UNIVERZITET U BEOGRADU

# Sistem za paralelizaciju neuronskih mreža na OpenCL akceleratorima

student:

Petar Jovanović

mentor:

dr Filip Marić

Master rad

Matematički fakultet
Katedra za računarstvo i informatiku

14. maj 2018

# *Sažetak*

Kao jedan od bitnih faktora za ponovnu popularnost neuronskih mreža (NN) u oblasti mašinskog učenja, navodi se dostupnost daleko moćnijih računara u odnosu na one koji su bili prisutni 1980-ih kada su se NN poslednji put našle u žiži interesovanja. Nova generacija neuronskih mreža, poznata pod imenom "duboke neuronske mreže" (DNN), zbog većeg broja skrivenih slojeva ima računarske zahteve koji su toliko veliki da se eksperimentisanje sa njima može značajno ubrzati izvršavanjem na grafičkim karticama. Zbog bolje dokumentacije i razvijenijeg alata, Nvidia CUDA je trenutno dominantna platforma u ovom polju. Ostale platforme koje se baziraju na OpenCL tehnologiji trenutno nisu jednako upotrebljive i pored vrlo konkurentnog hardvera.

Cilj ovog rada je razvoj alata za konstrukciju i paralelizaciju DNN zasnovanih na OpenCL tehnologiji, koji su implementirani po uzoru na Theano biblioteku, LISA grupe sa Univerziteta u Montrealu. Predloženi sistem podržava konstrukciju simboličke reprezentacije izraza sa n-dimenzionalnim tenzorima, uz automatsko računanje gradijenta funkcija, što može značajno olakšati implementacije DNN. Takođe u okviru rada su implementirane neke popularne arhitekture neuronskih mreža i njihove performanse su analizirane nad referentnim skupovima podataka (MNIST, CIFAR).

# Sadržaj

**Sažetak** **i**

**1 Uvod** **1**
1.1 Programiranje grafičkih kartica . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Postojeće biblioteke za razvoj neuronskih mreža . . . . . . . . . . . . . . . . 4

**2 Automatsko diferenciranje** **5**
2.1 Numerički izvod . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Simboličko diferenciranje . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Automatsko diferenciranje . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3.1 Automatsko diferenciranje unapred . . . . . . . . . . . . . . . . . . 9
2.3.2 Automatsko diferenciranje unazad . . . . . . . . . . . . . . . . . . 12

**3 Neuronske mreže** **15**
3.1 Perceptron . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2 Višeslojni perceptron . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.3 Treniranje neuronskih mreža propagacijom grešaka unazad . . . . . . . . . 19
3.4 Konvolutivne neuronske mreže . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**4 Implementacija bibilioteke za razvoj neuronskih mreža na OpenCL-u** **24**
4.1 Upravaljanje OpenCL platformom . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Apstraktna sintaksa i automatsko diferenciranje . . . . . . . . . . . . . . . . 26
4.3 Višedimenzioni nizovi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.4 Konvolucija kao množenje matrica . . . . . . . . . . . . . . . . . . . . . . . . 29
4.5 Propagacija greške unazad u konvolutivnoj mreži . . . . . . . . . . . . . . . 31
4.6 Primer upotrebe za definisanje višeslojnog perceptrona . . . . . . . . . . . 31

**5 Eksperimentalni rezultati** **35**
5.1 MNIST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.2 CIFAR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.3 Diskusija . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**6 Zaključak** **40**

**Bibliografija** **41**

# Glava 1

# Uvod

Neki od problema čije se rešavanje smatralo prekretnicom za veštačku inteligenciju, kao što je prepoznavanje objekata na slikama ili pobeđivanje ljudskih protivnika u igri Go, rešeni [1, 2] su u poslednjoj deceniji korišćenjem novih tehnika mašinskog učenja. Mašinsko učenje je pristup rešavanju problema u kome se funkcija rešenja ne programira direktno, nego se pravi model koji uz podešavanje parametara može da aproksimira funkciju rešenja, tj. da za date ulaze vrati izlaze koji su približno jednaki onima koje bi prava funkcija vratila. Prednost tih modela je to što se podešavanje njihovih parametara može automatizovati. U terminologiji mašinskog učenja ta procedura podešavanja parametara modela se naziva učenje ili trening.

Postoji više vrsta modela u mašinskom učenju [3, 4] kao što su linearna regresija, Bajesovo učenje, grafovski modeli, stabla odlučivanja, SVM itd. Model koji je u osnovi novih pomaka na pomenutim problemima je neuronska mreža. Više detalja o neuronskim mrežama je dato u poglavlju 3, ali ukratko, one se sastoje od više jedinica koje imaju određeni broj ulaza, kojima je pridružen vektor težina, sa kojim se računa vektorski proizvod, a onda se taj proizvod nekom nelinearnom funkcijom transformiše u izlaz. Takve jedinice su grubo uporedive sa biološkim neuronima, odakle dobijaju i ime. Neuroni su obično organizovani u slojeve, gde se izlazi jednog sloja dovode na ulaze sledećeg i signal se tako obrađuje kroz mrežu.

Radovi grupa istraživača sa univerziteta u Torontu [1, 5, 6], Montrealu [7] i Njujorku [8], podstakli su ponovno intenziviranje istraživanja neuronskih mreža u oblasti mašinskog učenja. Pristup koji je tada razvijen nazvan je duboko učenje [9] (eng. *deep learning*) zbog prodora u treniranju neuronskih mreža velike dubine, tj. sa velikim brojem slojeva.

Do uvođenja tehnika dubokog učenja, neuronske mreže su imale tri vrste problema koji su ih sprečavali da postignu bolje performanse. Prvi su bili algoritamski. Postojeće verzije algoritma za trening su imale nedostataka koji su ih sputavali u treniranju mreža veće dubine, te su u praksi bile ograničene na samo dva sloja. Drugi problem je bio nedostatak dovoljno velikih skupova podataka za trening, koji su se vremenom pojavili usled sve veće zastupljenosti interneta. Treća vrsta problema su bili računarski zahtevi koji su do nedavno bili preveliki za postojeći hardver. Međutim, sa izlaskom Nvidia CUDA platforme [10], grafičke kartice postaju dostupne za programiranje generalne namene. Zahtevna operacija, veoma bitna za neuronske mreže, je množenje matrica, a grafičke kartice mogu da je izvode brzinama reda veličine teraflopsa[1], koje su pre bile dostupne samo na superračunarima. Na CUDA tehnologiji razvijeno je više biblioteka koje olakšavaju efikasne implementacije neuronskih mreža. U vreme pisanja ovog rada, druga najpopularnija platforma za programiranje grafičkih kartica, OpenCL [11], ima daleko oskudniju podršku među postojećim bibliotekama za neuronske mreže.

