Implementacija triplastnega nevronskega omrežja z vzvratnim učenjem na zbirki ISOLET

Avtor: Jakob Kreft
Mentor: asist. dr. Klemen Grm
Profesor: izr. prof. dr. Simon Dobrišek

Univerza v Ljubljani-Fakulteta za elektrotehniko, Tržaška cesta 25, 1000 Ljubljana
jk6684@student.uni-lj.si, klemen.grm@fe.uni-lj.si, simon.dobrisek@fe.uni-lj.si

Implementation of three layer neural network with backpropagation on ISOLET dataset

We developed a simple neural netowrk from scratch with backpropagation without the use of popular machine learning libraries such tensorflow or pytorch and tested the performance of our implementation on ISOLET dataset. The performance was onpar with the implemenation with pytorch. We also compared the performance of four different optimizers (SGD, Adagrad, RMSprop, Adam) and compared how parameters such as layer size, learning rate and momentum affect the performance.

1 Uvod

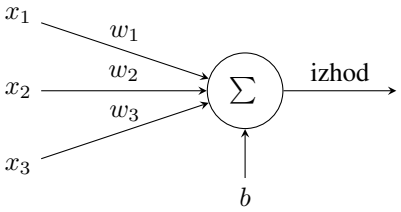
Pri tej vaji smo implementirali in raziskali učinkovitost triplastnega nevronskega omrežja, znanega kot perceptron. Osredotočili smo se na simulacijo in implementacijo perceptrona z uporabo vzvratnega učenja in gradientnega postopka, kot je opisano v literaturi [1]. Naš pristop vključuje testiranje omrežja z različnim številom nevronov v drugi vmesni prikriti plasti. Delo je zajemalo dva glavna eksperimenta: preizkus perceptrona na problemu XOR in na zbirki ISOLET, ki vključuje izgovorjave 26 angleških črk. Dodatno smo implementirali in preizkusili enako arhitekturo z uporabo knjižnice PyTorch, kar nam je omogočilo primerjavo med ročno implementiranim vzvratnim učenjem in naprednejšimi postopki gradientne optimizacije, ki so na voljo v knjižnici PyTorch. Tako smo primerjali tudi, kako različne optimizacijske metode (SGD, Adagrad, RMSprop, Adam) vplivajo na učinkovitost in uspešnost razvrščanja.

2 Metodologija

Naš cilj je bil razviti preprosto, a učinkovito nevronsko mrežo, brez uporabe popularnih knjižnic za strojno učenje, kot sta TensorFlow ali PyTorch. Nato razviti še kodo, ki uporablja knjižnico PyTorch, tako, da smo lahko lažje primerjali optimizacijske algoritme.

3 Razvoj nevronske mreže

Odločili smo se za implementacijo tri plastne nevronske mreže. Ta arhitektura vključuje vhodno plast, samo eno prikrito plast in izhodno plast. Vhodna plast je namenjena sprejemu značilk in je zato enake velikosti kot število značilk za vsak vzorec. Velikost skrite plasti lahko nastavljamo poljubno, dodajamo lahko tudi več skritih plasti. Izhodna plast pa je velikosti kot število razredov. Vsak nevron je povezan z vsemi nevroni iz prejšnje plasti.



Iz skice je razvidno, da pomnožimo vse vhode z njihovimi pripadajočimi utežmi, jih seštejemo in dodamo odmik, nato vrednost ustavimo v aktivacijsko funkcijo in dobimo rezultat nevrona, ki je lahko vhod v naslenjo plast.

Matematično lahko zapišemo tako:

$$a = f\left(\left(\sum_{i=1}^{n} x_i w_i\right) + b\right)$$

Kjer je:
- a izhod nevrona.
- x_i i-ti vhod v nevron.
- w_i urež povezana na i-ti vhod.
- b premik (angl. bias).
- n število vhodov v nevron.
- f aktivacijska funkcija (kot naprimer sigmoidna funkcija, ReLU, tanh, itd.).

