UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Laura Guzelj Blatnik

Nevronske mreže z vzvratnim razširjanjem napak v funkcijskem programskem jeziku

Delo diplomskega seminarja

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

Ljubljana, 2020

Kazalo

Nevronske mreže z vzvratnim razširjanjem napak v funkcijskem programskem jeziku

Povzetek

Diplomsko delo se poglobi v usmerjene nevronske mreže. Le te temeljijo na posnemanju možganskih funkcij, uporabljajo pa se za napovedovanje in klasifikacijo. Sestavljajo jih nevroni organizirani v sloje. Nevroni so med sabo povezani s sinapsami. S pomočjo algoritma za vzvratno razširjanje napake in učnih primerov mrežo naučimo odzivanja na neznane situacije. Algoritem temelji na spreminjanju uteži na sinapsah, učenje pa poteka dokler ni razlika med želeno in izračunano vrednostjo dovolj majhna. Poleg nevronskih mrež se delo osredotoči tudi na funkcijsko programiranje s poudarkom na programskem jeziku OCaml. Primer nevronske mreže z vzvratnim razširjanjem napak je tudi implementiran v programskem jeziku OCaml. Delovanje mreže je prikazano na konkretnem primeru, kjer je ovrednotena napaka mreže na izbrani podatkovni množici.

Feed-forward neural networks with backpropragtion in a functional programming language

Abstract

This paper focuses on feed-forward neural networks. Mimicking brain functions, neural networks are used for prediction and classification. Neural networks are composed of neurons organized in layers and connected by synapses. Utilizing the backpropagation algorithm and learning examples the network is able to learn how to respond to unknown situations. The idea behind backpropagation is to change weights on the synapses until the difference between computed and desired values is small enough. In addition, the work presents functional programming and the OCaml functional programming language. An implementation of a neural network with backpropagation in the OCaml programming language is presented at the end of the work. The work concludes with an application of the neural network to a real-world example, where the error of the network is evaluated on a specific data set.

Math. Subj. Class. (2010): 68T05, 68N18

Ključne besede: usmerjene nevronske mreže, vzvratno razširjanje napake, perceptron, umetna inteligenca, strojno učenje, funkcijski programski jezik OCaml

Keywords: feed-forward neural network, backpropagation, perceptron, artificial intelligence, machine learning, functional programming language OCaml

1. Uvod

Človeški možgani so kompleksen organ predvsem zaradi neštetih 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 sprememe in prepoznavati neznane situacije. Tako je 1949 Donald Hebb v svojem delu The Organization of Behavior [8] prvič predstavil idejo o umetnih nevronskih mrežah, kjer se je zgledoval predvsem po možganih.

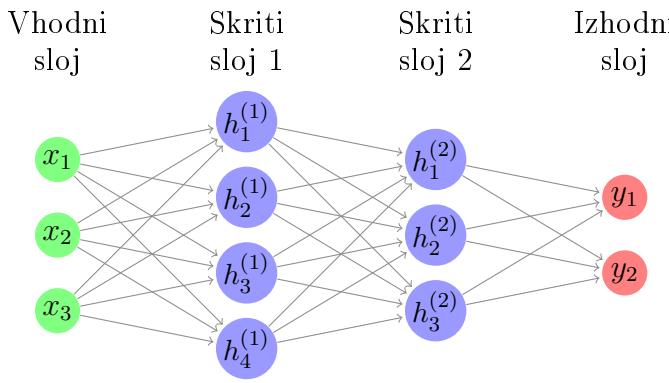
Dvoslojni perceptorn, kot prva nevrosnka mreža zmožna učenja, se je rodil nekoliko kasneje, leta 1962 je ta pojem prvi omenil Frank Rosenblatt v Principles of Neurodynamics [9]. Z razvojem strojne opreme se je zanimanje za nevronske mreže zopet povečalo v osemdesetih letih prejšnjega stoletja. Leta 1986 je več raziskovalcev neodvisno drug od drugega vpeljalo teorijo o večslojnih nevronskih 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 so v nekaterih svojih zmožnostih primerljive z zmožnostmi možganov. Kljub vsemu pa so možgani sposobni masrcičesa, česar računalniki trenutno ne zmorejo.

Diplomska naloga se poglobi v usmerjene nevronske mreže in njihovo učenje 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 preprostih operacij veliko hitrejše od možganov. Kot bomo videli v nadaljevanju, je osnovna ideja umetnih nevronskih mrež zelo podobna živčevju v možganih. Lahko bi rekli, da so umetne nevronske mreže zgolj abstraktna poenosatvitev delovanja možganov. V nadaljnjem besedilu se izraz nevronske mreže nanaša na umetne nevronske mreže, besedo "umetne" bom zaradi kratkosti izpuščala.