U ovom radu razmatra se implementacija biblioteke za brzo treniranje velikih neuronskih mreža, korišćenjem grafičkih kartica, bazirano na OpenCL tehnologiji. Automatsko diferenciranje je uvedeno u poglavlju 2 jer je najvažniji algoritam za treniranje (i razumevanje) neuronskih mreža specijalan slučaj automatskog diferenciranja unazad, i mislimo da ga je tako lakše razumeti. Poglavlje 3 uvodi neuronske mreze i sve pojmove relevantne za mreže korišćene u radu za testiranje razvijene biblioteke, kao što su konvolutivne neuronske mreže. Konvolutivne mreže su uključene u testove jer su one ključni deo mnogih arhitektura mreža koje se primenjuju na klasifikaciju slika, što je jedan od standardnih problema za poređenje pristupa. U poglavlju 4 opisana je implementacija biblioteke za razvoj neuronskih mreža na OpenCL platformi, koja je predmet ovog rada. Dat je i kraći primer upotrebe biblioteke. Poglavlje 5 predstavlja eksperimente koji su izvedeni sa razvijenom bibliotekom i daje izmerene rezultate.

U ostatku ovog poglavlja dat je pregled osnovnih pojmova u vezi sa programiranjem grafičkih kartica, kao i kratak osvrt na postojeći alat za razvoj neuronskih mreža na grafičkim karticama, i na OpenCL platformu.

## 1.1 Programiranje grafičkih kartica

Početna namena grafičkih kartica u računarima bila je ubrzavanje prikazivanja scena u računarskim igrama i ubrzavanje obrade slika i videa. Sa pojavom Nvidia CUDA [10] programskog interfejsa grafičke kartice su postale nova platforma za ubrzavanje programa koji su računski intenzivni. Prema fon Nojmanovoj arhitekturi [12] računara, glavne

[1] 1 TFLOPS = $10^{12}$ operacija u pokretnom zarezu po sekundi.

komponente centralne procesorske jedinice su aritmetičko-logička jedinica, koja izvršava računske operacije, i kontrolna jedinica koja upravlja tokom izvršavanja operacija i pomeranjem podataka unutar procesora.

Grafičke procesorske jedinice (GPU) predstavljaju procesore sa većim brojem nezavisnih jezgara gde svako jezgro ima jednu kontrolnu jedinicu i više aritmetičko-logičkih jedinica, kao i malu količinu brze lokalne memorije [13]. Osim procesorske jedinice na grafičkoj kartici se obično nalazi i posebna radna memorija. Takva arhitektura GPU čini pogodnim za tip paralelnog programiranja koji se prema Flinovoj podeli [14] naziva SIMD (iste instrukcije – višestruki podaci, eng. *single instruction multiple data*). Za programe koji sadrže izračunavanja koja se mogu isprogramirati na SIMD način, savremeni grafički procesori omogućavaju pristup računskim performansama koje se mere u teraflopsima.

Pristup paralelizovanju programa se sastoji u tome da se delovi koda koji se bave iterativnom obradom skupova podataka, predstave kao funkcije koje se nezavisno izvršavaju nad pojedinačnim elementima skupa podataka. U GPU programiranju, takve funkcije se nazivaju *kerneli*, a njihovo izvršavanje se može odvijati paralelno na svim jezgrima grafičke kartice. U vreme pisanja ovog rada, dominante platforme za programiranje grafičkih kartica su CUDA i OpenCL. Na obe platforme, programski kod se deli na deo koji se izvršava na procesoru (sa pristupom sistemskoj memoriji) računara na kome se nalazi grafička kartica, i kod koji se izvršava na samoj grafičkoj kartici (sa pristupom radnoj memoriji kartice), koji se naziva kernel funkcija.

Na CUDA platformi, kod se piše u C/C++ jeziku [15] sa dodatnom sintaksom[2] za pozive kernel funkcija. Osim sintakse, koriste se i funkcije i strukture uključene CUDA Runtime biblioteke, od kojih su neke dostupne samo u kodu kernela (na primer struktura za određivanje koordinata niti izvršavanja u nizu svih niti koje su pokrenute). CUDA programi se prevode CUDA kompilatorom (nvcc) koji radi kao standardni kompilatori, prevodeći izvorni kod u izvršive binarne fajlove.

Za razliku od CUDA platforme, OpenCL [11] ne uvodi sintaksna proširenja u C/C++ jezik, kojim se takođe programira. U OpenCL-u glavni program, koji inicijalizuje računanje na kartici i organizuje memorijske transfere između sistemske i memorije grafičke kartice, programira se u standardnom C/C++ jeziku i ne koristi se specifičan kompilator za prevođenje. Međutim, kod kernel funkcija u OpenCL-u se piše u C jeziku, sa dodatim makro definicijama i funkcijama za obeležavanje memorije kojoj neki pokazivač pripada, određivanje koordinata tekuće niti izvršavanja, postavljanje barijera i slično. U okviru OpenCL biblioteke koja se uključuje u glavni program, nalaze se funkcije za prevođenje

[2]Koristi se niz simbola $<<<$ ... $>>>$ uz ime kernel funkcije, pre liste parametara, na primer: *kernel* $<<< 32, 32 >>>$ (*parametri*). Brojevi dati izmedju znakova $<<<$ i $>>>$ predstavljaju dimenzije blokova niti izvršavanja koji će biti pokrenuti.

izvornog koda kernela, koje se pozivaju u toku rada glavnog programa. Izvorni kod kernela se u glavnom programu čuva kao string koji se može uneti direktno u kod, koristeći standardnu sintaksu za niske karaktera, ili se može učitati iz fajla, ili generisati u toku rada glavnog programa. Mogućnost generisanja koda kernela u toku izvršavanja čini OpenCL pogodnim za metaprogramiranje, tj. pisanje programa koji generišu programe.

## 1.2 Postojeće biblioteke za razvoj neuronskih mreža

Zajedno sa rastućom popularnošću neuronskih mreža, porastao je broj biblioteka za rad sa njima. Kako neuronske mreže mogu da se predstave kao operacije nad višedimenzionim nizovima, uklapaju se u model pogodan za izvršavanje na grafičkim karticama, te mnoge popularne biblioteke podržavaju ubrzavanje na njima, što značajno ubrzava trening i izvršavanje modela.

Neke od popularnih biblioteka koje su poslužile i kao inspiracija za razvoj sistema opisanog u ovom radu su: Theano [16], TensorFlow [17], Chainer [18], Torch [19], Keras [20], Caffe [21]. Ove biblioteke nude različite nivoe apstrakcije u konstrukciji neuronskih mreža. Keras i Caffe imaju unapred definisane popularne arhitekture neuronskih mreža i korisniku ostavljaju samo konfigurisanje hiperparametara[3] kao sto su dimenzije slojeva, brzina učenja i slično. Theano, TensorFlow, Torch i Chainer nude niži nivo apstrakcije koji omogućava programeru da sam konstruiše neuronske mreže bez ograničavanja na predefinisane arhitekture. Na tom nivou mreža se definiše kao simbolički matematički izraz sa višedimenzionim nizovima koji predstavljaju težine mreže. Taj izraz se kasnije prevodi u efikasne potprograme koji se izvršavaju na grafičkoj kartici. Korisnik biblioteke je time oslobođen potrebe da vodi računa o pomeranju podataka između sistemske i grafičke memorije, podešavanju pokretanja kernela, sinhronizaciji operacija i sličnom. Međutim, najveće olakšanje u razvoju neuronskih mreža dolazi od toga što ove biblioteke mogu automatski da izračunaju parcijalne izvode simboličkog izraza koji predstavlja mrežu, po svim parametrima mreže.

Cilj biblioteke opisane u ovom radu je da podrži takav pristup razvoju neuronskih mreža na OpenCL grafičkim karticama, među kojima mnoge ne podržavaju CUDA platformu i pored vrlo sposobnog hardvera nisu lako upotrebljive u tom polju.

