Korištenje web tehnologija za izradu i prikaz višeslojnih neuronskih mreža

Karačić, Vedran

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

DIPLOMSKI RAD

KORIŠTENJE WEB TEHNOLOGIJA ZA IZRADU I PRIKAZ VIŠESLOJNIH NEURONSKIH MREŽA

Vedran Karačić

Split, kolovoz 2016.

Prazna stranica

Sadržaj

Uvod

Neuronske mreže postaju sve veći dio svakodnevnog života, od filtriranja nepoželjne e-pošte [1] do uvježbavanja autonomnih automobila [2], jer mogu rješavati sve brže rastući broj složenih problema, no time se povećava i složenost njihove implementacije. Još od početka strojnog učenja isključivo su istraživači implementirali neuronske mreže zbog njihove složenosti. Nažalost još uvijek je za nestručne osobe složeno stvarati i koristiti neuronske mreže za svoje potrebe te ih je zbog toga potrebno učiniti pristupačnijima. Internet je postao platforma dostupna gotovo svima na svijetu, a sve veći broj programskih rješenja se prebacuje u oblik pogodan za izvođenje na internetu u obliku web aplikacija[1], pa je to logičan smjer i za neuronske mreže. Zbog potrebe za brzim i efikasnim izvođenjem aplikacija, istraživači uglavnom koriste programske jezike kao što su C++ i Java [3] koji kompajliraju programe neuronskih mreža u strojni jezik zbog čega dulje vrijeme internet pretraživači nisu mogli parirati takvim implementacijama po pitanju efikasnosti izvođenja. No web tehnologije su se razvile, a JavaScript programski jezik je postao jedan od najpopularnijih programskih jezika[2] na svijetu te postaje sve efikasniji što čini razliku između performansi web tehnologija i kompajlerskih jezika manjom. Navedeno je dalo povoda za razvoj programskih rješenja kao što su *ConvNetJS*, *Brain.js* i *SynapticJS*, koji su svi razvijeni u JavaScript programskom jeziku i koji omogućuju stvaranje, uvježbavanje i korištenje različitih arhitektura neuronskih mreža.

Uz to što bi približavanje neuronskim mreža što većem broju ljudi u vidu web aplikacija povećalo njihovu upotrebu i popularnost, omogućilo bi stvaranje različitih vizualizacija što bi poboljšalo samu edukaciju ljudi o strojnom učenju i smanjilo složenost neuronskih mreža, time otvarajući vrata još većoj mogućnosti popularizacije ovog područja čije mogućnosti primjene eksponencijalno raste.

[1] Objašnjeno činjenicom da u IEEE-vom popisu najpopularnijih jezika 2016, od prvih deset samo su tri koja se ne koriste za izradu web aplikacija.

2 http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

1. Općenito o neuronskim mrežama

Umjetna neuronska mreža je koncept koji je nadahnut načinom na koji funkcionira ljudski mozak. Prvi model neuronske mreže je stvoren 1943. od strane McCulloch i Pitts-a u kojem su dokazali da jednostavni tipovi neuronskih mreža mogu izračunati bilo koju aritmetičku ili logičku funkciju [4]. Njihov rad se smatra začetkom neuro-računalstva. Umjetna neuronska mreža se sastoji od neurona i veza između njih, slično kao što se ljudski mozak sastoji od neuron i sinapsi između njih. Kao i kod ljudskog mozga, neuroni se aktiviraju kada dosegnu vrijednost određenog praga. Postoje mnoge različite arhitekture umjetnih neuronskih mreža specijaliziranih za različite zadatke kao što su automatsko prevođenje, prepoznavanje uzoraka, prepoznavanje slika itd. Njihova primjena drastično raste zadnjih godina, od velikih internet tražilica, preko različitih primjena u ekonomiji do analiziranja ljudskog ponašanja na internetu.

Neuron je najmanja procesorska jedinica koja prima neki unos, pomnoži unos s težinom veze preko koje je došao taj unos, te izračuna aktivaciju neurona preko jedne od aktivacijskih funkcija i na kraju šalje to vrijednost sljedećem neuronu u mreži. Neuroni su uglavnom grupirani u slojeve ovisno o specifičnom zadatku koji odrađuju u mreži i o arhitekturi mreže. Povezani su vezama čije se težine mijenjaju za vrijeme faze učenja (koja se još zove i faza uvježbavanja i optimizacije).

Faza učenja se uglavnom sastoji od namještanja težina tako da je učinak mreže bliži željenom učinku za zadane unose. Učenje se realizira različitim algoritmima od kojih je najpopularniji algoritam unazadne propagacije (engl. *Backpropagation*) (BP).

Neke od značajki umjetnih neuronskih mreža su:

- Dobre su za procjenjivanje nelinearnih odnosa u uzorcima podataka
- Mogu raditi i s oštećenim i fragmentiranim podacima koji su tipični za senzorske podatke (kamere, mikrofone itd.) i prepoznati uzorke u njima
- Robustne su s obzirom na greške u podacima
- Mogu stvarati svoje veze između podataka kada te veze nisu eksplicitno zadane
- Mogućnost rada s velikim brojem unosa
- Prilagodljive su okolini

1.1. Povijest neuronskih mreža

Povijest umjetnih neuronskih mreža započinje s radiom McCulloh-a i Pitts-a iz 1943. što se nastavilo s Donald Hebb-om koji je napisao knjigu *The Organization of Behavior* u kojoj dolazi do zaključka da su učenja procesi koji mijenjaju strukturu mozga tako da se veze ili sinapse između određenih neurona kada bi se taj neuron aktivirao [5]. 1951. poznati profesor umjetne inteligencije s MIT-a, profesor Marvin Minsky se zainteresirao za ovo novo područje i stvorio prvo neuro-računalo (*SNARC - Stochastic Neural Analog Reinforcement Calculator*) koje se sastojalo od 40 neurona povezanih vakuumskim cijevima te je učilo na način koji je opisao Hebb [6] [7]. Bilo je uspješno s tehničkog pogleda, ali nije radilo ništa osobito interesantno s pogleda procesiranja informacija.

1957. i 1958. Rosenblatt, Wightman i ostali su razvili *Mark I* perceptron neuro-računalo [8]. Rosenblatt je kasnije postao poznat kao otac neuro-računalstva. 1959. Widrow i Hoff su razvili model nazvan ADALINE (engl. *Multiple ADAptive LINear Elements*) za prepoznavanje binarnih uzoraka koji je bio rana implementacija jednoslojnog perceptrona. [9]

Istraživačkom polju umjetnih neuronskih mreža je tada, po mnogim znanstvenicima nedostajalo rigoroznosti koja je bila potrebna za znanstveno polje, te je bilo puno velikih očekivanja za moći i mogućnosti neuronskih mreža u bližoj budućnosti. Sve to je navelo Minskya i Paperta da diskreditiraju istraživanje neuronskih mreža i da preusmjere financiranje tog polja na financiranje polja umjetne inteligencije. Napisali su knjigu *Perceptrons* u kojoj su naveli kao fatalni nedostatak neuronskih mreža njihovu nemogućnosti učenja za predviđanje nelinearne logičke funkcije kao što je isključivo ILI (XOR). [10]

Unatoč dokazanom nedostatku perceptrona, istraživački rad o umjetnim neuronskim mrežama se nastavio u drugim istraživačkim poljima kao što su procesiranje signala, biološko modeliranje i prepoznavanje uzoraka. Neki od najistaknutijih istraživača tog polja su počeli objavljivati svoje radove u ovo vrijeme.

Od 1980. mnogi istraživači su predlagali istraživanja razvoja neuro-računala i primjena neuronskih mreža. Jedan od njih je John Hopfield tko je u 1982. započeo rad u području neuronskih mreža te je skupa s ostalim predavačima uspio nagovoriti mnoge istraživače da se pridruže polju. On se bavio stvaranjem mreža koje imaju dvosmjernu vezu, gdje su prijašnje mreže imale samo jednosmjernu. Izmislio je jednu od ponavljajućih neuronskih mreža koja je nazvana po njemu - Hopfield mreža. [11]

1986. tri neovisna istraživača, Geoff Hinton, David Rumelhart i Ronald Williams su u svom radu nazvanom *„Learning internal representation by back propagation"* razvili ideju koja se danas naziva algoritam unazadne propagacije (engl. *Backpropagation*) (BP) koji prenosi greške pri prepoznavanju uzoraka unazad kroz mrežu. Tada su se ovakve mreže zvale sporo-učeće jer je za brzo učenje bilo potrebno i više sklopovlja. Tako je nastalo *Deep Thought* računalo koje je pobijedilo Larsena, šahovskog majstora, u šahu 1988., ali je izgubilo od Gary Kasparova u 1989. Njegov nasljednik, *Deep Blue*, je uspio pobijediti Kasparova 1996-1997 i tako postati prvi računalni sustav koje je pobijedio svjetskog šahovskog prvaka, označavajući time veliku prekretnicu za umjetnu inteligenciju i strojno učenje.
Od ostalih većih uspjeha strojnog učenja tu su IBM-ov *Watson* koji je pobijedio 2011. u kvizu *Jeopardy* i Google-ov *AlphaGo* koji je pobijedio profesionalnog igrača igre *Go* u 2015.

1.2. Tipovi učenja

Pod učenjem (ili uvježbavanjem ili optimizacijom) mreže se podrazumijeva mijenjanje vrijednosti parametara (težinskih vrijednosti veza između neurona) tako da je učinak mreže što bliži traženom učinku. Tri najpopularnija tipa učenja su nadgledano (engl. *supervised*), nenadgledano (engl. *unsupervised*) i pojačano (engl. *reinforced*) učenje te svako od njih ima različite algoritme za učenje.
Pod nadgledano učenje se podrazumijeva davanje mreži podatke za učenje koji su označeni, tj. za svaki unos postoji i željeni, ciljani izlaz i tako se zapravo ‘nadgleda’ nad učenjem mreže. Ova vrsta učenja se obično koristi za probleme klasifikacije i regresije. Cilj nadgledanog učenja je da se mreža nauči da daje izlazne vrijednosti koji su što moguće bliže ciljanim izlaznim vrijednostima. Razlika između ciljanog izlaza i stvarnog se naziva greška (ili gubitak ili trošak), gdje što manja greška znači i bolji učinak mreže. Ovakvo učenje se još zove i učenje popravljanja greške (engl. *error-correction learning*). Učinak mreže se mjeri jednim od mnogih algoritama koji uzimaju u obzir grešku mreže. Jedan od popularniji je algoritam prosječne kvadratne greške (engl. *mean-squared error*) (MSE):

$$E = \frac{1}{n}\sum_{i=1}^{n}(cilj_i - izlaz_i)^2$$

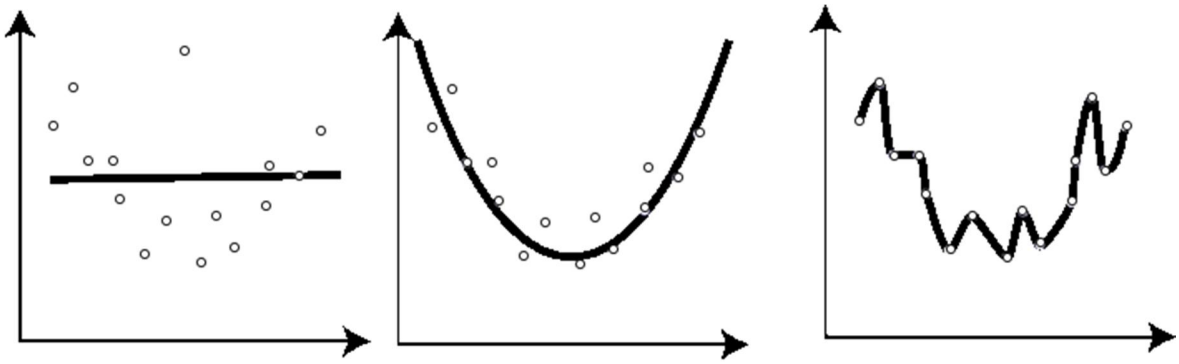
Neke od prednosti ovog algoritma su to što računa prosječnu vrijednost svih izlaznih neurona, te to što se velike greške dodatno kažnjavaju. Još jedan popularni algoritam je *unakrsna entropija*:

$$E = \frac{1}{n}\sum_{i=i}^{n}[cilj_i * ln(izlaz_i) + (1 - cilj_i) * ln(1 - izlaz_k)]$$

Popravci se vrše iterativno tako da za svaki unos mreža namješta težine svojih veza da se smanji vrijednost greške. Takvo smanjivanje se također vrši preko algoritama od kojih je najpoznatiji *gradijent pada* (engl. *Gradient descent*) (GD).

Uobičajeni problemi kod navedenih algoritama su - predobro i loše učenje modela (engl. *overfitting* i *underfitting*). *Overfitting* se dešava kada je odličan učinak mreže za dane podatke, ali vrlo loš za sve ostale. To znači da je mreža naučila primjere, ali nije naučila generalizirati nove podatke. Ovo se dešava kada je struktura mreže presložena. Način na koji se može zaštiti od ovog problema je podjela podataka tako da se ⅔ koristi za učenje a ⅓ za testiranje mreže. Nakon što se obavi proces učenja, s testnim podacima se testira učinkovitost mreže. Ako se uspostavi da je učinak mreže loš, potrebno je promijeniti broj parametara ili povećati broj podataka za učenje ili se koristiti nekom od metoda regularizacije. [12]

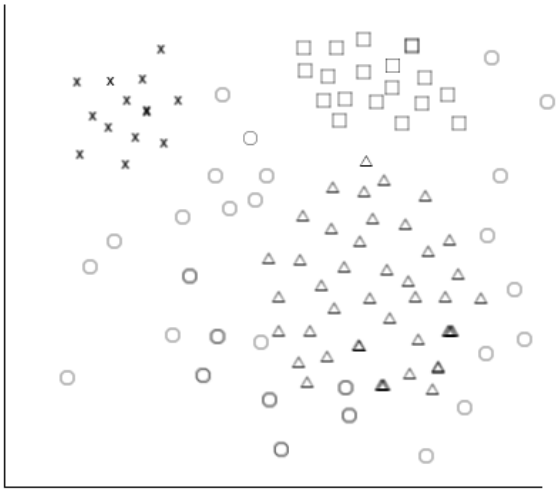
Za razliku od *overfittinga*, *underfitting* se dešava kada mreža ima loš učinak nakon faze učenja. U tom slučaju je potrebno dodati više parametara i / ili slojeva neurona.



Slika 1.1 Prikaz različito naučenih mreža za iste podatke. Na prvom grafu s lijeve strane je prikazana mreža koja je *underfitana*. Na srednjem grafu je mreža koja je pravilno uvježbana. Na desnom grafu je mreža koja je *overfitana*.

Suprotno od nadgledanog učenja je nenadgledano učenje, gdje ne postoje označeni traženi učinci mreže, nego se koriste neoznačeni podaci za učenje. Može se podrazumijevati kao

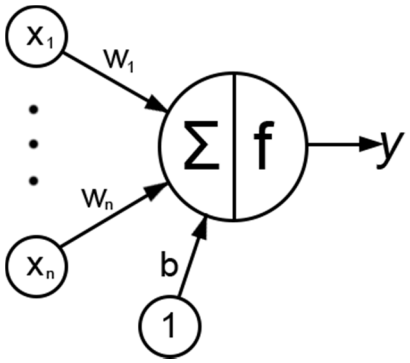
pronalazak uzoraka u potpuno nestrukturiranoj ‘buci’ podataka. Podaci za učenje mogu biti slike, zvukovi, pikseli u videozapisima itd. Kod ovakvih tipova učenja mreže same prepoznaju i klasificiraju uzorke u podacima uz pomoć klasifikacijskih algoritama kako npr. *K-sredina* grupiranje [13] koje grupira točke podataka gdje svaka točka pripada onoj grupi čijoj je sredini najbliža (Slika 1.1). Jednom kada se grupiraju potrebno je potvrditi grupacije tako što se izračuna koliko su udaljene točke od središta neke grupe ili koliko su središta grupa udaljena jedna od drugih.



Slika 1.2 Na slici su prikazane grupe podataka s oznakama križić, trokut i kvadrat dok su krugovi podaci koji nisu grupirani.

Ojačano učenje je tip učenja gdje se mreža uči tako što se pozitivnom povratnom informacijom (ili nagradom) ‘potiču’ određena ponašanja. Slično je pozitivnom pojačavanju koje se koristi u kognitivno-bihevioralnoj terapiji [14]. Može se podrazumijevati kao učenje metodom pokušaja gdje mreža uči iz svoje okoline i posljedica svojih radnji u toj okolini bilo iz iskustva ili od novih odluka. Razlikuje se od nenadgledanog učenja u ciljevima, gdje su ciljevi nenadgledanog učenja pronalazak sličnosti između točaka podataka, a ciljevi ojačanog učenja su pronalazak nagrađivanih radnji i povećavanje ukupne nagrade. Razlikuje se od nadgledanog učenja po načinu na koji se mreža uči, gdje nadgledano učenje mreži daje tražene izlazne podatke za određene ulazne podatke, dok u ojačanom učenju mreža dobiva signale nagrada čime je ovakav tip učenja mnogo fleksibilniji za npr. učenje igranja igra poput šaha.

1.3. Jednoslojni perceptron



Slika 1.3 Jednoslojni perceptron s ulazima $x_{1\text{-}n}$ skupa s težinama $w_{1\text{-}n}$, sklonosti b i izlazom y. Unutar neurana su prikazana njegova dva dijela, sumirajuća funkcija i funkcija izlaza.

