UNIVERSITY OF MARIBOR

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Tin Kramberger

Neuronske mreže

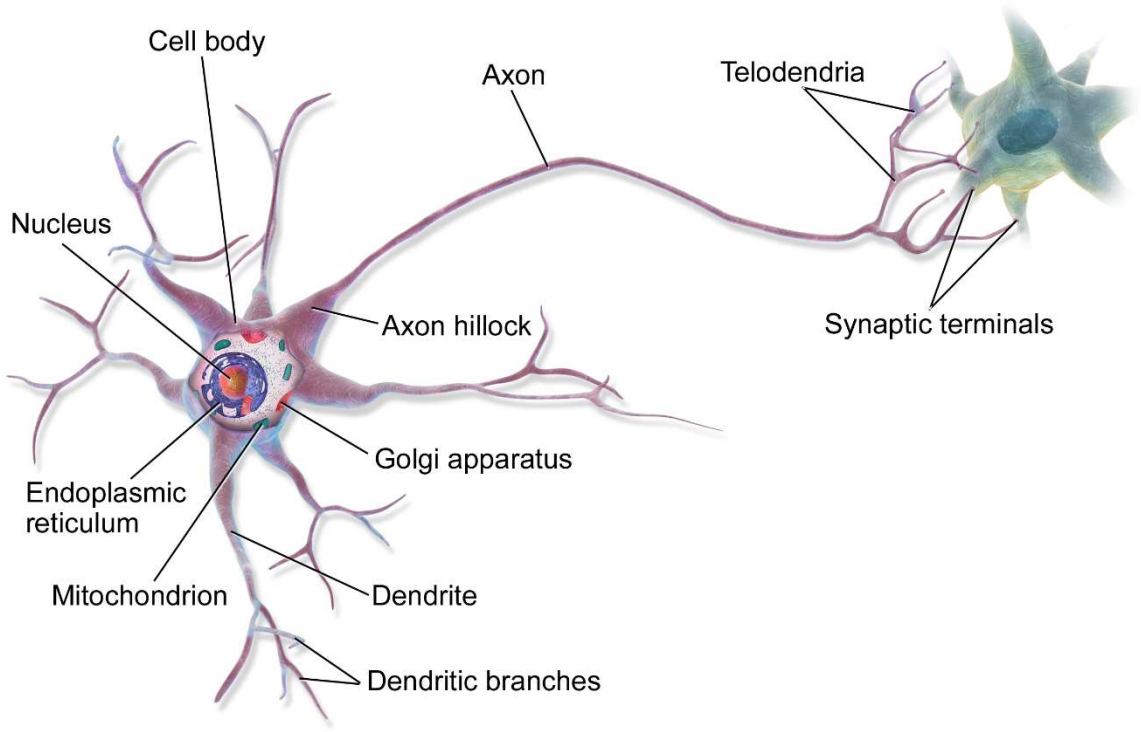
Maribor, 2019.

Sadržaj

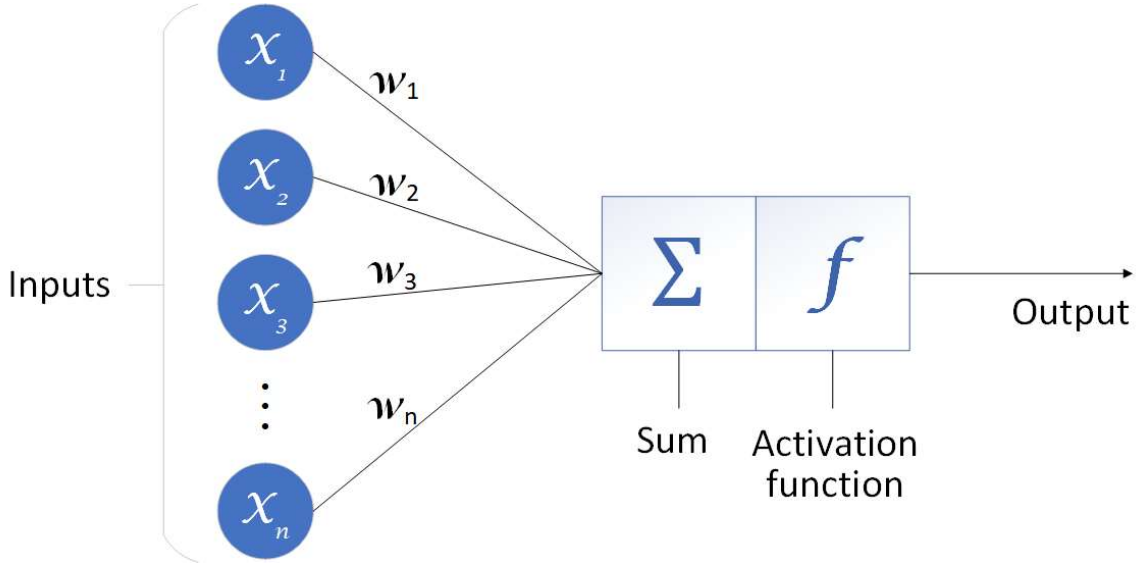
1. Uvod u neuronske mreže 1
1.1. Razlike između bioloških i umjetnih živčanih sustava 2
2. Aktivacijske funkcije neuronske mreže 3
2.1. Sigmoid 4
2.2. Hiperbolički tangens (tanh) 4
2.3. Aktivacijska funkcija ispravljene linearne jedinice (ReLU) 5
2.4. Propuštajući ReLU 5
2.5. Maxout 5
2.6. Aktivacijska funkcija eksponencijalne linearne jedinice (ELU) 5
2.7. Jezgreno bazirana ne parametarska aktivacijska funkcija (KAF) 6
2.8. Softplus aktivacijska funkcija 6
2.9. Softmax funkcija 6
3. Princip rada neuronske mreže 7
4. Učenje neuronske mreže 8
4.1. Funkcije gubitka 8
4.2. Gradijent i gradijentni pad 12
4.3. Propagacija unazad 15
4.4. Optimizatori gradijentnog pada 20

1. Uvod u neuronske mreže

Ideja neuronskih mreža izuzetno je stara i datira od pedesetih godina prošlog stoljeća, ali tek posljednjih godina neuronske mreže su područje računarstva koje se iznimno istražuje prvenstveno zbog sve bržeg računalnog sklopovlja u vidu računanja matematičkih operacija i interneta. Tome su najviše doprinijele grafičke kartice pomoću kojih je moguće računati kompleksne matrične račune koje se najviše koriste u neuronskim mrežama i dostupnosti velike količine podataka koji su potrebni za treniranje neuronskih mreža. Neuronske mreže su nastale na neki način kao simulacija bioloških procesa učenja. Živčani sustav inteligentnijih živih bića sastoji se od stanica koje se zovu neuroni. Neuroni su međusobno spojeni pomoću aksona i dendrita (Slika 1.). Na sličan način funkcionira i umjetni neuron. Dendrit bi bio ulaz u neuron do kojeg dolaze izlazi iz prethodnog sloja (X_1-X_n) nakon čega se računa suma ulaza nad kojom se poziva aktivacijska funkcija čija se vrijednost potom prosljeđuje dalje na sljedeći sloj. Aktivacijska funkcija služi umjetnoj neuronskoj mreži da joj se podari nelinearnost. Pojednostavljena shema umjetnog neurona nalazi se na Slika 2. .



Slika 1. Biološki neuron (Izvor: https://en.wikipedia.org/wiki/Neuron)



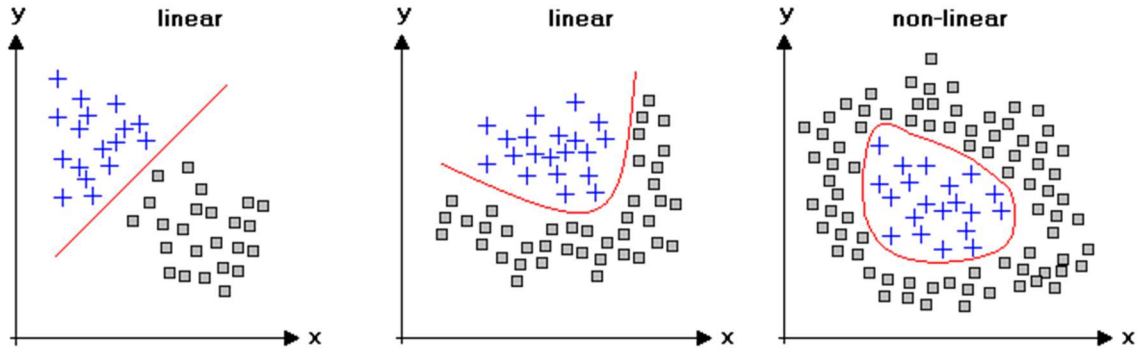
Slika 2. Umjetni neuron

1.1. Razlike između bioloških i umjetnih živčanih sustava

U trenutku pisanja ovog rada nije moguće u potpunosti simulirati rad mozga iz nekoliko razloga. Umjetni neuroni se ponašaju poprilično drugačije od prirodnih i na drugi način „ispaljuju“. Ljudski mozak ima oko 100 milijardi neurona i oko 100 trilijuna veza (sinapsi) i troši oko 20W energije. Za usporedbu je moguće uzeti najveću umjetnu neuronsku mrežu koja ima oko 10 milijuna neurona i jednu milijardu konekcija, a računalo koje je pokreće ima 16 tisuća CPU-a i troši oko 3 milijuna wata. Mozak ima 5 tipova senzora. Ne zna se točno kako mozak uči, ali vrlo vjerojatno ne uči računanjem parcijalnih derivacija i to je jedan od velikih nedostataka simulacije živčanog sustava.