[3] U novijim verzijama Keras podržava i niži nivo apstrakcije koji dozvoljava definisanje mreža pomoću slojeva.

# Glava 2

# Automatsko diferenciranje

U ovom poglavlju uvode se tehnike automatskog diferenciranja jer je pomoću njih lakše uvesti algoritam treniranja neuronskih mreža. Takođe, ove tehnike daju mogućnost automatizacije računanja gradijenta, što smanjuje prostor za greške pri implementaciji neuronskih mreža.

Izvod funkcije predstavlja brzinu promene izračunate vrednosti funkcije, tj. broj koji govori koliko se izračunata vrednost funkcije menja pri promeni ulaznog parametra. U geometrijskom smislu, izvod predstavlja koeficijent pravca tangente na grafik funkcije u tački u kojoj se računa. Ako funkcija ima više ulaznih parametara, izvod funkcije po jednom parametru se naziva parcijalni izvod i takođe predstavlja brzinu promene vrednosti funkcije u odnosu na promenu vrednosti tog parametra.

Gradijent skalarne funkcije više promenljivih (oblika $f : \mathbb{R}^n \to \mathbb{R}$), označen sa $\nabla$ u 2.1, je vektorsko polje čije su komponente parcijalni izvodi date funkcije po svakoj promenljivoj. On ima pravac najvećeg rasta vrednosti funkcije i intenzitet promene promene duž tog pravca. Gradijentne metode optimizacije, kao što je gradijentni spust, na osnovu njega menjaju parametre funkcije $f$ tako da njena izračunata vrednost bude minimalna ili maksimalna u zavisnosti od cilja optimizacije.

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right) \tag{2.1}$$

Postoji nekoliko načina da se automatizuje računanje izvoda, a neki od njih su: numerička aproksimacija, simboličko prepisivanje i automatsko računanje izvoda. U ostatku ovog odeljka uvodi se automatsko diferenciranje, uz prethodan osvrt na ostale metode.

## 2.1 Numerički izvod

Numerička aproksimacija je jednostavna za implementaciju i bazira se na aproksimaciji izvoda, datoj u jednačini (2.2). Promenljiva $\epsilon$ predstavlja malu vrednost koja se dodaje tački u kojoj računamo izvod.

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{2.2}$$

Zbog male vrednosti u $\epsilon$, greške u reprezentaciji brojeva na računaru se brzo akumuliraju ponavljanim računanjem tokom koraka optimizacije, što čini ovaj pristup manje upotrebljivim. Postoje načini da se te greške smanje, ali oni su van opsega ovog rada. Kako neuronske mreže imaju veliki broj parametara, ovaj način računanja izvoda nije efikasan jer zahteva mnogo prolaza unapred kroz neuronsku mrežu da bi se izračunala vrednost $f(x)$ za parcijalni izvod po svakom parametru.

## 2.2 Simboličko diferenciranje

Simboličko diferenciranje se odvija tako što se na podizraze formule koja se diferencira, rekurzivno primenjuju pravila prepisivanja po formulama za računanje izvoda za dati podizraz. To je i način na koji ljudi računaju izvode funkcija na papiru. Originalni izraz funkcije se prepisivanjem transformiše u izraz čija vrednost predstavlja izvod te funkcije. Ovakav pristup je iskorišćen u *Theano* [16] biblioteci koja je jedna od inspiracija za biblioteku razvijenu u ovom radu.

Ulaz u proceduru za simboličko računanje je funkcija, a izlaz je takođe funkcija, ali računanje njene vrednosti nekoj tački daje vrednost izvoda ulazne funkcije u datoj tački.

Jedan od problema koji se javljaju u simboličkom pristupu je da pravila za prepisivanje po definiciji izvoda, za dati izraz mogu da vrate izraz veće složenosti, što narušava čitljivost rezultujućeg izraza i povećava računske zahteve [22]. Da bi se to izbeglo, potrebno je uz primenu pravila za izvode, primenjivati i pravila pojednostavljenja izraza na dobijene medjurezultate. Takođe moguće je poslužiti se načinom na koji automatsko računanje izvoda, opisano u sledećem odeljku, razlaže proces računanja i predstaviti svaki korak simbolički.

## 2.3 Automatsko diferenciranje

Automatsko diferenciranje je vrlo blisko simboličkom, ali ono ne sadrži konstrukciju simboličkog izraza za izvod, već samo direktno računanje njegove vrednosti u datoj tački (valuaciji). Za primene u treniranju neuronskih mreža, to je dovoljno, iako simbolička reprezentacija izvoda može nekad pružiti koristan uvid, ako je dovoljno čitljiva. Ovaj pristup predstavlja primenu pravila izvoda za složene funkcije (eng. *chain rule*), datom u (2.3). Ako $f(g(x))$ i $g(x)$ iz (2.3) zamenimo redom sa $y$ i $w$, isto pravilo možemo napisati u Lajbnicovoj notaciji kao u (2.4).

$$f(g(x))' = f'(g(x))g'(x) \tag{2.3}$$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x} \tag{2.4}$$

Ako je $y$ složena funkcija kao u (2.5), pravilo važi za nju, pa se izvod može razviti kao u (2.6).

$$y = u_1 \circ u_2 \circ u_3 \circ \cdots \circ u_n(x) \tag{2.5}$$

$$\frac{\partial y}{\partial x} = \frac{\partial u_1}{\partial u_2} \cdot \frac{\partial u_2}{\partial u_3} \cdots \frac{\partial u_n}{\partial x} \tag{2.6}$$

Računanje izvoda po pravilu za složene funkcije može se odvijati zdesna nalevo u jednačini (2.6). Prvo se računa izvod najužeg podizraza koji sadrži promenljivu po kojoj se izvod radi. Zatim se računa izvod izraza koji sadrži obrađeni podizraz, u tački koja ima vrednost prethodno izračunatog izvoda. Ovaj postupak se nastavlja prema redosledu definisanom zagradama, a na kraju izračunata vrednost predstavlja vrednost parcijalnog izvoda izraza $y$ po promenljivoj $x$.

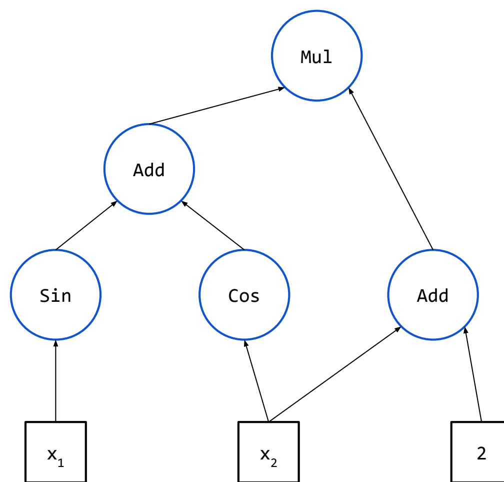
$$\begin{aligned}
\frac{\partial y}{\partial x} &= \frac{\partial u_1}{\partial u_2}\left(\frac{\partial u_2}{\partial x}\right) \\
&= \frac{\partial u_1}{\partial u_2}\left(\frac{\partial u_2}{\partial u_3}\left(\frac{\partial u_3}{\partial x}\right)\right) \\
&\cdots \\
&= \frac{\partial u_1}{\partial u_2}\left(\frac{\partial u_2}{\partial u_3}\left(\cdots\left(\frac{\partial u_n}{\partial x}\right)\right)\right)
\end{aligned}$$

Takođe, računanje se može odvijati sleva nadesno u formuli (2.6). Tada se u jednom prolazu dobijaju parcijalni izvodi po svim parametrima funkcije. Ovaj proces se odvija tako što se podizraz šireg izraza fiksira kao promenljiva po kojoj se radi parcijalni izvod šireg izraza.