2.2. **Struktura nevronskih mrež.** Nevronske mreže so sestavljene iz nevronov, ki matematično gledano niso nič drugega kot funkcije. Nevroni so med seboj



SLIKA 1. Perceptron z dvema skritima slojema nevronov

povezani s sinapsami, vsaka sinapsa ima svojo težo oziroma utež. Nevroni so organizirani v slojih. Nevroni v istem sloju med seboj niso povezani, vsak nevron je povezan z vsemi nevroni v naslednjem (oziroma prejšnjem sloju). Nevronska mreža vsebuje vhodni sloj nevronov, ti nevroni predstavljajo vrednosti vhodnih podatkov, in pa izhodni sloj nevronov, ki predstavlja izhodne vrednosti dobljene z nevronsko mrežo. V strojnem učenju se večinoma uporabljajo usmerjene nevrosnke mreže, kjer izračun vrednosti poteka od vhodnega sloja proti izhodnemu. Usmerjene nevronske mreže ne vsebujejo povratnih povezav, prav tako sinapse ne smejo tvoriti cikla.

Nevronska mreža lahko vsebuje le vhodni in izhodni sloj – takrat govorimo o dvoslojni nevronski 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 smiselno omeniti, da obstajajo tudi mreže z dvosmernimi povezavami. Usmerjeni nevronski mreži rečemo perceptron, glede na število slojev ločimo dvoslojni, tj. perceptron z vhodnim in izhodnim slojem, in večslojni perceptron.

2.3. **Delovanje perceptrona.** Vzemimo dvoslojni perceptron z m nevroni v vhodnem sloju, njihova stanja in izhode[1] označimo z $X_1, X_2, \ldots, X_m$ in n nevroni v izhodnem sloju, označimo jih $Y_1, Y_2, \ldots, Y_n$. Običajno sloj nevronov zapišemo v obliki vektorja:

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

[1]V vhodnem sloju so običajno izhodi nevronov enaki njihovim stanjem.

Uteži med nevroni predstavimo z matriko:

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

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:* Funkcijo kombinacije uporabljamo za izračun stanja i-tega nevrona na sledeč način:

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

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

$$A = WX,$$

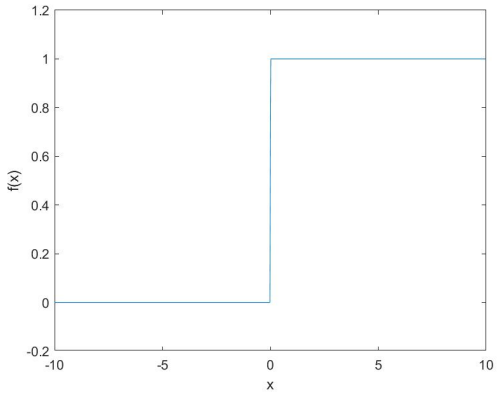
kjer je W matrika uteži, X pa vhodni vektor. Če upoštevamo še konstante aktivacije, je potrebno matriko uteži W in vektor X malo preoblikovati. Vektorju na konec dodamo še eno komponento z vrednostjo 1 in dobimo $X = (X_1, X_2, \ldots, X_m, 1)^\intercal$ matriki uteži pa dodamo še stolpec s konstantami 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, dobljeno s funkcijo kombinacije. Glede na namen uporabe nevronske meže lahko uporabimo različne funkcije aktivacije. Najpogosteje se uporabljaljata 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 vzvratnega razširjanja napake. 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 vzvratnem razširjanju napake. Odvisno na kateri interval želimo normalizirati izhodne vrednosti, lahko uporabimo različne sigmoidne funkcije. Primera sigmoidne funkcije sta:

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

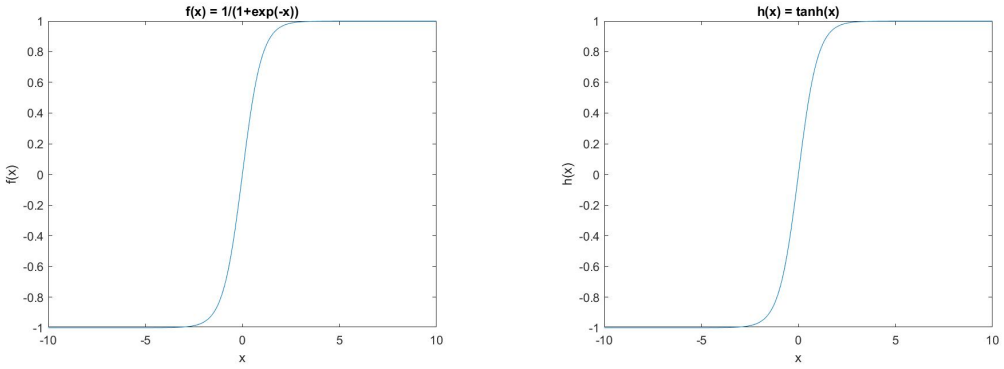
Kot je razvidno z grafov (slika 3) je na intervalu $(-2, 2)$ naklon obeh funkcij velik, zato na omenjenem intervalu mreža pri napovedovanju da jasne rezultate in učenje poteka hitro. Po drugi strani pa je zunaj tega intervala odvod majhen, tako je učenje mreže in posledično napovedovanje zelo počasno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

Kot bomo videli v nadaljevanju, pri učenju mreže z vzvratnim razširjanjem napake potrebujemo tudi odvod funkcije aktivacije, saj le ta vpliva na posodabljanje uteži. Odvod uporabimo na že izračunanih stanjih nevronov, pri zgornjih dveh sigmoidnih funkcijah lahko preprosto izrazimo z $f(x)$ oziroma $h(x)$ na naslednji način:

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

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

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



Slika 3. Sigmoidni funkciji aktivacije

2.3.3. *Konstanta aktivacije.* Sloju nevronov pogosto dodamo še en nevron - konstanto aktivacije. To je nevron z izhodom 1, ki posledično postane vhod za vse nevrone v naslednjem sloju. Naloga konstante aktivacije 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.

