VEGOVA | ELEKTROTEHNIŠKO-RAČUNALNIŠKA STROKOVNA ŠOLA IN GIMNAZIJA LJUBLJANA

Splošna matura:
Seminarska naloga pri računalništvu

APLIKACIJE IN INFORMACIJSKI SISTEMI

Mentor: Aleš Volčini prof.

Avtor: Jurij Juras, G 4. B

Ljubljana, April 2025

Povzetek

Živimo v dobi umetne inteligence, ki nas spremlja na vsakem koraku. Ampak ali sploh vemo, zakaj je tako pametna in kako je mogoče računalnik karkoli naučiti? Namen te seminarske naloge je ob primeru podrobno razložiti delovanje nevronskih mrež, pri čemer bo poseben poudarek na metodah učenja, ki jih te mreže uporabljajo.

Poleg tega bo naloga obravnavala primer programa, napisanega v Javi, ki se uči vzorec točk v dvodimenzionalnem prostoru. Program glede na koordinate točke določi, na kateri vzorec oz. na katero krivuljo točka spada. Poglobljeno bomo preučili, kako poteka analiza in prepoznavanje vzorcev s strani izbranega algoritma, ter opisali korake, ki so potrebni za dosego natančnih in učinkovitih rezultatov pri učenju.

Ključne besede: nevron, sloji, nevronska mreža, aktivacijske funkcije, program

Kazalo vsebine

Kazalo slik

1 Uvod

Nevronske mreže so temeljni gradnik umetne inteligence in ključnega pomena za razumevanje, kako lahko računalniki razmišljajo in se učijo. Gre za tehnologijo, ki poganja številne napredne funkcionalnosti sodobnega časa, od prepoznavanja obrazov in samovozečih avtomobilov do glasovno vodenih naprav.

Nevronske mreže so računalniški modeli, zasnovani po zgledu človeških možganov. Sestavljene so iz velikega števila medsebojno povezanih enot, imenovanih nevroni. Vsak nevron prejme določene vhodne informacije, jih obdela in posreduje naprej drugim nevronom v mreži.

Delovanje nevronskih mrež temelji na procesu, imenovanem usposabljanje (backpropagation). V tem procesu mreža prejme veliko količino podatkov in se sčasoma nauči prepoznavati vzorce ter povezave med njimi. Na ta način lahko natančneje napoveduje izide ali rešuje kompleksne probleme.

Mreže imajo več plasti, zaradi česar vhodne informacije potujejo skozi več nivojev nevronov, pri čemer vsak sloj opravi svojo obdelavo. S pomočjo matematičnih operacij in prilagajanja notranjih parametrov se mreža uči iz napak in tako izboljšuje predvidevanje rezultatov.

Nevronske mreže so izredno pomembne, ker omogočajo računalnikom, da obdelujejo kompleksne naloge na način, ki je bil še pred kratkim nepredstavljiv. Njihova sposobnost učenja in prilagajanja pomeni, da lahko opravljajo naloge, za katere programerji ne morejo napisati programa.

V svetu, že skoraj povsod srečamo programe, ki temeljijo na nevronskih mrežah in ljudem pomagajo pri raznih prepoznavanjih in nalogah:

- prepoznavanje bolezni, obrazov, besed
- gledena poslušano glasbo, predlagajo pesmi, ki so nam všeč
- izboljšajo nase pisanje

Nevronske mreže torej niso zgolj tehnološka zanimivost, temveč so postale del našega uskdana in vsak izmed nas se je že kdaj srečal z njimi. Le malo ljudi ve, kako delujejo in kako se učijo?

2 Sestava nevronske mreže

Sedaj si bomo na kratko ogledali sestavo nevronske mreže. Kasneje pa bomo ob pomoči izvorne kode tudi matematično razložili vse operacije in izračune, ki jih nevronske mreže uporabljajo.

Nevronske mreže so sestavljene iz več ključnih komponent, ki med seboj sodelujejo pri modeliranju kompleksnih vzorcev in odnosov v podatkovnih nizih.

2.1 Sloji

Sloji (Layers) so skupine nevronov z enako globino v nevronski mreži. Skupaj tvorijo strukturo modela in omogočajo mreži učenje zapletenih funkcij, ki povezujejo vhodne podatke z izhodi.

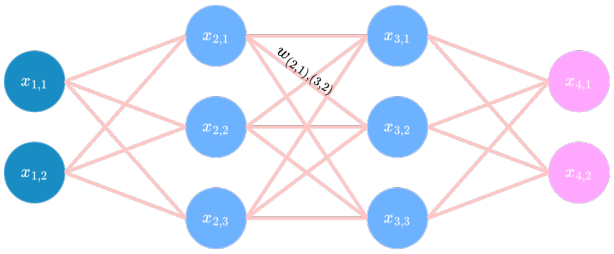
Nevronska mreža je organizirana kot zaporedje slojev, kjer vsak sloj prejme vhod iz prejšnjega sloja (ali neposredno iz vhodnih podatkov v primeru vhodnega sloja), izvede izračune in rezultate posreduje naslednjemu sloju. Ta postopek se nadaljuje do izhodnega sloja, ki poda končno napoved ali rezultat.

Sloje običajno delimo v tri skupine:

- **Vhodni sloj (obarvan temno modro, nevroni z ozanko $x_{1,i}$)** Predstavlja prvi sloj mreže. Njegova naloga je sprejem surovih vhodnih podatkov (npr. piksli slike, značilnosti iz tabele). Običajno ne izvaja nobenih transformacij, temveč podatke samo posreduje naprej.

- **Skriti sloji (obarvan svetlo modro, nevroni z ozanko $x_{2,i}$ in $x_{3,i}$)** To so vmesni sloji med vhodnim in izhodnim slojem. Vsak nevron v teh slojih izvede uteženo vsoto (forward propogation) svojih vhodov in uporabi aktivacijsko funkcijo (npr. ReLU, sigmoid). Več skritih slojev omogoča mreži, da se nauči kompleksnih, nelinearnih vzorcev v podatkih. Omogočajo hierarhično učenje – npr. pri slikah: nižje plasti prepoznajo robove, višje pa kompleksne oblike.

- **Izhodni sloj (obarvan roza, nevroni z ozanko $x_{4,i}$)** Zadnji sloj mreže, ki generira končni rezultat. Aktivacijska funkcija tega sloja je pogosto drugačna od aktivacijske funkcije v ostalih slojih. Največkrat je uporabljena funkcija softmax.



Slika 1: Nevronska mreža

Nevron v posameznem sloju je označen z dvema vrednostima naprimer: nevron $x_{3,2}$ je v tretjem sloju in drugi v stolpcu. Število nevronov v stolpcu pa je K. Uteži od nevrona do nevrona s eoznačujejo tako: utež iz nevrona $x_{2,1}$, do nevrona $x_{3,2}$ se označi z $w_{(2,1),(3,2)}$.

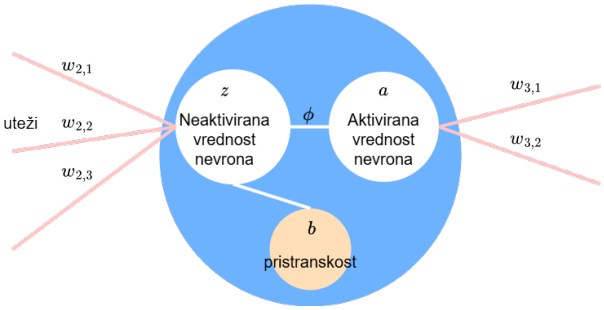
2.2 Nevron

Nevron je osnovni gradnik nevronske mreže. Navdih zanj izhaja iz nevronov, ki sestavljajo človeški živčni sistem. Nevrone delimo po plasteh. Vsak prejme vrednosti od nevronov iz prejšnje plasti $a_{2,i}$. Te vrednosti se pomnožijo z utežmi povezav med nevroni $w_{2,i}$. Nato se tem uteženim vsotam prišteje še pristranskost b, rezultat pa gre skozi aktivacijsko funkcijo ϕ.

$$z = w_{2,1}a_{2,1} + w_{2,2}a_{2,2} + w_{2,3}a_{2,3} + b$$

$$a = \phi(z)$$

Ta postopek določa, kako se vhodni signal prenese in preoblikuje skozi mrežo.



Slika 2: Slika nevrona $x_{3,2}$

2.3 Uteži

Vsi nevroni v mreži so med seboj povezani s povezavami, ki jih imenujemo uteži (weights). Vsaka povezava ima svojo numerično vrednost, ki določa, kako močno vpliva en nevron na naslednjega. Višja kot je vrednost uteži, večji vpliv ima signal iz prejšnjega nevrona na rezultat v naslednjem sloju.

Med procesom učenja te uteži prilagajamo. Cilj je, da se izhod mreže čim bolj približa želenemu rezultatu. Ta postopek se običajno izvaja s pomočjo algoritmov, kot je povratno širjenje napake, ki optimizira uteži glede na napako, izračunano med dejanskim in pričakovanim izhodom.

