f'(A_{1i}(l)) \sum_{j=1}^{N_2} \Delta_j^{(2)}(l) W_{ji}^{N(2)}$$

Uteži pa popravimo po sledeči formuli:

$$\begin{aligned}
 W_{ij}^{N(1)} &= W_{ij}^{(1)} - \eta \Delta_i^{(1)}(l) X_j(l) \\
 (17) \qquad &= W_{ij}^{(1)} - \eta f'(A_{1i}(l)) \left(\sum_{p=1}^{N_2} \Delta_p^{(2)}(l) W_{pi}^{N(2)} \right) X_j(l)
 \end{aligned}$$

Tako smo izpeljali pravilo za računanje spremembe uteži pri vzvratnem razširjanju napake.

3.4. Lastnosti vzvratnega razširjanja napake. Kljub temu, da je vzvratno razširjanje napake zelo priljubljeno pravilo pri učenju nevronske mreže, ima tudi nekaj pomanjkljivosti.

Gradientno iskanje temelji na iskanju lokalnega minimuma, posledično uteži ne konvergirajo vedno k globalno najboljši izbiri. Lahko se zgodi, da bo matrika uteži konvergirala k lokalnemu optimumu, taka mreža ne bo dajala zadovoljivih rezultatov. Kam bodo uteži konvergirale je odvisno predvsem od njihove začetne izbire. Če med učenjem pride do omenjenega problema moramo izbrati novo naključno matriko uteži in učenje ponoviti v upanju, da bodo uteži tokrat konvergirale k optimalnim.

Naslednji problem, ki se pojavi kot posledica gradientnega iskanja je določanje parametra η . Stopnja učenja določa kako hitro in kam bodo uteži konvergirale. Žal pa ni pravila, ki bi določalo kakšno vrednost je za η najbolje izbrati. Do take vrednosti lahko pridemo le s poskušanjem.

Slabost vzvratnega razširjanja napake je tudi to, da je za učenje nevronske mreže potrebno zelo veliko število učnih primerov. Ponavadi se to število giblje med nekaj 10000 in 100000. Posledično je učenje lahko zelo zamudno.

Potrebno je omeniti še časovno zahtevnost. Zaradi svoje kompleksne zgradbe so nevronske mreže časovno potratne in njihovo učenje praviloma ni hitro. Z večjo mrežo le to zahtevnost še povečamo. Kljub temu, pa lahko z različnimi implementacijami malo vplivamo na hitrost mreže.

Včasih se lahko zgodi, da napaka mreže med učenjem začne rasti. Pomembno je, da učenje takrat prekinemo, saj lahko sicer pride do preprileganja (ang. overfitting). Preprileganje pomeni, da mreža zelo dobro deluje na učnih primerih (napaka je tu majhna), aproksimacija za poljuben primer pa ni natančna. Zato je pomembno, da med učenjem ves čas spremljamo vrednost napake.

3.5. Pristranskost in varianca. Po končanem učenju lahko izračunamo pristranskost (ang. bias) in varianco nevronske mreže. Ta podatka povesta, kako uspešni smo bili pri aproksimaciji funkcije oziroma učenju nevronske mreže. Ker gre za aproksimacijo, bo pristranskost vedno prisotna, želimo pa si, da bi bila le ta čim manjša. Omenjeni količini izračunamo s pomočjo testnih primerov.

S t označimo točno vrednost, ki bi jo želeli dobiti, s \hat{t} pa vrednost, ki jo mreža vrne. Z $E[\hat{t}]$ označimo pričakovano napoved, to je povprečje vrednosti, ki jih

dobimo, če mrežo poženemo na več testnih primerih. Pristranskost mreže pri t -tem učnem primeru označimo z $N(t)$ in izračunamo sledeče:

$$(18) \quad N(t) = E[\hat{t}] - t.$$

Problem se pojavi, če točne vrednosti t ne poznamo. Včasih lahko iz ostalih podatkov sklepamo kakšna bi ta vrednost morala biti ali pa jo poskusimo oceniti na drugačen način. Pristranskost meri, kako dobro mreža deluje na učnih primerih. Majhna pristranskost pomeni, da je bilo učenje uspešno in da bo vsaj napoved na primerih, s katerimi smo mrežo učili, dobra. Če je pristranskost prevelika jo zmanjšamo tako, da izberemo večjo nevronske mreže.

