Razpoznavanje vzorcev

Poročilo izbirnega projekta:

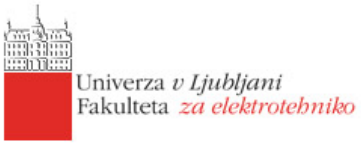
Simulacija tri-plastnega nevronskega omrežja

Študent: Tilen Tinta

Mentorja: izr. prof. dr. Simon Dobrišek, as. dr. Klemen Grm

Predmet: Razpoznavanje vzorcev

Datum: 9.1.2024



Vsebina

Ključne besede: XOR, isolet, nevronsko omrežje, feedforward, perceptron, backpropagation

---

1. Uvod

Kot zadnja laboratorijska vaja je bil tokrat izbirni projekt. V mojem primeru je bil to projekt 8 imenovan simulacija triplastnega nevronskega omrežja. Cilj tega je bilo napisati osnovno nevronsko mrežo ali natančneje tri-plastni perceptron, ga naučiti in testirati. Z enakimi pogoji je bilo nato potrebno narediti enako še z uradno knjižnico Pytorch. Dobljene rezultate iz obeh postopkov smo nato primerjali med seboj ter opazovali zanesljivost različnih postopkov učenja.

2. »Domači« tri – plastni perceptron

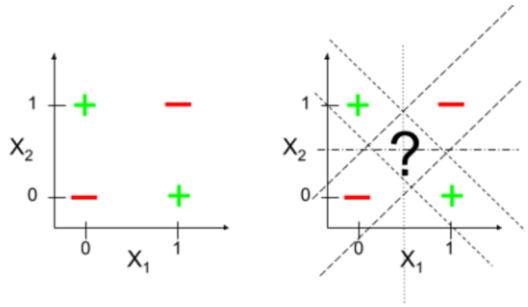
Kot že omenjeno je bila osnovna naloga, da sami napišemo kodo za tri plastno nevronsko omrežje. Za testiranje ali to pravilno deluje smo najprej uporabimo znan XOR problem. Ko smo se prepričani v pravilno delovanje naše mreže, smo lahko nato to preuredimo, da ustreza zbirki isolet. Enako smo tudi s to zbirko najprej mrežo naučimo in jo nato preizkusimo tako z učnimi kot z testnimi vzorci.

2.1. XOR problem

Eden najbolj osnovnih in enostavnih problemov za razumevanje in testiranje enostavnih nevronskih mrež je tako imenovan XOR problem. Logične funkcije kot so OR, AND, NOR, NAND so vse linearno ločljive, na kar funkcija XOR ni. Nelinearno ločljivost prikazuje tudi spodnja slika.

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 2-1: XOR

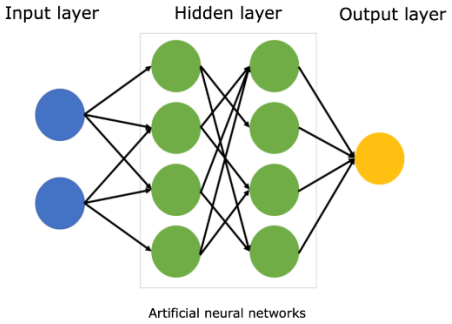


Slika 2-1: XOR problem, Vir: https://www.tech-quantum.com/solving-xor-problem-using-neural-network-c/

Iz te lahko vidimo, da z eno samo »črto« ni mogoče ločiti razredov med seboj glede na vhod. V ta namen se moramo poslužiti večjih mej oziroma nelinearnih mej, ki nam te razrede zaobjamejo.

2.2. Moja nevronska mreža

Za reševanje XOR problema ne zadostuje en nevron ampak zanj potrebujemo že zelo osnovno nevronsko mrežo. Natančneje tri-plastni perceptron. Sestavljen je iz treh plasti. Prvo ali vhodno plast v tem primeru sestavljata le dva nevrona zaradi natanko dveh vhodov, drugo oziroma skrito plast sestavljata ravno tako dva nevrona, tretjo plast imenovano tudi izhodna plast pa samo en nevron. Vsak nevron iz ene plasti je povezan z vsakim nevronom iz prejšnje plasti.