4 Implementacijske podrobnosti

Pri implementaciji smo se osredotočili na razvoj kode, ki bo delovala kot osnova za možno prihodnjo nadgradnjo. Pri izdelavi kode smo se zgledovali po implementaciji v knjigi NNFS [2] in učbeniku [1]. Prav tako smo si pomagali z razlagami na spletu [3, 4, 5, 6, 7, 8, 9, 10, 11] Vsak korak v postopku vzvratnega širjenja (backpropagation) je bil natančno izveden z upoštevanjem gradientov za posodobitev uteži med nevroni. Uteži smo na začetku naključno inicializirali. Implementacijo smo izvedli v programskem jeziku Python, brez uporabe knjižnic za strojno učenje.

4.1 Podrobnejši pregled kode `perceptron.py`

Naša implementacija nevronske mreže v Pythonu, poimenovane `perceptron.py`, vključuje sedem razredov, ki skupaj tvorijo celotno strukturo in funkcionalnost mreže. Ti razredi so: Layer, SigmoidActivation, ReLUActivation, SoftmaxActivation, CategoricalCrossentropyLoss, MomentumOptimizer, NeuralNetwork. Vsak razred ima svojo specifično vlogo v ustvarjanju in učenju mreže, od inicializacije in propagacije do izračuna napake in optimizacije.

Inicializacija plasti in nevronov:
Razred `Layer` ustvari osnovno strukturo nevronske mreže, to so uteži in odmiki za eno plast. Uteži v vsaki plasti so naključno inicializirane s standardno normalno distribucijo, kar pomaga preprečiti začetno simetrijo v učenju. Odmiki (biases) so inicializirani z vrednostjo nič, kar omogoča nevtralno aktivacijo na začetku učenja.

Propagacija naprej (Forward Pass):
Metoda `forward` v vsakem razredu uporablja vhodne podatke in trenutne uteži za izračun izhodnih vrednosti. Ta proces se nadaljuje skozi vse plasti mreže, s čimer se iz vhodnih podatkov izračuna končni izhod mreže.

Propagacija nazaj (Backward Pass):
Metoda `backward` izračuna gradient napake glede na uteži, odmike in vhodne podatke. Ta gradient se nato uporabi za posodobitev uteži in odmikov v mreži, kar je ključno za učinkovito učenje. Metoda temelji na verižnem pravilu parcialnega diferencialnega računa in je osnovni mehanizem za učenje v globokih nevronskih mrežah.

Aktivacijske funkcije:
Za aktivacijo nevronov smo implementirali tri različne funkcije:

- Sigmoidna funkcija: $\sigma(x) = \frac{1}{1+e^{-x}}$. Ta funkcija stiska izhodne vrednosti v interval med 0 in 1. V našem primeru za učenje na bazi ISOLET se je izkazala za slabo.
- ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$. Ta funkcija je znana po preprečevanju problema izginjajočih gradientov. Uporabili smo jo v skriti plasti.
- Softmax funkcija: $\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$. Uporabljena na izhodni plasti, ta funkcija pretvori izhode v verjetnosti za večrazredno klasifikacijo.

Izračun napake in optimizacija:
Za izračun napake smo uporabili kategorično križno entropijo, ki je standard za večrazredne klasifikacijske naloge. Formula za kategorično križno entropijo je: $L_{\text{ce}}(y, \hat{y}) = -\sum y \log(\hat{y})$. Optimizator z momentom izboljšuje posodobitve uteži z upoštevanjem prejšnjih sprememb uteži (momentum), kar prispeva k bolj gladki in hitrejši konvergenci.

Učenje in validacija:
Proces učenja poteka z uporabo mini-batch gradientnega spusta, kjer algoritem iterativno prilagaja uteži na podlagi učnih podatkov. V tem procesu funkcija `train` deli celotno učno množico na manjše skupine (mini-batches). Za vsak mini-batch algoritem najprej izvede napoved s pomočjo metode `forward`. Nato izračuna napako (loss) med napovedjo in pravimi vrednostmi z uporabo funkcije kategorične križne entropije. Ta napaka je osnova za izračun gradientov v metodi `backward`. Ti gradienti predstavljajo smernice za posodobitev uteži v mreži in so izračunani z uporabo verižnega pravila diferencialnega računa.