Nadalje definiramo še varianco

$$(19) \quad Var(\hat{t}) = E[E[\hat{t}] - \hat{t}]^2.$$

Kot je razvidno iz zgornje enačbe, $Var(\hat{t})$ ni odvisna od točnih vrednosti t . Varianca meri, kako točna bo napoved mreže pri neznanih podatkih. Stremimo k temu, da bi bila varianca čim manjša. Če je le ta prevelika, jo zmanjšamo tako, da povečamo število učnih primerov. Če je varianca še vedno velika, lahko poskusimo tudi z drugačno topologijo mreže.

Spomnimo se, kako se glasi napaka, ki jo pri posplošenem pravilu delta želimo minimalizirati. Če enačbo napake (enačba (7)) prepisemo v smislu novo označnih količin, dobimo:

$$(20) \quad E[(\hat{t} - t)^2] = E[(\hat{t} - E[\hat{t}] + E[\hat{t}] - t)^2]$$

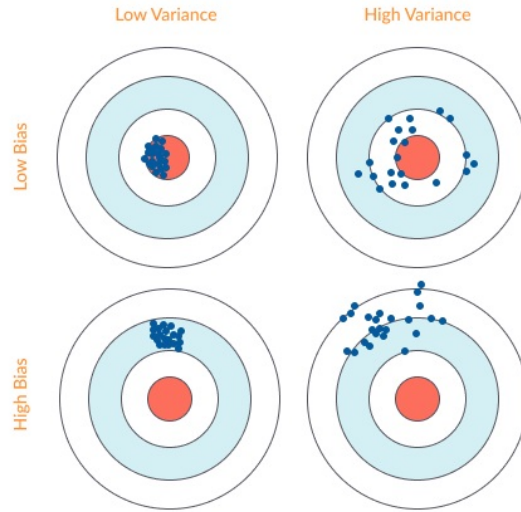
Upoštevamo še $E[t] = t$ in $E[E[\hat{t}]\hat{t}] = E[\hat{t}]^2$ in zapipšemo:

$$(21) \quad \begin{aligned} E[(\hat{t} - t)^2] &= (E[\hat{t}] - t)^2 + E[E[\hat{t}] - \hat{t}]^2 \\ &= N(t)^2 + Var(\hat{t}) \end{aligned}$$

Iz zgornje enačbe vidimo, da sta pristranskost in varianca mreže tesno povezani med seboj in z učenjem mreže poskušamo minimalizirati njuno vsoto. Pri minimumu napake mreže pa se vseeno lahko dobimo različne vrednosti omenjenih parametrov. Poglejmo si, kako te vrazlične vrednosti vplivajo na delovanje mreže.

Če se zgodi, da je pristranskost majhna, varianca pa velika, temu pravimo preprileganje nevronske mreže. To pomeni, da mreža zelo dobro deluje na učnih podatkih, ni pa sposobna znanja posplošiti na poljubne vhodne podatke. Po drugi strani lahko pride do neprileganja. To se zgodi, če sta tako pristranskost kot varianca veliki. Taka mreža ni sposobna dobro aproksimirati niti učnih podatkov, na dobro predikcijo splošnih podatkov lahko pozabimo.

Če pride do preprileganja moramo učenje ponoviti, le da tokrat izberemo drugačne vrednosti začetnih uteži. Postopek ponavljamo, dokler z delovanjem mreže nismo zadovoljni. Neprileganje lahko odpravimo tako, da spremenimo topologijo mreže. Lahko dodamo oziroma odstranimo nevrone, sloje ali pa dodamo kakšen parameter v vhodni sloj nevronov. Prav tako lahko dodamo nekaj učnih primerov



SLIKA 5. Pristranskost in varianca, vir[4]

oziroma izboljšamo njihovo kakovost. Načeloma velja, da več učnih primerov kot bomo pgnali, bolje bo mreža delovala.

3.6. Normalizacija podatkov. Različne vhodne vrednosti imajo lahko različne enote in se razlikujejo za kar nekaj velikostnih redov. Taki podatki pri učenju ne bodo dali dobrih rezultatov. Da optimiziramo delovanje mreže, je podatke pred učenjem potrebno normalizirati oziroma pretvoriti v enak velikostni red. Želimo si, da bi bili podatki majhni, tako bo učenje boljše. Obstaja več pravil za normalizacijo podatkov, najbolj pogosti sta min-max normalizacija in z-score normalizacija, ki je včasih omenjena tudi kot standarizacija. Kako bomo podatke normalizirali je odvisno od primera, ki ga obravnavamo.

3.6.1. *Min-max normalizacija.* Podatke normaliziramo po sledeči enačbi:

$$(22) \quad X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Tako podatke skrajšamo na interval $[0, 1]$.