Najjednostavniji oblik umjetne neuronske mreže je jednoslojni perceptron[3] (Slika 1.3) koji se sadrži od samo jednog neurona koji vrši binarnu klasifikaciju. Ova jednostavna struktura naučiti jednostavne matematičke i logičke operacije ali ne može logičke operacije kao što je isključivi ILI. [15] Nadgledani tip učenja se koristi za učenje ove vrste neuronskih mreža.

Jedini neuron u mreži ima prag t i aktivira se jedino ako je unos veći od vrijednosti praga. Unos se računa kao zbroj umnožaka težina ulaznih veza i ulaznih vrijednosti:

$$i = \sum_{n=0}^{m} w_m x_m + b$$

Gdje je b sklonost (engl. *bias*) - dozvoljava pomak aktivacijske funkcije[4]. Funkcija koraka se koristi kao aktivacijska funkcija u jednoslojnim preceptronima:

$$f(x) = \begin{cases} 0 \; za \; x < 0 \\ 1 \; za \; x \geq 0 \end{cases}$$

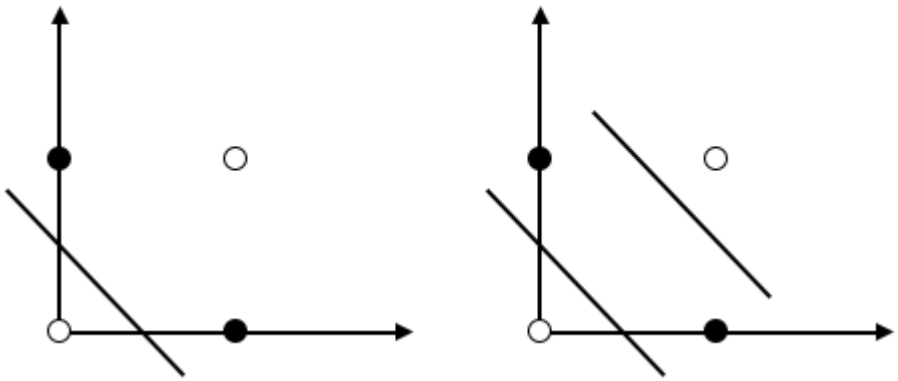
Jednoslojni perceptron se još naziva i Rosenblatt-ov perceptron, po svom izumitelju. U knjizi *Perceptrons* Minsky i Papert su opisali ograničenja jednoslojnih perceptrona te naveli da bi ta ograničenja vrijedila i za ostale izvedbe perceptrona što je uzrokovalo sumnju u računalne mogućnosti svih neuronskih mreža što je rezultiralo usporenju u istraživanju neuronskih mreža u 1970. [16]

[3] Linearni klasifikator koji odlučuje kojoj klasi pripada skup ulaznih vrijednosti.

[4] U slučajnu da su svi ulazni podaci nula i želimo neku specifičnu izlaznu vrijednost, bez sklonosti bi izlaz uvijek bio 1 ako se koristi funkcija koraka kao aktivacijska funkcija.

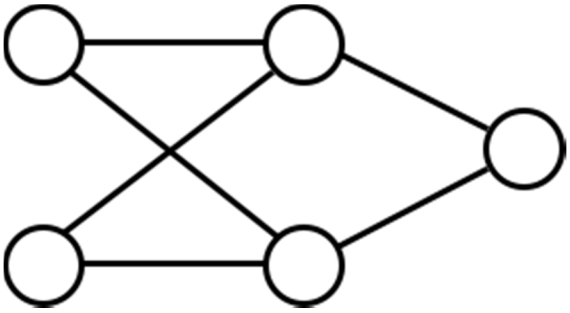
1.4. Višeslojni perceptron

Jedna od najpopularnijih struktura umjetnih neuronskih mreža je višeslojni perceptron koji nadilazi ograničenja jednoslojnog perceptrona. Ova se struktura sastoji od barem jednog skrivenog sloja neurona između ulaznog i izlaznog sloja gdje izlaz skrivenog sloja postaje ulaz izlaznog sloja ili ulaz sljedećeg skrivenog sloja. Takva struktura omogućava mreži da nauči bilo koju matematičku ili logičku operaciju zato što je sposobna baviti se ne-linearno razdvojivim problemima. [17]



Slika 1.4 Razlika u klasifikacijskim mogućnostima između jednoslojnog (lijevo) i višeslojnog (desno) perceptrona.

Kažemo za mrežu da je *N*-slojna kada se sastoji od *N* slojeva otežanih veza i *N* ne-ulaznih slojeva procesnih jedinica ili neurona. Tako da bi jednostavna dvoslojna mreža izgledala ovako:



Slika 1.5 Višeslojni perceptron s dva ulazna, dva skrivena i jednim izlaznim neuronom.

Broj skrivenih slojeva se ne može točno matematički odrediti ali optimalno je negdje između broja ulaznih neurona i izlaznih [18]. Ako ih nema dovoljno mreža neće biti u mogućnosti

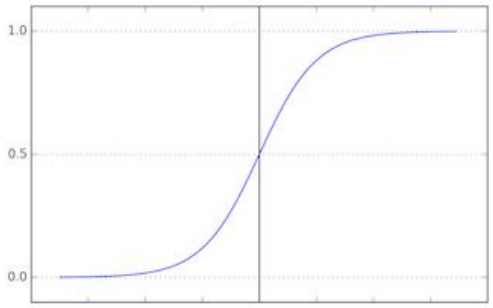
modelirati složene odluke (*underfitting*), a ako ih ima previše imat će problema s generalizacijom (*overfitting*). Neki od faktora za odlučivanje o broju skrivenih slojeva su:

- Broj ulaznih i izlaznih neurona
- Složenost klasifikacije
- Koji se algoritam za učenje koristi
- Šum u podacima

Postoje više algoritama koji se koriste kao aktivacijske funkcije neurona, a neki od korištenijih su:

- Sigmoid. Prima realni broj kao unos i smješta ga između 0 i 1 gdje veliki negativni brojevi postaju 0 a veliki pozitivni brojevi postaju 1. Povijesno ova aktivacijska funkcija je bila najpopularnija no ima manu zato što se sigmoid može zasititi na jednoj od krajnjih vrijednosti i time se poništava gradijent [19]. Zbog ovoga dodatno treba paziti na inicijalizirane vrijednosti jer ako su velike onda će doći do zasićenja.

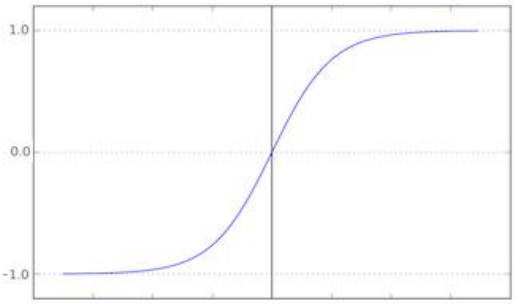
$$S(t) = \frac{1}{1 + e^{-t}}$$



Slika 1.6 Graf sigmoidne funkcije *S(t)*.

- Tangens funkcija (*tanh*). Prima realni broj kao unos i smješta ga između -1 i 1, no ima isti problem kao i sigmoid s zasićivanjem.

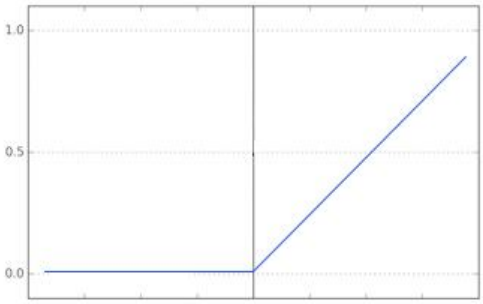
$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$



Slika 1.7 Graf tangens funkcije *tanh(x)*.

- ReLU (engl. *Rectified Linear Unit*). Vrlo popularna aktivacijska funkcija u zadnjih nekoliko godina. Naime ovakva aktivacijska funkcija uvelike ubrzava konvergenciju algoritma stohastičkog gradijenta pada za razliku od sigmoid i tanh funkcija. Također za razliku od sigmoid i tanh, ReLU nema nekih skupih operacija kao što je kvadriranje. Mana joj je to što uz veliku stopu učenja i veliki gradijent neke od ReLU jedinica mogu 'odumrijeti'[5] [20] tako da gradijent zauvijek ostane nula kod tih jedinica, no korištenjem pravilne stope učenja ovo postaje rijedak problem.

$$f(x) = \max(0, x)$$



Slika 1.8 Graf ReLu funkcije *f(x)*.

Najpopularniji algoritam za učenje višeslojnih perceptrona je algoritam unazadne propagacije, koji će se detaljnije obraditi u sljedećem poglavlju.

1.5. Algoritam unazadne propagacije

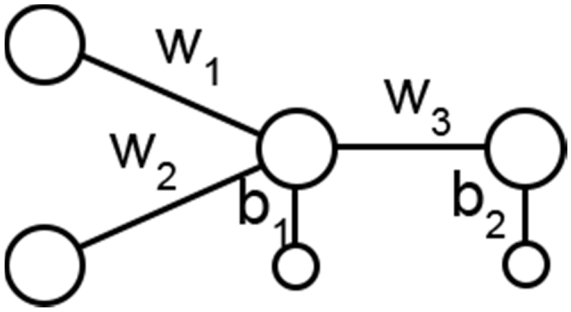
U srcu svake umjetne neuronske mreže stoji optimizacija ili učenje. Algoritam unazadne propagacije (engl. *Backpropagation*) (BP) je jedan od najpopularnijih algoritama za učenje višeslojnih perceptrona. Cilj BP algoritma je optimizirati težine veza u mreži tako da mreža može što bolje preslikavati ulazne vrijednosti u željene izlazne vrijednosti. BP se sastoji od dvije faze:

1) Prednja faza gdje se ulazne vrijednosti propagiraju kroz mrežu dok ne dođu do kraja.

[5] U slučaju velike vrijednosti sklonosti ovaj jedinica će uvijek davati nulu kao izlaznu vrijednost i zbog toga ni algoritam gradijenta pada neće ovu jedinicu prilagođavati.

2) Unazadna faza gdje se greška (razlika između željene i dobivene izlazne vrijednosti) izračuna i propagira se nazad kroz mrežu gdje se koristi određeni algoritam koji prilagođava težine veza da bi se smanjila ukupna greška.

BP algoritam se najbolje objašnjava kroz primjer. Koristit ćemo jednostavan primjer (Slika 1.9) dvoslojne mreže s dva ulazna neurona (*i*), jednim skrivenim (*h*) i jednim izlaznim (*o*):



Slika 1.9 Struktura mreže korištene u primjeru.

te ćemo nasumično postaviti težine veza:

Veza	w_1	w_2	w_3	b_1	b_2
Težinska vrijednost	0.3	0.7	0.5	0.4	0.1

Tablica 1.1 Težinske vrijednosti pojedinih veza neurona u primjeru.

Za početak ćemo odraditi prednju fazu gdje ćemo izračunati izlaznu vrijednost mreže kao i vrijednost greške. Kao primjere unosa uzet ćemo da su $i_1 = 1$, $i_2 = 2$, a željena izlazna vrijednost će biti 0.99.

Prvo ćemo izračunati izlaznu vrijednost skrivenog sloja. Svaki neuron se sastoji od dva dijela (Slika 1.3). Prvi zbraja umnoške ulaznih vrijednosti i težina ulaznih veza, a drugi izračunava vrijednost aktivacijske funkcije i tu vrijednost šalje sljedećem neuronu. Kako imamo samo jedan neuron u skrivenom sloju izračunat ćemo izlaznu vrijednost samo za taj neuron, u slučaju da ih je više, isti izračun bi radili za svaki dodatni neuron u skrivenom sloju.

$$in_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1 = 2.1 \quad (1)$$

Izračunali smo vrijednost unosa u skriveni neuron. Sada ćemo izračunati izlaznu vrijednost, a kao aktivacijsku funkciju ćemo koristiti sigmoid:

$$out_{h1} = \frac{1}{1 + e^{-in_{h1}}} = 0{,}890903179 \quad (2)$$

Izračunata izlazna vrijednost skrivenog neurona h_1 postaje ulazna vrijednost izlaznog neurona o_1 za kojeg ćemo odraditi isti izračun:

$$in_{o1} = w_3 * out_{h1} + b_2 * 1 = 0{,}545451589 \quad (3)$$

$$out_{o1} = \frac{1}{1 + e^{-in_{o1}}} = 0{,}633079683 \quad (4)$$

Izlaz izlaznog neurona o_1 je *0,633079683* i za tu vrijednost kažemo da je to izlazna vrijednost mreže, ali to nije ciljana izlazna vrijednost od *0.99* i zbog toga je potrebno izračunati vrijednost greške i popraviti težine veza u mreži da bi izlaznu vrijednost približili ciljanoj. Za izračun greške koristimo algoritam srednje kvadratne greške, gdje je *T* ciljana izlazna vrijednost a *y* dobivena izlazna vrijednost:

$$E = \frac{1}{n}\sum_{i=1}^{n}(T_i - y_i)^2 = (0.99 - 0{,}633079683)^2 = 0{,}127392113 \quad (5)$$

Nakon izračunavanja ukupne vrijednosti greške, prilagodba težina će se odradit uz pomoć algoritma gradijent pada, tako što će se izračunati parcijalne derivacije ukupne vrijednosti greške u odnosu na svaku pojedinu težinu (osim težina sklonosti), ta težina će se prilagoditi tako što će se od trenutne vrijednosti oduzeti umnožak parcijalne derivacije i stope učenja. Proces započinje od neurona u izlaznom sloju i ide prema neuronima u ulaznom sloju. Korištenjem lančanog pravila dobijemo:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial in_{o1}} * \frac{\partial in_{o1}}{\partial w_3} \tag{6}$$

Izračunat će se vrijednosti svih pojedinih faktora a na kraju njihov umnožak:

$$\frac{\partial E}{\partial out_{o1}} = (T - y) * (-1) = -0{,}356920317 \tag{7}$$

$$\frac{\partial out_{o1}}{\partial in_{o1}} = out_{o1} * (1 - out_{o1}) = 0{,}232289798 \tag{8}$$

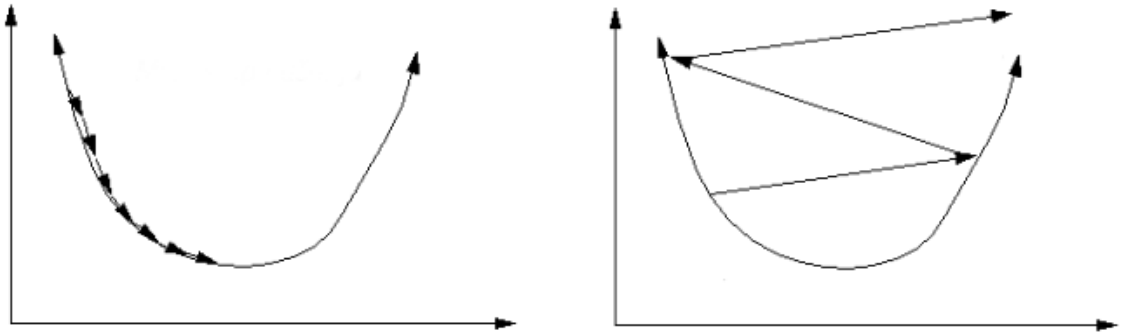
$$\frac{\partial in_{o1}}{\partial w_3} = out_{h1} = 0{,}890903179$$

Što daje rezultat:

$$\frac{\partial E}{\partial w_3} = -0{,}073863846$$

Kada se taj rezultat uračuna u formulu za prilagodbu težine i kada se uračuna stopa učenja $\alpha = 0.5$ dobije se nova vrijednost za w_3. Vrijednost stope učenja je velika samo zbog potrebe ovog primjera. Inače se uzima manja vrijednost jer u ovisnosti o stopi učenja ovisi koliko brzo će i hoće li uopće mreža konvergirati. Vrlo mala vrijednost stope rezultirati će vrlo sporoj konvergenciji, dok vrlo visoka može rezultirati u divergenciji i problemu *hill climbing*-a[6] (Slika 1.10). [17] U većini slučajeva spora konvergencija nije loša stvar i rezultira da mreža bude preciznija:

[6] Popularna analogija s penjanjem na brdo zavezanih očiju, gdje je cilj doći do samog vrha. Ako se penje vrlo kratkim koracima garantirano se dolazi do vrha ali vrlo sporo, dok se s ponjanjem jako velikim koracima može desiti da se vrh promaši i krene u sljedeću udubinu.



Slika 1.10 Prikaz konvergencije s (lijevo) malom stopom učenja i (desno) velikom stopom učenja.

$$w_3^+ = w_3 - \alpha * \frac{\partial E}{\partial w_3} = 0.5 \ - \ 0.5 \ * -0{,}073863846 \ = \ 0{,}536931923 \tag{9}$$

Težina se može ažurirati kada su sve vrijednosti za skriveni sloj izračunate da ne bi poremetilo rezultate.