Pomen konstante aktivacije najlažje predstavimo s pomočjo linearne funkcije. Z enačbo oblike $y = kx$ lahko zapišemo le premice, ki prečkajo koordinatno izhodišče. Če pa fuknkciji 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.* Vrednost izhodnega sloja 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:

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

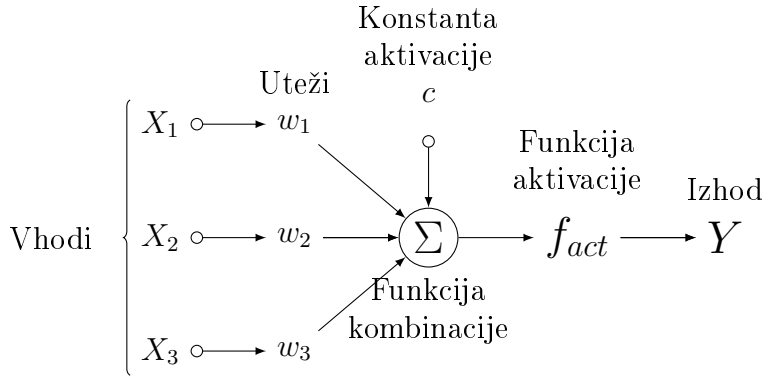
oziroma v vektorski obliki za cel sloj izhodnih nevronov:

$$Y = f(WX),$$

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

2.4. **Lastnosti nevronskih mrež.** 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 (znotraj sloja), v praksi pa to pomeni zelo hitre operacije. Zato lahko nevronske mreže uporabimo pri reševanju kompleksnih problemov.



SLIKA 4. Funkcija nevrona (vhodi X_1, X_2, X_3 predstavljajo izhode nevronov iz prejšnjega sloja).

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 priemrih 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 model)*. 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 nevronsko mrežo 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 najpomembenjša 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 prozvodnji,
- 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 izbrano mero tako, da bi se zastavljenemu cilju čimbolj približali.

Obstaja kar nekaj različnih pravil, kako umetno nevronsko mrežo 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, tj., razliko med želenim in dejanskim izhodom mreže.

Potrebno je omeniti še, da poleg nadzorovanega učenja poznamo tudi nenazdrovano 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 naključno 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:

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

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. Zaporedje takih popravkov nas namreč pripelje do lokalnega minimuma napake. Odvod napake po utežeh zapišemo sledeče:

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

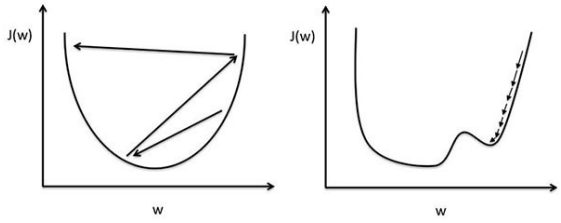
Sedaj lahko zapišemo enačbo za pravilo delta in nove vrednosti uteži W^N:

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

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

Zgoraj omenjeno *gradientno iskanje* v resnici 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. η poleg hitrosti konvergence vpliva še na robustnost na lokalne minumume - če je η premajhen se lahko zgodi, da gradiento iskanje najde le najbližji lokalni minimum, ki pa morda ni optimalen. V nasprotnem primeru je prevelik η preveč nestabilen, posledično lahko minimume povsem zgrešimo. Pomen izbire stopnje učenja je prikazan na sliki 5.

3.3. **Vzvratno razširjanje napake.** Vzvratno 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 – vzvratno razširjanje napake oz. *backpropragation* v angleščini).



SLIKA 5. Vpliv prevelike oziroma premajhne stopnje učenja na iskanje lokalnega minimuma, vir [10]

3.3.1. *Notacija.* Definirajmo notacijo, ki jo bomo uporabili pri izpeljavi vzvratnega razširjavnja napake.

- X_i označuje vrednost oz. izhod i-tega nevron v vhodnem sloju.
- H_{ki} označuje vrednost oz. izhod i-tega nevrona v k-tem skritem sloju.
- Y_i označuje vrednost oz. izhod i-tega nevrona 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 vzvratnega razširjanja napake.* Vzemimo splošen večsojen 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 vsabuje 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 skritem 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 sktiti 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 izhode 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:

$$e_i(l) = d_i(l) - Y_i(l), \tag{6}$$

oziroma v vektorski obliki:

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

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

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

Napako želimo minimalizirati in podobno kot 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 enostavo 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 pomgali z verižnim pravilom.