Ko so gradienti izračunani, jih optimizator (v našem primeru `MomentumOptimizer`) uporabi za posodobitev uteži in odmikov v mreži. To storimo s prilagajanjem uteži v nasprotni smeri gradienta, kar vodi k zmanjšanju napake pri naslednji iteraciji. Velikost prilagoditve je odvisna od stopnje učenja in momenta, ki določata, kako hitro se mreža 'uči'. Ta postopek se ponavlja skozi vse epohe učenja, pri čemer vsaka epoha predstavlja en prehod skozi celotno učno množico.

Funkcija `validate` se uporablja po končanem učenju za ocenjevanje uspešnosti modela na testnem naboru podatkov. Ta funkcija izvede napoved s pomočjo trenutno naučene mreže na testnih podatkih in izračuna izgubo ter natančnost. Izguba prikazuje povprečno napako modela na testnih podatkih, medtem ko natančnost prikazuje delež pravilno klasificiranih primerov. To omogoča oceno, kako dobro model generalizira na novih, nevidenih podatkih, kar je ključno merilo uspešnosti nevronske mreže.

4.2 Podrobnejši pregled kode `percep_torch.py`

V tem razdelku predstavljamo implementacijo nevronske mreže z uporabo knjižnice PyTorch, ki omogoča bolj modularno in enostavno izvedbo kompleksnih nevronskih mrež. Koda `percep_torch.py` ilustrira, kako lahko s pomočjo PyTorcha dosežemo podobne funkcionalnosti, kot smo jih implementirali ročno v `perceptron.py`. Ta implementacija nam je omogočala lažji preizkus različnih optimizacijskih funkcij.

Nalaganje in predobdelava podatkov:
Koda začne z nalaganjem in predobdelavo Isolet podatkovnih zbirk. Podatki so normalizirani s standardnim skaliranjem in kodirani s pomočjo enoličnega kodiranja (OneHotEncoder).

Definicija modela:
Model nevronske mreže je definiran z uporabo PyTorchovega modula `nn.Sequential`, ki omogoča zaporedno povezovanje plasti. Struktura mreže vključuje vhodno plast, eno ali več skritih plasti z ReLU aktivacijsko funkcijo in izhodno plast. Ta pristop omogoča enostavno prilagajanje arhitekture mreže.

Nastavitev kriterija in optimizatorja:
Za izračun napake uporabljamo PyTorchovo implementacijo križne entropije (`nn.CrossEntropyLoss`). Optimizacijsko funkcijo smo spreminali, najbolje pa se je odnesla Adam optimizatorja, ki je priljubljena zaradi svoje učinkovitosti pri različnih problemih globokih nevronskih mrež.

Učenje modela:
Proces učenja uporablja mini-batch gradientni spust. Med vsako epoho se izvede iteracija skozi celotno učno množico, pri čemer se za vsak mini-batch izvede napoved, izračuna izguba in posodobijo uteži s pomočjo optimizatorja.

Ocenjevanje modela:
Funkcija `evaluate_model` omogoča ocenjevanje uspešnosti modela na testnem naboru podatkov.

Ta implementacija z uporabo PyTorcha omogoča hitrejše in bolj učinkovito učenje ter ocenjevanje modela v primerjavi z našo ročno implementacijo. PyTorch z močnimi abstrakcijami in optimizacijami olajša razvoj in eksperimentiranje z globokimi nevronskimi mrežami.

5 Podatkovna zbirka ISOLET

Podatkovna zbirka ISOLET je zbirka, ki vsebuje izgovorjave 26 angleških črk, posnetih od več kot 150 govorcev. Vsak primer v zbirki je zvočna datoteka, pretvorjena v niz značilk, ki opisujejo vzorec. Vsak primer ima 617 značilk.

Naša funkcija `load_dataset` najprej naloži podatke iz dveh datotek: `isolet1+2+3+4.data` za učni nabor in `isolet5.data` za testni nabor. Uporaba ločenih naborov podatkov za učenje in testiranje omogoča objektivno oceno uspešnosti modela na nevidenih podatkih.

Funkcija značilke skalira in enolično kodira. Funkcija (`StandardScaler`) je uporabljena za skaliranje značilk, kar pomeni, da so vrednosti značilk prilagojene tako, da imajo povprečje nič in standardno deviacijo ena.