Slika 2-2: povezave med sloji, Vir: https://www.sciencelearn.org.nz/images/5156-neural-network-diagram

Vsaka povezava posebej ima svoje parametre oziroma uteži, katere s postopkom učenja ustrezno nastavljamo.

Cilj je, da ko mreži na vhoda podamo vrednosti X1 in X2 nam ta pravilno napove izhod oziroma naš Y.

2.2.1. Feedforward

Feedforward ali propagacija skozi mrežo je postopek pri katerem podatek, ki ga podamo na vhod potuje skozi nevronsko mrežo. Kakšen bo izhod oziramo kako bo ta potoval skozi je odvisno od parametrov mreže. Nanj vpliva število plasti, število nevronov v posamezni plasti, uteži posameznih povezav, biasi, uporabljena aktivacijska funkcija…

Za računanje izhodov posameznih nevronov se uporablja matrično množenje. To se izvaja po naslednji formuli.

$$H = \sigma(W_{ij} * I + B) \qquad (1)$$

Kjer je:

- H: izhodna vrednost nevrona
- σ*:* sigmoidna funkcija
- W_{ij}*:* matrika uteži posameznega vhoda
- I: matrika vhodnih vrednosti
- B: bias / odmik

Sigmoidna funkcija je matematična funkcija, ki priredi izhod nevrona, da se ta nahaja med 0 in 1. Ko je x enak 0 ima funkcija vrednost 0,5. Njena oblika ni linearna ampak spominja na črko s. Na področju strojnega učenja ji rečemo tudi aktivacijska funkcija in nam v nevronsko omrežje vnaša nelinearnost. Ta nam omogoča, da se naša mreža nauči kompleksnejših problemov. Zapišemo jo kot naslednjo formulo.

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

Bias ali zamik, ki bi mu lahko drugače rekli bolj elektrotehniško tudi »offset« je vrednost, ki vpliva na aktivacijo nevrona. Tudi tega ravno tako kot uteži nastavljamo pri učenju. Velikokrat ga predstavimo kot še en dodaten vhod. Tako ga lahko vključimo kar v matriko uteži kot dodaten stolpec na koncu matrike.

Tako uteži kot bias-i so na začetku določeni in se jih nato s postopkom učenja ustrezno spreminja. Za začetek so bile te naključno izbrane v območju med -1 in 1, kasneje pa spremenjene na -0,5 do 0,5. Ta sprememba je v mojem primeru ne glede na ostale parametre vedno podala boljši rezultat.

2.2.2. Učenje nevronske mreže

Pri inicializaciji kode za nevronsko mrežo so bile vse uteži ter bias-i postavljeni popolnoma naključno. Ko v takšno mrežo damo vhodne podatke je izhod ravno tako naključen, saj ta še ni naučena. V našem primeru je bilo učenje izvedeno s postopkom učenja z učiteljem. Pri tem postopku damo na vhod podatke oziroma značilke ter opazujemo izhode. Ker vemo kateremu razredu pripadajo ti podatki lahko na izhodu izračunamo napako med pravilno vrednostjo in dobljeno vrednostjo.

$$e = Prava\ v. - Dobljena\ v. \qquad (3)$$

Glede na rezultat izračuna nato po postopku vzvratnega učenja spreminjamo parametre mreže dokler ne dosežemo najboljšega razvrščanja. V nevronski mreži imamo več plasti in s tem postopkom lahko nastavimo le eno plast. Rešitev je, da računamo napako na vsaki plasti in tako dobimo podatek o potrebnih spremembah uteži za vsako plast posebej.