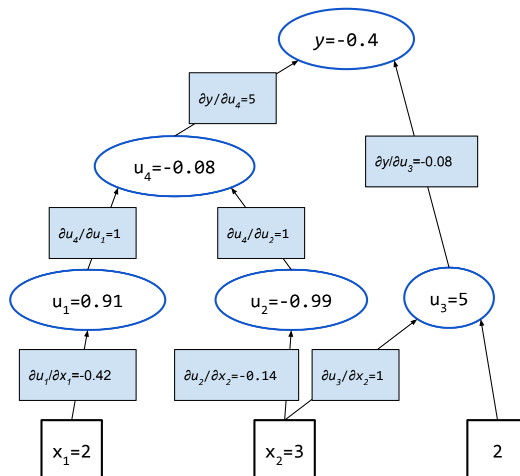
$$\frac{\partial y}{\partial x} = \left(\left(\left(\frac{\partial u_1}{\partial u_2}\right)\frac{\partial u_2}{\partial u_3}\right)\cdots\right)\frac{\partial u_n}{\partial x} \tag{2.7}$$

Ova dva načina računanja predstavljaju osnovu za dva tipa automatskog diferenciranja – računanje unapred (eng. *forward mode differentiation*) i računanje unazad (eng. *reverse mode differentiation*).

U oba slučaja, polazi se od predstavljanja date funkcije kao usmerenog acikličnog grafa. Na primer, ako imamo funkciju $f(x_1, x_2) = (sin(x_1) + cos(x_2)) \cdot (x_2 + 2)$ njen graf može izgledati kao slika 2.1a. Kada se računa vrednost izraza, polazi se od atoma (kvadrati u grafu) i ide se niz usmerene veze uz primenjivanje operacija u čvorovima, dok se ne stigne do čvora koji nema izlaznih veza, a vrednost koja se u njemu izračuna je i vrednost celog izraza.




(A) Izraz predstavljen kao usmereni graf




(B) Graf izračunavanja vrednosti izraza i izvoda

SLIKA 2.1: Graf izračunavanja vrednosti izraza i vrednosti izvoda funkcije $f(x_1, x_2) = (sin(x_1) + cos(x_2)) \cdot (x_2 + 2)$.

Svaki operator izraza može se proširiti da uz računanje vrednosti primene njegove operacije na ulaze, računa i vrednost izvoda u tački definisanoj datim ulazima. Dodata funkcija se razlikuje u zavisnosti od toga da li se izvod računa unapred ili unazad, što je detaljnije objašnjeno u naredna dva odeljka.

### 2.3.1 Automatsko diferenciranje unapred

Radi jednostavnije notacije, podizrazi u primeru sa slike 2.1a su zamenjeni sledećim promenljivima:

$$u_1 = sin(x_1)$$
$$u_2 = cos(x_2)$$
$$u_3 = x_2 + 2$$
$$u_4 = u_1 + u_2$$
$$y = u_4 u_3$$

Kada se zadaju vrednosti $x_1 = 2$ i $x_2 = 3$ i izračunaju vrednosti $u_i$ i ubace u graf umesto operatora, dobijamo stanje grafa na slici 2.1b. Takođe, prema pravilima za računanje izvoda za operatore $Sin$, $Cos$, $Add$, $Mul$, i izračunatim vrednostima u grafu, računaju se parcijalni izvodi operatora u odnosu na operande i smeštaju na odgovarajuće grane grafa (između operatora i operanda). Svaki izvod se računa istovremeno sa izračunavanjem vrednosti koju operator vraća.

Da bi se dobio parcijalni izvod cele funkcije po nekoj promenljivoj, potrebno je proći sve putanje u grafu od lista promenljive do korena[1] izraza, pomnožiti izvode koji se nalaze na njima i dobijene proizvode sabrati. Za tekući primer, postupak za nalaženje izvoda po $x_1$ i $x_2$ je dat jednačinama 2.8 i 2.9, gde se vidi da je postupak ekvivalentan pravilu traženja izvoda složenih funkcija. Postupak za nalaženje izvoda po parametrima $x_1$ i $x_2$ je predstavljen i u tabeli 2.1. U prvoj koloni tabele se nalaze vrednosti promenljivih, podizraza i celog izraza u poslednjem redu. Srednja kolona prikazuje računanje vrednosti parcijalnog izvoda po promenljivoj $x_1$, a desna kolona po $x_2$. U redovima tabele su predstavljeni koraci računanja, od samih promenljivih, podizraza koji ih sadrže do izraza cele funkcije na dnu tabele. Strelice uz levu ivicu kolona prikazuju smer računanja u svakoj koloni.

[1] Iako graf izraza striktno gledano, ne predstavlja drvo, zbog toga što čvor može imati više od jednog roditelja, pod korenom se podrazumeva čvor sa izlaznim stepenom 0 koji predstavlja poslednji operator koji je primenjen u konstrukciji izraza, a listovi su čvorovi ulaznog stepena 0 koji predstavljaju promenljive.

| Računanje vrednosti | Računanje izvoda po $x_1$ | Računanje izvoda po $x_2$ |
|---|---|---|
| $x_1 = 2$ | $\frac{\partial x_1}{\partial x_1} = 1$ | $\frac{\partial x_1}{\partial x_2} = 0$ |
| $x_2 = 3$ | $\frac{\partial x_2}{\partial x_1} = 0$ | $\frac{\partial x_2}{\partial x_2} = 1$ |
| $u_1 = sin(x_1) = 0.91$ | $\frac{\partial u_1}{\partial x_1} = \frac{\partial sin(x_1)}{\partial x_1} \cdot \frac{\partial x_1}{\partial x_1} = cos(x) \cdot 1 = -0.42$ | $\frac{\partial u_1}{\partial x_2} = \frac{\partial sin(x_1)}{\partial x_1} \cdot \frac{\partial x_1}{\partial x_2} = sin(x_1) \cdot 0 = 0$ |
| $u_2 = cos(x_2) = -0.99$ | $\frac{\partial u_2}{\partial x_1} = \frac{\partial cos(x_2)}{\partial x_2} \cdot \frac{\partial x_2}{\partial x1} = -sin(x_2) \cdot 0 = 0$ | $\frac{\partial u_2}{\partial x_2} = \frac{\partial cos(x_2)}{\partial x_2} \cdot \frac{\partial x_2}{\partial x_2} = -sin(x_2) \cdot 1 = -0.14$ |
| $u_3 = x_2 + 2 = 5$ | $\frac{\partial u_3}{\partial x_1} = \frac{\partial x_2}{\partial x_1} + 0 = 0$ | $\frac{\partial u_3}{\partial x_2} = \frac{\partial x_2}{\partial x_2} + 0 = 1$ |
| $u_4 = u_1 + u_2 = 0.91 - 0.99 = -0.08$ | $\frac{\partial u_4}{\partial x_1} = \frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_1} = -0.42 + 0 = -0.42$ | $\frac{\partial u_4}{\partial x_2} = \frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_2} = 0 + (-0.14) = -0.14$ |
| $y = u_4 u_3 = -0.08 \cdot 5 = \mathbf{-0.4}$ | $\frac{\partial y}{\partial x_1} = u_4 \frac{\partial u_3}{\partial x_1} + u_3 \frac{\partial u_4}{\partial x_1} = (-0.08) \cdot 0 + 5 \cdot (-0.42) = \mathbf{-2.1}$ | $\frac{\partial y}{\partial x_2} = u_4 \frac{\partial u_3}{\partial x_2} + u_3 \frac{\partial u_4}{\partial x_2} = (-0.08) \cdot 1 + 5 \cdot (-0.14) = \mathbf{-0.78}$ |