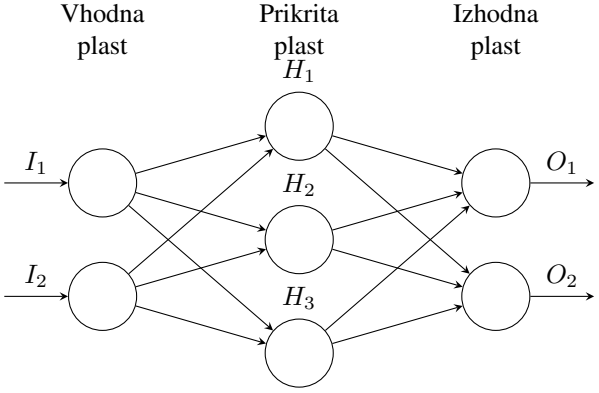
(`OneHotEncoder`) pa se uporablja za pretvorbo kategoričnih oznak - črk angleške abecede - v binarni format, ki ga lahko učinkovito obdela nevronska mreža. Ta postopek pretvori vsako oznako v vektor z dolžino, enako številu razredov (v našem primeru 26), kjer je samo en element vektorja enak 1, ostali pa so 0.

6 Rezultati

Pri vaji smo se osredotočili na testiranje štirih različnih optimizacijskih metod na podatkovni zbirki ISOLET, spreminjanjali število slojev, in število nevronov v skriti plasti, število epoh in momenta. Poleg tega smo pred tem izvedli učenje nevronskega sistema na klasičnem problemu XOR, da bi preverili osnovno delovanje in učinkovitost našega pristopa.

6.1 Učenje na problemu XOR

XOR (ekskluzivni ALI) je logična operacija, ki ima vrednost 1 samo, ko sta vhodni spremenljivki različni. Izbrali smo problem XOR, saj je znan kot primer, ki ne more biti rešen z enoplastno perceptronsko mrežo, kar predstavlja primer nelinearne klasifikacijske naloge. Naša koda, imenovana `xortron.py`, je uspešno rešila problem XOR z uporabo ene skrite plasti: nevronska mreža s strukturo 2-3-2 (dve vhodni enoti, tri v skriti plasti in dve v izhodni plasti). Poskusili smo tudi z dvoplastno strukturo 2-2, vendar se je izkazalo, da takšna mreža ni sposobna rešiti problema XOR.



Skica prikazuje povezavo med plastemi v kodi `xortron.py` Imamo dva vhoda, ki sprejemta en učni vzorec $[[0,0],[0,1],[1,0],[1,1]]$. Izhod bi sicer lahko bil en sam nevron, saj je pravilen rezultat lahko le $[0,1,1,0]$, vendar smo se zaradi posplošitve kode za obsežnejši problem ISOLET odločili za One-hot kodiranje tudi oznak pri tem enostavnem problemu torej: $[[0,1],[1,0],[1,0],[0,1]]$. Rezultat kode `xortron.py` po 1000 epohah problema XOR je bil naslednji:

$$\begin{bmatrix}[0,90 & 0,09]\\ [0,03 & 0,96]\\ [0,03 & 0,96]\\ [0,97 & 0,02]]\end{bmatrix}$$

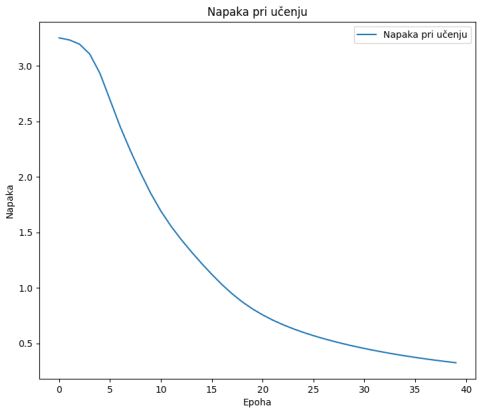
Iz česar je razvidno, da je problem rešen, če smatramo vse nad 0,5 kot 1 in pod 0,5 kot 0.