3.6.2. *Z-score normalizacija.* Cilj je podatke preoblikovati tako, da bomo dobili standardno normalno porazdelitev z povprečjem 0 in standardnim odkolonom 1. To dosežemo, če podatke preoblikujemo po sledeči enačbi:

$$(23) \quad X_{norm} = \frac{X - \mu}{\sigma},$$

pri čemer velja, da je μ povprečna vrednost in σ standardni odklon podatkov.

4. FUNKCIJSKI PROGRAMSKI JEZIK OCAML

Poleg funkcijskih poznamo še objektne in proceduralne programske jezike. Vsi načini programiranja imajo svoje prednosti in slabosti. Funkcijsko programiranje je zelo priljubljeno za pisanje generičnih algoritmov in algoritmov, ki temeljijo na rekurziji.

Način programiranja in programski jezik izberemo na podlagi več dejavnikov – včasih izberemo tak jezik, da bo program deloval kar se da hitro, drugač izberemo jezik v katerem bo zastavljeni problem najlažje rešljiv. Sama sem se za OCaml odločila, ker so implementacije nevronske mreže v funkcijskih programskih jezikih zelo redke in mi je to predstavljalo izziv.

4.1. Lastnosti funkcijskih programskih jezikov. V funkcijskem programiranju so osnovni objekti funkcije. To pomeni, da lahko funkcije sprejmejo ozrioma vračajo druge funkcije, poleg tega lahko funkcije shranimo v podatkovne strukture. Funkcijam, ki kot argument sprejmejo druge funkcije pravimo *funkcije višjega reda*. Ena izmed osnovnih funkcij višjega reda je na primer funkcija `map`. Le ta sprejme podatkovno strukturo, naprimer seznam ali tabelo, in na vsakem njenem elementu uporabi funkcijo, ki smo jo podali kot argument.

Listing 1 Primer uporabe funkcije `map`, ki funkcijo aktivacije uporabi na sloju nevronov, ki ga podamo s tabelo

```
val activation_layer : ('a -> 'b) -> 'a array -> 'b array = <fun>
let activation_layer funct layer =
    Array.map funct layer
```

4.1.1. Čiste funkcije. Funkcija je čista (ang. pufe function), če med izvajanjem ne sproži nobenih stranskih učinkov kot so npr. izpisovanje na zaslon, branje z tipkovnice, spreminjanje vrednosti spremenljivk... Poleg tega čiste funkcije pri enakih argumentih vedno vračajo enak rezultat, posledično lahko tako funkcijo vedno zamenjamo z njeno definicijo. Čiste funkcije v praksi pomenijo manj napak med programiranjem, če se le te že pojavijo, jih je lahko najti in odpraviti. V funkcijskem programiranju je večina funkcij čistih, saj funkcije načeloma vrednosti ne spreminjajo, ampak jih vračajo.

4.2. Lastnosti OCamla. OCaml je splošen programski jezik, kljub temu da ima veliko lastnosti funkcijskih jezikov. Poleg le teh, ima OCaml tudi lastnosti objektnega programiranja. Posledično OCaml odlikuje kar nekaj pozitivnih lastnosti. Izvajanje kode je hitro in učinkovito, jezik ima velik poudarek tudi na varnosti, zato do napak le redko prihaja.

Funkcije v OCamlu so večinoma čiste, ne drži pa to vedno. S pomočjo referenc lahko namreč spreminjamo vrednosti spremenljivk. Zaradi narave nevronske mreže sem to lastnost pri implementaciji izkoristila kar nekajkrat.

Dobra lastnost OCaml je tudi, da optimizira repno rekurzijo. Klic funkcije je repen, če se izvede zadnji. Pri rekurziji gre omeniti še OCamlovo sposobnost efektivnega ujemanja vzorcev.

OCaml ima dobro podprt sistem tipov. Tipi v OCamlu so statični, a se tudi tej lastnosti v določenih situacijah da izogniti z referencami. Jezik omogoča pisanje polimorfnih funkcij, tako da le te delujejo na poljubnem tipu vhodnih podatkov, kar je pri programiranju dostikrat zaželeno.

5. IMPLEMENTACIJA NEVRONSKE MREŽE

Del diplomske naloge je bila tudi implementacija usmerjene nevronske mreže v programskem jeziku OCaml. Cilj je bil implementirati generičen večslojni perceptron z vzratnim razširjanjem napake. Nadalje sem nevronske mreže tudi testirala na danem naboru podatkov in jo analizirala. Celotna koda se nahaja na koncu naloge, v prilogi 6.