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

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

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

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

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

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

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

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

$$\frac{\partial A_i(l)}{\partial W_{ij}^{(m+1)}} = \frac{\partial}{\partial W_{ij}^{(m+1)}} \left(\sum_{j=1}^{N_m} W_{ij}^{(m+1)} H_{mj}(l) \right) = H_{mj}(l) \tag{12}$$

Pri tem velja, da je f' odvod funkcije aktivacije.

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:

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

Označimo:

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

kjer se $\Delta^{(m+1)}$ navezuje na stanja nevronov v izhodnem sloju. Združimo vse zgornje rezultate in zapišemo:

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

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

Vzvratno razširjanje napake pride do izraza šele pri skritih slohjih. 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:

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

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:

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

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:

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

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:

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

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 nevronskih mrež, 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 vpliva na to 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 do neke mere vplivamo na hitrost učenja mreže.

Včasih se lahko zgodi, da napaka mreže med učenjem začne naraščati. 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 izven množice učnih primerov pa ni natančna. Zato je pomembno, da med učenjem ves čas spremljamo vrednost napake na primerih iz posebne množice testnih primerov, ki jih ne uporabljamo za učenje.

3.5. **Pristranskost in varianca.** Po končanem učenju lahko izračunamo pristranskost *(ang. bias)* in varianco nevronske mreže. Ti vrednosti 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 $\mathrm{E}[\hat{t}]$ označimo pričakovano vrednost napovedi, to je povprečna vrednost napovedi modelov naučenih iz različnih učnih množic. Pristranskost mreže pri t-tem učnem primeru označimo z $N(t)$ in izračunamo sledeče:

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

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 učnih primerih in primerih, ki so učnim podobni, dobra. Če je pristranskost prevelika jo zmanjšamo tako, da izberemo večjo nevronsko mrežo.

Nadalje definiramo še varianco

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

Kot je razvidno iz zgornje enačbe, $Var(\hat{t})$ ni odvisna od točnih vrednosti t. Varianca meri občutljivost napovedi na spremembe učne množice. Stremimo k temu, da bi bila varianca čim manjša. Če je le ta prevelika, jo zmanjšamo tako, da povečamo število učnih primerov, vendar to ni vedno mogoče. Če je varianca še vedno velika, lahko poskusimo tudi z drugačno topologijo mreže.

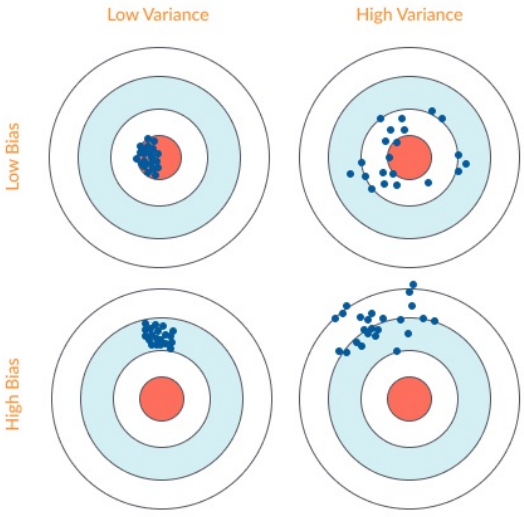
Spomnimo se, kako se glasi napaka, ki jo pri posplošenem pravilu delta želimo minimalizirati. Če enačbo napake (enačba (7)) prepišemo v smislu novo označnih količin, dobimo:

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

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

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

Iz zgornje enačbe vidimo, da sta pristranskost in varianca mreže tesno povezani med seboj in da 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.



Slika 6. Pristranskost *(ang. bias)* in varianca *(ang. variance)*, vir [4]

Če bomo izbrali kompleksno topologijo nevronske mreže (torej veliko število skritih plasti in nevronov - globoko mrežo) bo pristranskost majhna, a zaradi občutljivosti velikega števila uteži na male spremembe učne množice je varianca lahko zelo visoka. Zato v takem primeru obstaja nevarnost preprileganja. To pomeni, da bo mreža zelo dobro delovala na učnih podatkih, ne bo pa sposobna znanja posplošiti na poljubne vhodne podatke. Po drugi strani pri izboru enostavne topologije (torej topologije z malim številom skritih plasti in nevronov) opazujemo visoko pristranskost, a hkrati nizko varianco – v takih situacijah obstaja resna možnost neprileganja (ali podprileganja, *ang. underfitting*). Taka mreža ne bo sposobna dobro aproksimirati niti učnih podatkov, na dobro napoved 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 oziroma izboljšamo njihovo kakovost. Načeloma velja, da na več učnih primerih kot bomo mrežo učili, 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 jo včasih imenujemo tudi standarizacija. Kako bomo podatke normalizirali je odvisno od primera, ki ga obravnavamo.

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

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{22}$$

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

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

$$X_{norm} = \frac{X - \mu}{\sigma}, \tag{23}$$

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

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, drugič izberemo jezik v katerem bo zastavljeni problem najlažje rešljiv. Sama sem se za OCaml odločila, ker so implementacije nevronskih mrež 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.