2.4 Pristranskost

Pristranskost (bias) je dodatna vrednost, ki se prišteje skupni uteženi vsoti v nevronu, deluje kot zamik, ki pomaga pri pomikanju aktivacijske funkcije levo ali desno. Kar pomeni, da je lahko nevron aktiviran tudi takrat, ko so vsi vhodni podatki enaki nič. Njena glavna naloga je, da omogoča modelu večjo fleksibilnost in pomaga pri odpravljanju stroge linearnosti v izračunih. S tem omogoča, da se nevronska mreža bolje prilagodi podatkom in nauči tudi kompleksnejše vzorce.

2.5 Aktivacijske funkcije

Aktivacijske funkcije (activation function) v nevronskih mrežah so matematične funkcije, ki določajo izhod posameznega nevrona a glede na vhod z, ki ga prejme. Njihova glavna naloga je uvesti nelinearnost v model, kar mreži omogoča, da uči in modelira kompleksne vzorce (nelinearne odnose) v podatkih.

Če aktivacijske funkcije ne bi bilo, bi bila tudi globoka mreža le skupek linearnih transformacij, kar pomeni, da bi se obnašala kot ena sama linearna funkcija, ne glede na število slojev.

Poznamo več vrst aktivacijskih funkcij:

- **sigmoid:** $a = \phi(z) = \frac{1}{1+e^{-z}}$
- **ReLU:** $a = \phi(z) = \max(0, z)$
- **softmax** $a = \phi(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$

2.6 Funkcija izgube

Funkcija izgube ali funkcija napake (cost function) je matematična funkcija, ki meri, kako dobro ali slabo model deluje. Izračuna koliko daleč so vse i napovedi modela $\hat{y}_i$ (to so aktivirane vrednosti nevrona v zadnjem sloju) od željenih rezultatov y_i.

Cilj učenja v nevronskih mrežah je minimizirati vrednost funkcije izgube $J(\hat{y}, y)$. Ko optimiziramo mrežo, iščemo parametre (uteži in pristranskosti), ki zmanjšajo vrednost te funkcije.

Poznamo več vrst funkcij izgube:

- **Mean Squared Error:** $J(\hat{y}, y) = \frac{1}{K} \sum_{i=1}^{K} (y_i - \hat{y}_i)^2$
- **Mean Absolute Error :** $J(\hat{y}, y) = \frac{1}{K} \sum_{i=1}^{K} |y_i - \hat{y}_i|$
- **Categorical Cross-Entropy:** $J(\hat{y}, y) = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$

3 Implementacija v Javi

Sedaj, ko poznamo osnovne gradnike nevronske mreže, si lahko ogledamo, kako sem mrežo implementiral v programskem jeziku Java.

V nadaljevanju bom predstavil strukturo kode, ključne razrede ter kodo, ki omogoča delovanje mreže, od inicializacije nevronov do učenja prek povratnega širjenja napake. Implementirali bomo zgolj funkcije, ki za namen naloge potrebujemo.

3.1 Aktivacijske funkcije v Javi

Aktivacijskih funkcij je veliko, in če bi jih želeli ročno spreminjati ali če bi vsak nevron uporabljal drugačno funkcijo, bi moral nepotrebno podvajati kodo.

Zato sem se odločil, da aktivacijsko funkcijo shranim kot spremenljivko z uporabo vmesnika (interface). Na ta način lahko vsakič enostavno določim, katero funkcijo naj posamezeni nevron ali sloj uporabi, brez nepotrebnega podvajanja kode.

Najprej sem definiral interface:

```
public interface Function {
    double execute(double input);
    double execute(double value, double[] values);
}
```

s katerim potem lahko definiramo različne funkcije:

- ReLU funkcija je uporabljena pri vseh nevronih razen pri output

  ```
  public static Function ReLUFunction = new Function() {
      final String name = "ReLUFunction";
      @Override
      public double execute(double input) {
          return Math.max(0, input);
      }
      @Override
      public double execute(double value, double[] values) {
          return ReLUFunction.execute(value);
      }
  };
  ```

- Softmax funkcija se uporablja le v izhodnem sloju, saj zelo učinkovito poudari aktivirane oz. prižgane nevrone ter hkrati zmanjša vpliv tistih, ki niso pomembni oz. ugasnjeni.

  Deluje tako, da vsakemu izhodu priredi verjetnostno vrednost med 0 in 1, pri čemer je vsota vseh izhodov enaka 1. Na ta način lahko model jasno izrazi, kateremu razredu najbolj verjame.

  ```
  public static Function softMaxFunction = new Function() {
      final String name = "softMaxFunction";
      @Override
      public double execute(double input) {
          System.out.println("Wrong use of softMax function!");
          return 0;
  ```

```
        }
        @Override
        public double execute(double value, double[] values) {
            double total = Arrays.stream(values).map(Math::exp).sum();
            double output = Math.exp(value)/total;
            return output;
        }
    };
```

Če aktivacijske funkcije definiramo kot spremenljivke, jih lahko shranjujemo posebej za vsak nevron, kar nam močno olajša implementacijo in poveča prilagodljivost modela.

Ker pa bomo v fazi učenja računali tudi, kako posamezna aktivacijska funkcija vpliva na funkcijo izgube, bomo poleg same funkcije potrebovali tudi njen odvod. Odvod aktivacijske funkcije je ključen za povratno širjenje napake, saj omogoča izračun, kako naj se uteži prilagodijo, da se napaka zmanjša.

Naredimo Interface:

```
public interface Derivative {
    double execute(double input);
    double execute(double value, double[] values);
}
```

In nato definiramo odvod:

- ReLU odovd:

```
        public static Derivative ReLUDerivative = new Derivative() {
            final String name = "ReLUDerivative";
            @Override
            public double execute(double input) {
                return (input>0)?1:0;
            }
            @Override
            public double execute(double value, double[] values) {
                return ReLUDerivative.execute(value);
            }
        };
```

- Pri računanju vpliva softmax funkcije na funkcijo izgube pogosto ne potrebujemo eksplicitnega odvoda funkcije izgube. To velja v primeru, ko kot funkcijo izgube uporabimo navzkrižno entropijo (cross-entropy).

  Kombinacija softmax aktivacijske funkcije in navzkrižne entropijske izgube omogoča pomembno poenostavitev. Odvod izgube glede na vhod v softmax funkcijo je podan kar z:

$$\frac{\partial J}{\partial z_i} = \hat{y}_i - y_i$$

  Kjer je:

  - $\hat{y}_i$ – napovedana verjetnost za razred i (rezultat softmax funkcije),

- y_i – dejanska vrednost za razred i (0 ali 1),
- z_i – vhodna vrednost v softmax funkcijo za razred i,
- J – vrednost funkcije izgube.

To pomeni, da nam ni treba posebej računati odvoda funkcije softmax in funkcije izgube, saj je rezultat neposredno razlika med napovedjo in dejanskim izidom.

Za oba interfaca imamo dve različni execute funkciji. Ko računamo ReLu funkcijo, potrebujemo samo vrednost nevrona, medtem ko pri softmaxu potrebujemo tudi ostale vrednosti, saj vplivajo na vrednost aktivacijske vrednosti.

3.2 Funkcije izgube v Javi

Tako kot smo definirali aktivacijske funkcije, bomo na enak način definirali tudi funkcije izgube. Z uporabo vmesnika lahko vsako funkcijo izgube zapišemo kot svojo implementacijo, kar omogoča fleksibilnost pri izbiri metode.

Najprej naredimo interface:

```
public interface Function {
    double execute(double[] neuron_values, double[] expected_values);
}
```

S katerim potem lahko definiramo funkcijo navzkrižne entropije.

Funkcija izgleda tako:

```
public static Function cceFunction = new Function() {
    String name = "categorical_cross_entropy";
    @Override
    public double execute(double[] neuron_values, double[] expected_values)
        double cost = 0;
        for(int i = 0; i < neuron_values.length; i++){
            cost += expected_values[i]*Math.log(neuron_values[i]);
        }
        return -cost;
    }
};
```

S to funkcijo bomo lahko izračunali, koliko se napoved našega modela razlikuje od dejanskega rezultata – torej, kako napačno je model predvidel izhod.

Odvoda te funkcije ne bomo potrebovali, saj smo že pri razlagi softmax funkcije pojasnili, da v kombinaciji z navzkrižno entropijsko izgubo pride do matematične poenostavitve.