Ko imamo izračunane napake lahko te uporabimo za izračun spremembe uteži. To naredimo s sledečo formulo:

$$\Delta W_{ij} = lr * e * \sigma'(x) * H^T \qquad (4)$$

Kjer je:

- H: izhodna vrednost nevrona
- $\sigma'(x)$*:* odvod sigmoidne funkcije
- ΔW_{ij}*:* matrika uteži posameznega vhoda
- lr: learning rate – faktor učenja
- e: izračunana napaka

Ker vemo, da je odvod sigmoidne funkcije

$$\sigma'(x) = \sigma(x) * (1 - \sigma(x)) \qquad (5)$$

in da je naša vrednost napake izračunana iz izhodov že bila izračunana preko

sigmoidne funkcije, se ta enačba poenostavi v:

$$\Delta W_{ij} = lr * e * (O * (1 - O)) * H^T$$

(6)

Kjer je:

- H: izhodna vrednost nevrona
- O: matrika izhodnih vrednosti
- ΔW_{ij}: matrika uteži posameznega vhoda
- lr: learning rate – faktor učenja
- e: izračunana napaka

Z rezultatom izračuna popravimo vrednosti uteži, z enako enačbo le brez matrike H dobimo še vrednost gradienta. S tem izvedemo popravek bias vrednosti.

Ta postopek ponavljamo, kar nam povzroči, da vrednosti uteži in bias-ov konvergirajo k pravim vrednostim, katere bodo delovale kot ločilni pogoji v nevronski mreži.

2.2.3. Test prenaučenosti

Za učenje nevronske mreže moramo prej opisani postopek nastavljanja parametrov ponavljati. Ker seveda to ne moremo izvajati v nedogled ga moramo nekako omejiti. S prva sem za XOR problem enostavno nastavil fiksno vrednost ponovitev saj je bilo to za tako malo podatkov in tako enostavno mrežo to dovolj. V dodatni nalogi oziroma pri uporabi isolet podatkovne zbirke, kjer imamo veliko več parametrov pa je učenje smiselno ustaviti ko je točnost razpoznavanja največja.

V ta namen sem uporabil za učenje nastavljeno vrednost epoh. Te so v mojem primeru ponovitve učenja na celotni učni zbirki. Za podajanje značilk in prilagajanje parametrov nisem uporabil posebnih načinov kot je na primer z množicami (batchi)... Enostavno sem v vsaki iteraciji oziroma epohi učenja učno množico premešal in nato vzorec po vzorec podajal mreži ter prilagajal uteži.

Ob koncu vsake epohe sem trenutno naučeno nevronsko mrežo testiral z testno množico.

Če je ta dala boljšo točnost razpoznavanja od prejšnje iteracije sem učenje nadaljeval. Ko je bil rezultat testa enak ali celo manjši od prejšnje epohe naj bi se učenje ustavilo. Ker sem opazil, da to ni nujno najboljša naučenost mreže sem ob tem pogoju učenje ponovil še tri krat. Če se je v enem od treh ponovitev pojavila boljša točnost mreže sem nadaljeval z učenjem. S tem postopkom sem točnost mreže dvignil med 8 do 10%.

Težava, ki se tu pojavi je lahko ta, da je naša mreža preveč naučena na testne primere in poda samo pri teh tako velike točnosti saj njihovo razpoznavanje vpliva na količino učenja. Smiselno bi bilo, da bi te teste prenaučenosti opravljali z posebno množico vzorcev. Te bi izluščili kot novo skupino iz učne množice in z njimi testirali naučenost. Šele ko bi učenje končali, bi končno točnost določili s testnimi vzorci.

Tega v mojem primeru nisem izvedel saj učnih vzorcev ni veliko in bi z deljenjem te skupine imel še manj učnih vzorcev.

2.3. Isolet problem

Ko sem se prepričal v pravilno delovanje mreže, sem to začel izpopolnjevat in prilagajati da bo ustrezalo učenju zahtevane podatkovne zbirke. Zbirka imenovana Isolet je nabor 617-ih značilk pridobljenih iz posnetkov izgovorjenih črk angleške abecede. Te so bile posnete pri 150 govorcih, vsak pa je izgovoril vsako črko dvakrat. S tem so dobili 52 posnetkov za vsakega govorca. Te so nato razdelili v skupine po 30 govorcev ter tako dobili 5 skupin. Posnetke iz prvih 4 skupin uporabljamo kot učne vzorce, posnetke iz pete pa kot testne vzorce. Opaziti je možno da je vzorcev le 6238 in

ne 7800 kot bi bilo prav. Avtor omenja, da so bili ti odstranjeni zaradi šuma.