Sljedeće se računaju nove vrijednosti za w_1 i w_2, koje se računaju na isti način kao i w_3. Zbog jednostavnosti u ovom primjeru se koristi samo jedan izlazni neuron, u slučaju da ih je više potrebno je zbrojiti sve vrijednosti na koje izlaz skrivenog neurona može utjecati. Tako da u slučaju dva izlazna neurona jednadžba (7) bi bila zbroj parcijalnih derivacija za svaki izlazni neuron:

$$\frac{\partial E}{\partial out_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} + \frac{\partial E_{o2}}{\partial out_{o1}}$$

$$E_{o1} = \frac{1}{2}(T \ - \ out_{o1})^2 \text{ and } E_{o2} = \frac{1}{2}(T \ - \ out_{o2})^2$$

Jednadžba za izračunavanje parcijalne derivacije ukupne greške s obzirom na w_1 je:

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial in_{h1}} * \frac{\partial in_{h1}}{\partial w_1} \tag{10}$$

$$\frac{\partial E}{\partial out_{h1}} = \frac{\partial E}{\partial in_{o1}} * \frac{\partial in_{o1}}{\partial out_{o1}}$$

$$\frac{\partial E}{\partial in_{o1}} = \frac{\partial E}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial in_{o1}} = -2{,}366920317 \ * \ 0{,}232289798 \ = \ -0{,}549811442$$

$$\frac{\partial in_{o1}}{\partial out_{o1}} = w_3 = 0.5$$

$$\frac{\partial E}{\partial out_{h1}} = -0{,}274905721$$

$$\frac{\partial out_{h1}}{\partial in_{h1}} = out_{h1} * (1 - out_{h1}) = 0{,}097194705$$

$$\frac{\partial in_{h1}}{\partial w_1} = i_1 = 1$$

$$\frac{\partial E}{\partial w_1} = -0{,}274905721 * 0{,}097194705 * 1 = -0{,}02671938$$

Pa je nova vrijednost za w_1:

$$w_1^+ = w_1 - \alpha * \frac{\partial E}{\partial w_1} = 0.3 - 0.5 * -0{,}02671938 = 0{,}31335969$$

Kada se isto učini za w_2 dobije se:

$$w_2^+ = w_2 - \alpha * \frac{\partial E}{\partial w_2} = 0.7 - 0.5 * (-0{,}274905721 * 0{,}097194705 * 2) = 0{,}72671938$$

Kada su sve nove vrijednosti težina izračunate započinje sljedeća epoha u učenju i računa se nova izlazna vrijednost.

Prateći jednadžbe od 1 do 4 mreža daje novu vrijednost *0,641470079* što je veće od prošle vrijednosti (*0,633079683*). Sa svakom novom epohom i novi prilagođavanjem težina ta će vrijednost doći blizu ciljane vrijednosti od *0.99*.

Učenje se zaustavlja kada se ispuni neki od zadanih uvjeta:

- Dosegne se maksimalni broj epoha
- Dosegne se minimalna vrijednost za ukupnu grešku
- Kada mreža konvergira (mreža, odnosno ukupna greška uvijek konvergira samo je to jako spor proces pa se češće zadaju jedan od gore navedenih uvjeta za zaustavljanje učenja)

1.6. Algoritam gradijenta pada

Algoritam gradijent pada (GP) minimizira funkciju greške i najčešći je način optimizacije neuronske mreže. Neuronska mreža započinje s nasumičnim vrijednostima parametara zbog simetrije, jer ako su sve vrijednosti iste onda bi svi neuroni u skrivenom sloju imali iste izlazne vrijednosti i time bi se izračunavao isti gradijent tijekom izvršavanja BP algoritma i prilagodbe težina bi za svaku mrežu bile iste. GP algoritam te nasumične vrijednosti prilagođava u smjeru vrijednosti koje minimiziraju funkciju greške u smjeru negativnom od smjera funkcije gradijenta.

Funkcija greške omogućava kvantifikaciju kvalitete bilo kojeg skupa težina W. Cilj optimizacije je pronaći onaj skup W koji minimizira tu funkciju greške. Pronalazak najboljeg W odmah je nemoguće, ali iterativno pronalaženje boljeg W je moguće. Zbog opisanih razloga mreža započinje s nasumičnim vrijednostima parametara i iterativno ih prilagođava tako da svakom iteracijom vrijednost greške je manja.

Postoje tri vrste GP algoritma koje se međusobno razlikuju po količini podataka koje se koriste za njihove operacije te kako uravnotežuju preciznost (više podataka - veća preciznost) i vrijeme koliko im je potrebno da prilagodbu parametara (više podataka - više vremena).

Gradijent pada skupa (engl. *Batch gradient descent*) (BGD) ja standardni GP algoritam koji računa gradijent funkcije greške za čitav skup podataka za učenje. BGD se koristio u primjeru BP-a opisanom u jednadžbi (9) jer se koristio samo jedan testni podatak.

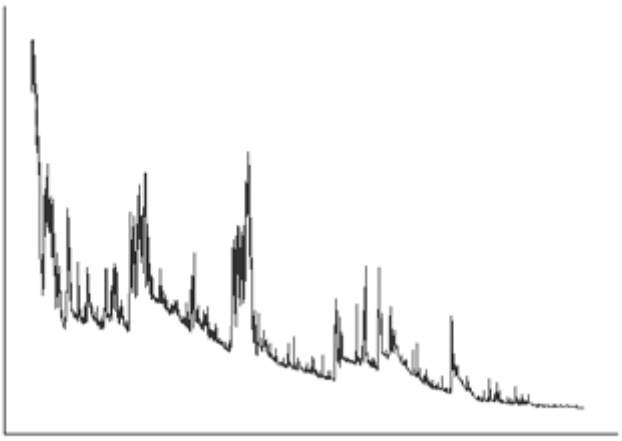
$$W = W - \alpha * gradijent(funkcijaGreške, podaci, parametri)$$

Zato što koristi čitavi skup podataka ovaj algoritam je vrlo spor [21], a ovisno o broju parametara i količini podataka, moguće je da uopće ne stane u memoriju. BDG će konvergirati u globalni minimum za konveksne površine i u lokalni minimum za ne-konveksne 'brjegovite' površine.

Stohastički gradijent pada (engl. *Stochastic gradient descent*) (SGD) za razliku od BGD računa gradijent funkcije greške za svaki pojedinačni podskup podataka u skupu podataka za učenje [22]:

$$W = W - \alpha * gradijent(funkcijaGreške, podaci_i, parametri_i)$$

SGD je zbog toga mnogo brži i moguće ga je ažurirati s novim podacima u toku izvršavanja (engl. *On-line updates*). [17] SGD vrši česte prilagodbe s velikim odstupanjima što rezultira u veliki fluktuacijama:



Slika 1.11 Vrijednosti stope greške kroz proces učenja s SGD algoritmom.

Na slici (Slika 1.11) se vidi kako SGD stalno preskače minimum što može predstavljati problem, ali ako se smanjuje stopa učenja pokazalo se da SGD konvergira kao i BGD.

Gradijent pada podskupa (engl. *Mini-batch gradient descent*) (MBGD) je kombinacija SGD i BGD gdje se izračunava gradijent i vrši prilagodba parametara za n-podskupe u sveukupnom skupu podataka [23]:

$$W = W - \alpha * gradijent(funkcijaGreške, podaci_{(i:i+n)}, parametri_{(i:i+n)})$$

Na ovaj način je moguće smanjiti odstupanja u prilagodbama parametara što dovodi do stabilnije konvergencije i bržeg izvođenja algoritma, iako postoje određeni problemi koji se trebaju riješiti [24]:

- Odabir pravilne vrijednosti stope učenja
- Raspored stope učenja: pokušaj prilagodbe stope učenja tako što se konstantno smanjuje po nekom određenom rasporedu ili kada stopa greške padne ispod

određenog praga. Ovakav raspored i prag se moraju unaprijed definirati i zbog toga se ne mogu prilagoditi karakteristikama skupa podataka za učenje

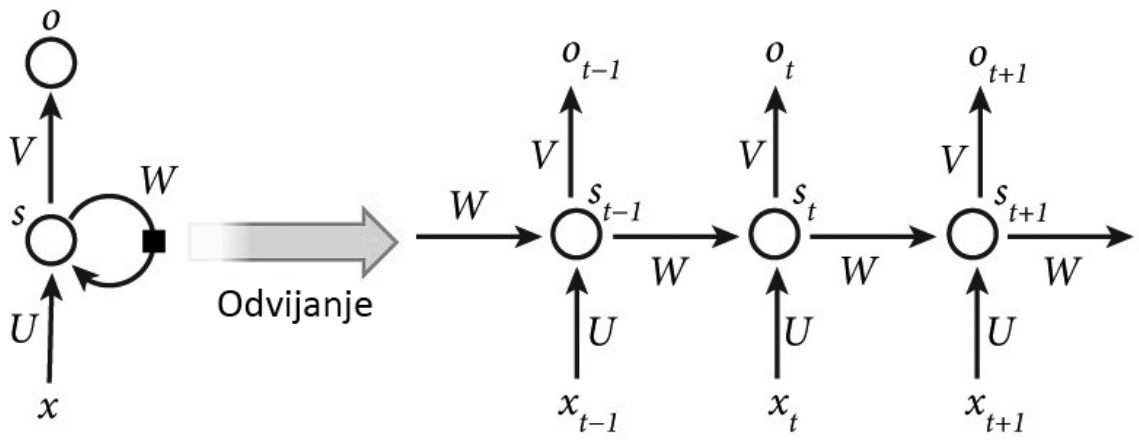
- Izbjegavanje upada u brojne pod-optimalne minimume

Postoje nekoliko popularnih rješenja tih problema [24], kao što su:

- *Momentum*: faktor u izračunu gradijenta koji 'ubrzava' gradijent ako ide u istom smjeru, a 'usporava' ako ide u različitim smjerovima.
- *Adagard*: prilagođava stopu učenja parametrima, tj. koristi različitu stopu učenja za svaki pojedinačni parametar u svakoj iteraciji.
- *Adadelta*: nadovezuje se na *Adagard* algoritam čineći ga manje 'agresivnim' tako što smanjuje stopu učenja.
- *RMSprop*: službeno neobjavljen algoritam kojega je stvorio Hinton. Radi na sličnom principu kao i *Adadelta*.
- *Adam* (*Adaptive Moment Estimation*): kombinacija gore navedenih algoritama

1.7. Povratna neuronska mreža

Višeslojne neuronske mreže su statične mreže i mogu imati samo unaprijed definirani broj unosa koji su međusobno neovisni i za svaki se preslikava u unaprijed definirani broj izlaza, što čini ovakvu vrstu mreža neučinkovitom za zadatke kao što su predviđanje sljedeće riječi u rečenici, prijevode gdje kontekst prošlih riječi utječe na sljedeće, kao ni primati sljedove različitih duljina kao unose. Povratne neuronske mreže (engl. *recurrent neural networks*) (RNN) se mogu koristit za preslikavanje sljedova unosa različitih duljina u sljedove izlaza različitih duljina. RNN mreže se zovu 'povratne' zato što izvode isti zadatak za svaki element u slijedu gdje je izlaz jedne takve operacije ovisan o izlazu prijašnje operacije. [25]

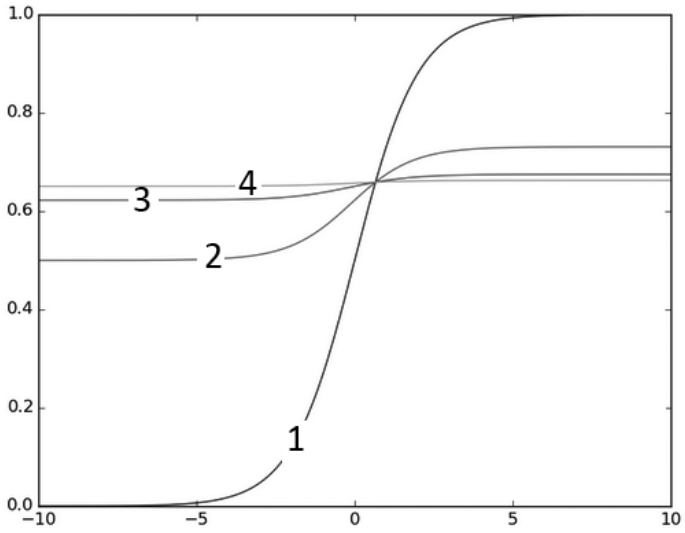


Slika 1.12 S lijeve strane je prikazana RNN mreža dok s desne 'odvijena' RNN, tj. kako izgleda kada se mreža pokrene tri puta. S *x* je označen ulaz, *s* je neuron u mreži, a *o* je izlazna vrijednost, *U*, *V* i *W* su vrijednosti težina veza između dijelova RNN mreže.

RNN mreže se mogu odviti u veliki broj slojeva što ih čini mrežama 'dubokog učenja'. Najvažnija značajka RNN mreža je njihovo skriveno stanje ili unutarnja memorija koja sprema kontekst slijeda.

RNN mreže se uče slično kao i višeslojne neuronske mreže. Svi parametri u mreži se dijele u svim vremenskim koracima i koristi se jedna vrsta BP algoritam za učenje što se razlikuje od BP po tome što gradijent ovisi o izračunima iz svih vremenskih koraka odvijanja učenja. Takav algoritam se zove algoritam unazadne propagacije kroz vrijeme (engl. *backpropagation through time*) (BPTT).

RNN mreže imaju poteškoća s učenjem uz pomoć BPTT algoritma ako su neke od ključnih riječi (često značenje neke riječi ovisi o riječima koje joj nisu u blizini) u kontekstu predaleko jedne od drugih jer se ne mogu više povezati. To se dešava ako mreža radi s veliki brojem sljedova i zato što se u mrežama dubokog učenja, pa samim time i RNN mrežama, slojevi povezuju množenjem, množiti gradijent s čak vrlo malim brojem može prouzročiti da gradijent 'eksplodira' (ako se množi s brojem većim od nule) ili 'nestane' (ako se množi s brojem manjim od nule). U slučaju 'eksplodiranja', gradijent nakon velikog broja množenja postane prezasićen i njegova vrijednost postaje prevelika, no taj problem se lako može riješiti tako da se težine 'zgnječe' pomoću nekih od matematičkih operacija kao što su sigmoid, tanh i slični. Dok s druge strane, kod nestajanja gradijenta vrijednosti težina postaju tako male da računala ne mogu s njima računati (Slika 1.13) i one postanu neupotrebljive u procesu učenja mreže što ovaj problem čini težim za riješiti.



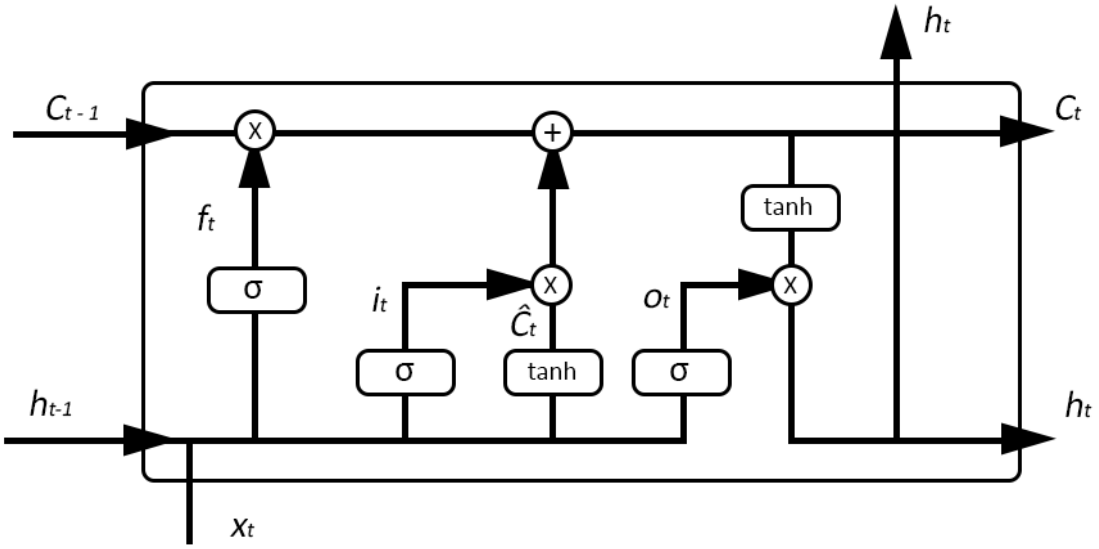
Slika 1.13 Funkcije višestrukih sigmoida. (1) jednostruki sigmoid, (2) dvostruki sigmoid, (3) trostruki sigmoid, (4) četverostruki sigmoid.

Problemi eksplodirajućeg i nestajućeg gradijenta kod RNN mreža je dobro dokumentiran i istražen[7] i glavni je razlog zašto izvorne RNN mreže se ne koriste tako često danas. [26]

1.8. Mreže duge kratkoročne memorije

Rješenje problema eksplodirajućeg i nestajućeg gradijenta u RNN mrežama su 90ih godina predstavili njemački istraživači Sepp Hochreiter i Jürgen Schmidhuber u svom radu "*Long short-term memory*" [26]. Mnogi istraživači su doprinosili ideji modernih mreža dugih kratkoročnih memorija (engl. *Long short-term* memory) (LSTM). Dugoročno pamćenje informacije je zadano stanje ovakvih mreža i zbog toga nemaju problema koje imaju RNN mreže.

[7] Grešku je prvi pronašao Hochreiter još 1991. i obradio kako temu svog diplomskog rada.



Slika 1.14 Jednostavna struktura LSTM mreže

LSTM mreža, kao i sve ostale ponavljajuće neuronske mreže, imaju oblik lanca ponavljajućih modula. U RNN mrežama taj modul je jedan jednostavni neuron, dok kod LSTM mreža postoje četiri različite jedinice ili ćelije koje na poseban način međusobno djeluju. Glavna značajka LSTM mreža je vrijednost stanja ćelija koja prolazi kroz sve vremenske korake mreže što omogućava jednostavni protok informacija bez da se nenamjerno mijenjaju. Mreža utječe na to stanje preko tri različita tipa vrata: ulazna, izlazna i vrata za zaboravljanje, koje određuju koje će informacije prolaziti kroz stanje. Sastoje se od jednostavnih slojeva neurona s, uobičajeno, sigmoidnom funkcijom, koji množe matrične vrijednosti. Sigmoid, zbog svoje mogućnosti da smjesti bilo koji broj u raspon od nule do jedan, odlučuje koliki će dio neke vrijednosti propustiti dalje, gdje nula znači da se ta vrijednost ne propušta, dok jedinica znači da se čitava vrijednost propušta. Za objašnjenje rada LSTM mreže proći ćemo jedan vremenski korak učenja kao primjer.