3.3 Nevron v Javi

Za definiranje nevronov sem ustvaril nov razred (class), saj ima vsak nevron več lastnosti, ki jih mora shranjevati, ter metode, ki izvajajo različne funkcionalnosti.

```
// vrednost neaktiviranega nevrona
public double neuron_value;
// vrednost aktiviranega nevrona
public double neuron_value_output;
```

```
// stevilo nevronom pred njim
public int neuron_before_number;
// stevilo nevronov za njim
public int neuron_next_number;
// vrednost utezi na posamezen nevron pred njim
public double[] weights;
//  pristranskost (zacne se z 0, z ucenjem pa se spreminja)
public double bias = 0;

// shranjuje vpliv nevrona na cost funkcijo
public double delta = 0;

 // aktivacijska funkcija, ki jo uporablja nevron
Function activationFunction;
// odvod aktivacijske funkcije
Derivative activateDerivative;

// cost funkcija se definira samo ce je nevron v zadnjem sloju
Neural_Network.Functions.Cost.Function costFunction;
```

Nevron inicializiramo z metodo:

```
public Neuron(int neuron_before_number, int neuron_next_number) {
    this.neuron_before_number = neuron_before_number;
    this.neuron_next_number = neuron_next_number;
    weights = new double[neuron_next_number];

    if(neuron_next_number!=0) {
        setNewWeights();
        setNewBias();
    }
}
```

Ko nevron inicializiramo, moramo nastaviti začetne uteži, saj so te ključne za prenos informacij med nevroni. To opravimo s pomočjo posebne funkcije, ki za vsak vhod ustvari naključno utež. Te uteži se nato med učenjem prilagajajo glede na napake modela.

Funkcija za inicializacijo uteži izgleda tako:

```
public void setNewWeights(){
    double max = 1;
    double min = -max;
    for (int i = 0; i < neuron_next_number; i++){
        double randomNum = min + ((max - min) * rand.nextDouble());
        weights[i] = randomNum;
    }
}
```

Vsakič, ko se mreža uči ali izvaja napoved, se mora vsak nevron nastaviti na novo vrednost glede na vhodne podatke. Poleg tega mora izračunati še svojo aktivirano vrednost, ki jo potem posreduje naprej v mrežo. Če se aktivacijska vrednost izračuna še

na podlagi ostalih vrednosti nevronov v sloju, potrebuje tudi vrednosti ostalih nevronov. Zato zapišemo dve različni funkciji:

```
public void setNeuron_value(double neuron_value) {
    this.neuron_value = neuron_value;
    if(activationFunction!=null) {
        this.neuron_value_output =
        activationFunction.execute(neuron_value);
    }
}
public void setNeuron_value(double neuron_value,
double[] neuron_values) {
    this.neuron_value = neuron_value;

    this.neuron_value_output =
    activationFunction.execute(neuron_value, neuron_values);
}
```

Nevron mora biti sposoben izračunati tudi odvisnost svoje vrednosti glede na aktivacijsko funkcijo. Odvod aktivacijske funkcije pove, kako občutljiv je izhod nevrona na spremembe vhodnih podatkov. Z njegovo pomočjo lahko določimo, kako naj se uteži spremenijo, da se napaka zmanjša.

Funkcija enostavno pokliče odvod:

```
public double getActivationDerivative(double[] neuron_values){
    return activateDerivative.execute(neuron_value, neuron_values);
}
```

3.4 Sloji v Javi

Sedaj ko smo uspešno definirali vse potrebne komponente nevronov, se lahko lotimo naslednjega koraka v gradnji naše nevronske mreže. Ta korak vključuje postavitev nevronov v sloje. Sloji omogočajo, da se podatki pravilno pretakajo skozi mrežo in omogočajo mreži, da se uči in opravlja kompleksne naloge.

V vsakem sloju shranjujemo naslednje podatke:

```
boolean last_layer = false;

//koliko nevronov je v prejsnjem sloju
public int neuron_before_number;

//koliko nevronov je v tem sloju
public int neuron_number;

//koliko nevronov je v naslednjem sloju
public int neuron_next_number;

// to so nevroni v tem sloju
public Neuron[] neurons;
// to so nevroni v naslednjem sloju
public Neuron[] neurons_next;
```

Ko inicializiramo sloj z metodo:

```
public Layer(int neuron_before_number, Neuron[] neurons,
int neuron_next_number, Functions functions){
    this.neuron_before_number = neuron_before_number;
    this.neuron_number = neurons.length;
    this.neurons = neurons;
    this.functions = functions;

    this.neuron_next_number = neuron_next_number;
    if(neuron_next_number ==0){
        this.neuron_next_number = neuron_number;
        last_layer=true;
    }

    neurons_next = new Neuron[this.neuron_next_number];

    setUpNeurons();
    setUpNextNeurons();
}
```

Nevrone v posameznem sloju je potrebno tudi ustrezno inicializirati. To vključuje nastavljanje začetnih uteži za vsako povezavo ter določanje ustreznih aktivacijskih funkcij. Vsak sloj bo imel dostop tako do svojih nevronov kot tudi do nevronov v naslednjem sloju. Ta povezava omogoča hitrejše računanje vrednosti, saj lahko nevroni v vsakem sloju enostavno komunicirajo z naslednjim slojem.

Nevrone inicializiramo z dvema funkcijama:

- Sloj inicializira svoje nevrone z metodo:

  ```
  public void setUpNeurons(){
      for(int i = 0; i<neuron_number;i++){
          neurons[i] = new Neuron(neuron_before_number,
          neuron_next_number);
          neurons[i].setFunctions(functions, last_layer);
      }
  }
  ```

- Pri inicializaciji nevronov v naslednjem sloju jim še ne dodelimo aktivacijske funkcije, saj to opravi naslednji sloj sam, ko inicializira svoje nevrone. To opravimo z metodo:

  ```
  public void setUpNextNeurons(){
      for(int i = 0; i<neuron_next_number;i++){
          neurons_next[i] = new Neuron(neuron_before_number,   0);
      }
  }
  ```

Ko informacije potujejo skozi nevronsko mrežo, vsakemu nevronu izračunamo njegovo vrednost, jo nastavimo in nato aktiviramo s pomočjo aktivacijske funkcije. Vse to pa nam opravi ena metoda, ki nastavi vrednosti vsem naslednjim nevronom:

```
public void calculateNextNeurons(){
    double[] new_next_values = new double[neuron_next_number];
    for (int i = 0; i < neuron_next_number; i++){
        new_neuron_next_values[i] = 0;

        for (int j = 0; j < neuron_number; j++){
            double neuron_value = neurons[j].neuron_value_output;
            double weight = neurons[j].weights[i];

            new_next_values[i] += neuron_value*weight;
        }

        new_next_values[i] += neurons_next[i].bias;

    }
    for(int i = 0; i < neuron_next_number; i++){
        double new_value = new_neuron_next_values[i];
        neurons_next[i].setNeuron_value(new_value, new_next_values);
    }
}
```

Sedaj sledi implementacija metode, ki izračuna, kako močno posamezen nevron vpliva na skupno vrednost funkcije izgube. Ta korak je ključen pri učenju, saj omogoča mreži, da ustrezno prilagodi uteži in izboljša svoje napovedi. Da bomo bolje razumeli logiko izračunov in postopkov, ki jih bomo v naslednjih korakih implementirali, si bomo najprej ogledali matematično razlago postopka.

Kot vemo, v zadnjem sloju naše nevronske mreže uporabljamo funkcijo softmax, pri kateri neaktiviran nevron vpliva na aktivacijsko vrednost tudi ostalih nevronov. Zato potrebujemo najprej izračunati, koliko vsaka vednost neaktiviranega nevrona v zadnjem sloju vpliva na vsako aktivirano vrednost nevronov $\frac{\partial a_{4,i}}{\partial z_{4,k}}$ in koliko aktivirana vrednost nevrona vpliva na funkcijo izgube $\frac{\partial J}{\partial a_{4,i}}$.

Zato lahko napišemo:

$$\delta_{4,k} = \frac{\partial J}{\partial z_{4,k}} = \sum_{i=1}^{K} \frac{\partial a_{4,i}}{\partial z_{4,k}} \cdot \frac{\partial J}{\partial a_{4,i}}$$

Vrednost $\delta_{4,k}$, ki jo dobimo pri tem izračunu za vsak $z_{4,k}$ je napaka vrednosti nevrona, ki jo shranimo v kodi kot delta vrednost nevrona. Zgornji izraz je sestavljen iz dveh odvodov in sicer iz odvoda funkcije softmax, ki ga zapišemo kot :

$$\frac{\partial a_{4,i}}{\partial z_{4,k}} = a_{4,i} \cdot (\delta_{i,k} - a_{4,k})$$

kjer je $\delta_{i,j}$ kronecker delta, katere vrednost ko sta i in k je 0, v nasprotnem primeru pa je vrednost 1. Odvod funkcije izgube, pa se zapiše kot:

$$\frac{\partial J}{\partial a_{4,k}} = -\sum_{i=1}^{2} \frac{\partial \hat{y}_i}{\partial a_{4,k}} \cdot \frac{y_i}{\hat{y}_i}$$

V to enačbo vstavimo odvod funkcije softmax in po vseh okrajšavah in računanju dobimo enostavno enačbo:

$$\delta_{4,k} = \frac{\partial J}{\partial z_{4,k}} = \hat{y} - y$$

Sedaj, ko znamo izračunati napake nevronov v zadnjem sloju, lahko izračunamo še odvisnost neaktiviranih nevronov v predzadnjem sloju na funkcijo izgube $\frac{\partial J}{\partial z_{3,k}}$. Vsaka neaktivirana vrednost tega nevrona vpliva samo na svojo aktivirano vrednost nevrona, zato za izračun odvisnosti uporabimo odvod aktivacijske funkcije:

$$\delta_{3,k} = \frac{\partial J}{\partial z_{3,k}} = \frac{\partial a_{3,k}}{\partial z_{3,k}} \cdot \frac{\partial J}{\partial a_{3,k}} = \sigma'(z_{3,k}) \cdot \frac{\partial J}{\partial a_{3,k}}$$