Vsak vzorec sestavlja 618 številskih vrednosti. Prvih 617 so vrednosti značilk, katere so že normalizirane. Zadnja vrednost podaja razred kamor sam vzorec spada.

Mrežo sem v ta namen prilagodil, da ima sedaj 617 vhodov, torej vsaka značilka svoj vhod. Ravno tako so bili prilagojeni izhodi, saj je v tem primeru 26 razredov, kar zahteva 26 izhodov če želimo, da se aktivira le en na razred. Število skritih nevronov sem za začetek pustil na 300, saj nisem imel predstave o točnosti delovanja mreže.

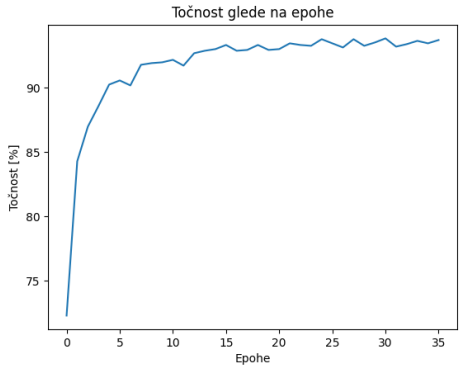
Z veliko poskusi sem se odločil za naslednje parametre:

- Faktor učenja: 0,05
- Nevroni skrite plasti: 200
- Max. epoh: 100
- Ponovitve prenaučenosti: 5

Ti parametri dajo sledeče končne vrednosti:

- Testni vzorci:
  Pravilnih: 1453,
  Napačnih:106,
  Točnost: 93.2%
- Učni vzorci
  Pravilnih: 6238,
  Napačnih: 0,
  Točnost: 100.0%

Učna uspešnost po posameznih epohah pa sledi:



Slika 2-3: Učna uspešnost po epohah

3. Nevronska mreža z PyTorch-om

Enako nevronsko mrežo sem poskušal implementirati še z Python-ovo knjižnico PyTorch. To je knjižnica katera vsebuje vse algoritme, katere se trenutno uporablja in so aktualni na področju globokega nevronskega učenja. Primerna je tako za domačo rabo kot za resno delo v industrijskih področjih.

Nevronsko mrežo z istimi lastnostmi za isti dataset sem poustvaril še s PyTorch-om, ter nato na tej preizkušal različne algoritme vzvratnega učenja. Ker je teh veliko sem se osredotočil le na štiri, ki so enostavni za implementacijo in pogosto uporabljeni. To so:

- SGD - Stochastic Gradient Descent
- Adam - Adaptive Moment Estimation
- RMSprop - Root Mean Square Propagation
- Adagrad - Adaptive Gradient Algorithm

Ker je bila moja prva implementacija učenja drugačna in ni bila po epohah na način kot je sedaj je za učenje računalnik potreboval veliko več časa. V ta namen sem izkoristil računanje na ločeni grafični kartici, katera podpira CUDA. To mi je samo učenje zelo pohitrilo. Z implementacijo epoh in nadzora učenja oziroma prenaučenosti se je čas zelo skrajšal tako da je to sedaj veliko manj opazno.