TABELA 2.1: Automatsko diferenciranje unapred za funkciju $f(x_1, x_2) = (sin(x_1) + cos(x_2)) \cdot (x_2 + 2)$, po parametrima $x_1$ i $x_2$.

$$\begin{aligned} \frac{\partial y}{\partial x_1} &= \frac{\partial u_1}{\partial x_1} \frac{\partial u_4}{\partial u_1} \frac{\partial y}{\partial u_4} \\ &= -0.42 \times 1 \times 5 \\ &= -2.1 \end{aligned} \tag{2.8}$$

$$\begin{aligned} \frac{\partial y}{\partial x_2} &= \frac{\partial u_2}{\partial x_2} \frac{\partial u_4}{\partial u_2} \frac{\partial y}{\partial u_4} + \frac{\partial u_3}{\partial x_2} \frac{\partial y}{\partial u_3} \\ &= -0.14 \times 1 \times 5 + 1 \times (-0.08) \\ &= -0.78 \end{aligned} \tag{2.9}$$

U algoritmima 1, 2 i 3 data je rekurzivna verzija računanja izvoda unapred u kojoj se računanje vrednosti i parcijalnog izvoda funkcije dešavaju u zasebnim prolazima kroz

graf. Efikasniji pristup sa samo jednim prolazom kroz graf se može izvesti prevođenjem algoritma u iterativnu proceduru, ali ona je izostavljena zbog konciznosti opisa. Algoritam počinje inicijalizacijom strukture koja će držati međurezultate računanja, zatim je popunjava izvršavajući računanje vrednosti u *evaluate* funkciji.

**Algorithm 1** Automatsko diferenciranje unapred

**Input:** $FG$ – graf funkcije u kome svaki čvor ima jedinstveni i dentifikator $id$ i metode za računanje vrednosti i parcijalnog izvoda operacije koju implementira,
$wrt$ – identifikator promenljive po kojoj se traži parcijalni izvod,
$valuation$ – valuacija, tj. vrednosti svih parametara u tački u kojoj se traži izvod.
**Output:** $y$ – vrednost izračunavanja funkcije date grafom $FG$,
$dy$ – vrednost parcijalnog izvoda funkcije date grafom $FG$ po promenljivoj sa identifikatorom $wrt$.

1: $dacc$ se inicijalizuje kao mapa čiji su ključevi identifikatori čvorova grafa $FG$ i za svaki list grafa vrednost u mapi se postavlja na 0, osim za list sa identifikatorom $wrt$, čija se vrednost postavlja na 1.
2: $cache$ se inicijalizuje kao prazna mapa, takođe sa identifikatorima čvorova kao ključevima.
3: $y \leftarrow \text{evaluate}(FG, valuation, cache)$
4: $dy \leftarrow \text{fwd_diff}(FG, valuation, dacc, cache)$
5: **return** $dy$

**Algorithm 2** evaluate

**Input:** $node$ – čvor u kome se računa vrednost operacije,
$valuation$ – mapa vrednosti svih listova podgrafa čiji je koren $node$,
$cache$ – mapa izračunatih vrednosti čvorova.
**Output:** $y$ sadrži vrednost izraza u čvoru $node$,
$cache$ sadrži $y$.

1: **if** is_atom($node$) **then**
2: $y \leftarrow valuation[\text{id}(node)]$
3: **else**
4: **for all** $op \in \text{operands}(node)$ **do**
5: evaluate($op, valuation, cache$)
6: **end for**
7: $y \leftarrow \text{perform}(\text{operation}(node), cache)$
8: **end if**
9: $cache[\text{id}(node)] \leftarrow y$
10: **return** $y$

Opisani pristup radi efikasno za funkcije koje nemaju puno ulaznih parametara, pošto se u jednom prolazu kroz graf računa parcijalni izvod po jednom parametru[2]. Za one koje imaju veliki broj ulaza, pogodnija je metoda traženja izvoda unazad, opisana u sledećem odeljku.

[2] U tabeli 2.1 i algoritmu 1 može se videti da je prvi korak računanja izvoda unapred postavljanje vrednosti izvoda ulaznih parametara na 0 osim za parametar po kome se parcijalni izvod traži, kome se dodeljuje vrednost 1. To onemogućava istovremeno računanje izvoda po svim ulaznim parametrima.

**Algorithm 3** fwd_diff

**Input:** *node* – čvor u kome se računa parcijalni izvod,
*valuation* – mapa vrednosti svih listova podgrafa čiji je koren *nodes*,

**Output:** *dy* – vrednost parcijalnog izvoda po *wrt* u čvoru *node*.

1: **if** is_atom(*node*) **then**
2: $dy \leftarrow dacc[\text{id}(node)]$
3: **else**
4: *opd* se postavlja na praznu mapu, koja za ključeve ima identifikatore čvorova koji su operandi tekućeg čvora, a vrednosti su vrednosti njihovih parcijalnih izvoda po varijabli koja ima vrednost 1 u *dacc*.
5: **for all** $op \in \text{operands}(node)$ **do**
6: $opd[\text{id}(op)] \leftarrow$ fwd_diff(op, valuation, dacc, cache)
7: **end for**
8: $dy \leftarrow \text{perform}(\text{operation}'(node, opd, valuation, cache))$
9: **end if**
10: **return** *dy*

### 2.3.2 Automatsko diferenciranje unazad

Kod automatskog diferenciranja unazad, akumulacija vrednosti parcijalnih izvoda za svaku promenljivu ide od izraza cele funkcije, tj. korena u grafu, ka promenljivima, tj. listovima, ili ako se gleda pravilo izvoda složenih funkcija (2.3), računanje se vrši sleva nadesno. U ovom slučaju, računanje parcijalnih izvoda ne ide paralelno sa izračunavanjem vrednosti funkcije, jer se međurezultati računanja vrednosti podizraza koriste unazad.