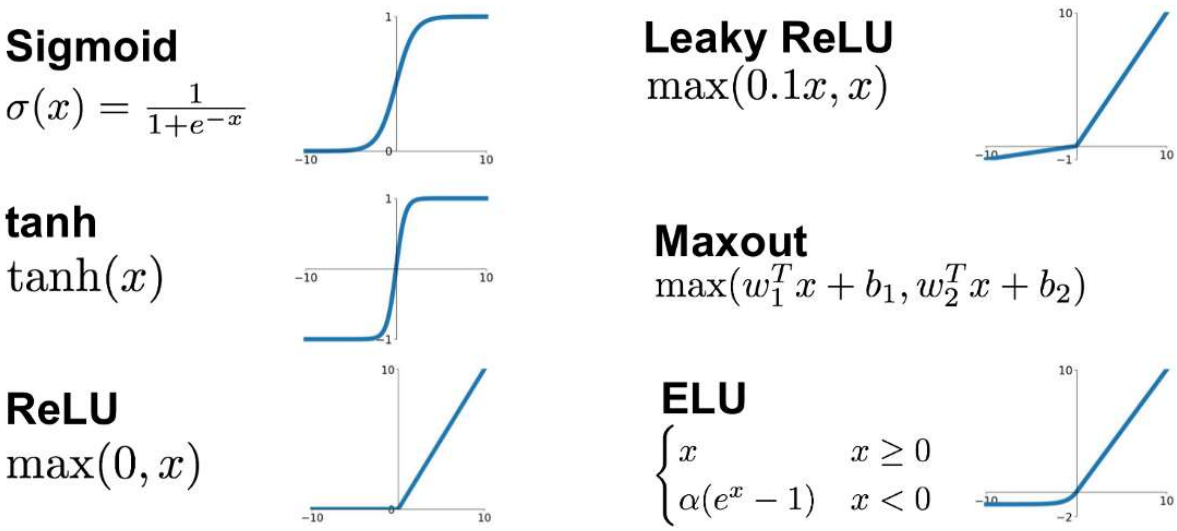
2. Aktivacijske funkcije neuronske mreže

Kao što je opisano na Slika 2. , nakon što se sumiraju ulazi u neuron, nad sumom je potrebno izvršiti funkciju aktivacije. Zapravo suma, tj. skalarni produkt vektora i nije najpravilniji izraz jer neke aktivacijske funkcije ne koriste skalarni produkt ulaznih vektora već uzimaju maksimalnu vrijednost (ReLu). Aktivacijske funkcije koriste se kako bi neuronska mreža bila nelinearna. U slučaju kada bi postojao samo linearan izlaz iz svakog sloja neuronske mreže, sve slojeve bi se moglo zamijeniti jednim slojem zbog nedostatka nelinearnosti.[1] Primjer linearnosti i nelinearnosti statističkih modela moguće je vidjeti na Slika 3. . Kao što je moguće vidjeti na slici bez nelinearnosti nije nikako moguće napraviti kompleksniji model klasifikacije, te bi se neuronska mreža zapravo mogla svesti na logističku regresiju.



Slika 3. Linearni i nelinearni modeli (Izvor: http://www.statistics4u.com)

Aktivacijske funkcije su se mijenjale i razvijale kroz povijest, a razlikuju se po matematičkim formulama, odnosno načinu na koji djeluju nad ulaznim setom podataka. Izuzetno su bitne prilikom konstruiranja neuronske mreže jer o odabiru aktivacijske funkcije ovisi brzina učenja i vrlo često preciznost i performanse same neuronske mreže. Aktivacijsku funkcija se odabire na razini sloja neurona, ne nad svakim neuronom zasebno. Osim što je bitna za nelinearnost mreže, nad aktivacijskom funkcijom potrebno je moći izračunati derivaciju. To je iz razloga što se prilikom učenja neuronske mreže koristi algoritam koji se zove propagacija unazad koji unazad po neuronskoj mreži propagira grešku i zahtjeva izračun gradijenta vektorskog polja. O algoritmu propagacije unazad biti će riječi u sljedećim poglavljima.



Slika 4. Aktivacijske funkcije(Izvor: https://medium.com)

2.1. Sigmoid

Sigmoidna funkcija je jedna od funkcija koja se prije najčešće koristila u neuronskim mrežama zbog jednostavnosti kod izračuna algoritma propagacije unazad. Kao što se može vidjeti na slici 4, sigmoidna funkcija nije centrirana po nuli i ima računanje eksponenta što je izuzetno zahtjevno za računanje. Osim toga, najveći je problem zasićenja gradijenata zato što funkcija ima kodomenu između 0 i 1 i vrijednosti mogu ostati konstantne zbog čega će gradijent imati manje vrijednosti. Zbog toga, moguće je da se ne dogodi promjena kod gradijentnog pada.

2.2. Hiperbolički tangens (tanh)

Hiperbolički tangens je neparna, monotono rastuća funkcija. Kodomena vrijednosti joj je -1 i 1 i centrirana je po nuli. Funkcija se definira kao u nastavku:

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Slično kao i kod sigmoidne funkcije postoji problem zasićenja gradijenata. Kada je ulaz veći od nule gradijenti će biti pozitivni ili negativni što može dovesti do problema eksplozije ili nestajanja gradijenata.

2.3. Aktivacijska funkcija ispravljene linearne jedinice (ReLU)

Aktivacijska funkcija ispravljene linearne jedinice (engl. Rectified Linear Unit Activation Function) ili skraćeno ReLU je najčešće korištena aktivacijska funkcija zbog jednostavnosti propagacije unazad i zato što nije zahtjevna za računanje. Kod nje je nemoguće dobiti zasićenje i konvergira brže od ostalih funkcija. Najveći problem ReLU funkcije je mrtav ReLU što se dogodi u slučaju ako je ulaz manji od nule, funkcija će uvijek davati nulu. Funkcija radi na način da propušta maksimalni ulazni parametar dalje. [2][3]

2.4. Propuštajući ReLU

Propuštajući ReLU (engl. Leaky ReLU) rješava problem mrtvog ReLU-a na način da dodaje dodatnu konstantu nagiba grafa. Na taj način je riješen problem mrtvog ReLU-a i ubrzano je učenje po istraživanjima zbog boljeg balansa. Postoji nekoliko različitih varijanti propuštajućeg ReLU-a, a najpoznatije su klasični propuštajući ReLU koji vrijednost nagiba ima kao konstantu i parametrizirani propuštajući ReLU koji parametar uči prilikom učenja neuronske mreže.[4]

2.5. Maxout

Maxout tip aktivacijske funkcije je kreiran 2013. godine. Za aktivaciju koristi maksimalnu vrijednost unutar grupe linearnih dijelova. Slična je ReLU aktivacijskoj funkciji, ali Maxout uspoređuje vrijednosti nad grupama kandidata, a ReLU ih uspoređuje s nulom. Eksperimentom je utvrđeno da u nekim slučajevima mreže građene Maxout aktivacijskim funkcijama postižu bolje rezultate od drugih mreža. Najveći je problem što je izuzetno skupa za izračun jer na neki način dodaje dodatni skriveni sloj i potrebno ju je koristiti uz dropout tehniku o kojoj će biti riječi kasnije.[5]

2.6. Aktivacijska funkcija eksponencijalne linearne jedinice (ELU)

Aktivacijska funkcija eksponencijalne linearne jedinice (engl. Exponential linear units) je također varijacija ReLU funkcije s boljom vrijednosti za $x<0$ koja je nastala 2015. godine. Ima identična svojstva kao i ReLU, ali nema problema s mrtvim ReLU-

ima, bolje izlazne vrijednosti od propuštajućeg ReLU-a. Najveća manjkavost je što ima više računanja zbog eksponencijalne funkcije. [6]

2.7. Jezgreno bazirana ne parametarska aktivacijska funkcija (KAF)

Jezgreno bazirana ne parametarska aktivacijska funkcija (engl. kernel-based non-parametric activation function) je nastala 2017 kako bi se dodatno povećala fleksibilnost neuronskih mreža. KAF je ne parametrizirana aktivacijska funkcija definirana kao jednodimenzionalna jezgrena aproksimacija. [7]