Kjer je σ ReLU funkcija. Izračun prvega člena enačbe smo že razložili, sedaj pa pojasnimoše drugi člen $\frac{\partial J}{\partial a_{3,k}}$. Pri tem členu iščemo, kako aktiviran nevron vpliva na končno izgubo. A ker nevroni niso v zadnjem sloju, ne moremo uporabiti enačbe od prej, uporabiti moramo drugačno. Kar vpliva na končno funkcijo izgube, so nevroni v zadnjem sloju. Nevroni v predzadnjem sloju pa so z utežmi povezani na le te, kar pomeni, da če ugotovimo, koliko nevroni vplivajo na nevrone v zadnjem sloju $\frac{\partial z_{4,i}}{\partial a_{3,k}}$. Ker pa vemo, koliko nevroni v zadnjem sloju vplivajo na izgubo, lahko zapišemo:

$$\frac{\partial J}{\partial a_{3,k}} = \sum_{i=1}^{2} \frac{\partial z_{4,i}}{\partial a_{3,k}} \cdot \frac{\partial J}{\partial z_{4,i}}$$

Kjer je $\frac{\partial J}{\partial z_{4,i}}$ delta nevrona v zadnjem sloju. Če se spomnimo formule za računanje vrednosti nevronov $z_{4,i} = w_{3,1}a_{3,1} + w_{3,2}a_{3,2} + w_{3,3}a_{3,3} + b$ lahko z njo tudi poenostavimo prvi člen $\frac{\partial z_{4,i}}{\partial a_{3,k}}$ v tej enačbi. Funkcijo odvajamo po $a_{3,k}$ dobimo, da nevron vpliva na naslednji nevron za točno svojo utež $w_{3,i}$.

Torej, če zapišemo vse skupaj:

$$\delta_{3,k} = \sigma'(z_{3,k}) \cdot \sum_{i=1}^{2} w_{(3,k),(4,i)} \cdot \delta_{4,i}$$

In nam je uspelo, sedaj lahko izračunamo odvisnost vsakega nevrona na funkcijo izgube.

Poglejmo še en primer: recimo, da želimo izračunati delto drugega nevrona v drugem sloju:

$$\delta_{2,2} = \sigma'(z_{2,2}) \cdot \sum_{i=1}^{3} w_{(2,2),(3,i)} \cdot \delta_{3,i}$$

Kaj pa sedaj lahko z izračunanimi deltami počnemo? Z njimi lahko spreminjamo posamezno utež in posamezno pristranskost, ki vplivata na končen rezultat mreže. Poglejmo si, kako izračunamo vpliv uteži, ki povezujejo predzadnji in zadnji sloj, na izgubo.

$$\frac{\partial J}{\partial w_{(3,k),(4,j)}} = \frac{\partial z_{4,j}}{\partial w_{(3,k),(4,j)}} \cdot \frac{\partial J}{\partial z_{4,j}} = \frac{\partial z_{4,j}}{\partial w_{(3,k),(4,j)}} \cdot \delta_{4,j}$$

Sedaj lahko to utež tudi spremenimo,vendar si moramopred tem izmisliti napredek učenja, kar je številka, s katero se počasi premikamo do idealne uteži. Tako lahko nastavimo naš napredek $l = 0.003$ in nastavimo novo vrednost uteži:

$$w_{(3,k),(4,j)} = w_{(3,k),(4,j)} - l \cdot \frac{\partial z_{4,j}}{\partial w_{(3,k),(4,j)}} \cdot \delta_{4,i}$$

Za izračun odvisnosti pristranskosti nekega nevrona v predzadnjem sloju $\partial b_{(3,k)}$ na celotno izgubo, sestavimo formulo:

$$\frac{\partial J}{\partial b_{(3,k)}} = \frac{\partial z_{3,k}}{\partial b_{(3,k)}} \cdot \frac{\partial J}{\partial z_{3,k}} = \frac{\partial z_{3,k}}{\partial b_{(3,k)}} \cdot \delta_{3,k}$$

In če funkcijo za izračun nevrona $z_{3,k}$ odvajamo po $b_{(3,k)}$, dobimo 1, zato lahko zapišemo končno formulo kot:

$$\frac{\partial J}{\partial b_{(3,k)}} = \delta_{3,k}$$

Nato ga spremenimo oz. nastavimo s formulo:

$$b_{(3,k)} = b_{(3,k)} - l \cdot \delta_{3,k}$$

Sedaj imamo vse matematične formule in jih lahko začnemo zapisovati s kodo. V razredu Layer naredimo metodo z imenom calculate deltas, ki izračuna odvisnosti nevronov od celotne izgube, po zgornjih formulah.

```
public void calculateDeltas(double batch_size){
    double[] neuron_values = getNeuron_values();
    if(last_layer){
        for (int i = 0;i < neuron_number;i++) {
            double d = neurons[i].neuron_value_output -
            neurons_next[i].neuron_value;
            neurons[i].delta = d/batch_size;
        }
    }else {
        for (int i = 0;i < neuron_number;i++) {
            double da = neurons[i].getActivationDerivative(
            neuron_values);
            for (int k = 0; k < neurons_next.length; k++) {
                neurons[i].delta += da * neurons[i].weights[k] *
                neurons_next[k].delta;
            }
       }
    }
}
```

Funkcijo kličemo od zadnjega do prvega sloja, saj delte nevronov temeljijo na deltah nevronov iz prejšnjega sloja, razen pri zadnjem sloju. Funkciji podamo tudi velikost skupin (batch), ki pove nevronski mreži, koliko točk naenkrat obdeluje in lahko glede na te točke izračuna povprečno vrednost napake in posledično tudi povprečno delto. To informacijo pa uporabljamo za to, da se spremenjene uteži hkrati prilagodijo večjim rezultatom, kar omogoči boljši rezultat pri večih primerih. Ko nastavimo delte, lahko s funkcijo:

```
public void calculateWeightsBiases(double l_r){
    for (Neuron neuron : neurons) {
        if(!last_layer){
            for (int i = 0; i < neurons_next.length; i++) {
                double d = neurons_next[i].delta;
```

```
                    double a = neuron.neuron_value_output;
                    neuron.weights[i] -= l_r * (d * a);
                }
            }
        }
        for(Neuron neuron_next : neurons_next){
            neuron_next.delta = 0;
        }
    }
```

Spremenimo uteži in pristranskosti tako, da bodo bolj pravilne. Na koncu funkcije, ko še zadnjič uporabimo delte nevronov v naslednjem sloju, nastavimo na 0, saj se bo vanje začela računati nova napaka batcha.

Do sedaj smo obdelali vse funkcije v razredu layer z izjemo ene, s katero pa glede na vhodne podatke in pravilne vrednosti izračunamo napako.

```
public double calculateCost(double[] expected_neuron_values) {
    return output_layer.calculateCost(expected_neuron_values);
}
```

3.5 Mreža v Javi

Vse, kar smo do sedaj definirali, bomo z razredom združili v celoto, ustvarili sloje, nastavili potrebne spremenljivke in poskrbeli za klic ustreznih metod.

Najprej definiramo vse potrebne spremenljivke:

```
public Layer input_layer; //bljiznica do vhodnega sloja
public Layer output_layer; //bljiznica do izhodnega sloja

//stevilo nevronov v vhodnem sloju
public final int input_neurons_number;
//stevilo nevronov v izhodnem sloju
public final int output_neurons_number;
//stevilo skritih slojev
public final int hidden_layers_number;
//celotno stevilo slojev
public int layer_number;
//seznam v katerem se shranjuje stevilo nevronov v posameznem sloju
public int[] neurons_numbers;

//seznam vseh slojev
public Layer[] layers;
//hitrost ucenja
private double learn_rate = 0.1;

//tukaj so shranjene vse funkcije, ki je jih bo dedelilo nevronom
Neural_Network.Functions.Functions functions;
```

Z bližnicami si okrajšamo klicanje in postavljanje nevronov v prvem in zadnjem sloju, sedaj pa potrebujemo oblikovati še nekaj metod, s katerimi bomo lahko upravljali z mrežo.

Na koncu še definiramo metodo, ki poskrbi za ustvarjanje in povezovanje slojev v nevronski mreži.

```
public void makeLayers(){

    layers[0] = new Layer(0, new Neuron[input_neurons_number],
    neurons_numbers[1], functions);

    for (int i = 0; i< hidden_layers_number; i++){
        layers[i+1] = new Layer(neurons_numbers[i],
        layers[i].neurons_next, neurons_numbers[i+2], functions);
    }
    layers[layer_number-1] = new Layer(neurons_numbers[layer_number-2],
    layers[layer_number-2].neurons_next, 0, functions);

    input_layer = layers[0];
    output_layer = layers[layers.length-1];
}
```

V vsakem sloju definiramo nove nevrone in povemo, koliko jih je v naslednjih. Nato vsak sloj svoje nevrone shrani, generira pa nove, ki bodo potem shranjeni v naslednjem sloju, na ta način do posameznega nevrona lahko dostopata dva sloja.