Svakom čvoru grafa izraza se dodeljuje pridruženi element $\bar{u}_i = \dfrac{\partial f}{\partial u_i}$ (eng. *adjoint*) koji predstavlja osetljivost izlaza na promene u datom čvoru. Prva faza se odvija tako što se izračunava vrednost izraza unapred i sačuvaju se svi međurezultati za svaki čvor izraza. U drugoj fazi, se računaju svi parcijalni izvodi uz propagiranje pridruženog elementa od izlaznog čvora ka ulaznima. Računanje izvoda počinje tako što se pridruženom elementu uz izlaz funkcije dodeli vrednost 1 a svim promenljivima se pridruženi element postavi na 0. Idući unazad, u svakom čvoru, pridruženi element se množi izvodom po svakom ulazu čvora i taj proizvod se prosleđuje kao pridruženi element čvora od koga dolazi ulaz. To se rekurzivno ponavlja dok se ne dođe do lista, tj. promenljive u kojoj se svi pridruženi članovi koji dolaze po granama grafa sabiraju a taj zbir predstavlja vrednost parcijalnog izvoda funkcije po toj promenljivoj. Prikaz računanja baziran na primeru sa slike 2.1 je dat u tabeli 2.2. Može se primetiti da se dobijene vrednosti izvoda (podebljano u tabeli) slažu sa vrednostima izračunatim u jednačinama 2.8 i 2.9.

Prvi prolaz za izračunavanje vrednosti izraza je prikazan u prvoj koloni tabele. Počinje dodeljivanjem vrednosti promenljivih, a nastavlja se izračunavanjem podizraza dok se u poslednjem redu ne dobije i vrednost celog izraza $y$.

Druga kolona prikazuje korake računanja parcijalnih izvoda, za svaki podizraz počevši od celog izraza za $y$ na dnu tabele do vrednosti parcijalnih izvoda za svaku promenljivu u prva dva reda tabele. Smer računanja je prikazan i strelicama koje se nalaze uz levu ivicu kolone.

| Računanje vrednosti | Računanje izvoda |
|---|---|
| $x_1 = 2$ | $\bar{x_1} = \bar{u_1} \cdot \frac{\partial u_1}{\partial x_1} = 5 \cdot cos(2) = \mathbf{-2.1}$ |
| $x_2 = 3$ | $\bar{x_2} = \bar{u_2} \cdot \frac{\partial u_2}{\partial x_2} + \bar{u_3} \cdot \frac{\partial u_3}{\partial x_2} = 5 \cdot (-sin(3)) - 0.08 \cdot 1 = \mathbf{-0.78}$ |
| $u_1 = sin(x_1) = 0.91$ | $\bar{u_1} = \bar{u_4} \cdot \frac{\partial u_4}{\partial u_1} = \bar{u_4} \cdot 1 = 5$ |
| $u_2 = cos(x_2) = -0.99$ | $\bar{u_2} = \bar{u_4} \cdot \frac{\partial u_4}{\partial u_2} = \bar{u_4} \cdot 1 = 5$ |
| $u_3 = x_2 + 2 = 5$ | $\bar{u_3} = \bar{y} \cdot \frac{\partial y}{\partial u_3} = \bar{y} \cdot u_4 = -0.08$ |
| $u_4 = u_1 + u_2 = -0.08$ | $\bar{u_4} = \bar{y} \cdot \frac{\partial y}{\partial u_4} = \bar{y} \cdot u_3 = 5$ |
| $y = u_4 u_3 = \mathbf{-0.4}$ | $\bar{y} = 1$ |

Tabela 2.2: Automatsko diferenciranje unazad za funkciju $f(x_1, x_2) = (sin(x_1) + cos(x_2)) \cdot (x_2 + 2)$, po oba njena parametra. Redosled praćenja operacija je prvo u levoj koloni od gore na dole, a zatim u desnoj od dole na gore.

Algoritam 4 daje pseudokod za opisanu proceduru računanja izvoda unazad. Ovo je pomoćna funkcija koja inicijalizuje keš i rečnik za držanje izračunatih parcijalnih izvoda. Promenljiva *adjoint* predstavlja pridruženi element i njegova vrednost se inicijalizuje na 1.

**Algorithm 4** Automatsko diferenciranje unazad

**Input:** $FG$ – graf funkcije čiji se izvod traži.
$valuation$ – vrednosti svih parametara u tački u kojoj se traži izvod.

**Output:** $grad$ – mapa izvoda za svaki parametar funkcije.

1: $cache$ se inicijalizuje kao prazna mapa gde je ključ identifikator čvora u $FG$, a vrednost izračunata vrednost izraza u tom čvoru.
2: $grad$ se inicijalizuje kao prazna mapa
3: evaluate($FG, valuation, cache$)
4: $adjoint \leftarrow 1.0$
5: rev_grad($FG, valuation, adjoint, grad, cache$)
6: **return** $grad$

Posle izračunavanja vrednosti funkcije, koje popuni keš međurezultatima izračunavanja, poziva se rekurzivna funkcija data u algoritmu 5. Ona ide kroz graf izraza od korena (najšireg izraza) ka listovima (atomima u izrazu) računajući izvode podizraza koje čuva u rečniku *grad*.

**Algorithm 5** rev_grad

**Input:** $node$ – tekući čvor u kome se računa izvod,
$valuation$ – mapa vrednosti svih parametara u tački u kojoj se računa izvod,
$adjoint$ – pridruženi element,
$grad$ – mapa za akumulaciju vrednosti parcijalnih izvoda po svakom parametru,
$cache$ – mapa izračunatih vrednosti svakog čvora $FG$.

1: **if** is_atom($node$) **then**
2:    $grad$[id($node$)] $\leftarrow$ $grad$[id($node$)] $+ adjoint$
3: **else if** is_constant($node$) **then**
4:    **return**
5: **else**
6:    **for all** $op \in$ operands($node$) **do**
7:       $a \leftarrow \overline{operation_{op}}(valuation, adjoint, cache)$
8:       rev_grad($op, valuation, a, grad, cache$)
9:    **end for**
10: **end if**

U sledećem poglavlju, biće više reči o specijalnom slučaju automatskog diferenciranja unazad koje predstavlja ključan korak u treniranju neuronskih mreža. Kako neuronske mreže imaju mnogo parametara, ovaj algoritam je pogodan jer u jednom prolazu daje parcijalne izvode za sve parametre.

# Glava 3

# Neuronske mreže

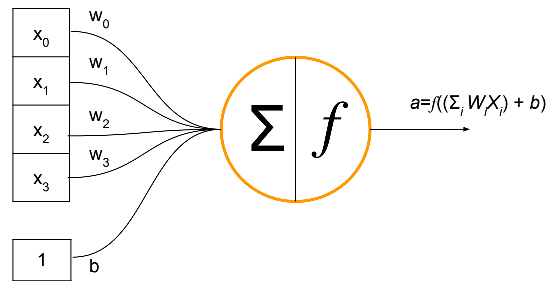
Neuronske mreže predstavljaju model računanja, inspirisan radom biloških neurona u mozgu, koji paralelno obrađuje podatke i koji se ne programira eksplicitno, nego se trenira nad primerima ulaznih i izlaznih podataka koje bi trebalo da generiše. Neuronske mreže sadrže veći broj jedinica koje odgovaraju pojedinačnim neuronima, i obično su organizovane u slojeve, koji predstavljaju faze kroz koje se ulazni podaci transformišu do izlaznih. Transformacija podataka se može odvijati u jednom prolazu kroz slojeve (eng. *feedforward*) ili mogu postojati ciklusi (eng. *recurrent*). Sve mreže korišćene u testovima u ovom radu su feedforward tipa.