6.2 Učenje na zbirki ISOLET

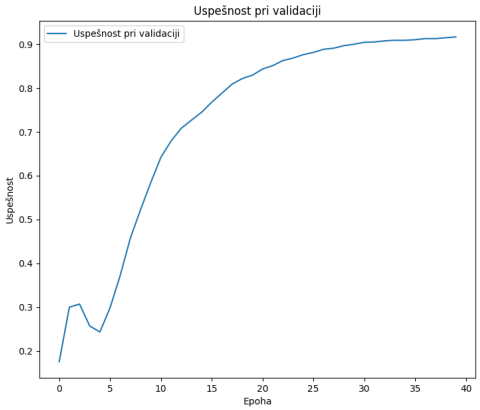
Našo implementacijo nevronske mreže smo trenirali in preizkusili na zbirki ISOLET. za lažjo vizalizacijo smo kodo `perceptron.py` nadgradili v `perceptron_viz.py`, ki nariše tudi grafe spreminjanja napake in spreminjanja uspešnosti razvrščanja na testni zbirki.

Najprej smo preizkusili delovanje nevronske mreže z momentom nastavlinim na 0.9 in dobili uspešnost razvrščanja na testni zbirki $0,95$. Potek skozi epohe prikazujeta grafa 3, 4. Rezultat je primerljiv z implementacijo z uporabo knjižnice PyTorch in optimizatorjem Adam.

Nato smo preizkusili delovanje nevronske mreže brez momenta oziroma nastaljenega na 0. Uspešnost se je posabšala in napaka se je počasneje zmanjševala. Tudi če smo povečali število epoh, nikoli nismo dosegli enako dobre uspešnosti kot z uporabo momenta. Potek spreminjanja napake in uspešnosti prikazujeta grafa 1 in 2.



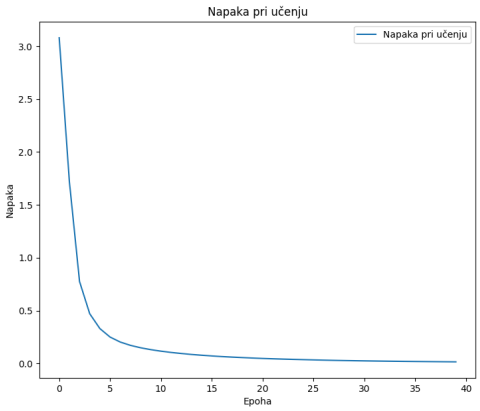
Slika 1: Graf spreminjanja napake skozi epohe brez optimizatorja z momentom.



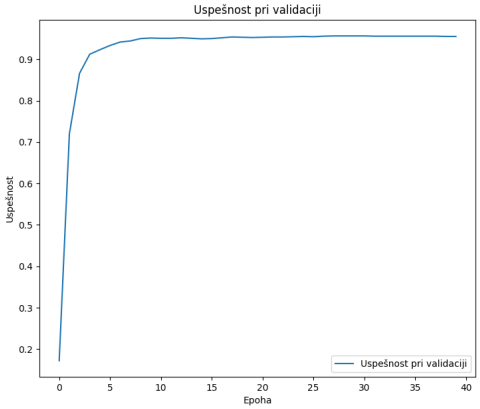
Slika 2: Graf spreminjanja uspešnosti skozi epohe brez optimizatorja z momentom.

6.3 Vpliv števila nevronov v prikriti plasti in vpliv učnega koraka

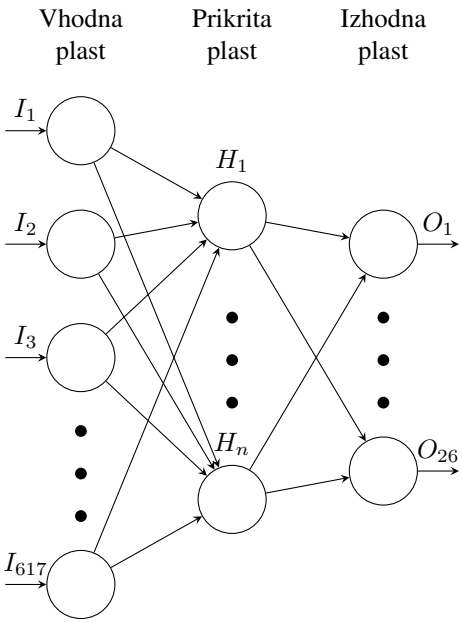
Z namenom preizkušanja različnih parametrov kot sta tudi velikost učnega koraka (learning rate) in število nevronov v prikriti plasti, smo kodo `percep_torch.py` predelali tako, da lahko poljubno nastavljamo meje dveh parametrov in natančnost na testni zbirki tudi izrišemo s pomočjo kode `heatmap_viz.py`, ki nam izriše toplotni graf uspešnosti razvrščanja (heatmap).