5.1. Primer nevronske mreže. Glavna funkcija, ki poskrbi, da se mreža nauči delovanja je `train_network`. Funkcija vrne seznam matrik uteži, sprejme pa naslednje vhodne podatke:

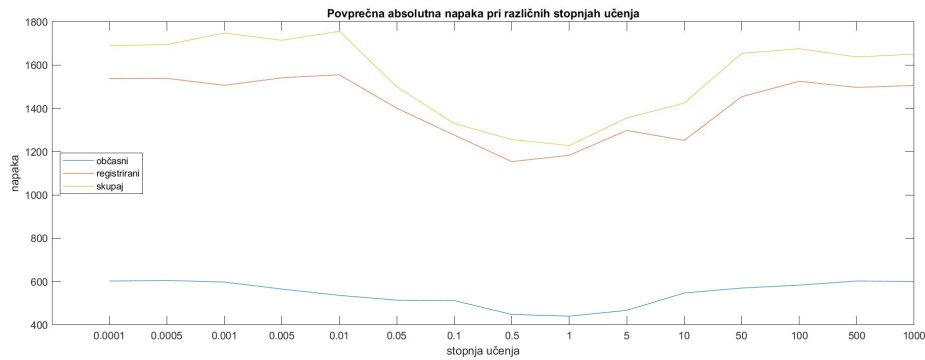
- `input_array`: tabela vhodnih vektorjev
- `output_array`: tabela pripadajočih zelenih izhodnih vektorjev
- `network_topology`: tabela, ki opisuje topologijo mreže s št. nevronov po slojih
- `rate`: stopnja učenja
- `bound`: meja, ki določa maksimalno vrednost začeno nastavljenih naključnih uteži
- `act_fun`: funkcija aktivacije
- `act_der`: odvod funkcije aktivacije

Posledično se signatura funkcije glasi:

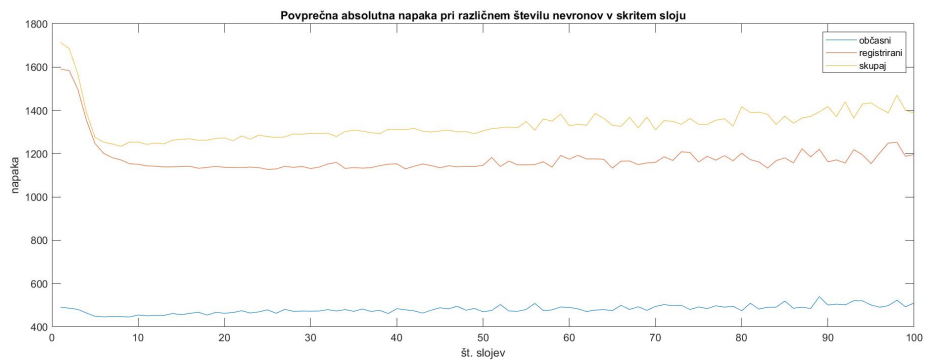
Listing 2 Signatura funkcije `train_network`

```
val train_network :  
    float array array -> float array array -> int array ->  
    float -> float -> (float -> float) -> (float -> float) ->  
    float array array array =  
<fun>
```

5.2. Evalvacija na naboru podatkov.



SLIKA 6. BLAAAA



SLIKA 7. DOPIŠI

6. PRILOGA

(* NEVRONSKA MREŽA *)

(*aktivacijska funkcija za posamezen nevron - sigmoida *)

```
let sigmoid neuron =
  1. /. (1. +. exp(-. neuron))
```

(*odvod sigmoide za posamezen izhodni nevron, sprejme output - to kar mreža dobi odvod sigmoide je $f'(x) = f(x) * (1 - f(x))$ *)

```
let d_sigmoid output =
  (output) * . (1. -. output)
```

(*aktivacijska funkcija za sloj, funct je funkcija aktivacije, ki jo želimo uporabiti *)

```

let activation_layer funct layer =
    Array.map funct layer

(*funkcija kombinacije za sloj in pripadajoče uteži,
zmnoži vektor (sloj neuronov) z matriko uteži,
vrne vektor*)
let combination_f layer weights =
    let w1 = Array.length weights.(0) and
        w2 = Array.length weights and
        l = Array.length layer in
    if w1 <> l then failwith "incompatible matrices!"
    else(
        let comb = Array.make w2 0. in
        for i = 0 to w2-1 do
            for j = 0 to w1-1 do
                comb.(i) <-
                    comb.(i) +. weights.(i).(j) *. layer.(j)
            done;
        done;
        comb
    )

(*ustvari float matriko z naključnimi utežmi velikosti nxm,
elementi so manjši od bound - float, spodnja meja je 0.,
m je št neuronov v prejšnjem sloju, n je št neuronov v tem sloju,
oba int vrne matriko mxn*)
let rand_weights n m bound =
    Array.init n
    (fun _
        -> Array.init m (fun _
            -> (Random.float 2.*.bound)-.bound))

(*vrne največji element v tabeli*)
let max_el arr = Array.fold_left max arr.(0) arr

(*ustvari tabelo matrik naključnih uteži s pravimi velikostmi,
sprejme vektor, ki opisuje topologijo neuronske mreže - int array,
in zgornjo mejo teh uteži - float*)
let create_weights_matrix network_topology bound =
    let n = (Array.length network_topology) - 1 in
    let weights = Array.make n [[]] in
    for i = 0 to n-1 do