Prva stvar što mreža učini je da izračuna koji dio stanja će izbrisati ili ‘zaboraviti’. Ovaj dio odrađuju vrata za zaboravljanje. Uzimajući u obzir vrijednosti za h_{t-1} i novi unos x_t izračunava sigmoid između nule i jedinice za svaki broj unutar stanja C_{t-1}:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

U sljedećem koraku mreža izračunava koje će nove informacije spremiti u vrijednosti stanja. Ovaj korak se sastoji od dva dijela gdje u prvom ulazna vrata odlučuju koje će se vrijednosti ažurirati, a tanh sloj (onaj isti kao i u RNN mreža) stvara nove vrijednosti koje bi se mogle dodati u stanje. Kombinacijom te dvije operacije ažurira se stanje:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] \; + \; b_{\mathrm{i}})$$

$$\hat{\mathrm{C}}_{\mathrm{t}} = tanh(W_c \cdot [h_{t-1}, x_t] \; + \; b_{\mathrm{C}})$$

Nakon što je mreža ustanovila koje će vrijednosti ažurirati, te odluke se primjenjuju u sljedećem koraku. Prvo se pomnoži staro stanje s vrijednostima vrata za zaboravljanje i time se uklanjaju vrijednosti za koje je odlučeno da se trebaju 'zaboraviti'. Nakon toga se pomnoži novo stanje s vrijednostima ulaznih vrata koja reguliraju koliko kojih novih vrijednosti će proći u stanje mreže:

$$C_t = f_t * C_{t-1} + i_t * \hat{\mathrm{C}}_{\mathrm{t}}$$

I na kraju je potrebno odlučiti koliko od kojih novih vrijednosti će se slati u sljedeći vremenski korak (sljedeću epohu učenja). Ova vrijednost će biti vrijednost stanja ali promijenjena za vrijednost izlaznih vrata. Prvo se izračunaju vrijednosti izlaznih vrata, poslije čega se vrijednosti stanja tanh operacija suzbija na vrijednosti između -1 i 1 a zatim se te vrijednosti pomnože s vrijednostima izlaznih vrata i dobiju se izlazne vrijednosti:

$$o_t = \sigma(W_i \cdot [h_{t-1}, x_t] \; + \; b_o)$$

$$h_t = o_t * tanh(C_t)$$

Izlazne vrijednosti $\underline{h_t}$ kao i stanje C_t se šalju u sljedeći vremenski korak i postaju h_{t-1} i C_{t-1} i gore opisani proces počinje iz početka.

Postoji mnogo različitih LSTM mreža od kojih je ova opisana najobičnija. Neki od značajnijih varijacija LSTM mreža su:

- Davanje vratima mogućnost saznanja vrijednosti stanja i uračunavanja tih vrijednosti u izračune vrijednosti o tome kako će utjecati na pojedine vrijednosti stanja. [27]
- Kombiniranje ulaznih vrata i vrata za zaboravljanje gdje mreža samo zaboravlja one vrijednosti koje će kasnije ažurirati, a ažurira samo one vrijednosti koje je 'zaboravila'
- *Gated Recurrent Unit* (GRU) kombinira ulazna vrata i vrata za zaboravljanje i spaja stanje s skrivenim stanjem (tanh sloj u mreži) [28]

2. JavaScript aplikacija

U sklopu ovog diplomskog rada napravljena je JavaScript aplikacija za izradu i prikaz višeslojnih neuronskih mreža. Motivacija iza ove aplikacije je napraviti sustav koji će relativno zahtjevne operacije kao što su učenje različitih jednostavnijih neuronskih mreža uspjeti obaviti isključivo u internet pregledniku, bez potrebe za serverom koji bi odradio svo 'teško dizanje'. Trenutno aplikacija podržava strukture višeslojnih perceptrona i LSTM mreže.

Za prikaz sučelja je korišten HTML i CSS, za upravljanje sučeljem je korišten *jQuery*, za prikaz strukture neuronske mreže *D3js*, a za samu izradu i učenje neuronskih mreža *SynapticJS*.

3.1. SynapticJS

SynapticJS[8] je jedan od JavaScript (JS) okvira (engl. *framework*) za izradu i učenje neuronskih mreža. Ovaj okvir omogućava stvaranja neurona u obliku JS objekata, povezivanje tih objekata u slojeve koje su također u obliku JS objekata, a kombinacijom tih slojeva stvara se neuronska mreža koja se uči pomoću posebnog *Trainer* objekta koji naveliko olakšava proces učenja mreža time što automatizira čitavi proces, a korisniku daje mogućnosti odabira raznih hiper-parametara kao što su stopa učenja, uvjeti zaustavljanja učenja, aktivacijska funkcija itd. Zbog tako modularne strukture moguće je stvoriti veliki broj različitih arhitektura višeslojnih neuronskih mreža i učiti ih na različite načine.

Ovaj okvir omogućava i stvaranja nekih od popularnih mreža pomoću *Architect* značajke koja uključuje arhitekture višeslojnih perceptrona, LSTM, strojeve tekućeg stanja (engl. *Liquid state machines*) i Hopfield-ove mreže. Stvaranje jedne od unaprijed definiranih arhitektura se može napraviti u jednoj linij:

```
var novaMreza = new Architect.Perceptron(2, 2, 1);
```

[8] http://synaptic.juancazala.com/

Za ovu aplikaciju se nije koristio *Architect* već se se mreže spajale sloj po sloj, čime je implementacija fleksibilnija (npr. moguće je definirati i veze između slojeva kao jedan-na-jedan, jedan-na-više i više-na-više):

```
function Perceptron(input, hidden, output) {
    var inputLayer = new Layer(input),
        hiddenLayers = [],
        hiddenLayer,
        outputLayer;

    _.each(hidden, function (neuronNum) {
      hiddenLayer = new Layer(neuronNum);
      hiddenLayers.push(hiddenLayer);
    });
    outputLayer = new Layer(output);

    inputLayer.project(hiddenLayers[0]);
    for (i = 1; i < hiddenLayers.length; i++) {
      hiddenLayers[i-1].project(hiddenLayers[i]);
    }
    hiddenLayers[hiddenLayers.length - 1].project(outputLayer);

    this.set({
        input: inputLayer,
        hidden: hiddenLayers,
        output: outputLayer
    });
}
```

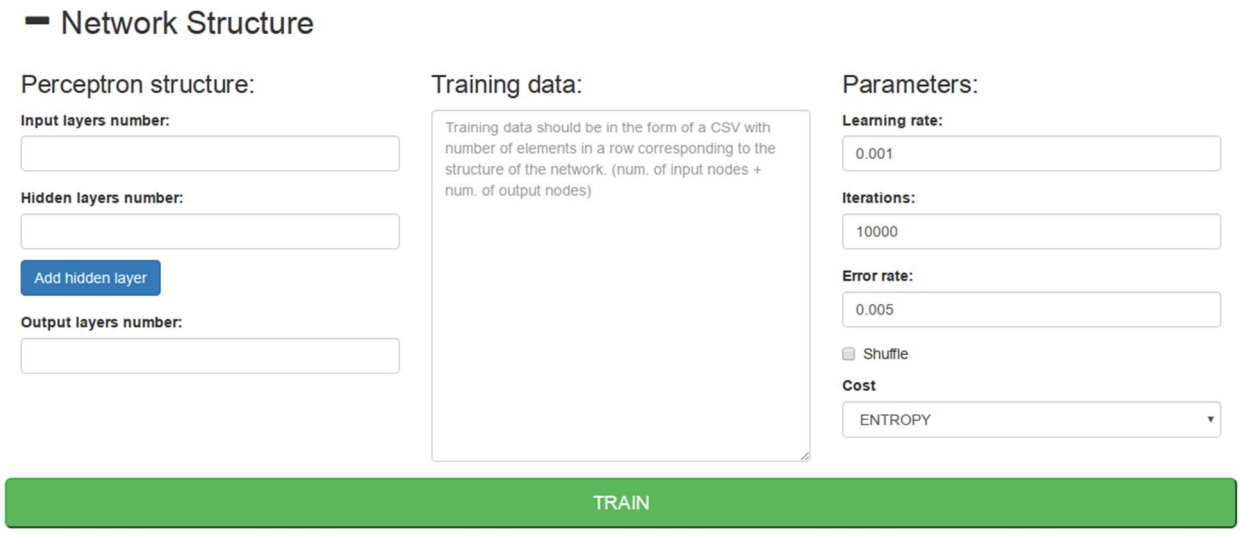
Kôd 2.1 Funkcija za stvaranje nove mreže višeslojnog perceptrona.

3.2. Sučelje aplikacije

Sučelje aplikacije je rađeno s ciljem olakšavanja stvaranja i učenja novih neuronskih mreža. Prilagodljivo je ekranima različitih veličina. Sučelje mreže višeslojnih perceptrona kao i LSTM mreža se sastoji od ista četiri dijela:

2.2.1. Dio za postavljanje hiper-parametera i podataka za učenje mreže

U ovom dijelu se definira struktura mreže, vrijednosti hiper-parametera za učenje, te gdje se unose podaci za učenje (Slika 2.1).

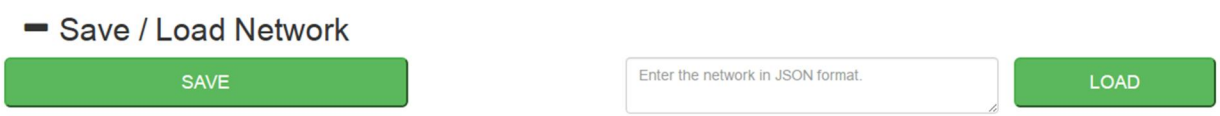


Slika 2.1 Dio sučelja u kojem se postavljaju vrijednosti hiper-parametara i unose se podaci za učenje neuronske mreže

Klikom na gumb *Add hidden layer* dodaje se novo polje za unos broja neurona u dodatnom skrivenom polju, pa je moguće definirati bilo koju strukturu mreže višeslojnih perceptrona za korištenje. Od funkcija greške moguće je odabrati jednu od sljedeće tri:

- Algoritam unakrsne entropije
- Algoritam prosječne kvadratne greške
- Binarna vrijednost greške

2.2.2. Dio za spremanje i učitavanje mreže

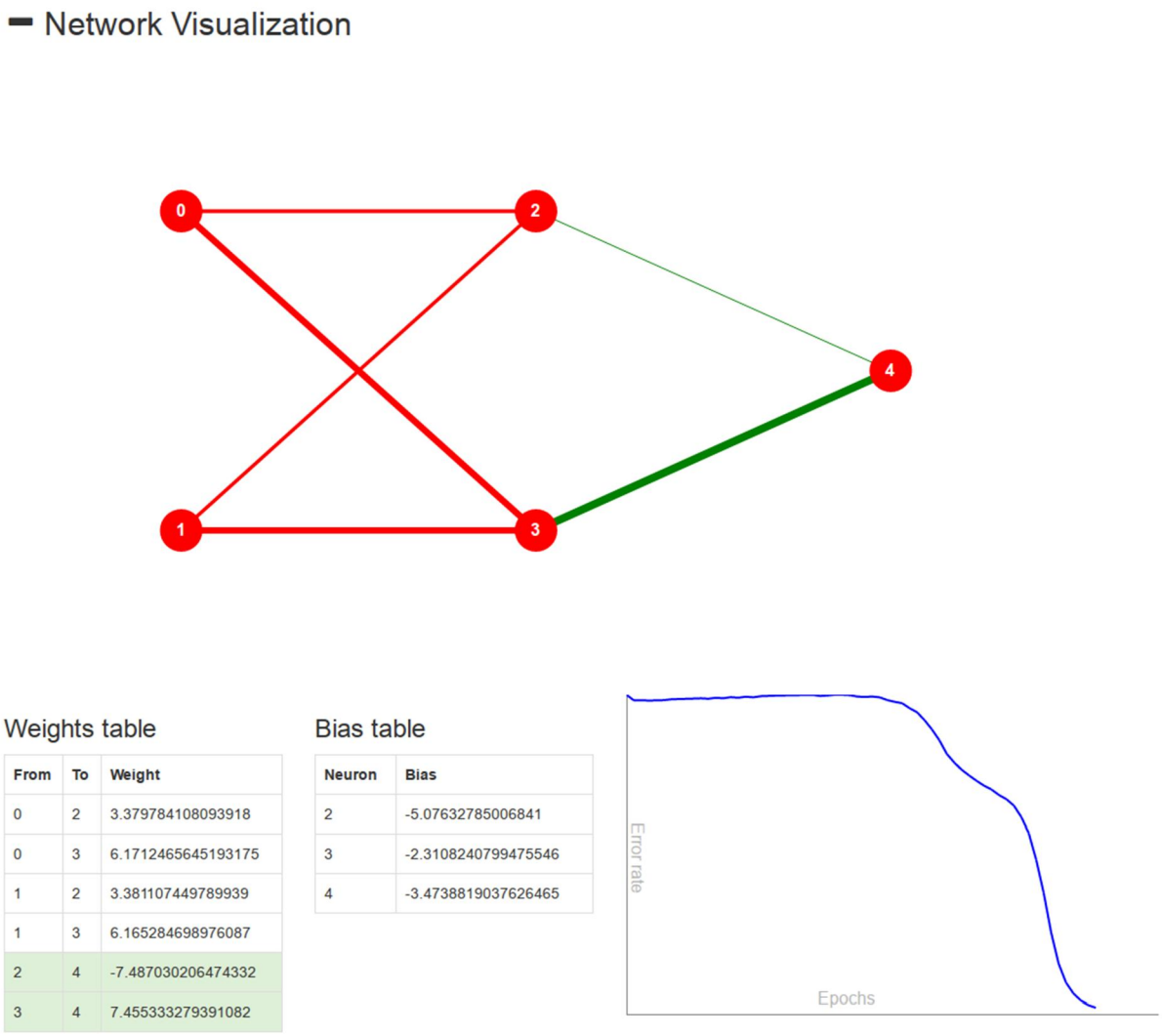


Slika 2.2 Dio sučelja u kojem se nalaze funkcionalnosti spremanja i učitavanja neuronskih mreža.

Klasični načini spremanja su ili spremanje u obliku datoteke ili spremanje u bazi podataka. Kako ova aplikacija nema u pozadini server koji bi upravljao s bazom podataka i kako JS

nije u mogućnosti upravljati s datotekama bez dodatnih programskih paketa, problem spremanja se riješio tako da se u pregledniku otvori nova kartica u kojoj je ispisna struktura naučene mreže u JSON formatu. Učitavanje mreže se vrši tako da se taj JSON objekt unese u polje za unos i stisne gumb *LOAD* što učitava spremljenu mrežu.

2.2.3. Dio za vizualizaciju mreže i procesa učenja

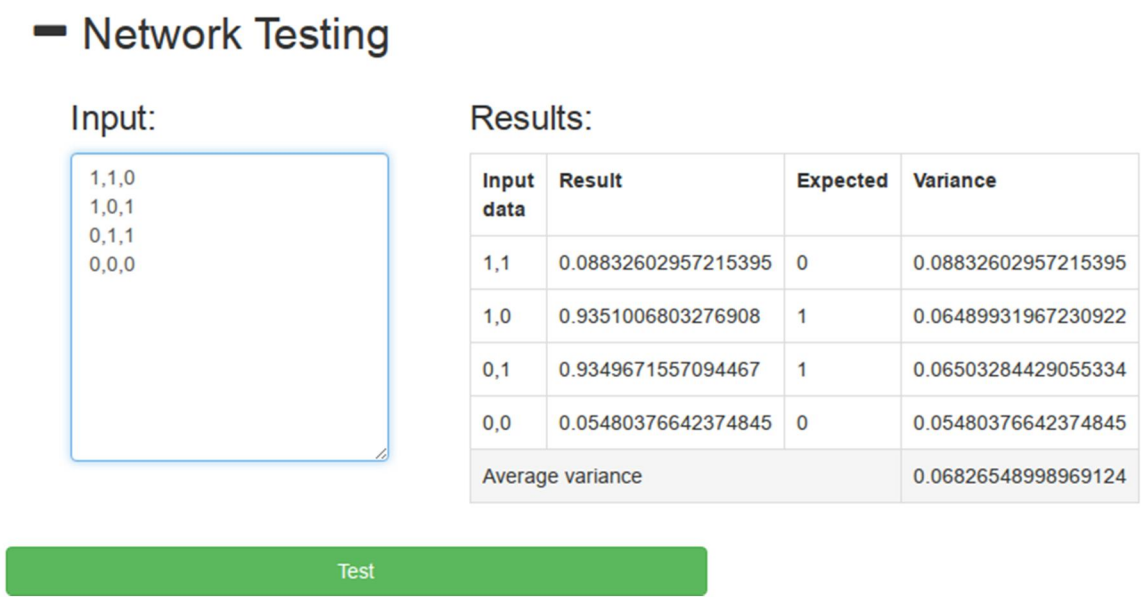


Slika 2.3 Dio sučelja za vizualizaciju neuronske mreže, procesa učenja i težinske vrijednosti veza između neurona i sklonosti u mreži.