Slika 3: Graf spreminjanja napake skozi epohe z momentom nastavljenim na 0,9.

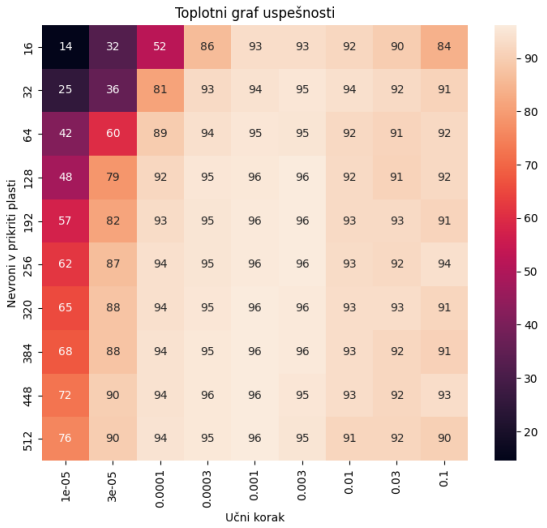


Slika 4: Graf spreminjanja uspešnosti skozi epohe z momentom nastavljenim na 0,9.



Skica prikazuje zgradbo našega nevronskega sistema. Imamo 617 vhodov, 26 izhodnih nevronov, število nevronov v prikriti plasti pa smo spreminjali.

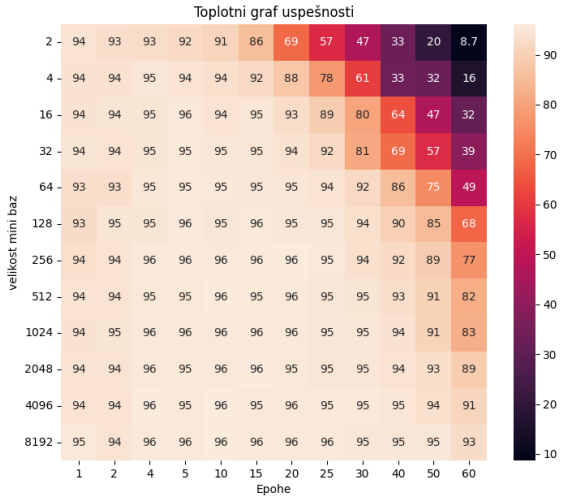
V tem testu smo spreminali učni korak od 0,00001 do



Slika 5: Toplotni graf uspešnosti razvrščanja, kjer smo spreminjali učni korak in število nevronov v prikriti plasti.

0,1 in število nevronov v skriti plasti od 16 do 512. Epoh je bilo vedno 10, velikost mini-baz pa 128. Iz grafa 5 je razvidno, da dobimo odlično uspešnost tudi pri majhnem številu nevronov v skriti plasti pri učnem koraku 0,001. Najslabši rezultat smo dobili pri majhnem številu nevronov in zelo majhnem učnem koraku.

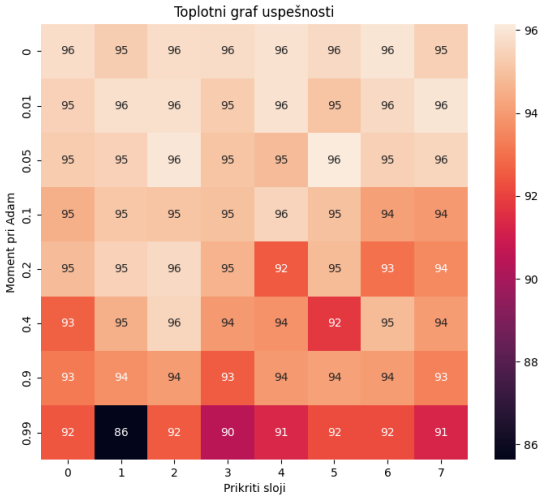
6.4 Vpliv števila epoh in velikosti mini baz



Slika 6: Toplotni graf uspešnosti razvrščanja, kjer smo spreminjali število epoh in velikosti mini baz.