```

```

        weights.(i) <-
            rand_weights (
                network_topology.(i+1) network_topology.(i) bound
            )

done;
weights

(*ustvari float matriko, ki predstavlja nevrone v mreži.
Sprejme int array, ki opisuje topologijo mreže
Ima velikost m*n, kjer je m št slojev in n max št neuronov v sloju,
na začetku so vse vrednosti 0.*)
let initialize_network network_topology =
    let layers_n = Array.length network_topology and
        max_layer = max_el network_topology in
    let network = Array.make_matrix layers_n max_layer 0. in
    network

(*delta = dE/dA*)

(*izračuna napako za izhodni sloj, vrne vektor*)
(*d je željen output, y to kar mreža dobi,
act_der je odvod funkcije aktivacije*)
let delta_output act_der d y =
    let d0 = Array.length d and
        y0 = Array.length y in
    if y0 <> d0 then failwith "wrong dimensions!"
    else (
        let e = Array.make d0 0. in
        for i = 0 to d0-1 do
            e.(i) <- -. (d.(i) -. y.(i)) *. (act_der y.(i))
        done;
        e
    )

(*izračuna napako za skrite sloje,
w so uteži enega sloja naprej(že popravljene!),
h_m so izhodi v tem sloju, d_out je delta naslednjega sloja,
vrne vektor*)
let delta_hidden w h_m d_out act_der =
    let a = Array.length h_m and
        d0 = Array.length d_out in
    let e = Array.make a 0. in

```

```

for i = 0 to a-1 do
  let sum = ref 0. in
  for j = 0 to d_0 - 1 do
    sum := !sum +. w.(j).(i) *. d_out.(j)
  done;
  e.(i) <- (act_der h_m.(i)) *. !sum;
done;
e

```

*(*w so uteži, ki jih popravljaš, rate je stopnja učenja,
delta je error - pomožne funkcije,
input je vektor nevronov en sloj pred utežjo (eno stopnjo nazaj),
vrne popravljene uteži *)*

```

let update_weights w rate delta input =
  let d_0 = Array.length delta and
    in_0 = Array.length input in
  for i = 0 to d_0 -1 do
    for j = 0 to in_0 -1 do
      w.(i).(j) <-
        w.(i).(j) -. rate *. delta.(i) *. input.(j)
    done;
  done;
w

```

*(*funkcija, ki vrne povprečje danih vhodnih podatkov
vhod: matrika s podatki
izhod: vektor z povprečji po stolpcih
)*

```

let mean data =
  let n = Array.length data in
  let n_a = Array.length data.(0) in
  let m = Array.make n_a 0. in
  for i = 0 to n-1 do
    for j = 0 to n_a - 1 do
      m.(j) <- m.(j) +. data.(i).(j)
    done;
  done;
  for j = 0 to n_a - 1 do
    m.(j) <- m.(j) /. (float_of_int n)
  done;
m

```



```

(*funkcija, ki vrne standardni odklon danih vhodnih podatkov
vhod: matrika s podatki
izhod: vektor standardnih odklonov po stolpcih
*)
let std data mean =
  let n = Array.length data in
  let n_a = Array.length data.(0) in
  let s = Array.make n_a 0. in
  for i = 0 to n-1 do
    for j = 0 to n_a - 1 do
      s.(j) <- s.(j) +. ((data.(i).(j) -. mean.(j))**2.)
    done;
  done;
  for j = 0 to n_a - 1 do
    s.(j) <- sqrt(s.(j) /. (float_of_int n))
  done;
  s

```

```

(*normalizira podatke z z-score normalizacijo
vhod: tabela, povprečje in standardni odklon
izhod: normalizirana tabela
*)
let norm_z_score a mean std_dev =
  let n = Array.length a in
  let arr = Array.copy a in
  for i = 0 to n-1 do
    arr.(i) <- (arr.(i) -. mean.(i)) /. std_dev.(i)
  done;
  arr