Za nastavljanje začetnih vrednosti nevronov oz. vrednosti za kalkuliranje uporabimo metodo:

```
public double[] feedNetwork(double[] input_neuron_values){
    input_layer.setNeurons(input_neuron_values, true);

    for (int i = 0; i<layer_number;i++) {
        if(layers[i]==output_layer)break;
        layers[i].calculateNextNeurons();
    }
    return output_layer.getNeuron_values_output();
}
```

V kateri najprej nastavimo nevrone v prvem sloju na željene vrednosti, nato pa s pomočjo povezav, ki so ustvarjene med sloji, izračunamo vrednosti nadaljnih nevronov. Ko izračunamo vrednosti tudi zadnjega sloja, aktivirane vrednosti nevronov zadnjega sloja vrnemo.

Za izboljšanje rezultatov pa uporabimo metodo za učenje:

```
public void learnNetwork(Batch batch){
    double[][] expected_neuron_values_batch = batch.results;

    for (int i = 0; i < batch.size; i++) {
        double[] predicted_values = feedNetwork(batch.data[i]);

        batch.checkPredictions(predicted_values, i);

        output_layer.setNextNeurons(expected_neuron_values_batch[i]);
        output_layer.calculateDeltas(batch.size);
        batch.cost += output_layer.calculateCost
```

```
            (expected_neuron_values_batch[i])/batch.size;
        }
        output_layer.calculateWeightsBiases(learn_rate);

        for (int i = layer_number-2; i >= 0; i--){
            layers[i].calculateDeltas(1);
            layers[i].calculateWeightsBiases(learn_rate);
        }
    }
```

V procesu učenja mreže, potrebujemo najprej izračunati, kako se mreža odzove na dane podatke, da lahko na podlagi tega izračunamo, za koliko se je zmotila. Za učenje dobimo celotno skupino vrednosti, torej začetne vrednosti in končne rezultate, združene v razredu batch. To nam pomaga, ko mrežo učimo z večimi primeri naenkrat, saj lažje dostopamo do določene vrednosti. Prav tako je zelo priročen, saj se uspešnost določenega batcha shranjuje kar v tem razredu, tako da vsak batch zase ve, kako uspešno je mreža prepoznala njegove vrednosti.

Ko mreža pregleda celotno skupino podatkov in izračuna povprečno napako na le-te, lahko začnemo z učenjem. Za vsak sloj upoštevamo napačne vrednosti prejšnjega, zato začnemo pri zadnjem. Ker so v zadnjem vrednosti že izračunane, naše preračunavanje začnemo pri predzadnjem. Najprej izračunamo napako sloja, nato pa mu glede na napako nastavimo uteži in pristranskost.

3.6 Skupine v Javi

Do sedaj smo si pogledali celotno kodo za učenje nevronskih mrež, za pošiljanje podatkov v nevronsko mrežo pa uporabimo razred skupine (batch), ki nam olajša delo s shranjevanjem vseh vrednosti, rezultatov in napak. Vse kar potrebuje poleg shranjevanja še znati, je nastavljanje začetnih vrednosti, rezultatov in preverjanje pravilnosti ugotovitve nevronske mreže.

Začnimo s podatki, ki jih shranjuje:

```
public int size; // velikost posameznege skupine
public double[][] data; // vhodni podatki
public double[][] results; // pravilni rezultati

public double cost = 0; //napakacelotnege skupine
public int correct_predictions_count = 0; // stevilo pravilnih ugotovitev

public int input_neuron_number; //stevilo vhodnih nevronov
public int output_neuron_number; //stevilo izhodnih nevronov
```

Razred inicializiramo z metodo:

```
public Batch(int batch_size, int input_neuron_number,
int output_neuron_number){
    this.size = batch_size;
    this.input_neuron_number = input_neuron_number;
    this.output_neuron_number = output_neuron_number;

    this.data = new double[batch_size][input_neuron_number];
```

```
    this.results = new double[batch_size][output_neuron_number];
}
```

S katero samo nastavimo vse potrebne vrednosti. Za nastavljanje vseh vhodnih in pričakovanih vrednosti uporabimo metodo:

```
public void setBatch(double[][] batch_data, double[][]
batch_results){
    this.data = batch_data;
    this.results = batch_results;
}
```

Pomembno je, da so začetne vrednosti na istem mestu v zaporedju, kot pravilen rezultat, saj le tako vemo, katere pravilne vrednosti veljajo za določeno začetno vrednost. Definiramo tudi metodo, da hkrati nastavimo določeno začetno vrednost in pričakovano vrednost:

```
public void setBatchI(double[] data, double[] results, int i){
    this.data[i] = data;
    this.results[i] = results;
}
```

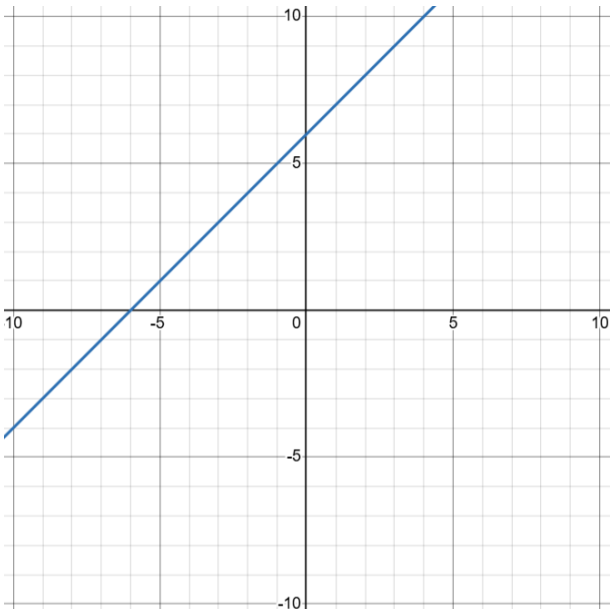
Razred mora samo še znati prepoznati, ali je vrednost, ki jo vrne mreža, pravilna ali napačna, zato definiramo metodo:

```
public void checkPredictions(double[] predicted_values,int i){
    if(predicted_values[0]>predicted_values[1]){
        if(results[i][0]==1)correct_predictions_count++;
    }else if(predicted_values[0]<predicted_values[1]){
        if(results[i][0]==0)correct_predictions_count++;
    }
}
```

Sedaj imamo pripravljeno vse za izgradnjo nevronske mreže, potrebujemo le še izziv, ki ga bomo dali nevronski mreži za reševanje.

4 Primer izziva

Za učenje nevronske mreže potrebujemo specifičen problem, katerega bo mreža skušala rešiti. Mreže lahko zasnujemo kot zelo obsežne in kompleksne, kar jih usposobi za obravnavo zapletenih izzivov. Vendar, da bi bolje razumeli delovanje mreže in da ne bi preveč obremenili računalnika med izvajanjem izračunov, lahko začnemo s preprostim primerom. Imamo koordinatni sistem, ki ga omejujeta vrednosti 10 in -10 in na njem narišemo funkcijo $f_1(x) = x + 6$.



Slika 3: Graf linearne funkcije

Na grafu si naključno izberemo n število točk. Polovico izbranih točk leži na funkciji $f_1(x)$, druga polovica pa nekje drugje v ravnini. V nadaljevanju bomo to uporabili za učenje nevronske mreže. Poskušali jo bomo naučiti, katere točke v tem koordinatnem sistemu ležijo in katere ne ležijo na $f_1(x)$.

4.1 Dfiniranje funkcij

Uvodni korak pri učenju mreže je, da potrebujemo naključno določiti točke na koordinatnem sistemu in jim pripisati, katere so in katere niso na $f_1(x)$.

Najprej zapišemo in v program vnesemo funkcijo, da jo bo lahko uporabljal. Kasneje bomo uporabljali tudi druge funkcije, zato zapišimo nov vmesnik:

```
public interface Function {
    int output_neuron_number = 2;
    boolean execute(double x, double y);
    double execute(double x);
}
```

Vsaka funkcija ima dve možnosti za izračunati. Lahko izračuna ali je točka (x, y) na tej funkciji, ter y pripadajoč x, kar bomo uporabili za računanje točk na funkciji, saj če bi z naključjem iskali točko, ki leži na funkciji, bi potrebovali ogromno časa, tako pa lahko izberemo naključen x in takoj določimo tudi y te točke, tako da bo na tej funkciji.

S pomočjo tega vmesnika lahko sedaj definiramo funkcijo:

```
public static Function diagonal = new Function() {
    public final int output_neuron_number = 2;
    @Override
    public boolean execute(double x, double y) {
        boolean f = y==x+6;

        return f;
    }

    @Override
    public double execute(double x) {
        return x+6;
    }
};
```

Sedaj, ko imamo definirano funkcijo, je potrebno nevronski mreži podati podatke, iz katerih se bo učila.