Funkcija mreže može biti generativna, u smislu da modeluje zajedničku raspodelu atributa i ciljnih promenljivih, ili diskriminativna ako modeluje uslovnu raspodelu ciljnih promenljivih za date vrednosti ulaznih atributa. Takođe se može gledati i kao prediktivna funkcija koja na osnovu ulaza predviđa izlaze.

Način organizacije slojeva predstavlja arhitekturu mreže, a neke od popularnijih arhitektura su višeslojni perceptron (MLP), konvolutivne mreže, rekurentne mreže, autoenkoderi, itd. Detaljniji pregled mnogih od njih je dat u knjizi Goodfellow et al. [23].

U ovom odeljku su opisani perceptron, višeslojni perceptron, kao osnovni modeli za uvođenje neuronskih mreža, i konvolutivna mreža, jer predstavlja jedan od najpopularnijih modela, pogotovo u klasifikaciji[1] slika, što će biti korišćeno kao test za biblioteku.

[1] Klasifikacija u mašinskom učenju predstavlja razvrstavanje ulaznih podataka prema zadatim kategorijama, tj. dodeljivanje odgovarajuće klase ulaznim podacima.




Slika 3.1: Perceptron

## 3.1 Perceptron

Perceptron je matematički model baziran na pojednostavljenim pretpostavkama o radu pojedinačnog biološkog neurona. Definisan je u radu Frenka Rozenblata [24], i predstavlja osnovni gradivni element neuronskih mreža. Na slici 3.1 dat je šematski prikaz transformacije podataka koju perceptron izvršava. Perceptron se sastoji od parametara koji se nazivaju težine, parametra koji se naziva prag (eng. *bias*) ili slobodni koeficijent, i funkcije aktivacije. Preciznije, u slučaju perceptrona koristi se step funkcija[2] za funkciju aktivacije. U jednačini 3.1 izlaz perceptrona označen kao promenljiva $a$ je definisan kao rezultat primene funkcije aktivacije $\sigma$ na zbir vektorskog proizvoda vektora težina $W$ i ulaznog niza $X$ i praga $b$.

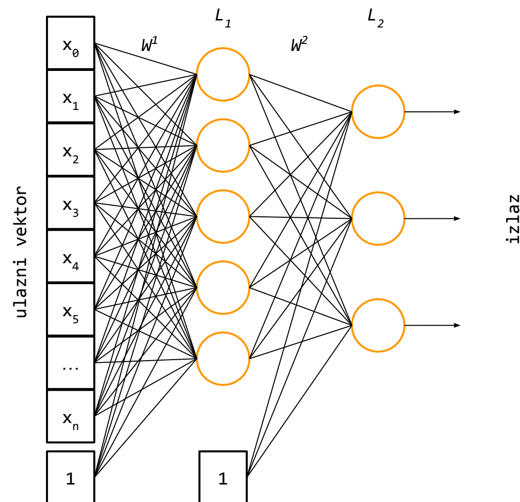
$$a = \sigma\left(\sum_{i=0}^{n} X_i W_i + b\right) \tag{3.1}$$

Parametri perceptrona određuju ravan u hiperprostoru dimenzija njegovog ulaza, zbog toga ovaj model može da klasifikuje samo instance ulaza koje pripadaju linearno separabilnim klasama. U zavisnosti od kodomena funkcije $\sigma$ rezultat se može interpretirati kao stepen pripadnosti ulazog niza jednoj klasi. Jedan perceptron može samo da klasifikuje ulaze u dve klase, a za više klasa se može koristiti više perceptrona gde svaki daje stepen pripadnosti jednoj od klasa.

## 3.2 Višeslojni perceptron

Kod višeslojnog perceptrona (eng. *Multilayer Perceptron, MLP*) više pojedinačnih neurona se grupišu u slojeve a izlazi jednog sloja se koriste kao ulazi za sledeći sloj. Zbog ovakve organizacije, ovaj tip neuronskih mreža se još naziva i *feed forward* mreža pošto signal u njoj propagira sa jednog sloja na sledeći, bez povratnih sprega i ciklusa.

[2] $step(x) = x >= b$, gde je $b$ vrednost praga.




SLIKA 3.2: Višeslojni perceptron sa jednim skrivenim i jednim izlaznim slojem.

Ovaj model je dovoljno opšt da može da aproksimira bilo koju funkciju ako se dozvoli neograničen broj neurona u skrivenom sloju i odabere odgovarajuća funkcija aktivacije [25].

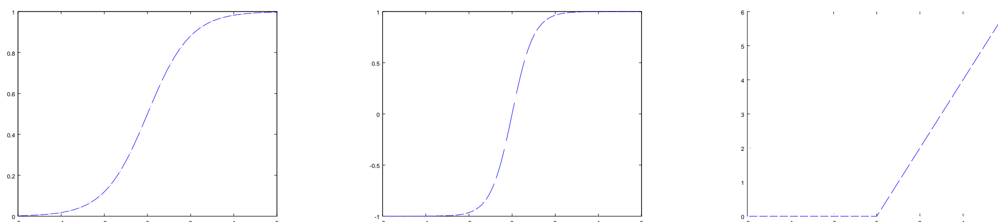
Na slici 3.2 prikazan je višeslojni perceptron sa jednim skrivenim slojem $L_1$ i izlaznim slojem $L_2$. Skriveni slojevi se nazivaju skrivenim jer se vrednosti njihovih aktivacija ne vide direktno u izlazu i funkcionalnost pojedinačnih neurona u njima nije unapred određena. Pod aktivacijom podrazumevamo izračunatu vrednost funkcije aktivacije koja predstavlja izlaz neurona, a aktivaciju sloja treba shvatiti kao vektor aktivacija svih neurona u tom sloju. Ulazni vektor na slici, označen sa $X$ je dimenzije 8, i uz njega se dovodi još jedno ulazno polje koje uvek ima vrednost jedan. Težine sa kojima se to dodatno polje množi su pragovi odgovarajućih neurona u sledećem sloju. Težine između slojeva su označene sa $W^l$, gde $l$ predstavlja broj sloja čijim neuronima pripadaju. Ulazni vektor možemo posmatrati kao nulti sloj čije vrednosti aktivacije su vrednosti ulaznog vektora.

$$a_i^l = \sigma\left(\sum_{j=0}^{k} a_j^{l-1} W_{j,i}^l + b_i^l\right), i \in [0, n) \tag{3.2}$$

Ako primenu funkcije aktivacije na vektor definišemo kao primenu te funkcije na svaki pojedinačni element vektora, npr. $\sigma([z_0, z_1, z_2]) = [\sigma(z_0), \sigma(z_1), \sigma(z_2)]$, jednačinu računanja aktivacije celog sloja kraće možemo zapisati u vektorskom obliku kao u (3.3).

$$a^l = \sigma(W^l a^{l-1} + b^l) \tag{3.3}$$

Takođe, radi pojednostavljenja notacije, uvešćemo $z$ kao vrednost vektorskog proizvoda ulaza i težina sabranog sa pragom: $z^l = W^l x + b^l$. Tada ceo proces računanja izlaza



(A) Sigmoidna funkcija. (B) Hiperbolički tangens. (C) Ispravljačka linearna aktivacija.