```

```

(*denormalizira podatke z z-score normalizacijo
vhod: tabela, povprečje in standardni odklon
izhod: denormalizirana tabela oz. prave vrednosti
*)
let denorm_z_score a mean std_dev =
  let n = Array.length a in
  let arr = Array.copy a in
  for i = 0 to n-1 do
    arr.(i) <- (arr.(i) *. std_dev.(i)) +. mean.(i)
  done;
  arr

```

```

(*funkcija, ki sprejme en učni primer in ustrezno popravi uteži
input: x - vhodni vektor
      d - željen izhodni vektor
      network - matrika neuronov v mreži,
                velikosti (št. slojev) x (max neuronov v sloju),
                vsi el. so 0.
      weights - tabela matrik uteži v mreži
      rate - stopnja učenja
      act_fun - funkcija aktivacije
      act_der - odvod funkcije aktivacije
output: popravljene uteži*)
let learning_example x d network weights rate act_fun act_der =
  let n = Array.length network in
    network.(0) <- x;
    (*forward propagation*)
    for i = 0 to n-2 do
      network.(i+1) <- activation_layer act_fun(
        combination_f network.(i) weights.(i)
      );
    done;
    let delta = ref(delta_output act_der d network.(n-1)) in
    for i = 0 to n-2 do
      weights.(n-2-i) <- update_weights (
        weights.(n-2-i) rate !delta network.(n-2-i)
      );
      delta := (delta_hidden (
        weights.(n-2-i) network.(n-2-i) !delta act_der
      ))
    done;
    weights

```

```

(* funkcija, ki nauči mrežo (uteži) pravilnega delovanja
vhod:
- input_array: tabela vhodnih vektorjev
- output_array: tabela pripadajočih zelenih izhodnih vektorjev
- network_topology: tabela s št. neuronov po slojih
- rate: stopnja učenja
- bound: zgornja meja, ko inicializiramo uteži
- act_fun: funkcija aktivacije

```

```

- act_der: odvod funkcije aktivacije
izhod:
- naučene uteži
*)
let train_network input_array output_array
    network_topology rate bound act_fun act_der =
    let network = initialize_network network_topology in
    let weights = create_weights_matrix network_topology bound in
    let mean_in = mean input_array in
    let std_dev_in = std input_array mean_in in
    let mean_out = mean output_array in
    let std_dev_out = std output_array mean_out in
    let n = Array.length input_array in
    for i = 0 to n-1 do
        let norm_input_array = norm_z_score (
            input_array.(i) mean_in std_dev_in) in
        let norm_output_array = norm_z_score (
            output_array.(i) mean_out std_dev_out) in
        let pom = ref (learning_example
            norm_input_array norm_output_array
            network weights rate act_fun act_der) in
        for i = 0 to (Array.length weights)-1 do
            weights.(i) <- !pom.(i);
        done;
    done;
weights

```

(* funkcija, ki nauči mrežo (uteži) pravilnega delovanja,
sprejme matriko z utežmi
vhod:

- input_array: tabela vhodnih vektorjev
- output_array: tabela pripadajočih željenih izhodnih vektorjev
- network_topology: tabela s št. nevronov po slojih
- rate: stopnja učenja
- w: vhodne inicializirane uteži
- act_fun: funkcija aktivacije
- act_der: odvod funkcije aktivacije

izhod:

```

- naučene uteži
*)
let train_with_input_weights input_array output_array
    network_topology rate w act_fun act_der =

```

```

let network = initialize_network network_topology in
let mean_in = mean input_array in
let std_dev_in = std input_array mean_in in
let mean_out = mean output_array in
let std_dev_out = std output_array mean_out in
let n = Array.length input_array in
for i = 0 to n-1 do
  let norm_input_array = norm_z_score (
    input_array.(i) mean_in std_dev_in) in
  let norm_output_array = norm_z_score (
    output_array.(i) mean_out std_dev_out) in
  let pom = ref (learning_example
    norm_input_array norm_output_array network w
    rate act_fun act_der) in
  for i = 0 to (Array.length w)-1 do
    w.(i) <- !pom.(i);
  done;
done;
w

(* funkcija, ki napove vrednost pri testnem primeru
vhod:
- input: vhodna tabela
- network: matrika, ki predstavlja nevrone v mreži
- weights: tabela matrik uteži
- act_fun: funkcija aktivacije
izhod:
- napoved
*)
let predict input network weights act_fun =
  let n = Array.length network in
  network.(0) <- input;
  for i = 0 to n-2 do
    network.(i+1) <- activation_layer act_fun (
      combination_f network.(i) weights.(i)
    )
  done;
  network.(n-1)

(* funkcija, ki napove vrednost pri testnem primeru
vhod:
- test_input: tabela vhodnih vektorjev