Algoritem 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
```

Čiste funkcije so pomemben koncept v funkcijskem programiranju. Funkcija je *čista (ang. pure 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, preden izvajanjem programa prevajalnik kodo preveri, zato do sintaktičnih 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 nevronskih mrež sem to lastnost pri implementaciji izkoristila kar nekajkrat.

Dobra lastnost OCamla 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 zaželjeno.

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 vzvratnim razširjanjem napake. Nadalje sem nevronsko mrežo tudi testirala na dani podatkovni množici in jo analizirala. Celotna implementacija se nahaja na koncu naloge, v prilogi 6, prav tako pa je dostopna tudi na povezavi `https://github.com/lauraguzeljblatnik/nevronske-mreze`.

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

- `input_array`: tabela vhodnih vektorjev,
- `output_array`: tabela pripadajočih želenih 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.

Signatura funkcije se glasi:

Algoritem 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>
```

Nadalje s pomočjo funkcije `predict` in naučenih uteži napovemo vrednosti za poljubne vhodne podatke. Le ta sprejme tabelo `input`, ki predstavlja vektor vhodnih podatkov, matriko `network`, ki predstavlja nevrone v nrevronski mreži, matriko matrik `weights`, ki predstavlja uteži med posameznimi sloji in funkcijo aktivacije `act_fun`, ki jo želomo uporabiti. Funkcija vrne tabelo, ki predstavlja vekor napovedanih vrednosti. Signatura je tako:

Algoritem 3 Signatura funkcije `predict`

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

5.2. **Vrednotenje na podatkovni množici.** Delovanje nevronske mreže sem preverila na podatkovni množici, ki sem jih pridobila s spletnega naslova `https://archive.ics.uci.edu/ml/datasets/bike+sharing+dataset`.

Vse funkcije, ki sem jih uporabila pri analizi delovanja nevronske mreže so vključene v nalogo in se nahajajo v prilogi 6.

5.2.1. *Podatkovna množica o izposoji koles.* Podatkovna množica vsebuje informacije o številu izposojenih koles za posamezen dan med letoma 2011 in 2012 v sistemu Capital Bikeshare v mestu Washington D. C. Število izposojenih koles je odvisno od naslednjih vhodnih podatkov:

- letni čas: 1 - zima, 2 - pomlad, 3 - poletje, 4 - jesen
- leto: 0 - leto 2011, 1 - leto 2012
- mesec: 1 do 12
- praznik: 1, če je bil dan praznik, 0 sicer
- dan v tednu: 0 do 6, 0 predstavlja nedeljo
- delovni dan: 1, če je dan delovni, 0 če je praznik ali vikend
- vreme: 1 - jasno, posamezni oblaki, delno oblačno, 2 - meglice, posamezni oblaki, oblačno, 3 - rahel sneg, rahel dež z razpršenimi oblaki, rahel dež v nevihtami, 4 - nalivi, toča, nevihte, sneg, megla
- temperatura: vrednosti so podane v stopinjah celzija in normalizirane sledeče: $t = {}^{(t-t_{min})}/_{(t_{max}-t_{min})}$, kjer je $t_{min} = -8$, $t_{max} = +39$
- občutek mraza: vrednosti so podane v stopinjah celzija in normalizirane sledeče: $t = {}^{(t-t_{min})}/_{(t_{max}-t_{min})}$, kjer je $t_{min} = -16$, $t_{max} = +50$
- vlažnost: normalizirana vrednost vlažnosti, vrednosti so deljene s 100, kar je maksimalna vrednost
- veter: normalizirana hitrost vetra, vrednosti so deljene s 67, kar je maksimalna vrednost

Uporabniki koles so razdeljeni na občasne in registrirane uporabnike, tako podatkovni nabor vsebuje informacije o številu obeh skupin in njuni vsoti. Izhodni podatki so torej:

- število občasnih uporabnikov,
- število registriranih uporanikov,
- število vseh uporabnikov (občasnih in registriranih).

Nabor podatkov vsebuje informacije o izposoji koles za 731 dni, od tega sem prvih 600 primerov uporabila za učenje, na 131 primerih pa sem preverjala delovanje nevronske mreže.

5.2.2. *Analiza delovanja nevronske mreže.* Delovanje mreže sem preverjala s pomočjo povprečne absolutne nakake. Namesto kvadrata napake sem izbrala absolutno napako. Absolutna napaka se meri na isti skali kot vrednosti same, posledično se jo lažje interpretira. Za učenje mreže pa absolutna napaka ni primerna, ker nima gladkega odvoda in se je ne da lepo optimizirati, zato tam uporabimo kvadrat napake. Povprečno absolutno napako sem izračunala tako, da sem za vsak testni primer izračunala napako med želeno vrednostjo in vrednostjo napovedano s pomočjo nevronske mreže. Formula za izračun napake močno spominja na enačbo (4), za l-ti učni primer se enačba glasi:

$$E(l) = |d(l) - Y(l)|.$$

Nato sem vzela povprečje teh napak za vse testne primere in tako dobila povprečno absolutno napako sledeče:

$$E_{povp}(l) = \frac{1}{n} \sum_{i=1}^{n} |d(l) - Y(l)|, \tag{24}$$

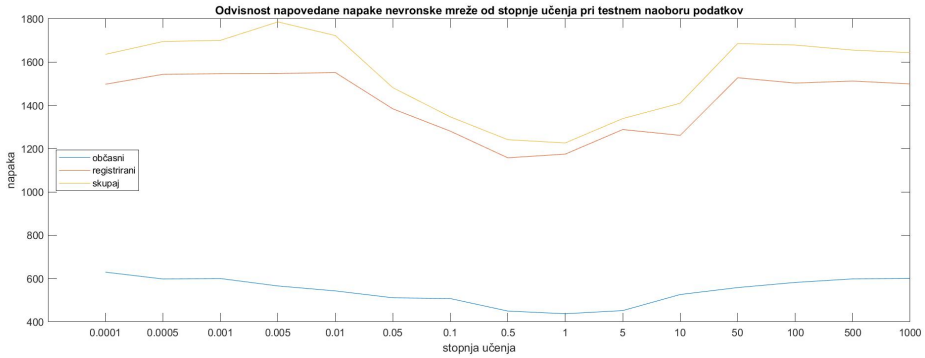
kjer n predstavlja število testnih primerov.

Da bi bili izračuni kar se da robustni glede na naključno izbrano začetno stanje, sem učenje mreže ponovila na petdesetih različnih začetnih utežeh in na koncu izračunala povprečje vseh napak. Napako sem računala posebej za vsakega izmed izhodnih parametrov.

S pomočjo funkcije `learned_vs_unlearned` pokažemo uspešnost učenja nevronske mreže. Funkcija za n primerov primerja povprečno absolutno napako za naučeno in nenaučeno mrežo. Če je napaka pri naučeni mreži manjša, v tabelo na mesto ustreznega izhodnega parametra prištejemo vrednost 1, če je napaka manjša pri nenaučeni mreži vrednost 1 odštejemo.

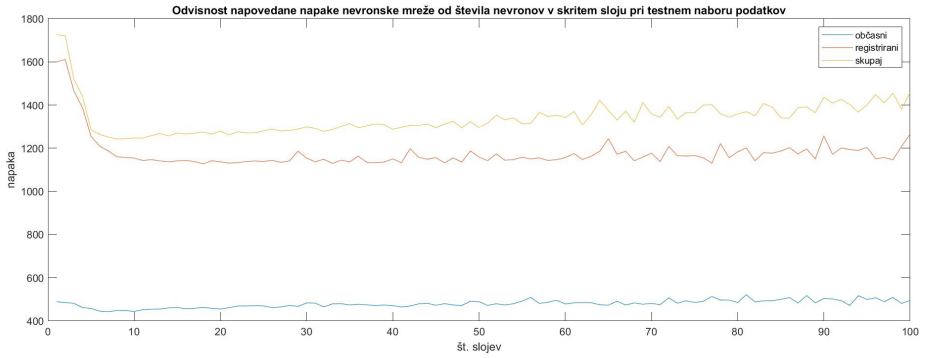
Če funkcijo poženemo na 50 različnih začetnih utežeh pri topologiji z enim skritim slojem z desetimi nevroni in stopnjo učenja 0.5, funkcija vrne naslednjo tabelo: `val count : int array = [|50; 50; 50|]`. To pomeni, da je bilo učenje uspešno, saj je bila napaka pri napovedi z naučenimi utežmi vsakokrat manjša kot napaka pri napovedi z nenaučenimi utežmi.

Če malo spremenimo topologijo in stopnjo učenja in za topologijo vzamemo dva skrita sloja z desetimi nevroni, za stopnjo učenja pa izberemo vrednost 0.1 opazimo, da mreža ne deluje več optimalno. Funkcija `learned_vs_unlearned` vrne tabelo `val count : int array = [|18; 32; 32|]`, kar pomeni da naučena in nenaučena mreža delujeta skoraj enako dobro. Ta primer prikazuje, kako pomembno je, kakšno topologijo in stopnjo učenja izberemo.



Slika 7. Vpliv stopnje učenja na učenje nevronske mreže.

5.2.3. *Analiza vpliva parametrov pri učenju nevronske mreže.* Zanimalo me je, kako stopnja učenja vpliva na delovanje nevronske mreže. Pri fiksni topologiji z eno skrito plastjo z desetimi nevroni sem spreminjala stopnjo učenja. Iz grafa na sliki 7, ki prikazuje povprečno absolutno napako pri petdesetih ponovitvah učenja v odvisnosti od stopnje učenja, je razvidno, da je napaka najmanjša, pri vrednostih $\eta = 0.5$ in $\eta = 1$. Kar je, glede na vpliv stopnje učenja pri gradientnem iskanju, povsem pričakovano.