2.8. Softplus aktivacijska funkcija

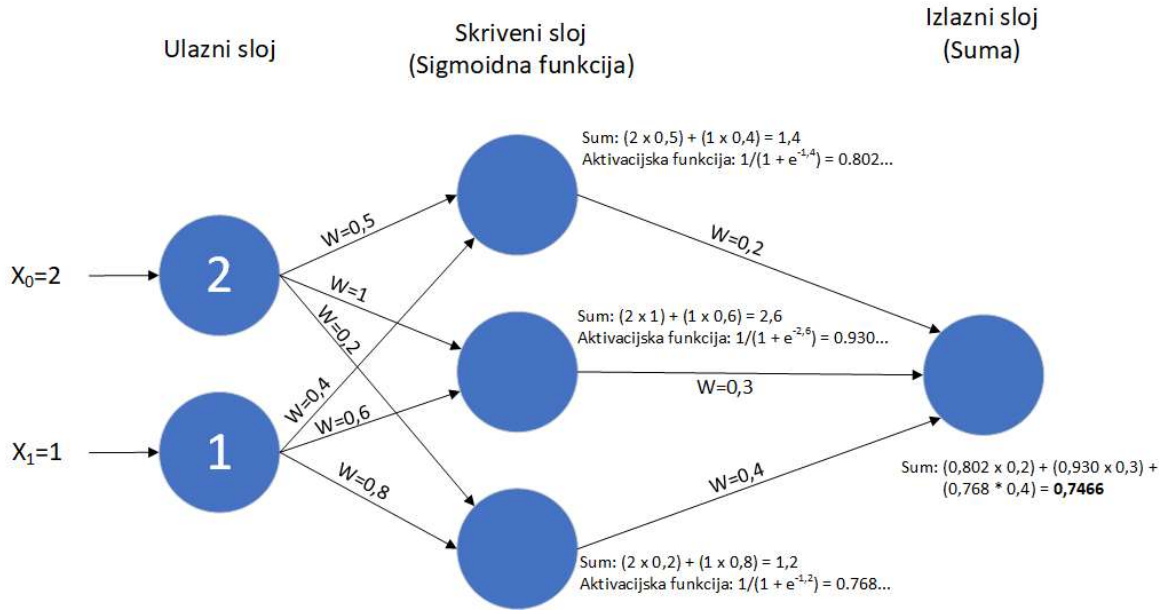
Softplus funkcija je uvedena 2001 godine i ima vrlo sličnu krivulju kao i ReLU, ali mekaniju krivulju oko nule i puno bolju derivaciju. Najveći je problem što slično kao kod ReLU funkcije može doći do mrtvog neurona.

2.9. Softmax funkcija

Softmax funkcija se koristi za zadnji sloj neurona kod klasifikacije (primjerice slika). Funkcija prima nenormalizirani vektor i normalizira ga u vjerojatnosnu distribuciju. Funkcija ima nezgodno ime i više je vezana za arg max funkciju nego max funkciju. [8][9]

3. Princip rada neuronske mreže

Neuronska mreža se sastoji od više neurona koji su spojeni u neuronsku mrežu. Neuronski mrežu moguće je gledati kao kompleksni računski graf u kojem je jedinica računanja neuron. [1] Mreže je moguće klasificirati na nekoliko različitih modela, ali sve rade na sličnom principu. Imaju ulaze iz kojih kroz slojeve trebaju kreirati smisleni izlaz koji su naučile postupkom učenja. Slojevi se sastoje od neurona koji su simulirani pomoću aktivacijskih funkcija. Svaki neuron posjeduje vlastitu težinu (w) koja se propagira na sljedeći sloj kroz aktivacijsku funkciju i na taj način se dobivaju izlazi. U primjeru na sliciSlika 5. moguće je vidjeti primjer jednostavne mreže s nasumično postavljenim vrijednostima ulaza i težina kao i izračun vrijednosti izlaza iz neuronske mreže.



Slika 5. Jednostavna neuronska mreža s izračunima

Na slici je izuzetno pojednostavnjeni model koji ima samo jedan nelinearni sloj s sigmoidnom aktivacijskom funkcijom, a izlaz je samo suma i ne bi bio pogodan za produkcijska rješenja kakva se očekuju od neuronskih mreža. Neuronska mreža će najčešće se sastojati od nekoliko nelinearnih skrivenih slojeva, a izlazni sloj će biti različit s obzirom na to što se od neuronske mreže očekuje da radi. Primjerice, u slučaju klasifikacije će se koristiti Softmax aktivacijska funkcija, dok će se u nekim drugim slučajevima generirati neki model iz ulaznih podataka i sl.[1], [8]

4. Učenje neuronske mreže

Učenje neuronske mreže radi se pomoću algoritma zvanog propagacija unazad kojeg je kreirao 1986. godine Rumelhart. [10] Kao što je naglašeno u prošlom poglavlju, na neuronsku mrežu moguće je gledati kao na kompleksni računski graf. Kako bi neuronska mreža funkcionirala i radila ono što se od nje traži potrebno ju je naučiti težinama. Težine neuronske mreže je moguće naučiti pomoću mnoštva kvalitetnih primjera što je najbitniji uvjet za kvalitetno učenje. Za učenje neuronske mreže potrebno se odlučiti za jedan od nekoliko optimizacijskih algoritama za učenje poput stohastičkog gradijentnog pada, Momentuma, Adagrada, Nesterov, Adadelta, Adam, Nadam, RMSprop, AdaMax i sl. [11] O tim algoritmima biti će riječi u kasnijim poglavljima, a za sada je najbitnije da su svi algoritmi za početak izuzetno slični i da kao jezgru koriste sličnu matematičku podlogu.

Prilikom učenja neuronske mreže prvo je potrebno poznavati ono što se uči, odnosno ulazne podatke i očekivane izlazne podatke. To je potrebno iz razloga kako bi se u svakom trenutku znalo koliko je neuronska mreža pogriješila i kako bi se ta pogreška propagirala kroz mrežu propagacijom unazad. Za to je potrebno poznavati dio matematike koja se bavi statističkim pogreškama. Funkcija koja računa pogrešku naziva se funkcija gubitka (Engl. Loss function). Postoji nekoliko varijanti kako se funkcija može izvesti i to je izuzetno ovisno o konkretnom problemu koji se pomoću neuronske mreže pokušava riješiti. Nakon računanja greške (gubitka) računa se gradijent kako bi se znalo u kojem smjeru treba mijenjati vrijednosti i propagirati ih po neuronskoj mreži pomoću algoritma propagacije unazad. O gradijentima će biti riječi nešto kasnije. Sada će biti objašnjene funkcije gubitka.

4.1. Funkcije gubitka

Kao što je bilo rečeno postoji nekoliko varijanti funkcija gubitka i vrlo je bitno da se koriste ispravno u vidu problema koji se pokušava riješiti neuronskom mrežom. Funkcije gubitka izračunava grešku između željenih rezultata i dobivenih rezultata. Funkcije gubitka moguće je podijeliti u dvije skupine:

- Regresijski gubitak
- Klasifikacijski gubitak

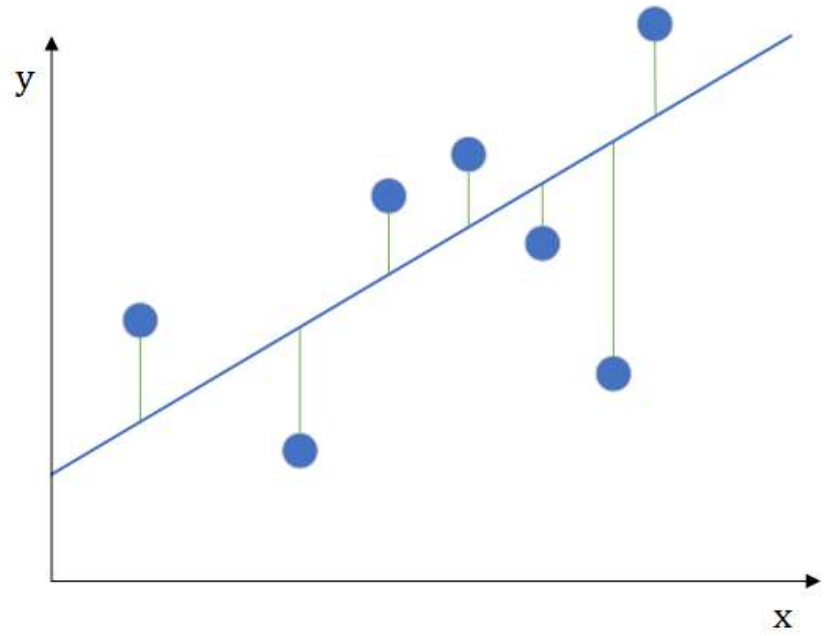
Naravno, ovisno o tome radi li neuronska mreža regresiju (neko određeno predviđanje podataka) ili radi klasifikaciju (primjerice klasifikacija slika po nekom parametru). Najosnovnije regresijske funkcije gubitka su:

- Srednja vrijednost kvadrata greške ili kvadratni gubitak
- L2 gubitak
- Srednji apsolutni gubitak
- L1 gubitak
- Srednja pogreška

Najčešće korištene regresijske funkcije su kvadratni gubitak i srednji apsolutni gubitak. Kvadratni gubitak, odnosno kvadrat srednje devijacije računa se pomoću sljedeće formule:

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Kao što je vidljivo u formuli izračunava se razlika dobivene vrijednosti i željene vrijednosti izlaza (varijable y_i i $\hat{y}_i$) nakon čega se kvadriraju, sumiraju i podjele s brojem izlaza. Proces je grafički prikazan na sliciSlika 6. gdje je linija željeni rezultat, a točke dobiveni rezultat. Razlika je udaljenost točaka od linije.