3.1. SGD algoritem

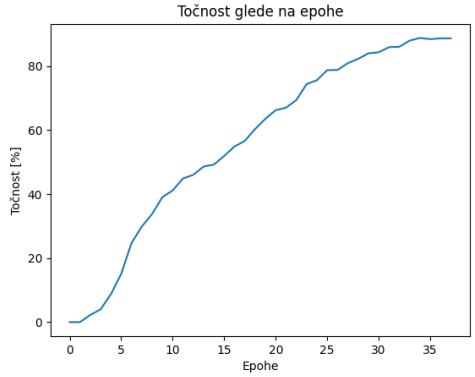
Temelji na gradientnem spustu, kar je podobno naši »domači« nevronski mreži. Razlika je v načinu, kako se posodabljajo parametri modela. Ta velikokrat uporablja množico vzorcev torej t.i. batch. To pohitri računanje oziroma kako hitro učenje konvergira k končni vrednosti, s tem zmanjša količino uporabljenega

pomnilnika za shranjevanje vzorcev, hkrati pa to zelo zmanjša možnost, da se optimizacija zatakne v lokalnem minimumu. Kot parametre ne potrebuje dodatnih posebnih argumentov le stopnjo učenja katero že uporablja »domača« mreža in moment, kateri le pohitri konvergiranje mreže ter preprečuje obtičanje v lokalnem minimumu. Ostali uporabljeni algoritmi temeljijo na enaki ideji.

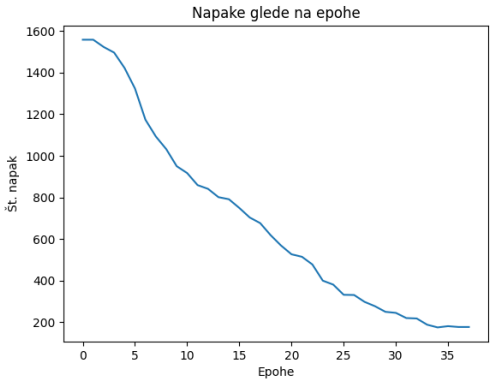
Rezultati učenja s tem algoritmom so sledeči:

- Testni vzorci:
  Pravilnih: 1428,
  Napačnih: 131,
  Točnost: 91.6%

- Učni vzorci:
  Pravilnih: 5971,
  Napačnih: 267,
  Točnost: 95.72%



Slika 3-1: Točnost - SGD



Slika 3-2: Napake - SGD

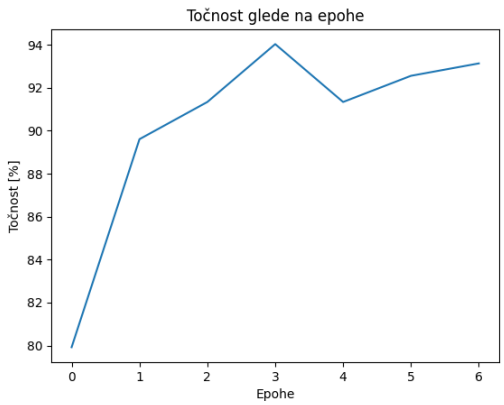
3.2. Adam algoritem

Algoritem temelji na dveh alg RMSprop in Momentum. Zelo je primeren za veliko dimenzionalne podatke kar v našem primeru so. Kot argument tudi ta sprejme stopnjo učenja dodatno pa še dva momenta. Označimo jo kot beta 1 in beta 2. Beta 1 nadzira odmik prvega momenta, kar je podobno kot zagon oziroma moment pri SGD-ju. Ta se navadno začne okrog 0.9, kar pomeni, da ohrani 90% prejšnjega pomika. Beta 2 nadzira odmik drugega momenta torej uteži kvadratov gradientov. Po navadi se začne okoli 0.999. Pomeni, da algoritem ohranja 99.9% prejšnjega odmika. Algoritem enako kot naša mreža najprej izračuna parametre nato pa jih z upoštevanjem bet šele spremeni.

Rezultat algoritma je:

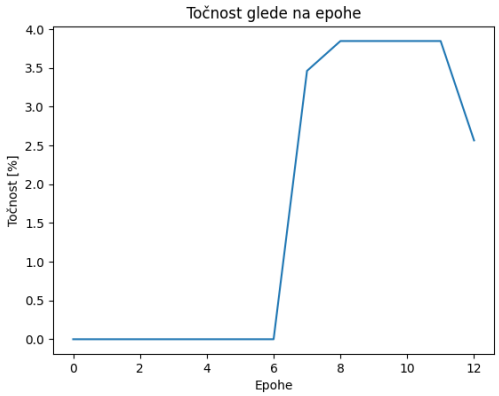
- Testni vzorci
  Pravilnih: 1461,
  Napačnih: 98,
  Točnost: 93.71%