4.2 Definiranje začetnih vrednosti

Nevronski mreži lahko podamo kakršnokoli vrednost. Lahko le x in y koordinati točke, lahko pa tudi njun seštevek, zmnožek... Zato naredimo vmesnik, s katerim bomo lahko beležili, kaj bomo podali nevronski mreži:

```
public interface Input {
    double execute(double x, double y);
}
```

Torej vsako vrednost, ki jo bomo dali nevronski mreži, bomo izračunali glede na x in y. Čeprav se nam nebi to prav nič pomagalo pri določanju, nevronska mreža z dodatnimi podatki zelo natančneje oceni položaj točke. Definirajmo nekaj funkcij za računanje začetnih vrednosti:

```
public static Input x = new Input() {
    @Override
    public double execute(double x, double y) {
        return x;
    }
};

public static Input y = new Input() {
    @Override
    public double execute(double x, double y) {
        return y;
    }
};

public static Input sum = new Input() {
    @Override
    public double execute(double x, double y) {
        return x+y;
```

```
    }
};

public static Input xPow2 = new Input() {
    @Override
    public double execute(double x, double y) {
        return x*x;
    }
};

public static Input yPow2 = new Input() {
    @Override
    public double execute(double x, double y) {
        return y*y;
    }
};

public static Input xy = new Input() {
    @Override
    public double execute(double x, double y) {
        return x*y;
    }
};
public static Input sinX = new Input() {
    @Override
    public double execute(double x, double y) {
        return Math.sin(x);
    }
};
public static Input sinY = new Input() {
    @Override
    public double execute(double x, double y) {
        return Math.sin(y);
    }
};
```

4.3 Generiranje podatkov

Definirano funkcijo in enačbe za računanje začetnih vrednosti, sedaj uporabimo, za določitev točk s prej predstavljenimi mejami. Zato ustvarimo nov razred. Najprej definiramo spremenljivke tega razreda:

```
public int dataset_length; // stevilo podatkov
public int input_neuron_number; // stevilo vhodnih nevronov
public int output_neuron_number; // stevilo izhodnih nevronov

public double[][] data; // vsi podatki
public double[][] results; // vsi rezultati
```

```
public int batch_size; // velikost skupine, za v nevronsko mrezo

//omejitvi to k
public static int min = -10;
public static int max = 10;


public Function[] functions; //seznam funkcij na grafu
private Input[] inputs; //

private static final Random rand = new Random();

//kazalec na funkcije
int function_pointer = 0;
```

Spremenljivke inicializiramo z metodo:

```
public DataSet(int dataset_length, Input[] inputs, Function[] functions, in
    this.dataset_length = dataset_length;
    this.input_neuron_number = inputs.length;
    this.functions = functions;
    this.output_neuron_number = functions.length+1;
    this.batch_size = batch_size;
    this.inputs = inputs;

    data = new double[dataset_length][this.input_neuron_number];
    results = new double[dataset_length][this.output_neuron_number];
}
```

Mreža bo imela v zadnjem sloju nevronov za eno več, kot je podanih funkcij, saj se odloča, ali leži točka na kateri funkciji ali nikjer. Tam, kjer bo sigurno prepričan, da točka leži, bo vrednost blizu 1 in tam, kjer bo prepričan, da funkcija ne leži, bo vrednost blizu 0. Za določanje točk, ki ležijo na funkciji, uporabimo metodo:

```
public double[] getPointFxIn(Function function){
    double x = getRanDouble(min, max);
    double y = function.execute(x);
    while(Double.isNaN(y)||!function.execute(x,y)||y<min||y>max){

        x = getRanDouble(min, max);
        y = function.execute(x);
    }

    return makeInput(x, y);
}
```

Metoda dokler ne dobi validnega rezultata, kar pomeni, da je znotraj omejenega območja in da y te točke je število, generira nove točke. Ko dobi točko, določi njene vrednosti za v mrežo z metodo:

```
public double[] makeInput(double x, double y){
    double[] output = new double[input_neuron_number];
```

```
    for(int i = 0; i <input_neuron_number;i++){
        output[i] = inputs[i].execute(x, y);
    }
    return output;
}
```

Metoda shranjen seznam izračunanih vrednosti za v nevronsko vrne. Za določanje točk izven vseh funkcij pa uporabimo metodo:

```
public double[] getPointFxOut(Function function){
    double x = getRanDouble(min, max);
    double y = getRanDouble(min, max);
    while(function.execute(x,y)){
        x = getRanDouble(min, max);
        y = getRanDouble(min, max);
    }
    return makeInput(x, y);
}
```

Dokler ne dobimo točke, ki ne leži na funkciji, generiramo nove točke in nato vrnemo vse vhodne vrednosti za nevronsko mrežo.

Za izračun vhodnih podatkov, potrebujemo vedeti, na kateri funkciji mora točka biti, če je to sploh primer. V metodi:

```
public double[] getData(){
    double[] output = getPointFxOut(functions[0]);
    if(function_pointer==output_neuron_number-1){
        for(int i = 0;i<output_neuron_number-1;i++){
            if(functions[i].execute(output[0], output[1])){
                output = getPointFxOut(functions[i]);
                i = -1;
            }
        }
        function_pointer = -1;
        return output;
    }
    output = getPointFxIn(functions[function_pointer]);
    return output;
}
```

z uporabo kazalca na funkcije lahko preverjamo, katera funkcija je bila uporabljena prej. In če kazalec ne kaže več na funkcije, pomeni, da je čas, da postavimo točko, ki ni na nobeni funkciji. Moramo pa biti pozorni, da točka, ki leži prosto v ravnini, slučajno ni del katerekoli funkcije, zato uporabimo zanko, ki to konstantno preverja. Metoda vrne vse vhodne podatke za v mrežo.

Za računanje pripadajočih pravilnih vrednosti, pa uporabimo metodo:

```
public double[] getResults(double[] data){
    double[] output = new double[output_neuron_number];
    boolean in_fx = false;
    for (int i = 0; i<output_neuron_number-1;i++){
```

```
            if(functions[i].execute(data[0], data[1])){
                output[i] = 1;
                in_fx = true;
                break;
            }
        }
        if(!in_fx) output[output.length-1] = 1;
        return output;
    }
```

Z zanko preverimo, na kateri funkciji leži točka in na mesto te funkcije v novem zaporedju, v katerem so prvotno vrednosti 0, vpišemo 1. Če pa točka ni na nobeni funkciji, 1 postavimo na zadnje mesto v seznamu.

Napisane metode pa pokličemo z metodo, ki ustvari vse podatke za v nevronsko mrežo in rezultate, ki jim pripadajo.

```
public void createDataSet(){
    for (int i = 0; i < dataset_length; i++) {
        data[i] = getData();
        results[i] = getResults(data[i]);
        function_pointer++;
    }
}
```

Metoda v seznam podatkov in rezultatov vpiše točke, kazalec pa se za vsako vpisano točko poveča, kar povzroči, da bo točka vsako ponovitev na drugi funkciji in na koncu na nobeni, potem pa spet od začetka. Preden s podatki nahranimo nevronsko mrežo, jih moramo spraviti v n velike skupine. Te skupine rezultatov ustvarjamo sproti iz vseh podatkov z metodo:

```
public Batch setRandomBatch(){
    Batch batch = new Batch(batch_size, input_neuron_number,
    output_neuron_number);

    LinkedList<Integer> already_used_r = new LinkedList<Integer>();

    for (int i = 0; i < batch_size; i++) {
        int r;
        while(already_used_r.contains(r = getRanInt(0,
        dataset_length)));

        batch.setBatchI(data[r].clone(), results[r].clone(), i);
        already_used_r.add(r);
    }
    return batch;
}
```

V novo skupino shranimo vse potrebne vhodne podatke in rezultate, ter pazimo, da se v eni skupini podatki ne ponovijo. Na koncu skupino vrnemo.

5 Vzpostavitev nevronske mreže

Po tem, ko uspešno definiramo vse potrebno, lahko generiramo podatke in jih damo nevronski mreži, da se iz njih uči. Naredimo glavni razred, ki bo ustvaril bazo in nevronsko mrežo. Razred mora vsebovati tri metode, ena metoda bo metoda, ki bo nevronsko mrežo učila, druga bo nekaj naključnih vrednosti spustila skozi mrežo, da dobimo občutek, kako dobro se izkaže na posameznih točkah in na koncu še metoda, ki bo vse poklicala. Pomembno je, da je nevronska mreža globalno definirana, saj bodo vse metode dostopale do nje, zato zapišemo:

```
private static final Neural_Network.Functions.Functions functions = new
Neural_Network.Functions.Functions(Functions.ReLUFunction,
Functions.softMaxFunction,
Neural_Network.Functions.Cost.Functions.cceFunction );

public static Network n;
```

Definiramo funkcije, ki jih bomo uporabljali, v zadnjem sloju bomo uporabili softmax, v ostalih pa ReLU. Sedaj še definiramo metodi:

```
public static void learn(int number_of_epoch, DataSet data_set){
    for (int i = 0; i < number_of_epoch; i++) {
        Batch batch = data_set.setRandomBatch();
        n.learnNetwork(batch);
        if(i%100==0) {
            System.out.println(i+"-"+batch.cost+"-"+
            batch.correct_predictions_count);
        }
    }
}
```

Kar je metoda za učenje, ki naredi dano število ponovitev učenja.