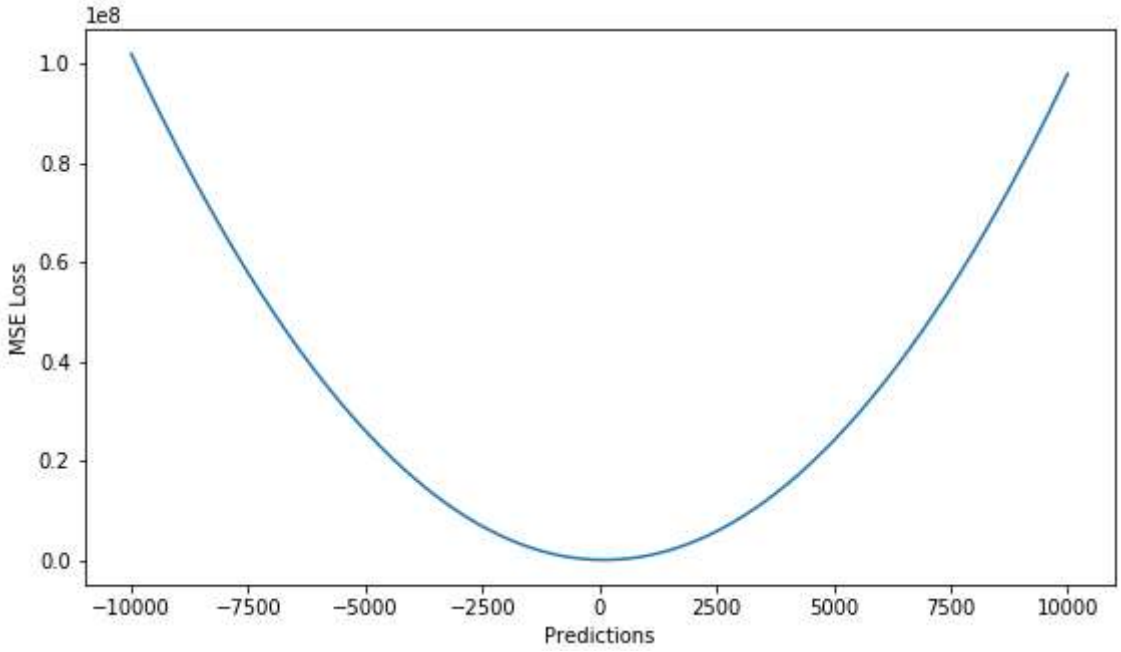


Slika 6. Regresija

Slika zapravo prikazuje klasičan graf linearne regresije koja je izuzetno čest način modeliranja korelacija među skalarnim vrijednostima. [12]

Kod računanja L2 gubitka formula je identična kao i kod srednje vrijednosti kvadrata greške, ali se suma razlike ne dijeli s brojem elemenata. L2 gubitak se računa pomoću formule:

$$L2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$



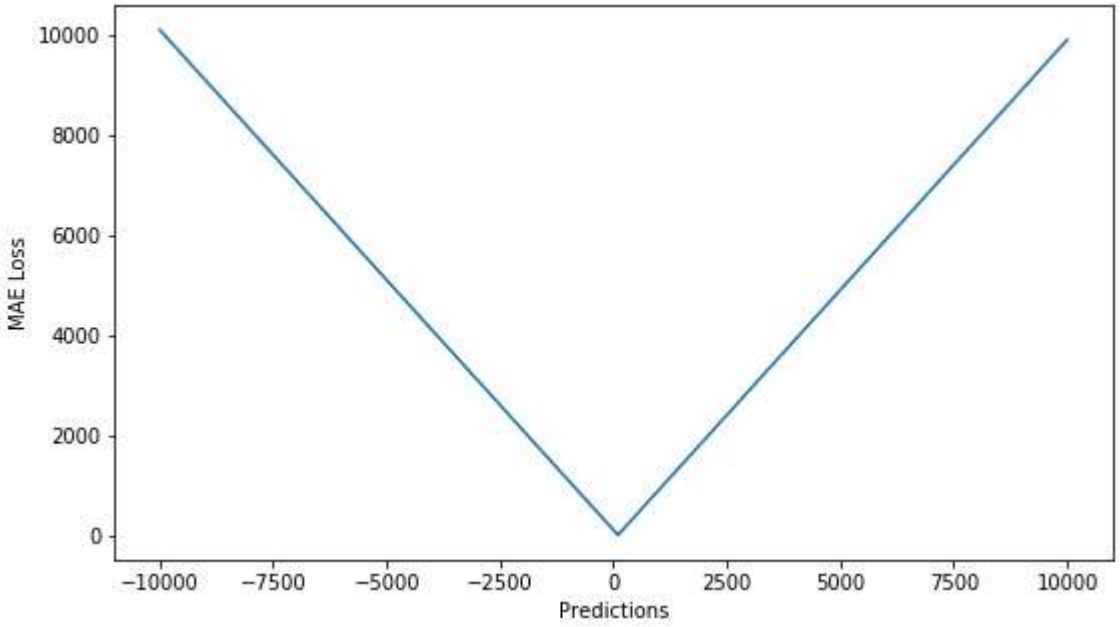
Slika 7. Izgled grafa funkcije kvadratnog gubitka

Srednji apsolutni gubitak je suma apsolutnih razlika između predviđenih i dobivenih vrijednosti. Formula za izračunavanje srednjeg apsolutnog gubitka slijedi u nastavku:

$$MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$$

Kao i kod L2 funkcije, L1 funkcija je identična srednjem apsolutnom gubitku, samo nema dijeljenje s brojem elemenata. Formula L1 funkcije gubitka slijedi u nastavku:

$$L1 = \sum_{i=1}^{n} |y_i - \hat{y}_i|$$



Slika 8. Izgled grafa funkcije srednjeg apsolutnog gubitka

Srednja pogreška je izuzetno nezgodna za korištenje jer može dati kako pozitivne, tako i negativne vrijednosti. Zbog toga je moguće da se te vrijednosti ponište prilikom računanja. Formula za računanje srednje pogreške je:

$$MBE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{n}$$

Kada se govori o klasifikaciji, najčešće funkcije su:

- *Hinge* gubitak ili više klasni SVM (Engl. Support vector machine))
- Gubitak križne entropije ili negativna vrijednost zapisivanja

Prilikom klasifikacije pomoću više klasnog SVM-a rezultat ispravne kategorije mora biti veći od sume rezultata svih ostalih (neispravnih) kategorija. Zbog toga se *hinge* koristi za klasifikaciju najveće margine (Engl. Maximum-margin), pogotovo za SVM. Iako ju nije moguće diferencirati, to je konveksna funkcija s kojom je jednostavno raditi. [13], [14] Matematička formula je:

$$SVMLoss = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

Gubitak križne entropije odnosno negativna vrijednost zapisivanja je najčešća funkcija koja se koristi prilikom klasifikacije. Gubitak križne entropije se povećava

kao što predviđena vjerojatnost odstupa od stvarnog kanala. Formula za križnu entropiju slijedi u nastavku:

$$CrossEntropyLoss = -(y_i \, log(\hat{y}_i) + (1 - y_i) \, log(1 - \hat{y}_i))$$

Moguće je primijetiti da se desna strana ne primjenjuje ukoliko je $y_i = 1$ zato što će biti jednak nuli. Ukoliko je $y_i = 0$, lijeva strana formule se ne primjenjuje zato što je jednaka nuli. Ukratko, množi se logaritam stvarno predviđene vjerojatnosti za istinitu klasu. Važan je aspekt taj da će križna entropija izuzetno kažnjavati predviđanje ako je uvjereno da je ispravno ali nije ispravno.