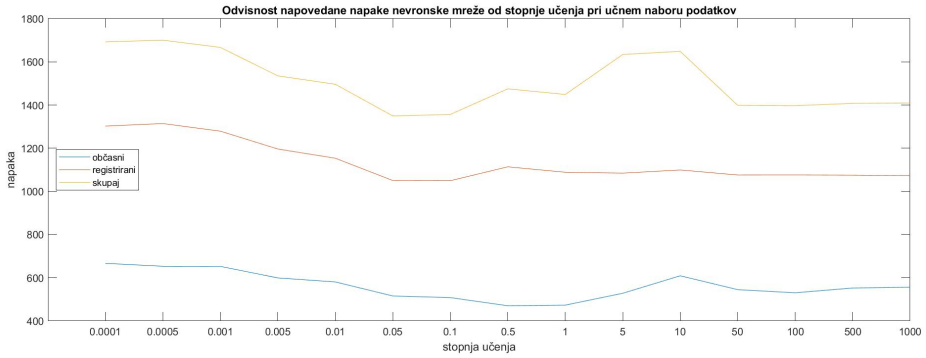


Slika 8. Vpliv števila nevronov v enem skritem sloju na učenje nevronske mreže.

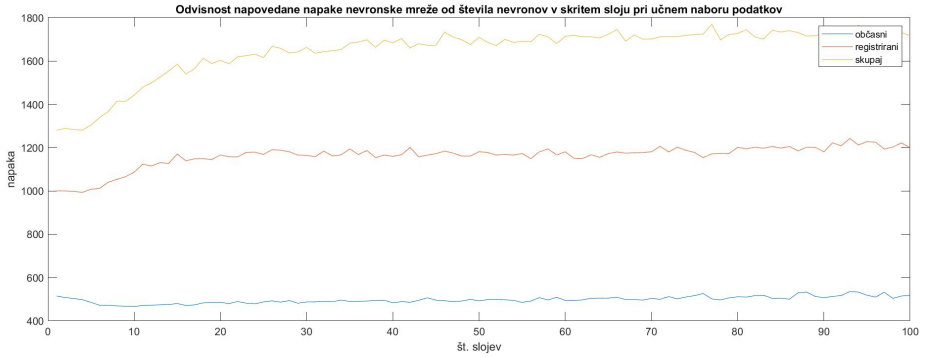
Nadalje sem analizirala, kakšna bi bila najbolj optimalna topologija za delovanje nevronske mreže. To sem testirala tako, da sem pri fiksni stopnji učenja $\eta = 0.5$ spreminjala število skritih nevronov. Ostala pa sem le pri enem skritem sloju, saj gre za relativno majhen podatkovni nabor in bi bila topologija z več skritimi sloji že prekompleksna, kar se kaže s slabim delovanjem nevronske mreže in posledično veliko napako pri napovedovanju. Graf na sliki 8 prikazuje povprečno absolutno napako pri petdesetih ponovitvah učenja v odvisnosti števila skritih nevronov. Iz

grafa je razvidno, da je napaka najmanjša pri topologiji z med pet in deset nevroni v skritem sloju.

Namesto na testni podatkovni množici lahko napako izračunamo na učni podatkovni množici, tako dobimo učno napako. Graf na sliki 9 prikazuje vpliv stopnje učenja na učno napako, medtem ko graf na sliki 10 prikazuje vpliv števila skritih nevronov. Če omenjena grafa primerjamo z grafoma na slikah 7 in 8, ugotovimo, da je pri manj kot sedmih nevronih v skritem sloju učna napaka manjhna, testna pa velika. Tako v omenjenem primeru pride do preprileganja. Podobno do preprileganja pride, v kolikor za parameter η izberemo vrednosti večje od 50. Do podprilegnja pride, ko je že učna napaka velika, to se zgodi pri topologijah z več kot 15 nevroni v skritem sloju. Zaključimo lahko, da bo mreža najbolje delovala,



Slika 9. Vpliv stopnje učenja na učenje nevronske mreže.



Slika 10. Vpliv števila nevronov v enem skritem sloju na učenje nevronske mreže.

če bo skriti sloj vseboval okoli 10 nevronov, za stopnjo učenja pa bomo izbrali vrednost med 0.5 in 0.1.