Ovaj dio se sastoji od tri dijela. Prvi je vizualizacija strukture mreže i veza između neurona u svakom sloju. Za ovaj dio se koristila popularni okvir za vizualizaciju u JS-u - *D3js*. Drugi dio su tablice težinskih vrijednosti veza i vrijednosti sklonosti. Pomoću ovih tablica moguće je ručno prolaziti kroz mrežu i izračunati izlaznu vrijednost. Treći dio je graf stope greške

kroz proces učenja. Ovog grafa nema kada se mreža učita kao JSON objekt u aplikaciju jer taj JSON objekt u sebi nema informacije o stopi greške za vrijeme učenja mreže.

2.2.4. Dio za automatsko testiranje naučene mreže



Slika 2.4 Dio sučelja za testiranje neuronske mreže

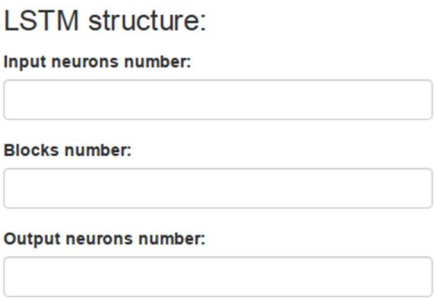
U ovom dijelu je moguće unijeti testne podatke i na osnovu njih dobiti rezultate mreže u tablici. Tablica za svaki redak pokazuje koje su ulazne vrijednosti, koje je izlazne vrijednosti mreža izračunala, koji je ciljani ili željeni izlaz za te ulazne vrijednosti i koliko je odstupanje. Na kraju se izračuna prosječno odstupanje mreže od ciljanih vrijednosti izlaza.

Dodatne informacije kao što su vrijeme potrebno za učenje mreže, broj iteracija i konačna vrijednost greške se ispisuju u JS konzoli (Slika 2.5).

```
Finished training in [6374ms].
Iterations: [97115]
Final error: [0.004996399772998699]
```

Slika 2.5 Dodatne informacije o procesu učenja mreže ispisane u JS konzoli.

Osim višeslojnog perceptrona, aplikacija može učiti i LSTM mreže. Razlika u sučelju je u dijelu za definiranje strukture mreže gdje se sada umjesto polja za unos broja neurona u skrivenom sloju nalazi polje za unos broja LSTM ćelija u vratima:



Druga razlika je u dijelu vizualizacije, naime s porastom broja ćelija u vratima LSTM mreže eksponencijalno se povećava i složenost strukture mreže stoga je vrlo teško napraviti kvalitetnu vizualizaciju u *D3js* okviru. Stoga se trenutno vizualizira samo stopa greške kroz proces učenja.

Od bitnih promjena u načinu implementacije, LSTM se jedino bitno razlikuje u stvaranju mreže, dok su ostale funkcije više-manje iste:

```
function LSTM(input, blocks, output) {
    var inputLayer = new Layer(input),
        inputGate = new Layer(blocks , 'inputGate'),
        outputGate = new Layer(blocks, 'outputGate'),
        forgetGate = new Layer(blocks, 'forgetGate'),
        memoryCell = new Layer(blocks, 'memoryCell'),
        outputLayer = new Layer(output),
        self;

    input = inputLayer.project(memoryCell);
    inputLayer.project(inputGate);
    inputLayer.project(outputGate);
    inputLayer.project(forgetGate);
    inputLayer.project(outputLayer);
    output = memoryCell.project(outputLayer);
    self = memoryCell.project(memoryCell);
    memoryCell.project(inputGate);
    memoryCell.project(outputGate);
    memoryCell.project(forgetGate);

    inputGate.gate(input, Layer.gateType.INPUT);
    forgetGate.gate(self, Layer.gateType.ONE_TO_ONE);
    outputGate.gate(output, Layer.gateType.OUTPUT);

    this.set({
```

```
        input: inputLayer,
        hidden: [inputGate, outputGate, forgetGate, memoryCell,
outputGate],
        output: outputLayer
    });
}
```

Kôd 2.2 Funkcija za stvaranje nove LSTM mreže.

Zbog velike modularnosti *SynapticJS* okvira moguće je stvoriti veliki broj različitih arhitektura neuronskih mreža. Od onih arhitektura koje ovaj okvir još podržava pored višeslojnog perceptrona i LSTM su:

- Strojevi tekućeg stanja
- Hopfield-ove mreže

SynapticJS nije jedini JS okvir za rad s neuronskim mrežama, od ostalih tu su još:

- Adnn - omogućava stvaranje neuronskih mreža kao sloj nad kodom za automatsku diferencijaciju.
- ConvNetJS - okvir koji je stvorio Andrej Karpathy za vrijeme postdiplomskog studija na Stanfordu. Ovaj okvir omogućava rad s neuronskim mrežama dubokog učenja.
- Brain - okvir za učenje jednostavnijih neuronskih mreža. Iako je službeno napušten projekt, još uvijek je vrlo popularan na *GitHub*-u.

3.3. Google-ov V8 JavaScript engine

Jedan od faktora koji je najodgovorniji za porast u performansama web tehnologija su engine-i web preglednika kao što je Google-ov V8 JS engine. Kao dio svog popularnog internet preglednika (*Chrome*) Google je izdao i V8 engine koji drastično povećava performanse izvođenja JS koda. Engine je napisan u C++ programskom jeziku i otvorenog je koda. Način na koji radi je da prevodi JS kod u strojni jezik koji onda izvodi *Just-In-Time* (*JIT*) kompajliranje. Engine se sastoji od interpretera (*Ignition*) i kompajlera (*TurboFan*). Interpreter interpretira i izvodi *bytecode*, dok kompajler kompajlira kod u strojni jezik ako je kod napisan na predvidiv način, ako ne onda se kod vraća u normalnu interpretaciju. [29]

Još od začetka V8 JS engine se unapređuje i trenutno ima bolje performanse za određene operacije nego neki programski jezici.[9]

3.4. Usporedba s *TensorFlow*-om

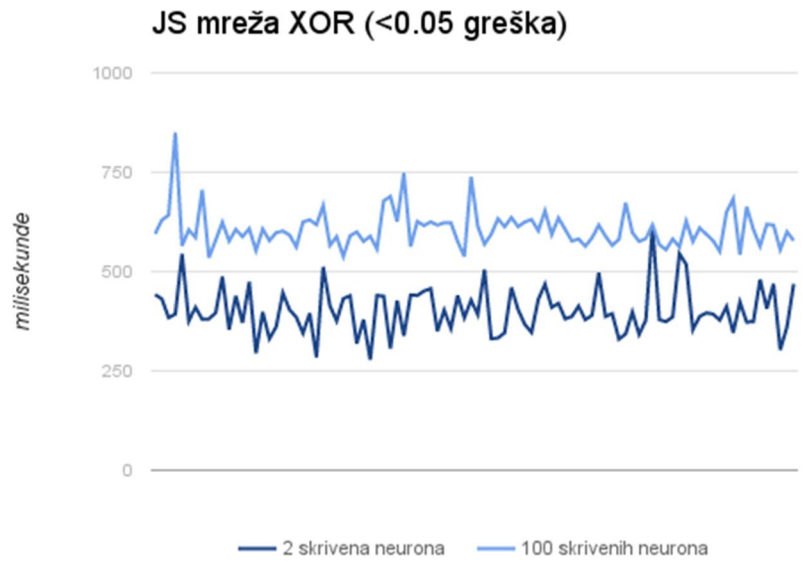
TensorFlow (TF) je okvir za strojno učenje kojega je razvio Google. Otvorenog je koda i trenutno je najpopularniji okvir za strojno učenje. Pisan je u C++ i Python programskim jezicima. Radi na način da strukturu mreže, sve potrebne operacije za učenje i sve ostale dodatne operacije definira unaprijed u Python-u, nakon čega stvara graf izvođenja od svih unaprijed definiranih operacija i tenzora (sve varijable i konstante u TF-u su tenzori). Naznačene operacije se izvode unutar sesije gdje se graf izvođenja prevodi u optimiziran C++ kod koji se kompajlira i izvede, nakon čega se rezultati te izvedbe vraćaju nazad u Python dio koji ih prikaže korisniku.

2.4.1. Usporedba učenja s XOR podacima

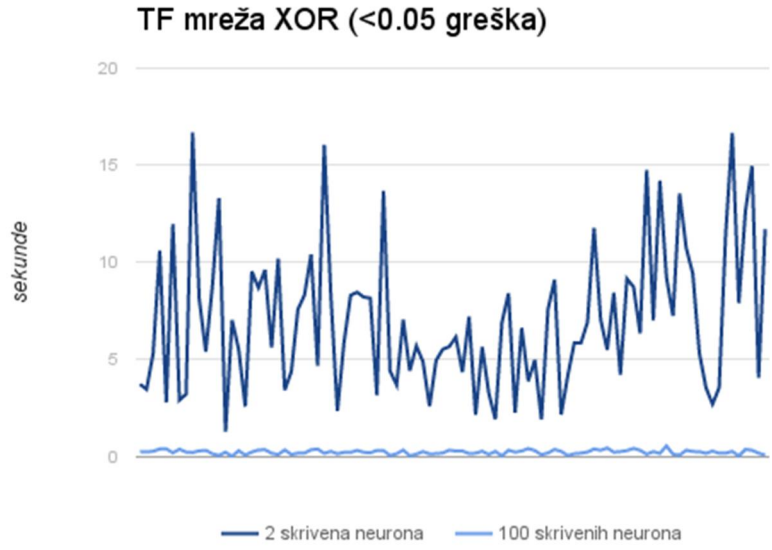
Za potrebe ovog rada stvoren je TF model višeslojnog perceptrona za predviđanje vrijednosti logičke operacije isključivog ILI (XOR) i čije će se performanse učenja usporediti s onima iz JS aplikacije. Za usporedbu su korištene neuronske mreže iste strukture, kao i ista stopa učenja (0.01). Za funkciju greške se koristio algoritam srednje kvadratne greške, a za optimizaciju BP algoritam i gradijent pada. Zbog promjenjive prirode gradijenta pada, svaki od eksperimenata se vršio po 100 puta.

Za XOR su potrebna dva ulazna neurona i jedan izlazni. Prva usporedba će utvrditi koliko je vremena potrebno da obje aplikacije nauče mrežu ispod 0.05 vrijednosti greške za strukture 2-2-1 (dva ulazna neurona, dva skrivena i dva izlazna) i 2-100-1. Zbog povećanja složenosti strukture mreže, povećat će se i broj operacija potrebno za izvođenje procesa učenja.

[9] http://raid6.com.au/~onlyjob/posts/arena/#speed



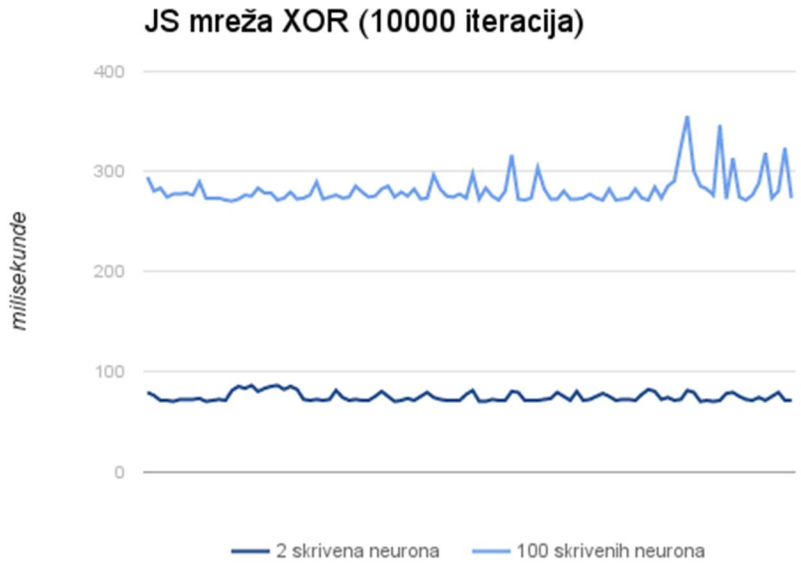
Slika 2.6 Vrijednosti vremena potrebnog za učenje JS mreže da prepozna XOR s manje od 0.05 vrijednosti greške prikazane u milisekundama. Četiri anomalije su isključene iz rezultata jer je mreži bilo potrebno više od minute za učenje, a zaustavilo se na 0.125 vrijednosti greške.



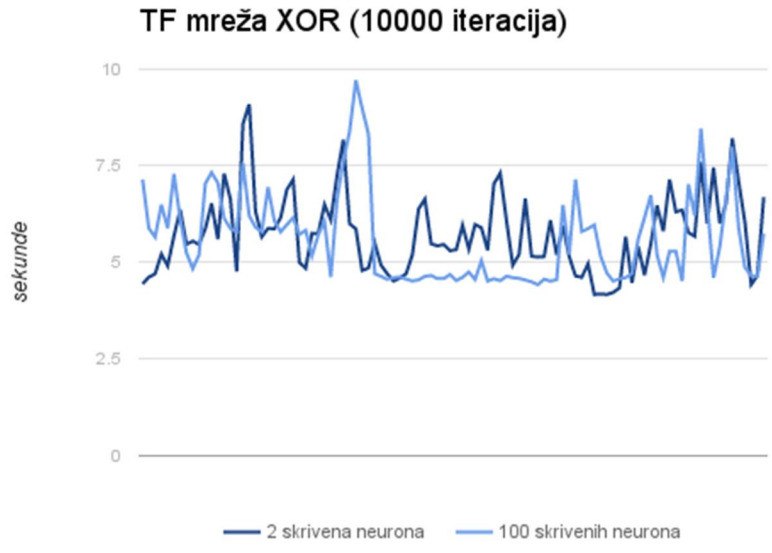
Slika 2.7 Vrijednosti vremena potrebnog za učenje TF mreže da prepozna XOR s manje od 0.05 vrijednosti greške prikazane u milisekundama. Zbog konzistentnosti podataka i u ovom grafu su isključene 4 najveće vrijednosti.

Iz rezultata je vidljivo da JS aplikacija (prosjek 401 ms) puno brže izvodi proces učenja nego TF aplikacija (prosjek 7165 ms) s jednostavnom strukturom. S složenijom strukturom vidi se da TF izvodi brže od JS aplikacije (prosjek 607 ms), i puno brže od jednostavne strukture.

Činjenica da TF za 2801.1% brže nauči mrežu sa složenijom strukturom od jednostavnije nameće pitanje je li JS mreža sporija zato što je okvir zakazao u izvedbi algoritama ili zato što je povećanje broja operacija drastično utjecalo na performanse. Na to pitanje će se odgovoriti sa sljedećom usporedbom gdje će se uspoređivati vrijeme potrebno da iste mreže odrade 10 000 iteracija što će dati uvida u stvarne performanse oba okvira.



Slika 2.8 Vrijednosti vremena potrebnog JS mreži da odradi 10000 iteracija učenja XOR-a.



Slika 2.9 Vrijednosti vremena potrebnog TF mreži da odradi 10000 iteracija učenja XOR-a.

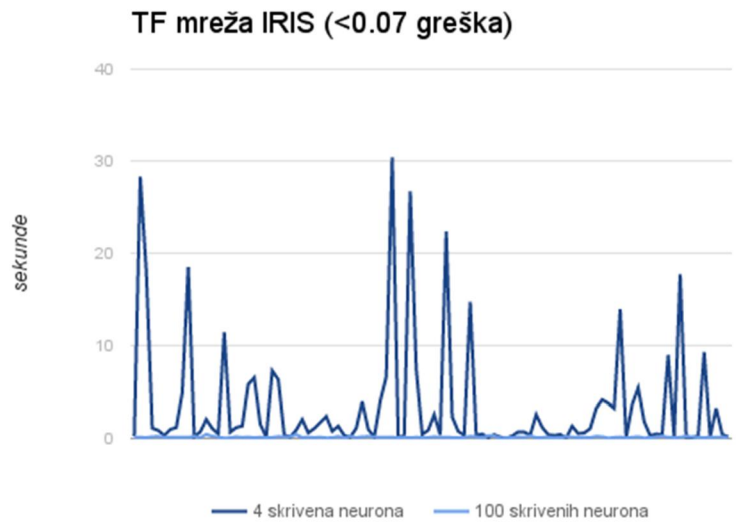
Prosječna vrijednost JS aplikacije za proces učenja do 10 000 iteracije mreže s dva skrivena neurona je bila 75 ms, a za 100 skrivenih neurona 281 ms što je uvećanje od 374.6%, dok je

to uvećanje za prošlu usporedbu bilo samo 151.3% što znači da je u prošloj usporedbi okvir zaista bolje naučio za XOR s više skrivenih neurona, ali je imao problema s performansama. S druge strane, izvedba TF mreže je relativno nepromijenjena za oba slučaja.

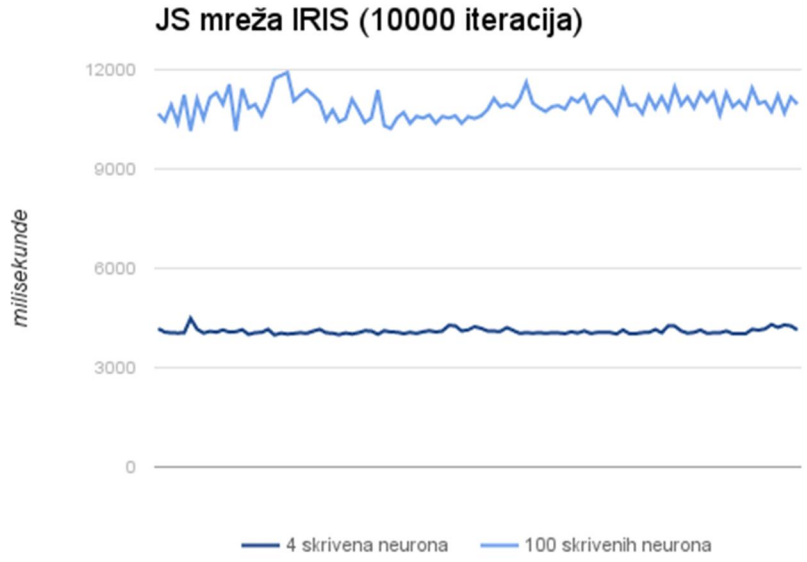
2.4.2. Usporedba učenja s IRIS podacima

Uz XOR izvela se usporedba učenja za prepoznavanje iris cvjetova iz IRIS skupa podataka, koji je popularno korišten za jednostavno prepoznavanje uzoraka. Sadrži vrijednosti širina i dužina čašičnih listića i latica od tri različita iris cvijeta. Ukupno se sastoji od 150 nelinearno razdvojivih skupova vrijednosti. Slijede rezultati istih usporedbi ali ovaj puta za IRIS skup podataka. Promijenjeni su brojevi ulaznih neurona na četiri, a izlaznih na tri, kao i prag za grešku na 0.07 zato što je TF aplikaciji trebalo eksponencijalno više vremena za učenje ispod 0.07.