Vse kar še potrebujemo, je metoda, ki bo klicala učenje nevronske mreže:

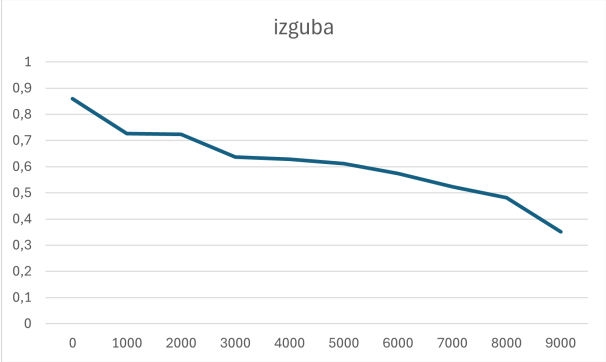
```
public static void main(String[] args) {
    DataSet data_set = new DataSet(1000, new Input[]{x, y},
    new Function[]{diagonal}, 100);

    data_set.createDataSet();

    n = new Network(new int[]{data_set.input_neuron_number, 5,5,
data_set.output_neuron_number}, functions);
    n.setLearn_rate(0.03);

    learn(9000, data_set);
}
```

V tem primeru naredimo bazo podatkov s tisočimi točkami, mreži bomo dali za input x in y funkcijo, ki jo bomo uporabljali, je linearna in velikost skupin je 100. V mreži smo definirali začetnih nevronov toliko kot je v bazi podatkov začetnih vrednosti in izhodov v nevronski mreži toliko, kolikor funkcij mora mreža prepoznati. Nastavili smo dva vmesna sloja s petimi nevroni, pri čemer je hitrost učenja 0.03. Mreži nato pokažemo 9000 skupin

po 100 točk. Na koncu pa na devetih primerih preverimo pravilnost nevronske mreže. Lastnosti mreže, ki bomo v nadaljevanju zapisali kot: $n([x, y], 5, 5, [d, 0])$, kjer d pomeni, da je točka na diagonali in 0 pomeni, da točka ni nikjer na funkciji.

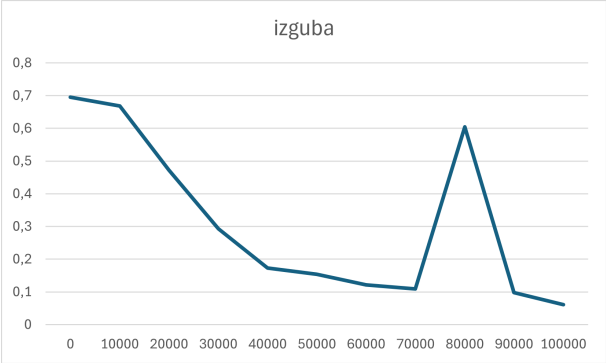
6 Testiranje

Čas je, da narejeno nevronsko mrežo stestiramo. Zaenkrat imamo mrežo $n([x, y], 5, 5, [d, 0])$ testirali jo bomo na 9000 skupinah po 100. (Vsi grafi bodo kazali izgubo z odvisnostjo od števila naučenih skupin.)



Slika 4: $n([x, y], 5, 5, [d, 0])$

Mreža iz izgube $0,85$ pride kar pod $0,4$. Pa poskusimo dobiti še boljši rezultat. Recimo, da mrežo stestiramo na 100000 skupinah po 100.



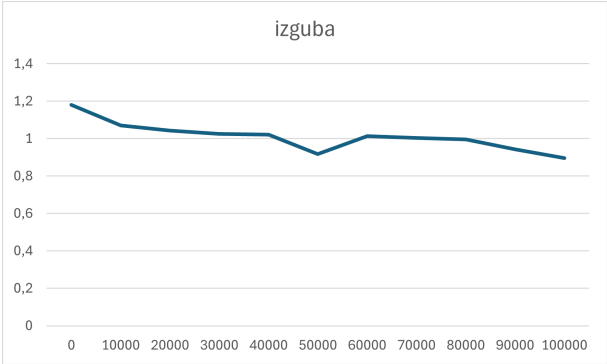
Slika 5: $n([x, y], 5, 5, [d, 0])$

Mreža se je zelo dobro izkazala, saj vidimo, da je končna izguba že pod $0,1$. A na 80000 ponovitvah izgleda, da je za veliko točk napačno odgovoril. Kako pa se naša mreža odzove na druge funkcije? K linearni funkciji dodajmo še funkcijo kroga s polmerom 4. Spišimo metodo:

```
public static Function circle = new Function() {
    public final int output_neuron_number = 2;
    private int r = 4;
    @Override
    public boolean execute(double x, double y) {
        boolean f = x*x+y*y == r*r;

        return f;
    }

    @Override
```

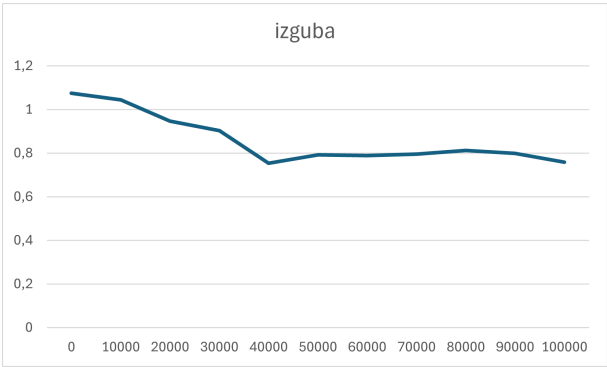
```
    public double execute(double x) {
        double a =Math.acos(x/r);
        return rand.nextDouble()>0.5?Math.sin(a)*r:-Math.sin(a)*r;
    }
};
```

Ponovno sestavimo mrežo $n([x, y], 5, 5, [d, c, 0])$ in stestirajmo.



Slika 6: $n([x, y], 5, 5, [d, c, 0])$

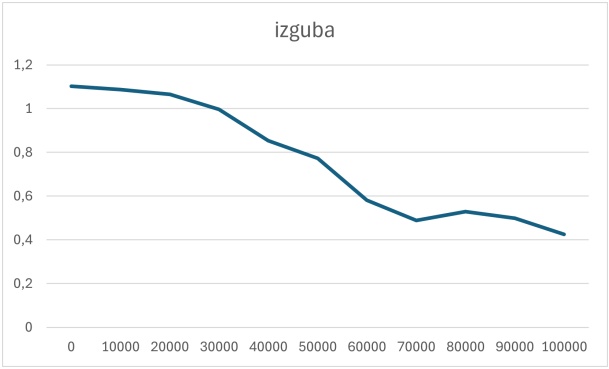
Vidimo, da naša mreža bolj težko prepoznava, kje leži točka. Poskusimo jo izboljšati, z dodajanjem začetnih vrednosti, torej: $n([x, y, x * y, x + y, sin(x), sin(y)], 5, 5, [d, c, 0])$



Slika 7: $n([x, y, x * y, x + y, sin(x), sin(y)], 5, 5, [d, c, 0])$

Že bolje, a da se še izboljšati uporabimo: $n([x, y, x*y, x+y, sin(x), sin(y)], 7, 5, 4, [d, c, 0])$.
Mreža postaja že bolj natančna. Otežimo ji delo z dodatno funkcijo:

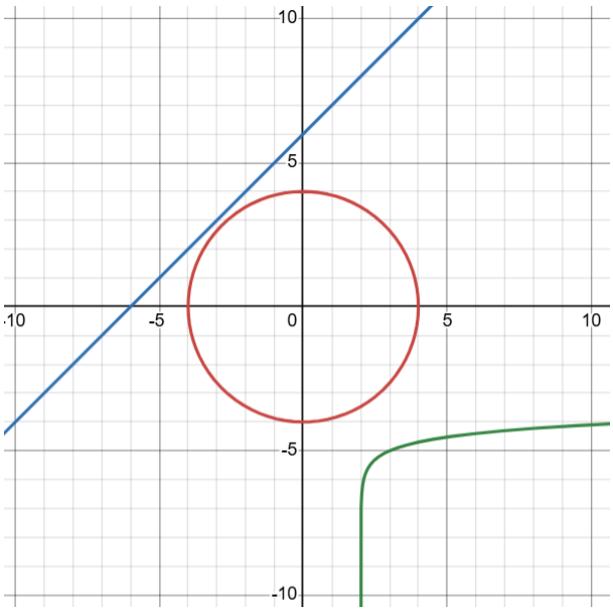
```
public static Function log = new Function() {
    public final int output_neuron_number = 2;
    @Override
    public boolean execute(double x, double y) {
        boolean f = y==Math.log10(x-2)-5;

        return f;
    }

    @Override
    public double execute(double x) {
```