6. Priloga

```
(* NEVRONSKA MREŽA *)

(*pomožna funkcija:
        aktivacijska funkcija za posamezen nevron - sigmoida *)
let sigmoid neuron =
        1. /. (1. +. exp(-. neuron))

(*pomožna funkcija:
        odvod sigmoide za posamezen izhodni nevron, sprejme output -
        to kar mreža izračuna. Odvod sigmoide je f'(x) = f(x)*(1-f(x)),
        funkcijo x -> x(1-x) uporabimo na vrednostih f(x) *)
let d_sigmoid output =
        (output)*.( 1. -. output)

(*pomožna funkcija:
        aktivacijska funkcija za sloj(vektor), funct je funkcija
        aktivacije, ki jo želimo na uporabiti *)
let activation_layer funct layer =
        Array.map funct layer

(*pomožna funkcija:
        funkcija kombinacije za sloj in pripadajoče uteži,
        zmnoži vektor (sloj nevronov) 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
        )

(*pomožna funkcija, ki ustvari float matriko z naključnimi
        utežmi velikosti nxm,
vhod:
- n: št. nevronov v tem sloju
- m: št. nevronov v prejšnjem sloju
- bound: vrednosti v matriki so med bound in (-bound)
izhod:
```

```
- matrika uteži velikosti mxn *)
let rand_weights n m bound =
        Array.init n
        (fun _
                -> Array.init m (fun _
                        -> (Random.float 2.*.bound)-.bound))

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

(*pomožna funkcija, ki ustvari tabelo matrik naključnih
uteži s pravimi velikostmi
vhod:
- network_topology: vektor, ki opisuje topologijo
        nevronske mreže - int array,
- bound: zgornja meja uteži - float
izhod: tabela matrik naključnih uteži prave velikosti*)
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

(*funkcija, ki ustvari float matriko, ki predstavlja nevrone v mreži
vhod:
- network_topology: tabela, ki opisuje topologijo mreže
izhod:
- float matrika velikosti mxn, kjer je m št. slojev in n max
št. nevronov v sloju, na začetku so vse vrednosti v matriki 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*)

(*funkcija, ki izračuna vektor delta za izhodni sloj
vhod:
- d: tabela z želenimi izhodi
- y: tabela z izhodnimi vrenostmi mreže
- act_der: odvod funkcije aktivacije
izhod:
- tabela vrednosti delta *)
```

```
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
        )

(*funkcija, ki izračuna vektor delta za skrite sloje
vhod:
- w: matrika uteži ene sloj naprej(že popravljene!)
- h_m: tabela izhodov v v sloju kjer računamo delto
- d_out: tabela delta v naslednjem sloju
izhod:
- tabela vrednosti delta *)
let delta_hidden w h_m d_out act_der =
        let a = Array.length h_m and
                d_0 = 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

(*funkcija, ki ustrezno popravi vrednosti v matriki uteži
vhod:
- w: matrika uteži, ki jo spreminjamo
- rate: stopnja učenja
- delta: ustrezen tabela napake dE/dA
- input: tabela vhodnih podatkov/vrednosti
izhod:
- ustrezno 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 s 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:
- data: matrika s podatki
- mean: povprečje podatkov
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


(*funkcija, ki normalizira podatke z z-score normalizacijo
vhod:
- a: tabela z vektorji podatki
- mean: vekotr povprečja podatkov
- std_dev: vektor standardnega odklona podatkov
izhod: normalizirana tabela vektorjev podatkov
*)
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

(*funkcija, ki denormalizira podatke z z-score normalizacijo
vhod:
- a: tabela z vektorji podatkov
- mean: vekotr povprečja podatkov
- std_dev: vektor standardnega odklona podatkov
izhod: denormalizirana tabela vektorjev podatkov
*)
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
vhod:
- x: vhodni vektor
- d: željen izhodni vektor
- network: matrika nevronov v mreži,
        velikosti (št. slojev) x (max nevronov v sloju),
        vsi el. so 0.
- weights: tabela matrik uteži v mreži
- rate: stopnja učenja
- act_fun: funkcija aktivacije
- act_der: odvod funkcije aktivacije
izhod: popravljene uteži*)
let learning_example x d network weights rate act_fun act_der =
        let n = Array.length network in
                network.(0) <- x;
        (*forward propragation*)
           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 želenih izhodnih vektorjev
 - network_topology: tabela s št. nevronov 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
na podanih začetnih utežeh
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


(*funkcija, ki n-krat požene evaluate na n
različnih mrežah, n krat nauči in pogleda napako
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

(*funkcija, ki n-krat požene evaluate na n
različnih mrežah, mreže ne nauči, le izračuna napako
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
pri nenaučenih utežeh
 *)
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

(*funkcija, ki prešteje, kolikorat je bilo
učenje z naučeno mrežo boljše (napaka je bila manjša),
kot učenje z nenaučeno mrežo
 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:
- tabela s številom uspešnih učenj po izhodnih
        parametrih
 *)
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


(*funkcija, ki za različne stopnje učenja izračuna
```

```
povprečno absolutno napako pri fiksni topologiji
vhod:
- r_vect: tabela vrednosti stopenj učenja
- 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
izhod:
- povprečna absolutna napaka za vsako stopnjo
        učenja iz r_vect
 *)
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

(*funkcija, ki za različna števila nevronov v skritem
sloju izračuna povprečno absolutno napako
pri fiksni stopnji učenja
vhod:
- min_hidden: najmanjšte št. sktirih nevronov
- n: število za kolikor želimo povečati min_hidden
- train_input: tabela vhodnih učnih vektorjev
- train_output: tabela vhodnih testnih vektorjev
- test_input: tabela izhodnih učnih vektorjev
- test_output: tabela izhodnih testnih vektorjev
- act_fun: funkcija aktivacije
- act_der: odvod funkcije aktivacije
- rate: stopnja učenja
izhod:
- povprečna absolutna napaka za vsako topologijo
        z od min_hidden do n nevroni v skritem sloju
 *)
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 i1 = Array.length test_input.(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 [|i1; min_hidden + t_i;o1|]
                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 model model, katerega delovanje težko razložimo
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
undeffitting 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`.

[8] D. O. Hebb, *The organization of behavior: A neuropsychological theory*, Wiley, New York, 1949.

[9] F. Rosenblatt, *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*, Spartan Books, Washingotn, D.C., 1962.

[10] *Learning rate gradient descent*, [ogled 12. 6. 2020], dostopno na `https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/learning-rate-gradient-descent/`.