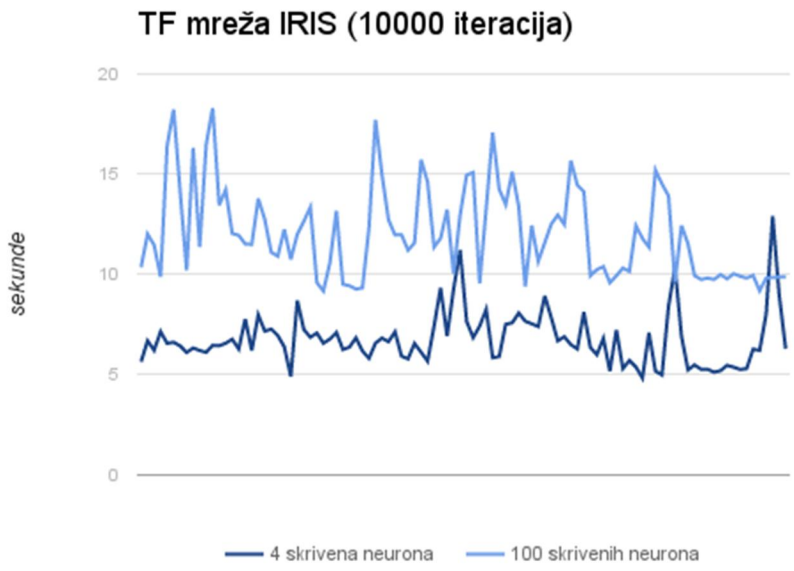
Slika 2.10 Vrijednosti vremena potrebnog za učenje JS mreže da prepozna iris cvjetove s manje od 0.07 vrijednosti greške prikazane u milisekundama.



Slika 2.11 Vrijednosti vremena potrebnog za učenje TF mreže da prepozna iris cvjetove s manje od 0.07 vrijednosti greške prikazane u milisekundama.



Slika 2.12 Vrijednosti vremena potrebnog JS mreži da odradi 10000 iteracija učenja s IRIS podacima.



Slika 2.13 Vrijednosti vremena potrebnog TF mreži da odradi 10000 iteracija učenja s IRIS podacima.

U ovom slučaju, JS aplikaciji je bilo potrebno mnogo više vremena za učenje do naznačene granične vrijednosti greške sa 100 skrivenih neurona nego s 4, s prosječno 1489.9% povećanjem za potrebno vrijeme, dok je vrijeme izvođenja za 10 000 iteracija povećano samo 266.3% što znači da je okvir imao poteškoća s učenjem s IRIS skupom podataka zbog lošije implementacije algoritama.

Dok je jedva vidljivo u prvoj usporedbi, vrijeme učenja TF mreže se smanjilo s prosječno 3528 ms na prosječno 140 ms, što je smanjenje od 2512.5%. U drugoj usporedbi, vrijeme da TF mreža dovrši 10 000 iteracija se povećalo za 178.6% i time utvrđuje da je konstantno bolje izvodila proces učenja s više skrivenih neurona, dok je vrijeme potrebno za učenje s IRIS skupom podataka povećano za skoro duplo.

Jedan od osnovnih primjera za uvod u TF je učenje mreže da prepozna brojeve koristeći se MNIST[10] skupom podataka koji se sastoji od slika ručno pisanih brojeva od nule do devet, gdje unos ima 784 vrijednosti što znači da je ovaj okvir optimiziran za rad s mnogo većim strukturama. Nažalost nije bilo moguće usporediti ove dvije mreže za učenje s MNIST skupom podataka jer se preglednik rušio kod učenja s 784-784-10 mrežom u JS aplikaciji.

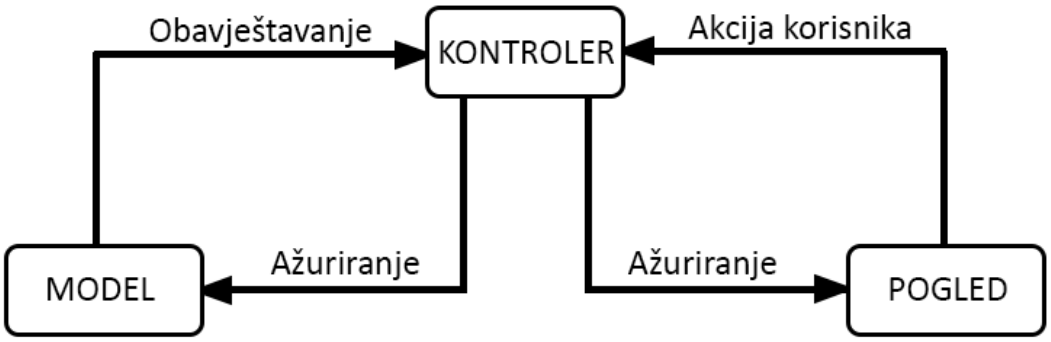
[10] LeCun, Yann, Corinna Cortes, and Christopher JC Burges. "The MNIST database of handwritten digits." 1998.

3. Python aplikacija

Web aplikacije u novije vrijeme ne žive u potpunosti u pregledniku kao JS aplikacija ovog diplomskog rada, nego u pozadini (engl. *backend*) se nalazi server koji obavlja 'teško dizanje' dok u pregledniku (engl. *frontend*) imaju jednostavni prikaz podataka dobivenih od servera. Ovakav tip aplikacije bi imao koristi za umjetne neuronske mreže jer bi se kompletniji okvirovi kako što je *TensorFlow* mogli povezivati s različim *frontend* okvirima i time se mogu iskoristiti prednosti oba sustava. Trenutno najpopularnija arhitektura za web aplikacije tog tipa je model-pogled-upravitelj (engl. *model-view-controller*) (MVC).

3.1. MVC arhitektura

MVC, kao što i samo ime naznačuje, se sastoji od tri komponente:



Slika 3.1 MVC komponente i njihova međusobna interakcija.

Model objekti učahure podatke specifične za aplikaciju i definiraju logiku i operacije koji upravljaju tim podacima i procesiraju ih. Npr. model može predstavljati neki lik iz igrice ili kontakt iz imenika. Objekti ovog tipa mogu imati na-jedan (engl. *to-one*) ili na-više (engl. *to-many*) veza s ostalim objektima modela, tako da je ponekad ova komponenta zapravo graf objekata. Većina stalnih podataka aplikacije bi se trebala nalaziti u objektima modela. Zbog toga što modeli predstavljaju znanje povezano sa specifičnom problematikom oni bi se mogli ponovno iskoristiti u poljima slične problematike. Idealno bi bilo, a vjerojatno će se i realizirati u budućnosti, da postoji standardni način spremanja naučenih modela neuronskih mreža tako da se mogu ponovno iskoristi u širokom rasponu aplikacija za različite svrhe. Npr. model neuronske mreže koji je naučen da prepoznaje ljudska lica u slikama ili videima bi se mogla iskoristiti za označavanje osoba na slikama na socijalnim mrežama kao i za

prepoznavanje ljudi u nadzornim sustavima. U slučaju MVC arhitekture idealno je da modeli nemaju eksplicitnu povezanost s pogledima koji prezentiraju te podatke i dopuštaju korisnicima njihovu izmjenu - ne bi se trebali baviti problematikom korisničkih sučelja i prezentaciji podataka.

Kako se modeli bave upravljanjem podataka, aplikacije se oslanjaju na različite vrste baza podataka da odrade spremanje objekata modela. Trenutno postoji mnogo specijaliziranih sustava za upravljanje bazama podataka od kojih neki najpopularnijih trenutno u industriji su:

- MySQL je sustav za upravljanje bazama podataka koji je besplatan i otvorenog je koda. Već dugo vremena je najpopularniji sustav u svijetu baza podataka. Prednosti su mu jednostavnost rukovanja i instalacije, te integracije u brojne druge sustave u koje se ubrajaju i web aplikacije.
- PostgreSQL je trenutno jedan od, ako ne i najnapredniji sustav za upravljanje bazama podataka. Također je besplatan i otvorenog koda. Ono po čemu se ovaj sustav ističe je to da je moguće raditi prilagođene funkcije u različitim programskim jezicima, te je moguće definirati vlastite tipove podataka, indeksa i slično.
- MongoDB je također besplatni sustav za upravljanje bazama i otvorenog je koda, no za razliku od prošla dva nije SQL sustav već NoSQL, tj. ne koristi se tablicama i redcima već kolekcijama i dokumentima. Dokumenti se sastoje od ključeva i vrijednosti za svaki ključ, a kolekcija je skup više dokumenata. Sustav dozvoljava dinamičnu strukturu tako da je moguće u kolekciji imati dokumente različitih, promjenjivih struktura. Sprema podatke u BSON format što je binarna reprezentacija podataka u JSON obliku.

Pogledi su objekti u aplikaciji koje korisnici vide. Ovakvi objekti znaju kako će se prikazati korisniku i kako će reagirati na korisnikovo djelovanje. Iako im je glavna uloga prikaz podataka korisniku iz modela aplikacije i omogućavanje promjena tih podataka, ipak nisu direktno povezani s modelom. Obično se sastoje od HTML, CSS i JavaScript koda koji se prikazuje u internet pregledniku.

Kontroleri su veza između jednog ili više pogleda i jednog ili više modela. Pomoću njih pogledi dobivaju uvid u podatke spremljene u bazi podataka. Ova komponenta je također veza između sustava i korisnika. Omogućava korisniku interakciju sa sustavom tako što

prikaže određene poglede u određenim mjestima na površini internet preglednika, potom šalje unose korisnika drugim dijelovima sustava bilo da je to prikaz drugog pogleda ili podešavanje i spremanje novih podataka u model(e). Također ova komponenta je ta koja mijenja poglede na osnovi promjena u modelima.

MVC je temeljni standard od kojega su nastale mnoge varijacije kao što su MVP (*model-view-presenter*), MVVM (*model-view-view-model*) i sl. koje počivaju na istim načelima kao i MVC, a to su:

- Modularnost
- Mogućnost ponovnog korištenja i proširenja koda
- Razdioba poslovne logike od prezentacijske logike
- Dozvoljava istodobni rad više programera na različitim dijelovima aplikacije
- Olakšano održavanje

3.2. Django

Drugi dio projekta vezanog za ova diplomski rad se sastoji od web aplikacije koja se pokreće na serveru i MVC je arhitekture. Za izradu aplikacije je korišten Django okvir koji je pisan u Python programskom jeziku i osnovan na model-pogled-predložak arhitekturi (engl. *model-view-template*) (MVT). MVT se razlikuje od MVC po tome što pogledi opisuju koji podaci se mogu vidjeti, a ne kako se podaci mogu vidjeti, zatim taj pogled delegira predlošku kako će se ti podaci prikazati.

Django dodatno promovira modularnost time što se jedan Django projekt sastoji od jedne ili više Django aplikacije. Django aplikacija je samostalna aplikacija koja je smještena u posebne direktorije unutar projekta tako da bi njeno isključivanje iz projekta značilo samo brisanje tog direktorija bez kompliciranih izmjena čitavog projekta.

Modele u Djangu čine Python klase koje nasljeđuju Djangovu `Model` klasu. Definiraju se unutar *models.py* datoteke koja se automatski stvori kada se stvori nova Django aplikacija. Osnovni dio svakog modela su polja koja se u Django definiraju tako da se stvore nove instance unaprijed definiranih polja i pošalju im se potrebni argumenti. Tako da za npr. polje u kojem će se upisivati neki kratki tekst je potreban sljedeći kôd:

```
from django.db import models
…
class Krug(models.Model):
      novo_polje = models.CharField(max_length=255)
…
```

U slučaju `CharField` polja potrebno je naznačiti kolika će biti maksimalna duljina unosa. Postoje i drugi parametri koji se mogu dodati i mogu biti specifični samo za neki tip polja, ili opći kao što je `unique` koji naznačuje treba li neko polje biti jedinstveno u tablici ili `default` koji naznačuje zadanu vrijednost nekog polja. Osim polja, uobičajena je praksa da se poslovna logika nalazi unutar modela, tako da je moguće definirati metode unutar klase koje mogu upravljati podacima u modelu. Npr. ako imamo model kruga i polja za njegove x i y koordinate kao i duljina polukruga, onda bi jedna takva metoda mogla biti izračun površine tog trokuta.

Nakon što se definiraju modeli potrebno je stvoriti migracijske datoteke pomoću naredbe:

```
python manage.py makemigrations
```

koja će stvoriti direktorij `migrations` s migracijskim datotekama u kojima se nalazi skup instrukcija pomoću kojih Django operira s bazom podataka. U slučaju primjera modela kruga, prva migracija će stvoriti novu tablicu unutar baze, a za svaku izmjenu modela potrebno je ponovno napraviti migraciju da bi Django napravi te izmjene unutar baze. Migracije se pokreću s naredbom:

```
python manage.py migrate
```

Nakon što se stvore tablice u bazi, s modelima se upravlja pomoću `objects` upravitelja, tako da za dohvaćanje kruga s ID vrijednosti 1 potrebno je pokrenuti naredbu:

```
Krug = Krug.objects.get(id=1)
```

što onda Django prevede u SQL instrukciju:

```
SELECT * FROM krug;[11]
```

Django pogledi se definiraju u *views.py* datoteci unutar aplikacija koja se također automatski stvori i dijele se na dvije vrste:

- Funkcionalni pogledi - definirani su u obliku Python funkcija i obično se koriste za neke jednostavnije radnje koje se mogu izvesti unutar jedne funkcije.
- Klasni pogledi (engl. *Class-based views*) (CBV) - definirani su u obliku Python klasa i koriste se za složenije radnje. Jedna od većih prednosti je nasljeđivanje koje olakšava njihovu implementaciju bez smanjivanja mogućnosti. Django dolazi s nekoliko ugrađenih klasa, npr. `TemplateView` s kojom se može u jednoj liniji naznačiti koji će se predložak koristiti i klasa će automatski podatke poslati tom predlošku da ih prikaže.

URL putanje se povezuju s pogledima unutar *urls.py* datoteke gdje se definiraju unutar `urlpatterns` liste u kojoj se svaki element sastoji od putanje pisane u obliku regularnog izraza i klase ili funkcije koja će rukovati zahtjevima što se šalju na tu URL putanju. Npr ako imamo pogled koji vraća sve spremljene stadione, URL putanja bi izgledala ovako:

```
urlpatterns = [
    url(r'^stadioni$',StadionView.as_view(),name='stadioni'),
]
```

Predlošci u Django prikazuju podatke korisniku i služe kao sučelje za interakciju između korisnika i aplikacije. Podatke koje predložak dobiva se zovu kontekst i u JSON su formatu. Predlošci mogu biti u različitim oblicima (*Django template*, *Mako*, *Jinja2* itd.), sličnost im je da su kombinacija Python koda i HTML-a a razlikuju se u sintaksi i mogućnostima. Prikaz imena svih stadiona u predlošku bi se mogao napraviti na sljedeći način s pretpostavkom da se nalazi Python riječnik sa svim stadionima i njihovim imenima u `stadioni` varijabli u kontekstu:

```
...
```

[11] Točna SQL instrukcija što se generira je opširnija i izvan okvira ovog diplomskog rada, ali osnovna operacija je ista.

```
<ul>
    <% for stadion in stadioni %>
    <li>{{ stadion.ime }}</li>
    <% endfor %>
</ul>
...
```

Python kôd se u Django predlošcima piše unutar `<% %>` oznaka, dok prikaz podataka se piše unutar `{{ }}` oznaka. Postoji i mnogo dodatnih oznaka i filtera koji se mogu koristiti u predlošcima, npr. za prvo slovo pretvoriti u veliko za svako ime koristi se `title` oznaka:

```
{{ stadion.ime | title }}
```

3.3. REST API

API je skraćenica od *Application Programming Interface*, i ovom slučaju se odnosi na HTTP API što je način na koji se dijele podaci aplikacije preko interneta. Npr. Twitter ima svoj API od kojeg se mogu zatražiti *tweetovi* da se prikažu u vlastitoj aplikaciji. U tome je moć API-a, tj. u mogućnosti da se podaci iz više različitih aplikacija mogu iskoristiti za neku novu aplikaciju. API-u se može pristupiti u točkama koje se zove krajnje ili pristupne točke (engl. *endpoint*) i preko njih se može komunicirati sa serverom preko HTTP protokola. Npr. ako bi željeli napraviti aplikaciju koja preko linkova dozvoljava objavljivanje, uređivanje, brisanje i pregled stadiona, imali bi sljedeće pristupne točke:

```
http://www.stadioni.com/pregled_stadiona
http://www.stadioni.com/dodaj_stadion?ime=Poljud
http://www.stadioni.com/uredi_stadion?id=5&ime=Wembley
http://www.stadioni.com/izbrisi_stadion?ime=Maksimir
```

Problem nastaje kada se različito implementiraju API-i pa se za svaku aplikaciju mora prolaziti kroz dokumentaciju i implementirati specifična logika za interakciju sa svakom aplikacijom. REST arhitektura se predstavila kao rješenje tog problema.