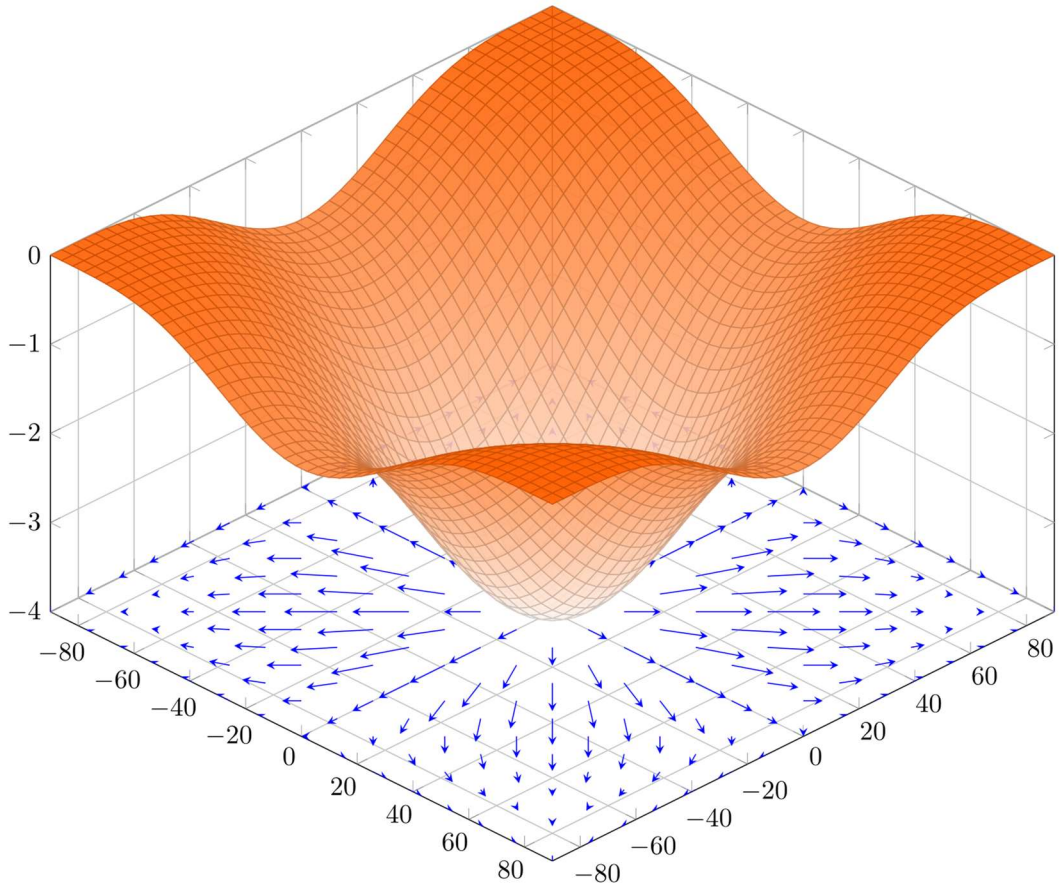
Osim navedenih i objašnjenih funkcija postoji izuzetno širok spektar funkcija koje je moguće upotrijebiti. U trenutku pisanja rada najupotrebljavanije funkcije su: srednji kvadrat greške, srednja apsolutna greška, srednja apsolutna postotnea greška, srednja apsolutna logaritamska greška, *hinge,* binarna križna entropija, kategorička križna entropija, rijetka kategorička križna entropija, *kullback leibler* divergencija, *poisson*, kosinusna blizina i mnoge druge. Prilikom odabira funkcije gubitka prvo je potrebno proučiti problem i izvore, pa pokušati zaključiti koja bi najbolje odgovarala, a naravno i testiranje dolazi u obzir jer kod strojnog učenja se na izuzetno malo različitom problemu druge funkcije poprilično drugačije ponašaju i daju bolje ili lošije rezultate. [15]–[21]

4.2. Gradijent i gradijentni pad

Nakon što se izračuna funkcija gubitka, pomoću iste je potrebno izračunati gradijent. Gradijent je više varijabilna generalizacija derivata i jedan je od osnovnih koncepata u analizi vektora i teorije nelinearnih mapiranja. Derivaciju je moguće definirati nad funkcijama s jednom varijablom, za funkciju s više varijabli ulogu derivacije preuzima gradijent. Gradijent je vektorski bazirana funkcija za razliku od derivacije koja je skalarna. Kao i derivacija, gradijent određuje nagib tangente grafa funkcije, odnosno gradijent pokazuje smjer gdje je najveće povećanje funkcije. Na sliciSlika 9. moguće je vidjeti ponašanje gradijenta na grafu funkcije. Gradijent je označen plavim strelicama.

Na gradijentni pad je moguće figurativno gledati kao na brdo na čiji sam vrh se mora popeti slijep čovjek u što manje koraka. U početku se čovjek kreće u velikim

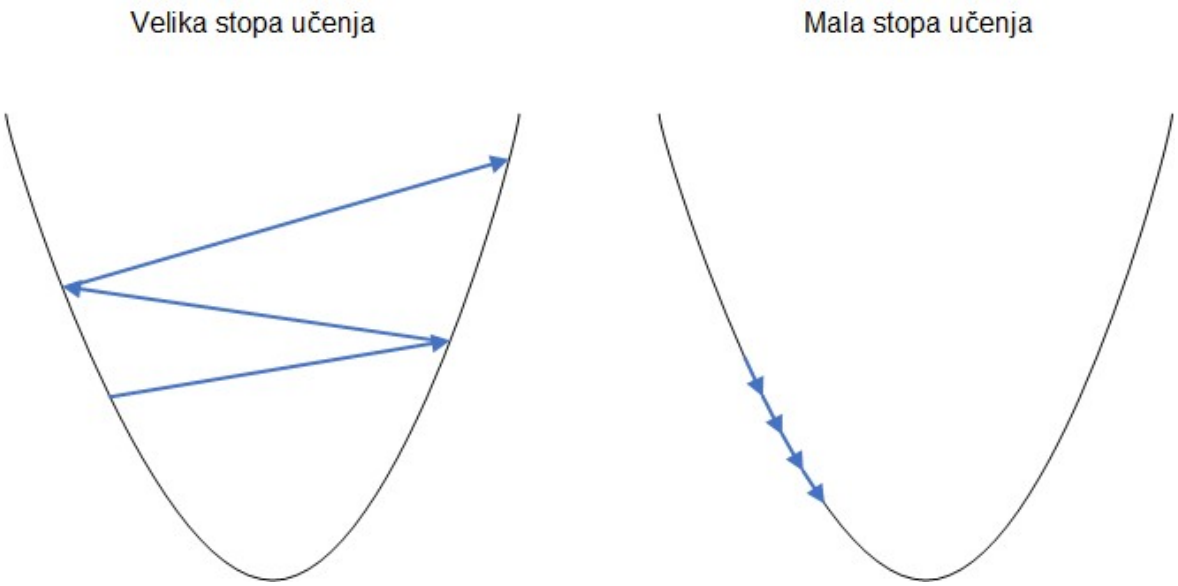
koracima po najvećem nagibu, dok nakon toga radi sve manje i manje korake kako ne bi prešao vrh brda. Ta procedura se opisuje matematički pomoću gradijenta i zove se gradijentni pad jer je vrh planine zapravo dno i gradijentni pad je minimizacijska funkcija.



Slika 9. Gradijent funkcije $f(x,y) = -(\cos^2 x + \cos^2 y)^2$ (izvor: wikipedia.org)

U neuronskim mrežama gradijent je potrebno izračunati kako bi se odredio smjer greške (smjer kretanja) i koje težinske vrijednosti je potrebno namjestiti kako bi se unaprijedilo rješenje mreže. Gradijent mjeri koliko se izlaz funkcije promijeni ako se malo promjene ulazi.

Gradijentni pad osim izračuna gradijenta treba i stopu učenja. To je jedan broj koji će odraditi veličinu koraka prema minimumu funkcije. U slučaju grafa na slici Slika 9. minimum bi bio u točki X=0, Y=0. Taj parametar ne smije biti ni prevelik ni premalen. Ako je stopa učenja prevelika, algoritam će premašiti cilj, odnosno minimum, a ako je premala, proces učenja će trajati predugo.



Slika 10. Gradijentni pad s obzirom na stopu učenja

Prilikom odabira stope učenja najbolje je kreirati graf funkcije gubitka kojeg mogu kreirati svi današnji programski okviri za rad s neuronskim mrežama. Na grafu će biti odmah vidljivo radi li se o ispravnoj ili neispravnoj stopi učenja.