```

- *test_output*: tabela izhodnih vektorjev
- *weights*: tabela matrik uteži
- *network_topology*: tabela, ki opisuje topologijo mreže
- *act_fun*: funkcija aktivacije
- *mean_in*: tabela povprečnih vrednosti vhodnih podatkov
- *std_in*: tabela standardnih odklonov vhodnih podatkov
- *mean_out*: tabela povprečnih vrednosti izhodnih podatkov
- *std_out*: tabela standardnih odklonov izhodnih podatkov

izhod:

```

- povprečna napaka med dobljeno in želeno vrednostjo
*)
let evaluate test_input test_output weights network_topology
  act_fun mean_in std_in mean_out std_out =
  let network = initialize_network network_topology in
  let n = Array.length test_input in
  let o1 = Array.length test_output.(0) in
  let error = Array.make_matrix n o1 0. in
  for i=0 to n-1 do
    let prediction = predict (
      norm_z_score (test_input.(i) mean_in std_in)
      network weights act_fun ) in
    let norm_pred = denorm_z_score (
      prediction mean_out std_out ) in
    for j=0 to o1-1 do
      let e_j = abs_float (
        norm_pred.(j) -. test_output.(i).(j)) in
      error.(i).(j) <- e_j;
    done;
  done;
  let e_m = mean error in
  e_m

```

(*n-krat požene evaluate na n različnih mrežah,
n krat nauči in pogleda napako*)
(* funkcija, ki napove vrednost pri testnem primeru
vhod:

- *train_input*: tabela vhodnih učnih vektorjev
- *train_output*: tabela vhodnih testnih vektorjev
- *test_input*: tabela izhodnih učnih vektorjev
- *test_output*: tabela izhodnih testnih vektorjev
- *network_topology*: tabela, ki opisuje topologijo mreže

- *act_fun*: funkcija aktivacije
- *act_der*: odvod funkcije aktivacije
- *rate*: stopnja učenja
- *n*: število ponovitev

izhod:

- *povprečna napaka med dobljeno in želeno vrednostjo*
*)

```
let analysis_error train_input train_output test_input
  test_output network_topology act_fun act_der rate n =
  let o1 = Array.length test_output.(0) in
  let e = Array.make_matrix n o1 0. in
  let mean_in = mean train_input in
  let mean_out = mean train_output in
  let std_in = std train_input mean_in in
  let std_out = std train_output mean_out in
  for i=0 to (n-1) do
    let trained_w = train_network train_input train_output
      network_topology rate 1.0 act_fun act_der in
    let error = evaluate test_input test_output trained_w
      network_topology act_fun
      mean_in std_in mean_out std_out in
    e.(i) <- error;
  done;
  let e_m = mean e in
  e_m

let analysis_unlearned train_input train_output test_input
  test_output network_topology act_fun act_der rate n =
  let o1 = Array.length test_output.(0) in
  let e = Array.make_matrix n o1 0. in
  let mean_in = mean train_input in
  let mean_out = mean train_output in
  let std_in = std train_input mean_in in
  let std_out = std train_output mean_out in
  for i=0 to (n-1) do
    let w = create_weights_matrix network_topology 1.0 in
    let error = evaluate test_input test_output w
      network_topology act_fun
      mean_in std_in mean_out std_out in
    e.(i) <- error;
  done;
  let e_m = mean e in
```