V tem testu smo spreminjali število epoh od 1 do 60 in velikost mini baz od 2 do 8192. Iz grafa 6 je razvidno, da dobimo odlične uspešnosti že po eni sami epohi, s povečevanjem števila epoh nad 20 pa opazimo, da uspešnost začne upadati, kar nakazuje na prenaučenje mreže na učni zbirki in slabšo generalizacijo znanja. Pri premajhni velikosti mini-baz je prišlo do poslabšanja uspešnosti. Iz grafa pa je ne razvidno to, da se je hitrost učenja bistveno razlikovala. velikost mini baze 256 in 5 epoh je pokazalo odlično uspešnost in hkrati zelo učinkovito in hitro učenje.

6.5 Vpliv števila plasti in velikosti momenta



Slika 7: Toplotni graf uspešnosti razvrščanja, kjer smo spreminjali število prikritih plasti in velikosti momenta.

V tem testu smo spreminjali število skritih plasti nevronske mreže od 0 (brez skritih plasti) do 7, poleg tega, smo spreminjali tudi moment pri optimizatorju Adam od 0, do 0.99. Iz grafa 7 je razvidno, da je problem rešljiv že brez skritih plasti, kar ponazarja, da je klasifikacija primerov iz Isolet zbirke linearen problem. Razvidno je tudi, da nam je moment v optimizaciji Adam škodoval.

6.6 Primerjava optimizacijskih algoritmov

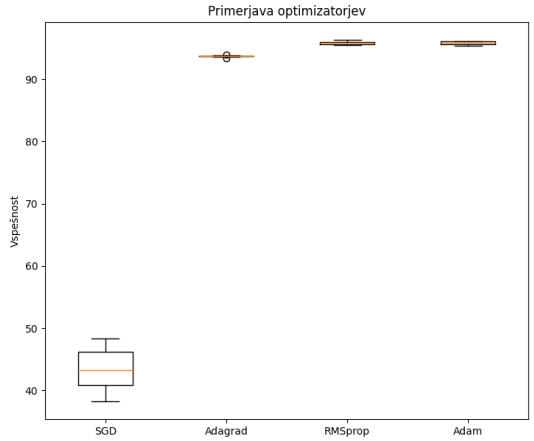
V tem testu smo preizkusili štiri različne metode optimizacije: Stohastični gradientni spust (SGD), Adagrad, RMSprop in Adam. Pri tem smo ohranili konstantne parametre, kot so število nevronov v skriti plasti, hitrost učenja in momenta, kar nam je omogočilo neposredno primerjavo med algoritmi in njihovim vplivom na učinkovitost učenja ter končno uspešnost klasifikacije.

- **Stohastični gradientni spust (SGD)**: Ta metoda na podlagi učnih podatkov mini baze izračuna gradient in posodobi uteži. Je preprost in zato lahko SGD traja dlje da konvergira. V našem primeru se je izkazal za najslabšega 8.
- **Adagrad**: Ta optimizator samodejno prilagaja učni korak za vsako parametrično dimenzijo. Vendar pa se je v našem primeru Adagrad izkazal za manj učinkovitega, saj njegova stalna akumulacija kvadratov gradientov včasih vodi do prehitrega zmanjšanja učnega koraka.
- **RMSprop**: Ta optimizator je variacija Adagrada, ki odpravlja njegov problem stalnega zmanjševanja stopnje učenja. RMSprop to doseže z uporabo

drsečega povprečja kvadratov gradientov, kar omogoča boljše prilagajanje učnega koraka. Ta pristop je se je v našem primeru izkazal za odličnega.

- **Adam (Adaptive Moment Estimation)**: Adam kombinira prednosti RMSpropa in Adagrada, saj ne samo, da ohranja drseče povprečje kvadratov gradientov (kot RMSprop), temveč tudi drseče povprečje samih gradientov. Ta pristop omogoča Adamu, da prilagodi stopnjo učenja na bolj dinamičen način, kar lahko vodi do hitrejše in stabilnejše konvergence. V naših testih je Adam skupaj z RMSpropom dosegel najvišje uspešnosti.