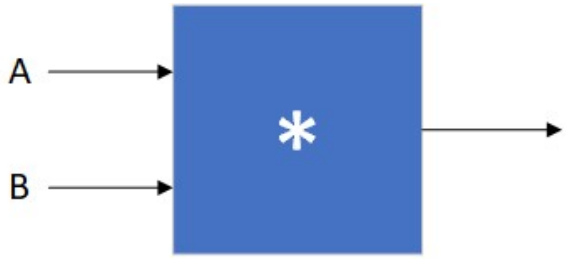
Slika 11. Funkcija gubitka s obzirom na stopu učenja

U sljedećim poglavljima će biti detaljan opis procedure učenja neuronskih mreža pomoću primjera u sklopu poglavlja o propagaciji unazad i kako se koristi gradijentni pad za propagaciju greške po neuronskoj mreži. [22]–[25]

4.3. Propagacija unazad

Propagacija unazad je algoritam za učenje neuronskih mreža koji je kreirao David E. Rumelhart 1986. godine. Propagacija unazad je skraćeni oblik pisanja, dok je dulji oblik pisanja propagiranje greške unazad zato što se greška izračunava pomoću funkcije gubitka nakon čega se distribuira odnosno propagira kroz neuronsku mrežu. To je algoritam koji se koristi kako bi se izračunali gradijenti koji su potrebni kako bi se podesile težinske vrijednosti u neuronskoj mreži. [8] Algoritam propagacije unaprijed je zapravo generalizacija delta pravila na višeslojne unaprijedne mreže (Engl. feedforward network) koja je moguća zbog korištenja pravila lanca. Pravilo lanca u matematici je pravilo za računanje derivacije kompozitne funkcije (funkcija koja se sastoji od više funkcija). [22], [26]

Kako bi se što više laički prikazao problem propagacije unazad, biti će uzet jednostavan programski kod koji će raditi na jednostavnim vratima zbrajanja i množenja umjesto na neuronima koji imaju aktivacijske funkcije. Kao jezik u kojem će biti objašnjena propagacija unazad uzeta je Java jer je više-manje sve ustanove pokrivaju kao jezik.



Slika 12. Vrata množenja

Vrata uzimaju dva realna broja A i B i množe ih. Jednostavan programski kod napisan u Javi izgleda kao u nastavku:

```
public static double forwardMultiplyGate(double a, double b) {
  return a * b;
}
//poziv funkciji vraća -6
forwardMultiplyGate(-2, 3);
```

Pitanje koje se nameće je kako podesiti ulazne parametre da daju neke druge vrijednosti. U ovom jednostavnom slučaju žele se dobiti vrijednosti koje su veće od

-6. Ako se varijabla A smanji za 0,01 i B se smanji za 0,01, A * B će iznositi -5,95 što je veći broj od -6. Dobiveni broj je bolji za 0,05. Kako bi se to moglo programski napraviti? Moguće je napisati programski kod koji nasumično traži bolje rješenje pomoću *random* funkcije. Naravno, takvo traženje rješenja će biti izuzetno dugotrajno za bilokakve malo veće neuronske mreže. Za ovaj problem gotovo pa može poslužiti. Primjer programskog koda slijedi u nastavku:

```
public static void main(String[] args) {
  double a = -2, b = 3;// Ulazne vrijednosti
  // Pokušaj nasumične promjene varijabli
  double tweak = 0.01f;
  double best = -50;// Mali broj s kojim će se uspoređivati
  double best_a = a, best_b = b;

  for (int i = 0; i < 100; i++) {
     // Male promjene a
     double a_try = a + tweak * (Math.random() * 2.0 - 1.0);
     // Male promjene b
     double b_try = b + tweak * (Math.random() * 2.0 - 1.0);
     double out = forwardMultiplyGate(a_try, b_try);
     if (out > best) {
           best = out;
           best_a = a_try;
           best_b = b_try;
     }
  }
  System.out.println("Najbolje su: a>" + best_a + ", >" + best_b);
}

public static double forwardMultiplyGate(double a, double b) {
  return a * b;
}
```

Iz razloga što nasumično određivanje vrijednosti nije nikako najbolji i najbrži način učenja, potrebno je izračunati gradijent. Kao najosnovniji primjer računanja gradijenta bio bi numerički gradijent. On će pomoći računanju smjera u kojem je potrebno pomaknuti vrijednosti kako bi rezultat bio veći od -6. Intuicija govori da je potrebno povećati varijablu a, a smanjiti varijablu b. Ta intuicija o kojoj se govori je zapravo derivacija funkcije. Derivacija po varijabli a glasi kao u nastavku:

$$\frac{\partial f(a,b)}{\partial a} = \frac{f(a + h, b) - f(a,b)}{h}$$

Varijabla h je veličina promjene (u programskom kodu naznačeno kao „tweak“). Cijeli simbol s lijeve strane je notacija za derivaciju funkcije $f(a,b)$ s obzirom na

varijablu a. S desne strane horizontalna crta označava razlomak. Vrata naime daju neku određenu izlaznu vrijednost $f(a, b)$ nakon čega se promijeni jedan od ulaza za mali iznos h nakon čega se pročita novi izlaz $f(a + h, b)$. Oduzimanje te dvije vrijednosti govori promjenu, a dijeljenje s h je normalizira. Primjer programskog koda koji opisuje gore navedeno slijedi u nastavku:

```
double a = -2, b = 3; // Ulazne vrijednosti
double out = forwarMultiplyGate(a, b);
double h = 0.0001;
//derivacija po a
double aph = a + h; //-1.9999
double out2 = forwarMultiplyGate(aph, b); //-5.9997
double a_derivative = (out2 - out) / h;  // 3.0
//derivacija po b
double bph = b + h; //3.0001
double out3 = forwarMultiplyGate(a, bph); //-6.0002
double b_derivative = (out3 - out) / h; // -2.0
```

Ako se prođe po programskom kodu može se vidjeti da je dodavanje vrijednosti varijable h dalo veću vrijednost izlaza. Dijeljenje s nasumičnom vrijednosti h postoji kako bi se normalizirala vrijednost koju vraćaju vrata. Matematički gledano vrijednost h bi trebala biti beskonačno malena, ali zadana vrijednost u praksi dovoljno dobro funkcionira kako bi dala dobre rezultate. Kada se govori o derivaciji, govori se o jednoj ulaznoj varijabli, a kada se govori o gradijentu govori se o svim ulaznim varijablama. Gradijent je napravljen od derivacija svih ulaza koje se nalaze u vektoru. To je moguće vidjeti u sljedećem primjeru:

```
double step_size = 0.01;
double out = forwardMultiplyGate(a, b); //Do sada -6
a = a + step_size * a_derivative; //a postaje -1.97
b = b + step_size * b_derivative; //b postaje 2.98
double out_new = forwardMultiplyGate(a, b); //-5.87
```

Veći korak nije uvijek nužno dobar. Kao što je već bilo napomenuto, veličina stope učenja mora biti određena s obzirom na problem i s obzirom na arhitekturu neuronske mreže. Ovo rješenje bi bilo rješenje pomoću računanja numeričkog gradijenta. Postoji i bolje rješenje, a to je analitički gradijent. Naime, numerički gradijent se mora računati posebno za svaki ulaz što je i dalje računalno zahtjevan, odnosno skup proces. Koristeći analitički gradijent, moguće je izračunati gradijent iz cijelog izraza. Kako bi se to moglo napraviti, potrebno je koristiti parcijalne

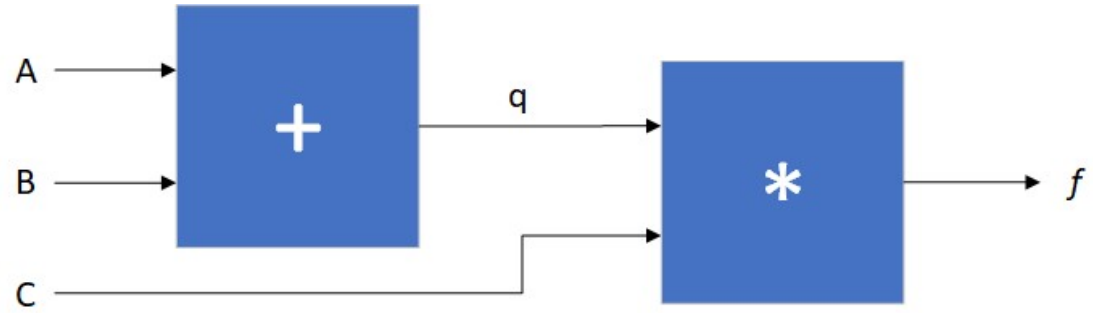
derivacije. U nastavku se nalaze parcijalne derivacije prethodne korištene funkcije *(a * b)* po varijabli *a* i po varijabli *b*:

$$\frac{\partial f(a,b)}{\partial a} = b \qquad \frac{\partial f(a,b)}{\partial b} = a$$

Programski kod koji radi pomoću analitičke derivacije:

```
double a = -2, b = 3;
double out = forwardMultiplyGate(a, b);
double a_gradient = b; // po kompleksnoj derivaciji
double b_gradient = a;
double step_size = 0.01;
a += step_size * a_gradient; // -1.97
b += step_size * b_gradient; // 2.98
double out_new = forwardMultiplyGate(a, b); // -5.87.
```

Ovo je najjednostavniji primjer računanja gradijenta na jednostavnim vratima množenja. Kako bi se stvari dodatno zakomplicirale, moguće je dodati još jedna jednostavna vrata zbrajanja kao na slici Slika 13.