- Učni vzorci:
  Pravilnih: 6216,
  Napačnih: 22,
  Točnost: 99.65%



Slika 3-3: Točnost - Adam

Slika 3-4: Napaka - Adam



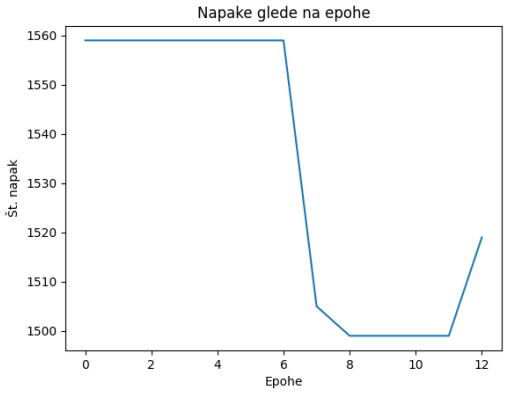
Slika 3-5: Točnost - RMSprop

3.3. RMSprop algoritem

Algoritem je zanimiv saj je uradno bil zelo nenavadno zasnovan in predstavljen v spletnem tečaju. Njegova prednost je da je sposoben zelo hitro reševat zelo velike spremembe v gradientih, kar je v nadaljevanju opaziti kot ne najbolj primeren za našo mrežo oziroma bi začetni parametri potrebovali posebno pozornost da bi ta pravilno deloval. Sprejme dva argumenta. To sta stopnja učenja ter faktor razpada (decay). Slednji določa kako hitro se stari gradienti pozabijo in ne upoštevajo pri računanju. Navadno je ta med 0.9 in 0.99. Dodamo mu lahko tudi argument, kateri preprečuje deljenje z nič in je zelo majhna vrednost.

Rezultati niso dobri, kar bi lahko pomenilo, da ta ni primeren za naš primer ali da vsi podani parametri niso pravilno nastavljeni. To je pri tem algoritmu tudi zelo značilna težava.

- Testni vzorci:
  Pravilnih: 60,
  Napačnih: 1499,
  Točnost: 3.85%
- Učni vzorci:
  Pravilnih: 240,
  Napačnih: 5998,
  Točnost: 3.85%



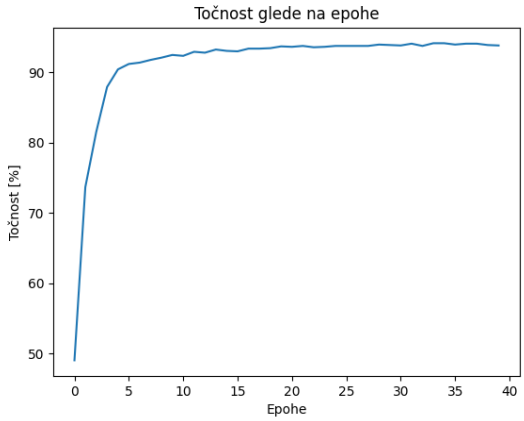
Slika 3-6: Napaka - RMSprop

3.4. Adagrad algoritem

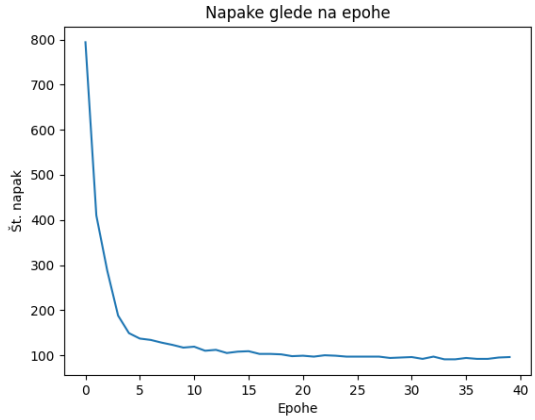
Bil je posebej zasnovan za reševanje dveh glavnih problemov pri učenju nevronskih mrež. To sta: izbira ustrezne stopnje učenja in prilagajanje te stopnje za vsak parameter posebej. Avtomatično prilagaja stopnjo učenja za vsak parameter modela na podlagi frekvence, s katero se parameter pojavlja v podatkih. To je uporabno pri delu z redkimi podatkovnimi značilnostmi, ker omogoča, da algoritem naredi večje posodobitve za redke parametre in manjše posodobitve za pogoste. Predstavljen je bil šele 2011. Argumenti so ponovno podobni kot pri večini saj večino teh ki jih potrebuje za izračun nastavlja sam in so odvisni od samih podatkov za učenje.