REST je postala najvažnija tehnologija za web aplikacije i njena važnost će samo rasti jer sve nove web tehnologije se razvijaju po načinu rada API-a. REST je skraćenica od *Representational State Transfer* iako je arhitektura za umrežene aplikacije za hipermediju,

koristi se primarno za izradu web servisa koji su jednostavni, održivi i skalabilni. Servis koji se temelji na REST arhitekturi se zove *RESTful* servis. REST ne ovisi o protokolu, ali *RESTful* aplikacije skoro je uvijek koriste s HTTP protokolom.

Postoji korelacija između najčešćih operacija u aplikacijama i HTTP metoda:

Akcija aplikacije	HTTP metoda[12]
Pregled	GET
Stvaranje	POST
Izmjena	PUT / PATCH
Brisanje	DELETE

Tablica 3.1 Usporedba čestih operacija u aplikacijama i HTTP metoda.

Veliku većinu vremena se koristi GET metoda, što u prijevodu znači "dobiti". Prikaz stranica u internet pregledniku se radi pomoću GET metode koja dobavlja čitavi sadržaj, tj. sve resurse i podatke koje aplikacija ili web stranica želi korisniku prikazati na određenom linku. Slanje podataka serveru, kao npr. ispunjavanje neke forme, se vrši pomoću POST metode. PUT i PATCH su metode za izmjene postojećih podataka na serveru od koji PUT mijenja sve podatke dok PATCH mijenja samo djelomično. Npr. ako imamo prikazane podatke o nekom korisniku i želimo izmijeniti broj telefona, PUT metoda će serveru nakon izmjene poslati sve prikazane podatke skupa s izmijenjenim brojem telefona, dok će PATCH poslati samo novi broj telefona i time PATCH štedi na internetskom prometu.

Primjer REST-a

Koristeći se primjerom sa stadionima, možemo stvoriti *RESTful* API na sljedeći način:

- Za implementaciju pristupne točke za pregled svih stadiona, URL bi ovako izgledao

  ```
  GET http://www.stadioni.com/stadioni
  ```

- Za stvaranje novog stadiona

  ```
  POST http://www.stadioni.com/stadioni
  ```

[12] Nisu navedene sve metode, još postoje i HEAD, TRACE i CONNECT.

```
Podaci:
      Ime = Poljud
```

- Dobivanje samo jednog stadiona se može realizirati koristeći ID tog stadiona

```
GET http://www.stadioni.com/stadioni/5
```

- Ažuriranje tog stadiona se može raditi na sljedeći način

```
PUT http://www.stadioni.com/stadioni/5
Podaci:
      Ime = Wembley
      Grad = London
```

- Brisanje stadiona bi se radi ovako:

```
DELETE http://www.stadioni.com/stadioni/87
```

Poznato je da u Sjedinjenim Državama košarkaški klubovi često dijele stadion s hokejaškim klubom iz istog grada pa bi bilo zgodno dodati i informacije o tome. U tome bi najbolje pomogli ugniježđeni resursi. REST upravlja isključivo s resursima, tako da je svaki navedeni URL link do resursa, u ovom primjeru stadiona. Ti resursi mogu biti i slike i audio-video zapisi itd. Implementacija jednog ugniježđenog resursa za upravljanjem klubovima koji igraju na istom stadionu se može implementirati na sljedeći način:

- Dobavljanje svih klubova koji igraju na stadionu s ID-em 78:

```
GET http://www.stadioni.com/stadioni/78/klubovi/
```

- Dobavljanje specifičnog kluba:

```
GET http://www.stadioni.com/stadioni/78/klubovi/1/
```

- Dodavanje novog kluba stadionu:

```
PUT http://www.stadioni.com/stadioni/78/
Podaci:
      Ime = New York Knicks
```

Još jedan bitan dio REST-a je povratna informacija koju zahtjev vrati nazad u obliku HTTP status koda. Ti status kodovi mogu biti:

- 2xx = uspješni zahtjev
- 3xx = usmjerenje na drugi URL
- 4xx = greška korisnika
- 5xx = greška servera

Od kojih su najčešći uspješni statusi:

- 200 - OK (zahtjev se uspješno izvršio)
- 201 - Stvoreno (uspješno izvršen POST zahtjev za dodavanje novog resursa)
- 202 - Prihvaćeno (uspješno izvršen DELETE zahtjev za brisanje resursa)

Dok su česte pogreške:

- 400 – Loš zahtjev (krivi podaci)
- 401 – Neovlašten zahtjev (ako je za raditi zahtjev potrebna ovjera autentičnosti)
- 404 – Resurs nije pronađeno (URL ne postoji)
- 405 – Nedozvoljena metoda
- 409 – Konflikt (kada se pokušava napraviti već postojeći resurs)

Kada se rade HTTP zahtjevi može se specificirati u kojem će formatu biti odgovor. Npr. ako je traženi resurs web stranica, onda će se zahtijevati odgovor u HTML obliku, za sliku će to biti *image* itd. JSON je odabran kao format za REST API zahtjeve, zbog svoje jednostavne i čitljive sintakse s kojom je jednostavno upravljati. Kod REST zahtjeva se JSON format specificira na sljedeći način:

```
GET /stadioni
Accept: application/json
```

A odgovor bi bio u sljedećem formatu:

```
[{
    id: 5,
    ime: 'Poljud'
},
...
{
    id: 78,
    ime: 'Madison Square Garden',
    klubovi: {
        id: 1
        ime: 'New York Knicks'
    }
}]
```

U aplikaciji ovog diplomskog rada je korišten Django REST okvir (engl. *Django REST framework*) (DRF) koji upravlja REST dijelom aplikacije i pruža mnogo dodatnih pogodnosti.

3.4. Struktura aplikacije

Drugi dio aplikacije je rađen u Python-u uz pomoć Django okvira. Cilj je bio iskoristiti snažni i punopravni okvir za rad s umjetnim neuronskim mrežama za potrebe web aplikacije i time spojiti mogućnosti tog okvira s dostupnošću web aplikacije. Za okvir namijenjen upravljanju neuronskim mrežama odabran je *TensorFlow* zbog mnogobrojnih značajki i činjenice da je trenutno najpopularniji okvir za rad s neuronskim mrežama, dok je za web dio odabran Django zbog nekoliko razloga:

1. Pisan je u Python programskom jeziku u čemu je pisan i *TensorFlow*.
2. Dolazi s jednostavnim serverom tako da nije potrebno dodatno instalirati i konfigurirati taj dio.
3. Modeli funkcioniraju preko ORM-a (*Object-relational mapping*) što omogućava pisanje modela i upita na bazu u Python-u, te pruža mogućnost da se vrsta baze podataka može vrlo jednostavno zamijeniti s jednom linijom, tj. mijenja se sustav koji služi kao most između baze podataka i aplikacije i prevodi Python kôd u instrukcije za odabranu bazu podataka (u slučaju MySQL baze, Python kôd bi se preveo u SQL instrukcije).
4. Jednostavno uključivanje *TensorFlow* okvira u aplikaciju.

Osim navedenih, glavnih tehnologija, korišteno je i nekoliko dodatnih.

- Git i GitHub. Git je sustav za kontrolu verzija programa (engl. *Version control system*) pomoću kojega je moguće lako pratiti promijene koje se rade na projektu i jednostavno je vratiti čitavi sustav na neku stariju veziju. Jako je popularan u svijetu programiranja jer programeri mogu bez puno problema raditi na istom projektu istodobno bez straha da će neke od promjena nestati zbog djelovanja drugih programera. GitHub je web aplikacija za rad s git-om online i najpopularnija je platforma za dijeljenje otvorenog koda.
- Virtualno okruženje (engl. *Virtual environment*) ili skraćeno *virtualenv* je Python-ov *sandbox* sustav u kojem je moguće instalirati pakete koji neće utjecati na ostatak sustava, tj. čim se *virtualenv* deaktivira, pristup instaliranim paketima koje su instalirani unutar *sandbox* sustava nije moguć. Time se omogućava instaliranje paketa koji bi mogli doći u konflikt s paketima instaliranim na operacijskom sustavu,

kao npr. ako je potrebno instalirati noviju verziju Django okvira a nekoliko projekta već koristi staru verziju i ne bi se mogli pokrenuti na novoj.

- DRF - ovo je okvir koji omogućava jednostavan rad s REST API-ima. Dolazi kao dodatak Django-u i ima ugrađeno mnogo funkcionalnosti koji olakšavaju izradu i rad s REST API-em. Za ovaj projekt su se koristili DRF-ova funkcionalnost za serijalizaciju podataka i različite klase za prikaz podataka unutar jedne API pristupne točke.

Prvi dio projekta, JS aplikaciju, je bilo jednostavno uključiti kao novu aplikaciju unutar Django projekta. Za pristup toj aplikaciji stvorene su sljedeće dvije lokacije:

- Višeslojni perceptron:

  `/js/mlp`

- LSTM:

  `/js/lstm`

TensorFlow dio aplikacije funkcionira isključivo preko komunikacije s API pristupnim točkama, čime se pojednostavljuju prikazi informacija time što se mogu priključiti različiti paketi za prikaz i vizualizaciju podataka u koje spada i onaj korišten za JS aplikaciju i još važnije, jednostavno je ostvariti komunikaciju s drugim, nevezanim aplikacijama. Trenutno postoji samo jedan model, ili tablica u bazi, koji sprema neuronsku mrežu oblika višeslojnog perceptrona. Prije učenja se spremaju informacije o strukturi mreže, načinu učenja i ovlastima, a nakon, informacije o naučenoj mreži (putanja do spremljenog *TensorFlow* modela, zadnja vrijednost funkcije greške itd.).

Zbog jednostavnosti, prije učenja mreže sustav nema kontakta s *TensorFlow* okvirom. Tek kada učenje započne i pošalju se podaci za učenje, ako već nije spremljena putanja do podataka, sustav stvori graf izvedbe s onom strukturom i hiper-parametrima mreže koji su spremljeni pri stvaranju objekta modela i započne proces učenja. Zbog toga što, trenutno, *TensorFlow* ima mogućnost spremati mrežu samo kao binarnu datoteku, spremanje mreže se radi tako da se unutar `saved_models` direktorija spremaju te binarne datoteke tako da se svaka spremi u poddirektorij koji dobiva naziv po ID-u modela spremljenog u bazi.

Korištenje naučene mreže se vrši tako da se na naznačeni API pristupnim točkama, koji u sebi sadrži ID modela, pošalje lista ulaznih podataka za koje se želi dobiti izlaz i sustav vrati izlazne vrijednosti koji je ta mreža izračunala za dane ulazne podatke.

Pristupne točke se mogu podijeliti na javne i privatne. Javnima može svatko pristupati i koristiti dok privatne mogu samo ili korisnik koji ju je stvorio ili grupa kojoj taj korisnik pripada. Razlog za postojanje ograničenja pristupa modelima je taj što neki korisnici ili istraživačke grupe možda ne bi željeli da itko izvan njih ima pristup modelima jer, npr. rade na razvoju modela za prepoznavanje varanja u financijskom sektoru što bi trebalo biti privatno, ili ako se koriste podacima za učenje mreže koji se sastoje od privatnih medicinskih zapisa neke bolnice koja želi napraviti model za unaprijeđeno dijagnosticiranje neke učestale bolesti.

3.4.1. Javne API pristupne točke

Javne API pristupne točke za *TensorFlow* dio su sljedeći:

- Za prikaz svih TF modela šalje se zahtjev s GET metodom na sljedeću pristupnu točku:

  ```
  /api/public/models/
  ```

- Za dodavanje novog TF modela šalje se POST zahtjev na tu istu pristupnu točku.

  Podaci koji se šalju na ovoj pristupnoj točki preko POST metode su:

Ime	Tip podatka	Opis
num_inputs	Cijeli broj	Broj ulaznih neurona.
num_outputs	Cijeli broj	Broj izlaznih neurona.
num_hidden	Lista cijelih brojeva	Lista brojeva skrivenih neurona gdje prvi element označava koliko je neurona u prvom skrivenom sloju, drugi u drugom itd.
learning_rate	Pozitivan decimalni broj	Vrijednost stope učenja.
cost	Tekst	Funkcija greške. Može biti ili „MSE“ (srednja kvadratna

		greška) ili „ENTROPY“ (unakrsna entropija).
optimizer	Tekst	Algoritam za učenje mreže. Može biti „GD“ (gradijent pada).
activation	Tekst	Aktivacijska funkcija neurona. Može biti „SIGM“ (sigmoid) ili „TANH“.
permission_type	Cijeli broj	Tip ovlasti za ovaj model. Može biti 0 (javni model), 1 (samo korisnik ima pravo pristupa), 2 (grupa kojoj korisnik pripada ima pravo pristupa).

Kada se svi ti podaci pošalju stvori se novi objekt klase `TFModel` koji unutar sebe sprema sve poslane podatke i sadrži pet dodatna polja:

- `trained` – naznačuje da li je model naučen
- `file_path` – naznačuje gdje je naučeni model spremljen
- `latest_cost` – zadnja vrijednost funkcije greške
- `permission_user` – ako je odabran tip ovlasti 1, u ovo polje se dodaje ID korisnika ovlaštenog za pristup ovom model
- `permission_team` – ukoliko je odabran tip ovlasti 2, u ovo polje se dodaje ID grupe ovlaštene za pristup ovom modelu

- Za prikaz detalja jednog modela šalje se GET zahtjev na sljedeću pristupnu točku:

  `/api/public/models/<ID modela>`

- Za korištenje nekog specifičnog modela za predvidjeti izlaz od poslanih ulaznih podataka šalje se POST zahtjev na tu pristupnu točku s `input_data` parametrom koji je lista listi u kojima se nalaze ulazni podaci. Ova pristupna točka vraća natrag listu izlaznih vrijednosti koje je mreža izračunala za svaku od ulaznih vrijednosti koje je dobila.

- Za učenje određenog modela potrebno je poslati POST zahtjev na sljedeću pristupnu točku s podacima iz tablice:

```
/api/public/models/<ID modela>/train/
```

Ime	Tip podatka	Opis

training_data	Lista	Podaci za učenje. Oblik:
		`[[`
		`  [1, 2, 3],`
		`  [1, 2]`
		`],`
		`...`
		`]`
		gdje se svaka podlista unutar liste sastoji od liste ulaznih podataka i liste ciljanih izlaza.

min_error	Decimalni broj	Vrijednost praga funkcije greške. Ako vrijednost greške prijeđe prag, učenje završava.
iterations	Cijeli broj	Maksimalni broj iteracija učenja kojih će mreža odraditi. Ako se ovaj podatak ne pošalje, mreža se ograniči na 100 000 iteracija

3.4.2. Red zadataka

Kako složenost struktura mreža i broj podataka za učenje raste tako raste i vrijeme potrebno za njihov učenje pa je vrlo lako preopteretiti sustav da postane nedostupan. Rješenje ovog problema je uvedeno u obliku reda (engl. *Queue*). U aplikaciji se stvara *singleton* objekt (objekt koji može biti instanciran samo jednom i ne može se izbrisati) pokretanjem migracija, naime u Django-u je moguće vlastoručno stvarati migracije što se obično radi kada se želi popuniti baza s unaprijed definiranim podacima, u ovom slučaju je to *Queue* objekt. Ovaj objekt u sebi sadrži vrijednosti za pauziranje izvođenja (polje pozitivnog cijelog

broja nazvano *timeout* koje označava koliko će se vremenski izvođenje pauzirati između zadataka, i boolean polje *pause* koji ako je istinito (*True* vrijednost) pauzira izvođenje zadataka dok se ne prebaci u laž (*False* vrijednost)). Osim toga, *queue* objekt sadrži i metode za pokretanje zadataka.

Zadatak (engl. *Task*) je Django model u kojemu se definiraju zadaci koje aplikacije treba izvoditi (trenutno samo učenje mreža). Model se sadrži od sljedećih polja:

- `priority` – cijeli broj koji označava visinu prioriteta za taj zadatak. Administrator pri stvaranju novih korisnika dodjeljuje prioritet tom korisniku koji sustav iskoristi za popuniti ovo polje pri stvaranju novog zadatka
- `created_at` – vrijeme i datum kada je zadatak stvoren
- `started_at` – vrijeme i datum kada je zadatak započeo
- `finished_at` – vrijeme i datum kada je zadatak završio
- `model` – strani ključ koji označava za koji `TFModel` objekt se izvršava zadatak
- `training_data_csv_name` – ime CSV datoteke u kojoj su spremljeni podaci za učenje
- `min_error` – prag funkcije greške
- `iterations` – prag za broj iteracija
- `status` – status zadatka koji može biti: 0 (čeka da se izvrši), 1 (izvršava se), 2 (završen), 3 (greška se desila u izvršavanju zadatka)
- `error` – u slučaju greške ovdje se spremi koja se točno greška dogodila

Da bi se omogućilo izvršavanje zadataka u redu potrebno je prebaciti vrijednost polja *QUEUE* unutar *settings.py* datoteke na *True,* inače bi se učenja mreža izvršavale kako bi zahtjevi za njim pristizali. Kada se vrijednost prebaci, svaki zahtjev za učenjem neke mreže stvara zasebni objekt *Task* modela tj. zadatak, postavi se stanje na 0 i čeka se red. *Queue* objekt izvršava zadatke poredane po vremenu stvaranja i prioritetu tako da se prvi izvršavaju oni koji su prije stvoreni i imaju najveći prioritet. Zadatak u tom trenutno se prebacuje u stanje 1 i izvršava proces učenja mreže. Kada se izvrši proces zadatak stanje prebacuje na 2 (ili 3 ako se desila neka greška), sprema sve ostale podatke u svojim poljima i poziva *Queue* objekt da započne sljedeći zadatak. *Queue* objekt radi provjeru je li postavljena pauza, u slučaju da nije provjerava zadnja tri zadatka koja su se odradila. Ako su sva tri zadatka imala grešku u izvođenju znači da sustav ne radi pravilno i zaustavlja se. U slučaju da zadnja tri zadatka nisu sva imala grešku, *Queue* objekt provjerava izvodi li se trenutno neki zadatak

(postoji li zadatak sa statusom 1). U slučaju da postoji preskače se daljnje izvođenje, a ako ne postoji, *Queue* objekt pokreće sljedeći zadatak koji je po redu.