Slika 13. Vrata zbrajanja i množenja

Izraz koji je potrebno izračunati je $f(a,b,c) = (a + b)c$, a programski kod koji ga računa bi u Javi izgledao kao u nastavku:

```
public static void main(String[] args) {
  double a = -2, b = 5, c = -4;
  double f = forwardCircuit(a, b, c); // izlaz je -12
}
public static double forwardMultiplyGate(double a, double b) {
  return a * b;
}
public static double forwardAddGate(double a, double b) {
  return a + b;
}
public static double forwardCircuit(double a, double b, double c){
  double q = forwardAddGate(a, b);
  return forwardMultiplyGate (q, c);
}
```

U ovom slučaju će gradijent biti računan od malo drugačijih varijabli. Potrebno je podijeliti računanje za svaka vrata zajedno. Zadnja vrata računaju q * c, a prednja vrata računaju a + b. Sve parcijalne derivacije po svim varijablama slijede u nastavku:

$$\frac{\partial f(q,c)}{\partial q} = c \qquad \frac{\partial f(q,c)}{\partial c} = q \qquad \frac{\partial f(a,b)}{\partial a} = 1 \qquad \frac{\partial f(a,b)}{\partial b} = 1$$

Nakon toga može se upotrijebiti pravilo lanca (Engl. chain rule) kako bi se izračunao gradijent od q s obzirom na a i b.

$$\frac{\partial f(q,c)}{\partial a} = \frac{\partial q(a,b)}{\partial a} \frac{\partial f(q,c)}{\partial q}$$

Programski kod koji računa gore navedene formule dan je u nastavku:

```
double a = -2, b = 5, c = -4;
double q = forwardAddGate(a, b); //q je 3
double f = forwardMultiplyGate(q, c);// izlaz je -12

//Kao i u prošlom primjeru gradijent za množenje
//wrt stoji za "with respect to"
double derivative_f_wrt_c = q; // 3
double derivative_f_wrt_q = c; // -4
//Derivacija zbrajanja je 1
double derivative_q_wrt_a = 1.0;
double derivative_q_wrt_b = 1.0;
//Pravilo lanca
double derivative_f_wrt_a = derivative_q_wrt_a *
derivative_f_wrt_q;//-4
double derivative_f_wrt_b = derivative_q_wrt_b *
derivative_f_wrt_q;//-4
//Gradijent je izračunat, stavljamo ga na jedno mjesto (nepotreban
kod, samo da se vidi gradijent)
double[] grad_f_wrt_abc = {derivative_f_wrt_a, derivative_f_wrt_b,
derivative_f_wrt_c};

double step_size = 0.01;
a = a + step_size * derivative_f_wrt_a; // -2.04
b = b + step_size * derivative_f_wrt_b; // 4.96
c = c + step_size * derivative_f_wrt_c; // -3.97

// Rezultat je veći izlaz:
q = forwardAddGate(a, b); // q postaje 2.92
f = forwardMultiplyGate(q, c); // izlaz je -11.59
```

Ovaj programski kod opisuje jedan korak propagacije unazad. Na identičan način će učiti neuronska mreža, samo što umjesto jednostavnih računskih operacija poput

množenja i dijeljenja koristi kompleksnije aktivacijske funkcije kako bi se postigla nelinearnost modela. Primjerice, za sigmoidnu funkciju kao što je već prikazano, vrijedi formula:

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

Derivacija sigmoidne funkcije slijedi u nastavku:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Naravno, učenje neuronske mreže nije praktično obavljati ručno jer zahtjeva izuzetno mnogo računanja, ali na ovim jednostavnim primjerima dan je opis kako neuronska mreža uči pomoću propagacije unazad. Protokol za učenje neuronske mreže izgleda kao u nastavku:

- Uzme se nasumična podatkovna točka i pusti se kroz mrežu
- Mjeri se koliko je dobro mreža predvidjela rezultat pomoću funkcije gubitka
- Izvodi se propagacija unazad dobivene greške
- Podešavaju se vrijednosti s obzirom na stopu učenja
- Petlja kreće od početka, odnosno iterira

Takav protokol se naziva stohastički gradijentni pad i to je jedan od algoritama koji se koristi kao algoritam optimizacije neuronskih mreža o kojima će biti govora u narednim poglavljima. [10], [15], [27], [28]

4.4. Optimizatori gradijentnog pada

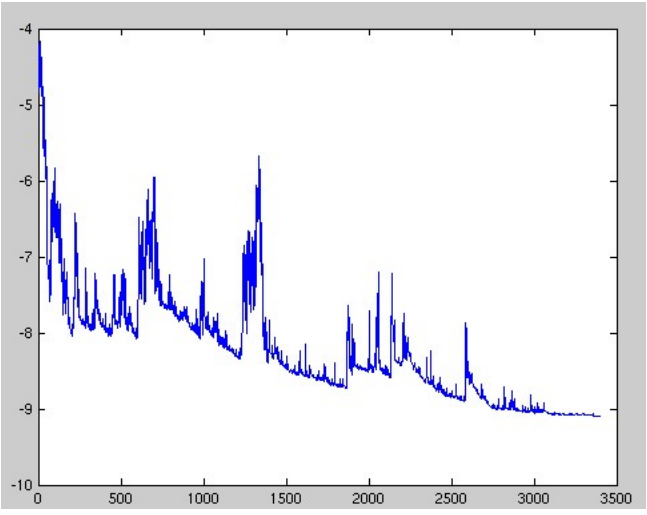
U prošlim poglavljima je objašnjen gradijentni pad, gradijent i propagacija unazad. Sve te algoritme koristi algoritam optimizatora gradijentnog pada. Najčešći algoritmi koji se koriste u modernim neuronskim mrežama su Adadelta, Adagrad, Adam, Adamax, Nadam, RMSprop, Vanilla, stohastički gradijentni pad i mnogi drugi. Različiti optimizatori postoje iz povijesnih razloga i razloga što neuronska mreža nije namijenjena rješavanju samo jednog problema, već može rješavati mnogo različitih problema iz različitih područja koji traže drugačiji pristup problemu. Razlikuju se također s obzirom na količinu podataka koje je potrebna, točnost i vrijeme izvođenja.

4.4.1. Vanilla gradijentni pad

Vanilla gradijentni pad, odnosno *batch* gradijentni pad računa gradijent funkciju gubitka s obzirom na parametre θ za cijeli set podataka (Engl. dataset). Dakle, ova verzija gradijentnog pada će računati gradijent za cijeli set podataka kako bi napravio jedno ažuriranje (Engl. update). Zbog tog problema, Vanilla algoritam je izuzetno spor i zahtjeva ogromne količine radne memorije kako bi cijeli set podataka stao u nju prilikom računanja gradijentnog pada.

4.4.2. Stohastički gradijentni pad

Stohastički gradijentni pad (u danjem tekstu SGD) je metoda koja računa parametre θ za svaki primjer treniranja iz seta podataka. Vanilla radi redundantne računske operacije za velike setove podataka jer računa gradijente za slične primjere prije svakog ažuriranja. SGD radi jedno od jednom jedno ažuriranje. Dok Vanilla jako često ostane u lokalnom minimumu, SGD zbog svoje fluktuacije može preskočiti lokalni minimum i pronaći globalni. Najveći je problem SGD-a konvergencija do apsolutnog minimuma funkcije jer se može dogoditi promašivanje. Postoje metode koje tijekom učenja smanjuju stopu učenja pa je taj problem pomoću njih znatno umanjen.



Slika 14. Fluktuacija SGD-a

4.4.3. Gradijentni pad mini seta

Gradijentni pad mini seta (Engl. Mini-batch gradient descent) koristi najbolje od oba svijeta kako bi napravio ažuriranje za svaki mini set iz seta podataka. Na taj način smanjuje varijaciju ažuriranja parametara i na taj način dolazi do stabilnije

konvergencije i može koristiti optimiziranije računanje matricama (prednosti novih grafičkih kartica). Veličina seta ovisi o problemu koji se rješava, ali je negdje u rangu od 50 do 256. Kada se koristi SGD, najčešće se koristi i ovaj algoritam jer se već nalazi u programskim okvirima za treniranje neuronskih mreža.

4.4.4. Momentum

Problem kod SGD algoritma je upravo šverdanje, odnosno krivudanje ukoliko krivulja nema dovoljno jasno izražen put prema globalnom optimumu. Naravno, krivudanje usporava učenje jer se nepotrebno gubi vrijeme. Kada bi se povećala stopa učenja, stvari se mogu popraviti, ali i pogoršati zbog promašivanja globalnog optimuma. Momentum algoritam rješava taj problem pomoću dodavanja vrijednosti parametra izglađivanja (Engl. smoothing parameter) na vektor kretanja. Na neki način, kao da se lopta dodatno proguruje nizbrdo. Rezultat je brža konvergencija i reducirana oscilacija kretanja.