Rezultati mreže naučene s tem algoritmom so naslednji:

- Testni vzorci:
  Pravilnih: 1462,
  Napačnih: 97,
  Točnost: 93.78%
- Učni vzorci:
  Pravilnih: 6041,
  Napačnih: 197,
  Točnost: 96.84%



Slika 3-7: Točnost - Adagrad



Slika 3-8: Napaka - Adagrad

4. Zaključek

V tem izbirnem projektu smo lahko res podrobno spoznali in se poglobili v delovanje in matematiko za enostavnimi nevronskimi mrežami. Sami smo lahko dobili občutek kompleksnosti mrež in kako kateri parameter vpliva na delovanje, ter različnih načinov učenja.

V prvem delu smo spoznali problem in rešitev, ki lahko na prvi pogled izgleda zelo enostaven. Na koncu se izkaže, da je rešitev kar precej kompleksna in hkrati zanimiva.

Drugi del projekta, ki je deloval na precej večji množici podatkov nam je podal osnovno idejo kompleksnejše mreže in nam s tem pokazal, da lahko do istega rezultata oziroma dobre uspešnosti razpoznavanja pridemo le s spremembo parametrov mreže iz prve naloge.

Tretji del, ki zahteva uporabo knjižnice PyTorch nam šele poda pravo kompleksnost nevronskih mrež. Pri tem smo lahko preizkušali različne učne algoritme ter pri tem spoznali uspešnost vsakega ter kompleksnost računanja, ki jo vsak ima.

Pri zadnjem delu sem izkoristil še dodatno možnost, saj knjižnica to že podpira. Lahko sem preizkusil delovanje iste kode na različni strojni opremi. Ker se pri resni uporabi v podatkovnih centrih za učenje nevronskih mrež uporablja grafične kartice še posebej podjetja Nvidia, ki imajo CUDA jedra me je zanimala hitrost izvajanja moje kode. Pri manjši količini podatkov hitrost niti ni pretirano opazna. Ta razlika se pojavi šele ko zahtevamo večje število epoh ali spremenimo arhitekturo izvajanja kode.

Enako sem projekt izkoristil za pisanje bolje strukturirane kode. Ker mi objektno programiranje ni tuje zaradi C#, nisem pa najbolj domač s Python-om sem to poskušal implementirati še tukaj. Kljub temu, da še vedno vsa sintaksa ni najbolj po Python-ovih pravilih je struktura veliko bolj pravilna in pregledna kot koda iz preteklih laboratorijskih vaj.

5. Reference

- Prosojnice predavanj: Inteligentni sistemi v Avtomatiki, prof. Simon Dobrišek
- Prosojnice predavanj: Razpoznavanje vzorcev, prof. Simon Dobrišek
- Prosojnice vaj: Razpoznavanje vzorcev, as. Klemen Grm
- Nikola Pavešić, Razpoznavanje vzorcev (Ljubljana 2012)
- Spletni forumi za uporabo python funkcij
- 3Blue1Brown: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
- The Coding Train: https://www.youtube.com/playlist?list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh
- Make Your Own Neural Network knjiga: https://github.com/harshitkgupta/StudyMaterial/blob/master/Make%20Your%20Own%20Neural%20Network%20(Tariq%20Rashid)%20-%20%7BCHB%20Books%7D.pdf