Zadatke je moguće pratiti slanjem GET zahtjeva na sljedeću API pristupnoj točki:

```
/api/tasks/
```

3.4.3. Privatne API pristupne točke

Za razliku od javih, privatnim pristupnim točkama mogu pristupiti samo korisnici koji se prijave (preko standardne Django forme za prijavu). Privatne pristupne točke se isto nazivaju kao i javne samo što je umjesto `public` rute postavljen naziv `private`, pa tako npr. za ispis svih javnih i privatnih modela kojima korisnik ili grupa čiji je korisnik član imaju pristup se nalazi na sljedećoj lokaciji:

```
/api/private/models/
```

Kod ovih pristupnih točki moguće je naznačiti kakve ovlasti su potrebne pri rukovanju pojedinih mreža pri njihovom stvaranju. Osim toga, pri stvaranju zadataka učenja mreža, zadatcima se dodjeljuje prioritet koji se dobije iz podataka o korisniku i time ti zadaci dobiju prioritete koji su u najmanju ruku veći od javnih zadataka čiji prioritet je najniži.

Upravljanje korisnicima i grupama kao i dodjeljivanje razina prioriteta radi administrator preko standardnog Django administracijskog alata.

3.4.4. Prednosti i nedostaci

Ovako napravljeni sustav je dobar iz više razloga:

1. Podaci se mogu prikazati na više načina, tj. različiti okviri za prikaz podataka se mogu spojiti na API i prilagoditi interakciju korisnika i sustava onako kako im odgovara. Isto tako mogu se napraviti bilo kakvi oblici forma za unos podataka koje komuniciraju preko API pristupnih točaka sa sustavom te njihova implementacija, izgled, raspored itd. ne utječe na rad sustava.
2. Aplikacija je rađena poštujući već ustaljenu praksu i arhitekturu velike većine modernih web aplikacija (većina današnjih web aplikacija funkcionira na načelima MVC i REST API arhitektura).

3. Zbog tog što je rađena na ustaljeni način, ostale, nevezane aplikacije se mogu spajati na ovu bez previše izmjena u načinu rada (jer i ostale su vrlo vjerojatno na MVC i REST API primjeru rađene) i mogu očekivati standardizirani način komunikacije što omogućava veliku upotrebljivost ovog sustava kao *third-party* aplikaciju u mnogim drugim aplikacijama koje ne nužno moraju biti vezane uz web tehnologije.

Naravno ovakav sustav nije savršen i ima nekoliko nedostataka:

1. Nedostatak standardiziranog formata neuronskih mreža za njihovu ponovnu uporabu u drugim aplikacijama. Ovo nije nedostatak ovog sustava, već nedostatak čitavog web ekosustava treniranja i uporabe neuronskih mreža jer je ovo polje vrlo mlado i čak ni najpopularniji i najrazvijeniji okviri nemaju jedinstven sustav spremanja mreža. Rješenje bi bilo razviti standard za ostvarivanje tako nečega što je jednostavno za jednostavne modele koje koristi ovaj projekt, ali teško za moderne strukture mreža.
2. Sustav u trenutnom stanju ne može stvoriti ni trenirati složenije neuronske mreže od višeslojnog perceptrona. Da bi sustav mogao raditi s nekom drugom strukturom, npr. s LSTM mrežama, potrebno je ručno napraviti novi model i ručno prilagoditi njegovo uvježbavanje. Takav način rada maksimalno ograničava upotrebu sustava za trenutne potrebe naprednijih korisnika jer za postizanje što veće točnosti nerijetko se rade male izmjene strukture neuronske mreže što može značiti ubacivanje specijaliziranog sloja neurona što sustav trenutno nije u mogućnosti napraviti.
3. Sustav je trenutno dosta krhak jer čim se poremete spremljeni uvježbani modeli onemogući se proces učenja za sve ostale modele i često administrator mora ručno uklanjati nastali problem i nerijetko iznova stvarati modele. Ovakav način rada nije pogodan za višekorisničko korištenje sustava.
4. Pojedini okviri za prikaz podataka imaju ograničeno koliko mogu čekati na odgovor servera što može stvoriti probleme pri procesu učenja jer taj proces nerijetko traje puno dulje nego što je ograničenje čekanja odgovora. Stoga sekvencijalni način rada nije moguć za te okvire.

Zaključak

Web tehnologije su dovoljno uznapredovale da mogu parirati klasičnim programskim jezicima u izradi i izvođenju procesa učenja neuronskih mreža s obzirom na performanse. Kao što je spomenuto u uvodu, strojno učenje se primjenjuje u eksponencijalno rastućem broju poslova. Pojednostavljivanje procesa izrade i učenja neuronskih mreža bi strojno učenje približilo većem broju ljudi i time njima omogućilo da se služe strojnim učenjem za svoje osobne potrebe. U prvom dijelu rada, u JS aplikaciji, dokazano je da program koji je rađen u isključivo web tehnologijama može u nekim pogledima parirati programskom okviru koji se izvodi u strojnom jeziku. Čitav proces izrade i učenja neuronske mreže je sveden na unos nekoliko vrijednosti što je puno jednostavnije od ručnog stvaranja neuronske mreže. Preciznost mreže i ukupno vrijeme izvođenja procesa učenja nije previše odstupalo u usporedbi s programskim okvirom, no nedostatak sustava je što je ograničen na samo dvije različite arhitekture neuronskih mreža.

U drugom dijelu je dokazano da je moguće programski okvir korišten u prvom dijelu iskoristiti da se izvodi u pozadini servera koji omogućava stvaranje i učenje neuronskih mreža na webu. Ovaj sustav pojednostavljuje stvaranje i učenje neuronskih mreža u programskom okviru i time zadržava jednostavnost korištenja što ima i JS aplikacija a pri tome ima i prednosti u performansama što ih donosi programski okvir *TensorFlow*, no, kao i sustav iz prvog dijela, i ovdje je nedostatak ograničenost izbora arhitektura.

U oba slučaja glavni nedostatak je ograničenost izbora arhitektura. Ovaj izbor se može povećati ručnim dodavanjem novih arhitektura ali time bi sustav još uvijek bio ograničen. Način na koji bi se riješila ograničenost je dati korisniku veću slobodu i mogućnost stvaranja arhitektura neuronskih mreža, no time bi se povećala složenost sustava i zbog sve veće složenosti neuronskih mreža koji se danas koriste samo je pitanje vremena kada bi taj sustav zastario do te mjere da nije više upotrebljiv za modernu uporabu.

Literatura

[1] A. Rothwell, L. Jagger i W. Denn, »Intelligent SPAM detection system using an updateable neural analysis engine«. Patent US6769016 B2, 27 Jul 2004.

[2] S. Huang i W. Ren, »Use of neural fuzzy networks with mixed genetic/gradient algorithm in automated vehicle control,« *IEEE Transactions on Industrial Electronics,* svez. 46, br. 6, 1999.

[3] M. Fourment i M. Gillings, »A comparison of common programming languages used in,« *BMC Bioinformatics,* 2008.

[4] W. McCulloch i W. Pitts, »A logical calculus of the ideas immanent in nervous activity,« *The bulletin of mathematical biophysics,* svez. IV, br. 4, p. 115–133, 1943.

[5] D. O. Hebb, The organization of behavior: A neuropsychological theory, New York: Wiley & Sons, 1949.

[6] M. Minsky, *A neural-analogue calculator based upon a probability model of reinforcement,* Cambridge: Harvard University Psychological Laboratories, 1952.

[7] »History of Information,« [Mrežno]. Available: http://www.historyofinformation.com/expanded.php?id=4343. [Pokušaj pristupa 2016].

[8] C. Dr. Tappert, »Pace University,« [Mrežno]. Available: http://csis.pace.edu/~ctappert/srd2011/rosenblatt-contributions.htm. [Pokušaj pristupa 2016].

[9] »Wikipedia,« Wikimedia Foundation Inc., [Mrežno]. Available: https://en.wikipedia.org/wiki/ADALINE. [Pokušaj pristupa 2016].

[10] M. Minsky i S. Papert, Perceptrons, M.I.T. Press, 1969.

[11] J. J. Hopfield, »Neural networks and physical systems with emergent collective computational abilities,« *Proceedings of the national academy of sciences,* svez. 8, br. 79, pp. 2554-2558, 1982.

[12] M. A. Nielsen, Neural networks and Deep Learning, Determination Press, 2015.

[13] J. MacQueen, »Some methods for classification and analysis of multivariate observations.,« u *Fifth Berkeley symposium on mathematical statistics and probability.*, Berkeley, 1967.

[14] R. S. Sutton i A. G. Barto, Reinforcement Learning:, Cambridge: MIT Press, 1998.

[15] J. A. Bullinaria, »Networks of Artificial Neurons, Single Layer Perceptrons; School of Computer Science, University of Birmingham,« 2015. [Mrežno]. Available: http://www.cs.bham.ac.uk/~jxb/INC/l3.pdf. [Pokušaj pristupa 2016].

[16] »Wikipedia,« Wikimedia Foundation Inc., [Mrežno]. Available: https://en.wikipedia.org/wiki/Perceptrons_(book). [Pokušaj pristupa 2016].

[17] S. O. Haykin, »Multilayered Perceptrons,« u *Neural Networks and Machine Learning (3rd edition)*, Pearson, 2008.

[18] J. Heaton, »Feedforward Neural Networks,« u *Introduction to Neural Networks for Java, 2nd Edition*, Heaton Research, Inc, 2008, pp. 158-159.

[19] X. Glorot i Y. Bengio, »Understanding the difficulty of training deep feedforward neural networks,« u *International Conference on Artificial Intelligence and Statistics*, Chia Laguna Resort, Sardinia, Italy, 2010.

[20] A. Karpathy, F. Li i J. Johnson, »CS231n Convolutional Neural Network for Visual Recognition,"." Online Course (2016).,« University of Stanford, 2016. [Mrežno]. Available: http://cs231n.stanford.edu/. [Pokušaj pristupa 2016].

[21] R. D. Wilson i T. R. Martinez, »The general inefficiency of batch training for gradient descent learning,« *Neural Networks,* svez. 16, br. 10, p. 1429–1451, 2003.

[22] L. Bottou, »Large-Scale Machine Learning with Stochastic Gradient Descent,« u *International Conference on Computational Statistics*, Paris, 2010.

[23] A. Ng, »Machine Learning,« Coursera, 2016. [Mrežno]. Available: https://www.coursera.org/learn/machine-learning/lecture/9zJUs/mini-batch-gradient-descent. [Pokušaj pristupa 2016].

[24] S. Ruder, »sebastianruder.com,« 19 Jan 2016. [Mrežno]. Available: http://sebastianruder.com/optimizing-gradient-descent/index.html#minibatchgradientdescent. [Pokušaj pristupa 2016].

[25] »Wikipedia,« Wikimedia Foundation Inc., [Mrežno]. Available: https://en.wikipedia.org/wiki/Recurrent_neural_network.

[26] S. Hochreiter i J. Schmidhuber, »LONG SHORT-TERM MEMORY,« *Neural Computation,* svez. 9, br. 8, pp. 1735-1780, 1997.

[27] F. A. Gers i J. Schmidhuber, »Recurrent nets that time and count,« u *IEEE-INNS-ENNS International Joint Conference*, Barcelona, 2000.

[28] K. Cho, B. van Merrienboer i e. al., »Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation,« *arXiv,* 2014.

[29] H. Woo, »Red Hat Developers,« RedHat, 31 May 2016. [Mrežno]. Available: http://developers.redhat.com/blog/2016/05/31/javascript-engine-performance-comparison-v8-charkra-chakra-core/. [Pokušaj pristupa 2016].

Sažetak

Ovaj rad obrađuje relativno novo područje izrade i prikaza višeslojnih neuronskih mreža korištenjem web tehnologija i radi usporedbu između uvježbavanja neuronskih mreža pomoć web tehnologija i trenutno najpopularnijih rješenja. U prvom dijelu rada objašnjava se što su neuronske mreže, kako je došlo do njihovog razvoja, koji su neki od popularnih tipova učenja i po čemu se razlikuju i kako radi jedna popularna arhitektura neuronske mreže 'plitkog učenja' i jedna 'dubokog učenja', obje koje se mogu stvoriti, uvježbavati i koristiti u projektu ovog rada. Posebno se obrađuje algoritam učenja koji je, uz male razlike, korišten za obje arhitekture, u svrhu razumijevanja složenosti procesa učenja preko kojega će se kasnije uspoređivati performanse različitih implementacija sustava za rad s neuronskim mrežama. U sljedećem dijelu prikazuje se dio projekta ovog rada, tj. aplikacije koja je izrađena isključivo u web tehnologijama kao dokaz da su se web tehnologije dovoljno razvile za omogućavanje razvoja takvih aplikacija. Kako je 'duboko učenje' postalo jako popularno u području umjetne inteligencije pa tako i u svijetu, kao primjer da web tehnologije mogu podržati i tu vrstu odabrana je LSTM (*Long Short-term Memory*) arhitektura kao jedne od popularnih u području obrade prirodnog jezika. Nakon toga performanse se uspoređuju s *TensorFlow*-om, trenutno najpopularnijim rješenjem za rad s neuronskim mrežama i koji uvježbavanje mreže izvodi u strojnom jeziku. U zadnjem dijelu prikazuje se drugi dio projekta koji se sastoji od web aplikacije koja se u pozadini koristi *TensorFlow*-om za brže i efikasnije uvježbavanje i korištenje neuronskih mreža. Aplikacija se temelji na modernim praksama razvoja web aplikacija i napravljena je na način koji osigurava jednostavno nadograđivanje.

Summary

This thesis explores the relatively new field of constructing and displaying artificial neural networks using web technologies and compares the performance of neural networks trained with web technologies and with the currently most popular machine learning framework. In the first part of this thesis neural networks are explained, what they are, how they evolved, what some of the more popular learning types are and what their differences are and how a shallow learning and a deep learning neural network operate, both of which can be created, trained and used in the practical part of the thesis. The learning algorithm is particularly explored, which is the one that the both supported neural network architectures use, with slight difference, all to demonstrate the complexities included in the training process which is later used to compare the performances of two different implementations of neural network applications. The next part of the thesis shows the web application that was created using purely web technologies as a proof that the technologies have evolved sufficiently to allow creating such applications. Because deep learning became very popular in the field of artificial intelligence and is widely used in the world, as an example that the web application can support that kind of neural networks, the LSTM (Long Short-term Memory) architecture is selected. The LSTM network is popular in the field of natural language processing. After that the performance of the web application is compared with *TensorFlow,* the currently most popular neural network framework which performs the training by compiling the code and then running it. In the last part of this thesis explores the second part of the project which uses *TensorFlow* for faster and more efficient training and usage of neural networks. The application is based on current web development standards and is created to be easily extendable.

Skraćenice

API (engl. *Application Programming Interface*) – programsko sučelje koje omogućava pristup sustavu za kojega je napisano.

BGD (engl. *Batch Gradient Descent*) – varijanta algoritma gradijenta pada.

BP (engl. *Backpropagation*) – algoritma unazadne propagacije.

BPTT (engl. *Backpropagation through time*) – algoritam unazadne propagacije kroz vremenski period.

CSS (engl. *Cascading Style Sheet*) – stilovi koji određuju izgled web stranice.

GD (engl. *Gradient Descent*) – algoritam gradijenta pada.

GRU (engl. *Gated Recurrent Unit*) – vrsta LSTM mreže.

HTML (engl. *Hiper Text Markup Language*) – skup oznaka za prikaz podataka na webu.

HTTP (engl. *Hypertext Transfer Protokol*) – protokol za web.

JS (engl. *JavaScript*) – Javascript programski jezik.

JSON (engl. *JavaScript Object Notation*) – format za razmjenu podataka.

LSTM (engl. *Long Short-term Memory*) – vrsta ponavljajućih neuronskih mreža.

MBGD (engl. *Mini Batch Gradien Descent*) – varijanta algoritma gradijenta pada.

MSE (engl. *Mean Squared Error*) – funkcija za izračunavanja greške neuronske mreže.

MVC (engl. *Model View Controller*) – arhitektura web aplikacija.

MVT (engl. *Model View Template*) – arhitektura web aplikacija.

MNIST (engl. *Mixed National Institute of Standards and Technology database*) – skup podataka koji označavaju ručno napisane brojeve.

REST (engl. *Representational State Transfer*) – arhitektura web aplikacija.

RNN (engl. *Recurrent Neural Network*) – vrsta neuronskih mreža.

SGD (engl. *Stochastic Gradient Descent*) varijanta algoritma gradijenta pada.

TF (engl. *TensorFlow*) – programski okvir za stvaranja i upravljanje neuronskim mrežama-

URL (engl. *Uniform Resource Locator*) – adresa resursa na internetu.