Iz grafa 8 je razvidno, da sta Adam in RMSprop dosegala najvišje natančnosti, kar nakazuje na njuno učinkovitost v našem primeru. Nasprotno pa Adagrad ni izkazal optimalne uspešnosti, medtem ko je bil SGD manj učinkovit, z nižjo končno natančnostjo. To potrjuje, da lahko izbira primernega optimizacijskega algoritma bistveno vpliva na uspešnost nevronske mreže.



Slika 8: Graf kvantilov primerjave optimizacijskih algoritmov SGD, Adagrad, RMSprop in Adam.

7 Zaključek

V tem delu smo uspešno razvili in testirali triplastno nevronsko mrežo brez uporabe priljubljenih knjižnic za strojno učenje. Naša implementacija je temeljila na vzvratnem učenju in gradientnem postopku. Preizkusili smo jo na problemu XOR in na podatkovni zbirki ISOLET. Uspešnost implementacije na zbirki ISOLET je bila primerljiva z rezultati, doseženimi z uporabo knjižnice PyTorch, kar potrjuje učinkovitost našega pristopa.

Eksperimentalno smo preizkusili vpliv različnih optimizacijskih algoritmov, kot so SGD, Adagrad, RMSprop in Adam, na uspešnost klasifikacije. Ugotovili smo, da sta se Adam in RMSprop izkazala za najučinkovitejša, medtem ko sta Adagrad in zlasti SGD pokazala slabše rezultate. To poudarja pomembnost izbire ustreznega optimizacijskega algoritma za določen problem strojnega učenja.

Analiza vpliva števila plasti in nevronov v skriti plasti je razkrila, da je problem ISOLET mogoče rešiti tudi brez skritih plasti, kar kaže na linearno naravo problema.

Poleg tega smo s preizkušanjem različnih velikosti učnih korakov, števila epoh in velikosti mini-baz pridobili vpogled v to, kako ti parametri vplivajo na učenje in uspešnost nevronske mreže. Ugotovili smo, da lahko optimalna izbira teh parametrov znatno izboljša uspešnost in učinkovitost modela in prepreči tveganje prenaučenja.

Nadaljnje raziskave bi lahko vključevale razširitev naših testov z uporabo različnih podatkovnih zbirk.

8 Priloge

Vse omenjene kode prilagam poleg dokumenta v mapi.
`perceptron.py`,
`perceptron_viz.py`,
`percep_torch.py`,
`percep_heat.py`,
`OptCompare.py`,
`xortron.py`,
`heatmap_data.npy`,
`heatmap_viz.py`,
`isolet1+2+3+4.data`,
`isolet5.data`

Literatura

[1] N. Pavešić, *Razpoznavanje vzorcev*, 1 2012.

[2] H. Kinsley and D. Kukieła, *Neural Networks from Scratch (NNFS) https://nnfs.io*, 1 2020.

[3] K. Sinha, "Simple neural net backward pass," 11 2021. [Online]. Available: https://nasheqlbrm.github.io/blog/posts/2021-11-13-backward-pass.html

[4] D. Cornelisse, "How to build a three-layer neural network from scratch," 2 2018. [Online]. Available: https://www.freecodecamp.org/news/building-a-3-layer-neural-network-from-scratch-99239c4af5d3/

[5] J. Brownlee, "How to Code a Neural Network with Backpropagation In Python (from scratch)," 10 2021. [Online]. Available: https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

[6] A. Gołda, "Backpropagation," 2005. [Online]. Available: https://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

[7] M. A. Nielsen, "Neural networks and deep learning," 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap1.html

[8] 3Blue1Brown, "But what is a neural network? — Chapter 1, Deep learning," 10 2017. [Online]. Available: https://www.youtube.com/watch?v=aircAruvnKk

[9] C. Olah, "Neural Networks, Manifolds, and Topology – colah's blog," 4 2014. [Online]. Available: http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

[10] J. Howard, "Practical deep learning for Coders - Practical Deep learning," 2019. [Online]. Available: https://course.fast.ai/

[11] K. Sinha, "Kaushik Sinha - Simple neural net backward pass," 11 2021. [Online]. Available: https://nasheqlbrm.github.io/blog/posts/2021-11-13-backward-pass.html