Slika 8: $n([x, y, x * y, x + y, sin(x), sin(y)], 7, 5, 4, [d, c, 0])$

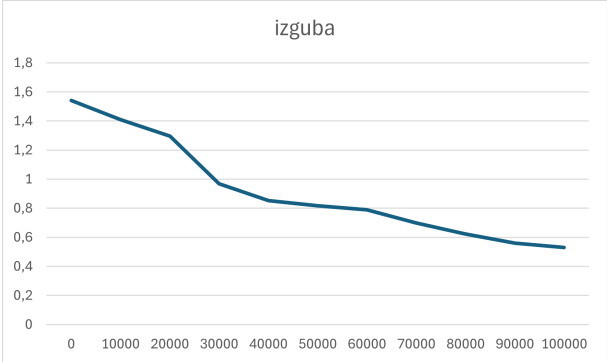
```
            return Math.log10(x-2)-5;
        }
};
```

Dodajmo mreži še to funkcijo: $n([x, y, x * y, x + y, sin(x), sin(y)], 7, 5, 4, [d, c, l, 0])$. Nato geometrijska ravnina, na katero postavljamo točke in ugiba, katera leži na kateri funkciji, izgleda takole:



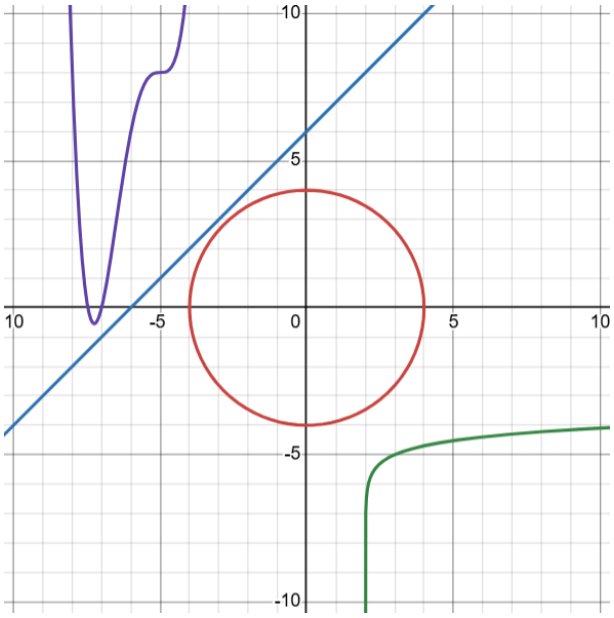
Slika 9: Funkcije na koordinatnem sistemu

Mrežo testiramo:



Slika 10: $n([x, y, x * y, x + y, sin(x), sin(y)], 7, 5, 4, [d, c, l, 0])$

Dobro se odnese! Dodajmo še funkcijo, ki je bolj kompleksna, in si oglejmo koordinatni sistem:

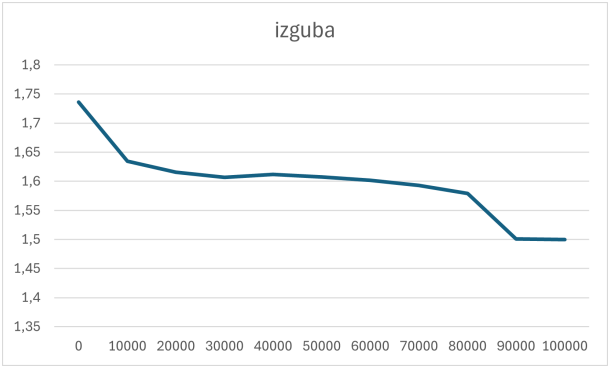


Slika 11: Funkcije na koordinatnem sistemu

Funkcijo zapišemo takole:

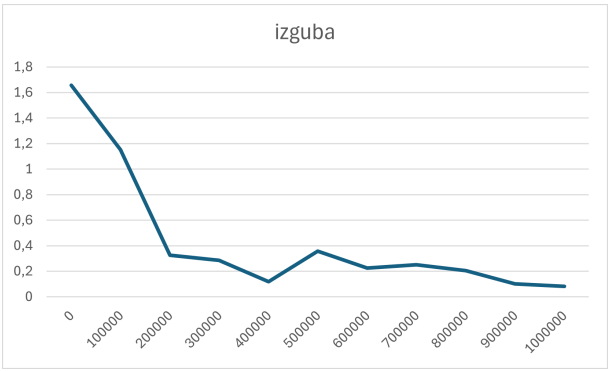
```
public static Function polinom = new Function() {
    public final int output_neuron_number = 2;
    @Override
    public boolean execute(double x, double y) {
        boolean f = y==Math.pow(x+5, 4)+3*Math.pow(x+5, 3)+8;
        return f;
    }
    @Override
    public double execute(double x) {
        return Math.pow(x+5, 4)+3*Math.pow(x+5, 3)+8;
    }
};
```

Ob testiranju mreže $n([x, y, x*y, x+y, sin(x), sin(y)], 7, 5, 4, [d, c, l, p, 0])$ lahko narišemo graf:



Slika 12: $n([x, y, x * y, x + y, sin(x), sin(y)], 7, 5, 4, [d, c, l, p, 0])$

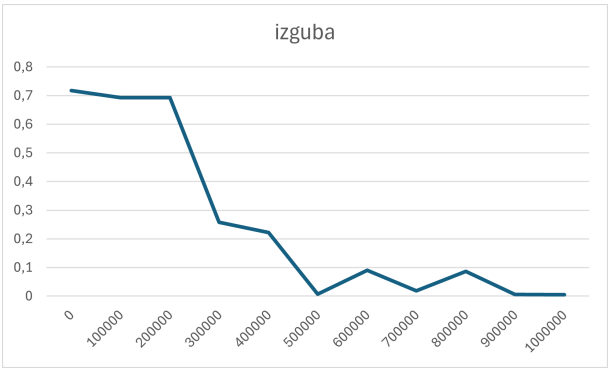
Izguba se ni premaknila niti pod 1, kar je precej slab rezultat. Nastavimo mrežo na $n([x, y, x * y, x + y, sin(x), sin(y), x * x, y * y], 9, 8, 8, 7, 5, 4, [d, c, l, 0])$, ker smo dodali tako veliko nevronov, se bodo uteži počasneje povečevale oz. zmanjševale, zato nastavimo več ponovitev, naprimer 1000000. In poglejmo rezultat:



Slika 13: $n([x, y, x * y, x + y, sin(x), sin(y), x * x, y * y], 9, 8, 8, 7, 5, 4, [d, c, l, p, 0])$

Mreža se zelo dobro izkaže! Iz začetne zgube, ki je skoraj 2, pride do zgube pod 0.1. Bi se dalo priti do zgube 0? Pri mrežah z mnogo nevroni je zelo veliko računanja, kar dolgo traja, zato je bolje, da probamo priti do izgube 0, na lažjih funkcijah, kot je samo krog. Uporabimo mrežo: $n([x, y, x * y, x + y, sin(x), sin(y), x * x, y * y], 9, 7, 5, 4, [c, 0])$, in iz podatkov narišemo graf (slika 14):

Super je uspelo! izguba se giblje med 0.1 in 0 kar je zelo presenetljivo.



Slika 14: $n([x, y, x * y, x + y, sin(x), sin(y), x * x, y * y], 9, 7, 5, 4, [c, 0])$

7 Zaključek

Nevronska mreža že deluje in zna prepoznavati točke v dvodimenzionalnem prostoru. To je šele začetek – z nevronskimi mrežami lahko rešujemo najrazličnejše probleme, kar sem si tudi zadal za naslednje korake.

S tem projektom sem se veliko naučil o nevronskih mrežah, a zavedam se, da je pred menoj še ogromno znanja, ki ga moram osvojiti. Moj naslednji cilj je, da se bo ta nevronska mreža naučila igrati igro štiri v vrsto in to na način, da bo igrala sama proti sebi.

Največji izziv pri projektu je bilo učenje same mreže. Sprva sem želel graditi na kodi, ki sem jo napisal pred dvema letoma, a sem že po nekaj dneh programiranja ugotovil, da vsebuje ogromno napak. Popravljanje teh me je stalo toliko časa in energije, da sem vmes skoraj obupal, a na srečo nisem.

8 Viri

- Kvarkadabra - časopis za tolmačenje znanosti [Dostopano: 13.4.2025], URL:
  `https://kvarkadabra.net/2017/08/kako-zgraditi-umetne-mozgane`
- Umetna Inteligenca – AI [Dostopano: 13.4.2025], URL:
  `https://umetnainteligenca.blog/kaj-je-nevronska-mreza-in-kako-deluje`
- Derivative of the Softmax Function and the Categorical Cross Entorpy [Dostopano: 8.4.2025], URL:
  `https://medium.com/data-science/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1`
- Activation function [Dostopano: 8.4.2025], URL:
  `https://en.wikipedia.org/wiki/Activation_function`
- Tenzor reshaping [Dostopano: 10. 4. 2025], URL:
  `https://en.wikipedia.org/wiki/Tensor_reshaping`

Izjava o avtorstvu

Izjavljam, da je poročilo v celoti moje avtorsko delo, ki sem ga izdelal samostojno.

Datum: Jurij Juras