4.4.5. Nesterov momentum

Nesterov momentum je nadogradnja klasičnog momentum algoritma. Naime, postoji problem kod momentuma da se „guranje lopte“ ne uspori pri kraju, tako da se često događa da se optimum prvo promaši, pa se vraća nazad na isti. Taj problem rješava Nesterov momentum na način da prilikom ažuriranja gleda malo unaprijed kako bi prilagodio parametar izglađivanja. [29]

4.4.6. Adagrad

Adagrad algoritam koji optimizira stopu učenja s obzirom na parametre na način da provodi veće korake za rjeđe parametre i manje korake za češće parametre. Zbog toga je idealan algoritam za rad s raspršenim podacima. Adagrad povećava robusnost SGD-a i koristi se prilikom treniranja velikih neuronskih mreža. Eliminira potrebu za podešavanjem stope učenja.

4.4.7. Adadelta

Adadelta je dodatak na Adagrad algoritam koji ne pohranjuje prijašnje gradijente kao što radi Adagrad, već ih svede na fiksnu veličinu nakon čega se oni stariji gube. To dovodi do redukcije smanjenja stope učenja. [30]

4.4.8. RMSprop

RMSprop i Adadelta su bili razvijani neovisno jedan o drugome i izuzetno su slični kako bi riješili problem s Adagrad algoritmom. RMSprop je identičan prvom ažurirajućem vektoru Adadelte.

4.4.9. Adam

Predviđanje adaptivnog momenta (Engl. Adaptive Moment Estimation, Adam) je još jedan od algoritama koji računa vlastitu stopu učenja poput Adagrad i RMSprop algoritma, ali eksponencijalno zaglađuje gradijent prvog reda kako bi ukomponirao momentum u ažuriranje.

4.4.10. AdaMax

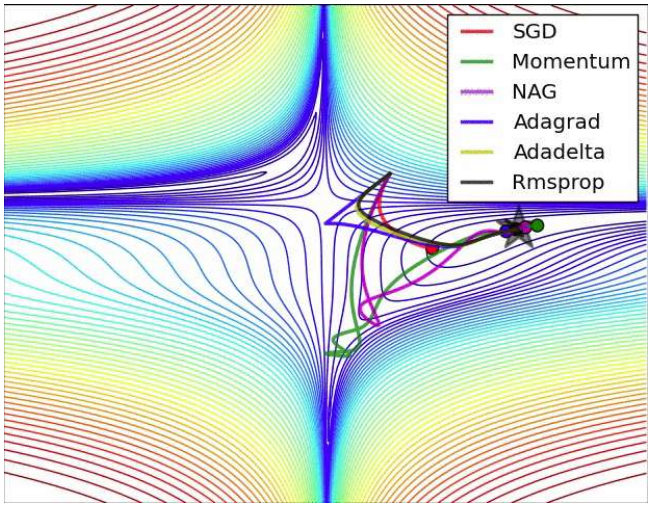
Algoritam je izuzetno sličan Adam algoritmu uz minimalne promjene oko stabilizacije vršnih vrijednosti. Autori Kingma i Ba izmjenjuju Adam algoritam i dokazuju da uz izmjene doprinose bržoj konvergenciji ka globalnom optimumu. [31]

4.4.11. Nadam

Nadam algoritam (Engl. Nesterov-accelerated Adaptive Moment Estimation) kombinira algoritme Adam i Nesterov accelerated gradient. [32]

4.4.12. Koji algoritam odabrati

Prije odabira algoritma potrebno je razviti određenu intuiciju kako će se algoritam ponašati u traženom slučaju. Odličnu vizualnu usporedbu moguće je naći na stranicama: https://imgur.com/a/Hqolp



Usporedba optimizacijskih algoritama (Izvor: https://imgur.com/a/Hqolp)

Ako su kod ulaza podataka raštrkani podaci najvjerojatnije će se najbolji rezultati postići s nekim od optimizatora koji koriste adaptivnu stopu učenja. SGD će naći minimum, izuzetno je robustan, ali će mu trebati neko dulje vrijeme za navedene radnje. Jako mnogo radova još uvijek koristi SGD zato što je dobra stara testirana metoda koja funkcionira. [11], [29], [38]–[43], [30]–[37]

5. Izvori

[1] C. C. Aggarwal, *Neural Networks and Deep Learning*. Springer, 2018.

[2] A. Maas, A. Hannun, and A. Ng, “Rectifier Nonlinearities Improve Neural Network Acoustic Models,” 2013.

[3] A. F. Agarap, “Deep Learning using Rectified Linear Units (ReLU),” no. 1, 2018.

[4] S. Qiu, X. Xu, and B. Cai, “FReLU: Flexible Rectified Linear Units for Improving Convolutional Neural Networks,” 2017.

[5] I. J. Goodfellow, D. Warde-farley, and A. Courville, “Maxout Networks,” 2013.

[6] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” Nov. 2015.

[7] S. Scardapane, S. Van Vaerenbergh, S. Totaro, and A. Uncini, “Kafnets: kernel-based non-parametric activation functions for neural networks,” pp. 1–35, 2017.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[9] J. S. Bridle, “Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition,” in *Neurocomputing*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 227–236.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[11] S. Ruder, “An overview of gradient descent optimization algorithms,” Sep. 2016.

[12] A. C. Rencher and W. F. Christensen, *Methods of multivariate analysis*, Third edition. 2012.

[13] “Max-Margin Classifications,” in *Machine Learning for Multimedia Content Analysis*, Boston, MA: Springer US, 2007, pp. 235–266.

[14] C. Cortes and V. Vapnik, “SUPPORT-VECTOR NETWORKS,” *Mach. Learn.*,

vol. 20, no. 973, pp. 273–297, 1995.

[15] A. Natekin, A. Knoll, and O. Michel, “Gradient boosting machines, a tutorial,” 2013.

[16] “Max-Margin Classifications,” in *Machine Learning for Multimedia Content Analysis*, Boston, MA: Springer US, pp. 235–266.

[17] P. Grover, “5 Regression Loss Functions All Machine Learners Should Know,” 2018. [Online]. Available: https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0.

[18] R. Parmar, “Common Loss functions in machine learning,” 2018. [Online]. Available: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23.

[19] K. Janocha and W. M. Czarnecki, “On Loss Functions for Deep Neural Networks in Classification,” *Schedae Informaticae*, vol. 25, no. December, pp. 49–59, 2016.

[20] “Loss Functions and Optimization Algorithms. Demystified.” [Online]. Available: https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c. [Accessed: 18-Feb-2019].

[21] “Losses - Keras Documentation.” [Online]. Available: https://keras.io/losses/. [Accessed: 18-Feb-2019].

[22] M. Hazewinkel, *Encyclopaedia of mathematics*. Springer-Verlag, 2002.

[23] B. A. Dubrovin, A. T. Fomenko, and S. P. (Sergeĭ P. Novikov, *Modern geometry--methods and applications*. Springer-Verlag, 1992.

[24] J. Dean *et al.*, “Large Scale Distributed Deep Networks,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231.

[25] “Gradient Descent in a Nutshell – Towards Data Science.” [Online]. Available: https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0. [Accessed: 19-Feb-2019].

[26] O. Hern and P. Rico, “A Semiotic Reflection on the Didactics of the Chain Rule,” vol. 7, pp. 321–332, 2010.

[27] A. Karpathy, “Intro to neural nets for programmers.” [Online]. Available: http://karpathy.github.io/neuralnets/.

[28] P. Jay, “Back-Propagation is very simple. Who made it Complicated?” [Online]. Available: https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c.

[29] Nesterov Yurii E., “A method for solving the convex programming problem with convergence rate O (1/k^ 2),” *Dokl. akad. Nauk Sssr*, vol. 269, pp. 543–547, 1983.

[30] M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method,” Dec. 2012.

[31] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Dec. 2014.

[32] T. Dozat, “INCORPORATING NESTEROV MOMENTUM INTO ADAM,” *ICLR 2016 Work.*, no. 1, pp. 2013–2016, 2016.

[33] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, 2011.

[34] G. E. Hinton, “Neural networks for machine learning, Coursera Video.” 2012.

[35] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning,” in *Proceedings of the 26th Annual International Conference on Machine Learning - ICML ’09*, 2009, pp. 1–8.

[36] J. Pennington, R. Socher, and C. Manning, “Glove: Global Vectors for Word Representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.

[37] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient BackProp,” 1998, pp. 9–50.

[38] C. Darken, J. Chang, and J. Moody, “Learning rate schedules for faster stochastic gradient search,” in *Neural Networks for Signal Processing II*

Proceedings of the 1992 IEEE Workshop, pp. 3–12.

[39] Y. Kawamoto, K. Chatzikokolakis, and C. Palamidessi, “On the Compositionality of Quantitative Information Flow,” Nov. 2016.

[40] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” Feb. 2015.

[41] N. Qian, “On the momentum term in gradient descent learning algorithms,” *Neural Networks*, vol. 12, no. 1, pp. 145–151, Jan. 1999.

[42] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, “Advances in optimizing recurrent networks,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8624–8628.

[43] B. Yoshua, B.-L. Nicolas, and P. Razvan, “ADVANCES IN OPTIMIZING RECURRENT NETWORKS.”