e_m

```
let learned_vs_unlearned train_input train_output test_input
  test_output network_topology act_fun act_der rate n =
  let o1 = Array.length test_output.(0) in
  let count = Array.make o1 0 in
  let mean_in = mean train_input in
  let mean_out = mean train_output in
  let std_in = std train_input mean_in in
  let std_out = std train_output mean_out in
  for i=0 to (n-1) do
    let w = create_weights_matrix network_topology 1.0 in
    let error_unlearned = evaluate test_input test_output
      w network_topology act_fun
      mean_in std_in mean_out std_out in
    let trained_w = train_with_input_weights
      train_input train_output network_topology
      rate w act_fun act_der in
    let error_learned = evaluate test_input test_output
      trained_w network_topology act_fun
      mean_in std_in mean_out std_out in
    for j=0 to (o1-1) do
      if (error_unlearned.(j) > error_learned.(j))
      then count.(j) <- count.(j) + 1
      else count.(j) <- count.(j) - 1
    done;
  done;
count

let bias test_input test_output weights network_topology
  act_fun mean_in std_in mean_out std_out =
  let network = initialize_network network_topology in
  let n = Array.length test_input in
  let o1 = Array.length test_output.(0) in
  let out = Array.make_matrix n o1 0. in
  for i=0 to n-1 do
    let prediction = predict (
      norm_z_score test_input.(i) mean_in std_in)
      network weights act_fun in
    let norm_pred = denorm_z_score prediction
      mean_out std_out in
    out.(i) <- norm_pred;
```

```

done;
let o_m = mean out in
let bias = Array.make_matrix n o1 0. in
for i=0 to n-1 do
  for j=0 to (o1-1) do
    bias.(i).(j) <- test_output.(i).(j) -. o_m.(j);
  done;
done;
bias

let variance test_input weights network_topology act_fun
mean_in std_in mean_out std_out =
let network = initialize_network network_topology in
let n = Array.length test_input in
let o1 = network_topology.(Array.length network_topology - 1) in
let out = Array.make_matrix n o1 0. in
for i=0 to n-1 do
  let prediction = predict
    (norm_z_score test_input.(i) mean_in std_in)
    network weights act_fun in
  let norm_pred = denorm_z_score prediction mean_out std_out in
  out.(i) <- norm_pred;
done;
let o_m = mean out in
let v = Array.make_matrix n o1 0. in
for i=0 to n-1 do
  for j=0 to (o1-1) do
    v.(i).(j) <- (out.(i).(j) -. o_m.(j))**2.;
  done;
done;
let var = mean v in
var

(*max_rate in min_rate so potence 10*)
let rate_analysis r_vect train_input train_output test_input
test_output network_topology act_fun act_der =
let o1 = Array.length test_output.(0) in
let mean_in = mean train_input in
let mean_out = mean train_output in
let std_in = std train_input mean_in in
let std_out = std train_output mean_out in
let n = Array.length r_vect in

```



```

let r = Array.make_matrix n o1 0. in
for r_i=0 to n-1 do
    let error = analysis_error train_input train_output
        test_input test_output network_topology
        act_fun act_der r_vect.(r_i) 50 in
    r.(r_i) <- error;
done;
r

let topology_analysis min_hidden n train_input train_output
test_input test_output act_fun act_der rate =
let o1 = Array.length test_output.(0) in
let mean_in = mean train_input in
let mean_out = mean train_output in
let std_in = std train_input mean_in in
let std_out = std train_output mean_out in
let t = Array.make_matrix n o1 0. in
for t_i=0 to n-1 do
    let error = analysis_error train_input train_output
        test_input test_output [|11; min_hidden + t_i;3|]
        act_fun act_der rate 50 in
    t.(t_i) <- error;
done;
t

```

SLOVAR STROKOVNIH IZRAZOV

backpropagation vzvratno razširjanje napake
bias (neuron) (nevron) konstante aktivacije
bias (value) pristranskost (v statističnem smislu)
black box "črna škatla" (nezmožnost razlaganja odločitev nevronske mreže)
feed-forward (neural network) usmerjena (nevronska mreža)
higher-order functions funkcije višjega reda
overfitting preprileganje
pure function čista funkcija
supervised learning nadzorovano učenje
step function stopničasta pragovna funkcija
undefitting neprileganje

LITERATURA

- [1] I. Kononenko in M. Kukar, *Machine Learning and Data Mining: Introduction to Principles and Algorithms*, Horwood Publishing Limited, UK, 2007

- [2] *Neural Networks*, [ogled 15. 4. 2020], dostopno na <https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html>.
- [3] *Understanding Activation Functions in Neural Networks*, v: Medium, [ogled 20. 4. 2020], dostopno na <https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>.
- [4] *Neural Network Bias: Bias Neuron, Overfitting and Underfitting*, [ogled 25. 4. 2020], dostopno na <https://missinglink.ai/guides/neural-network-concepts/neural-network-bias-bias-neuron-overfitting-underfitting/>.
- [5] S. Sathyanarayana, *A Gentle Introduction to Backpropagation*, v: Numeric Insight, Inc Whitepaper, 2014 [ogled 20. 4. 2020], dostopno na https://www.researchgate.net/publication/266396438_A_Gentle_Introduction_to_Backpropagation
- [6] H. Fanaee-T in J. Gama *Event labeling combining ensemble detectors and background knowledge*, v: Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg, [ogled 15. 3. 2020], dostopno na <https://archive.ics.uci.edu/ml/datasets/bike+sharing+dataset>.
- [7] <https://ocaml.org/learn/description.html>, [ogled 5. 5. 2020], dostopno na <https://ocaml.org/learn/description.html>.