SLIKA 3.3: Popularne funkcije aktivacije

propagacijom unapred u mreži sa dva sloja (kao na slici 3.2) definišu sledeće formule:

$$
\begin{aligned}
z^1 &= W^1 X + b^1 \\
a^1 &= \sigma(z^1) \\
z^2 &= W^2 a^1 + b^2 \\
a^2 &= \sigma(z^2)
\end{aligned}
$$

Za funkciju aktivacije $\sigma$ popularni izbori su logistička funkcija (3.4), hiperbolički tangens (3.5), a od nedavno i ispravljačka linearna funkcija (3.6) (eng. *rectified linear unit*), čiji grafici su redom prikazani na slici 3.3.

$$sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{3.4}$$

$$tanh(z) = \frac{e^z + e^{-z}}{e^z - e^{-z}} \tag{3.5}$$

$$relu(z) = max(z, 0) \tag{3.6}$$

Za izbor broja slojeva i broja neurona po sloju mreže ne postoje teorijske smernice, pa se preporučuje eksperimentisanje sa tim tzv. hiperparametrima modela. Izbor funkcije aktivacije se može napraviti na osnovu njenog kodomena i toga šta se želi postići sa konkretnom mrežom. Na primer, relu aktivacija je postala popularna u dubokim mrežama jer su kod nje problemi nestajućeg i eksplodirajućeg gradijenta umanjeni.

## 3.3 Treniranje neuronskih mreža propagacijom grešaka unazad

Propagacija grešaka unazad (eng. *backpropagation*) je specijalan slučaj automatskog diferenciranje, tj. diferenciranje unazad. Ovaj algoritam je otkrivan više puta [26] u istoriji, ali kao rad koji je prepoznao korisnost algoritma za treniranje višeslojnih neuronskih mreža obično se navodi papir Rumelharta, Hintona i Vilijamsa iz 1986 [27]. Efikasna propagacija grešaka unazad predstavlja ključni algoritam za treniranje dubokih neuronskih mreža.

U suštini, propagacija grešaka unazad je gradijentna metoda optimizacije. To znači da ona koristi gradijent funkcije koju optimizuje (u ovom kontekstu neuronske mreže), koji sadrži parcijalne izvode po svim parametrima (tj. težinama u neuronskoj mreži). Vektor gradijenta, u prostoru parametara, pokazuje u pravcu najvećeg rasta vrednosti funkcije, a na osnovu njega metoda može da napravi korak u tom pravcu, sa istim ili suprotnim smerom u zavisnosti od toga da li se funkcija maksimizuje ili minimizuje.

Ideja algoritma propagacije grešaka unazad je da se definiše diferencijabilna funkcija greške koja zavisi od izlaza i parametara mreže, i da se nekom gradijentnom metodom optimizacije parametri mreže podešavaju tako da se greška minimizuje. Za funkciju greške se obično bira skalarna funkcija koja zavisi od izlaza mreže i tačnih vrednosti koje bi mreža trebalo da vrati, i računa vrednost proporcionalnu njihovoj razlici. Radi konciznosti, u ovom odeljku razmatraćemo usrednjenu kvadratnu razliku kao funkciju greške, datu u jednačini 3.7. Parametri $W$ i $b$ su parametri za optimizaciju, a $X$ i $Y$ su fiksirani ulazni parametri koje optimizacija ne menja. $f$ predstavlja model, tj. funkciju koju model računa sa datom konfiguracijom parametara (težina $W$ i pragova $b$). Funkcija greške iz 3.7 je pogodna i zato što je njen izvod $(Y - f)\nabla f$.

$$J(W, b; X, Y) = \frac{1}{2}\|Y - f(W, b; X)\|^2 \tag{3.7}$$

Ako se svaki sloj mreže gleda kao funkcija težina, pragova i ulaza, cela mreža predstavlja kompoziciju takvih funkcija. Automatskim diferenciranjem unazad se mogu dobiti parcijalni izvodi po težinama i pragovima mreže za sve slojeve u jednom prolazu. Ti izvodi se onda koriste za ažuriranje svaih parametara po formulama 3.8 i 3.9. Parametar $\alpha$ predstavlja brzinu učenja koja može zavisiti i od broja iteracija[3]. Kao optimizacioni metod je izabran gradijentni spust (eng. *gradient descent*), koji pravi korake u pravcu najbržeg opadanja vrednosti funkcije greške.

[3]Često se $\alpha$ smanjuje tokom iteracija da bi se optimizacija fokusirala na užu okolinu u prostoru parametara koji pretražuje.

$$W^l_{i,j} = W^l_{i,j} - \alpha \frac{\partial J}{\partial W^l_{i,j}} \tag{3.8}$$

$$b^l_i = b^l_i - \alpha \frac{\partial J}{\partial b_i} \tag{3.9}$$

U literaturi uvođenje propagacije grešaka unazad [28–31] obično daje kompaktniji opis algoritma bez eksplicitnog uvođenja automatskog diferenciranja unazad, koji je prikazan u algoritmu 6. Za podsetnik oko notacije, $J$ označava funkciju greške, $a$ označava aktivaciju sloja neurona a $a_cache$ je promenljiva koja čuva niz tih aktivacija, $\sigma$ se odnosi na funkciju aktivacije neurona, $z$ je vrednost zbira vektorskog proizvoda težina i ulaza, i pragova za dati sloj neurona, sloj je indeksiran u eksponentu u zagradama kao u $W^{(l)}$. Takođe, operator $\odot$ predstavlja Adamarov proizvod, tj. vektor proizvoda pojedinačnih elemenata operanada.

**Algorithm 6** Backprop

**Input:** ulazni podaci skupa za trening $X$, očekivani izlazi skupa za trening $Y$, svih težina po slojevima mreže $W$, skup svih pragova po slojevima mreže $b$, broj slojeva u mreži $L$.

**Output:** skup parcijalnih izvoda funkcije greške po težinama $\nabla_W J(W, b; x, y)$, skup parcijalnih izvoda funkcije greške po pragovima $\nabla_b J(W, b; X, Y)$.

1: $a_cache \leftarrow forward_eval(W, b, X)$
2: inicijalizuje se niz $\delta$ koji će držati signale greške po slojevima za svaki neuron.
3: $\delta^{(L)} = -(Y - a_cache^{(L)}) \odot \sigma'(z^{(L)})$
4: **for** $l = L - 1$ down to 1 **do**
5: $\quad \delta^{(l)} = W^{(l)} \delta^{(l+1)} \odot \sigma'(z^{(l)})$
6: **end for**
7: **for** $l = L$ down to 1 **do**
8: $\quad \nabla_{W^{(l)}} J(W, b; X, Y) = \delta^{(l+1)} a_cache^{(l)}$
9: $\quad \nabla_{b^{(l)}} J(W, b; X, Y) = \delta^{(l)}$
10: **end for**

Može se primetiti da je ta verzija algoritma ekvivalentna automatskom diferenciranju unazad, opisanom u odeljku 2.3.2, i to po sledećim stavkama:

- Na liniji 1 u algoritmu 6, počinje se evaluacijom mreže unapred i čuvaju se međurezultati (u $a_cache$).
- Niz $\delta$ je ekvivalentan pridruženom članu.
- Računanje $\delta$ izlaza koje počinje na liniji 3 je ekvivalentno drugom koraku računanja pridruženog člana za funkciju greške (tj. operatoru najvišeg nivoa u grafu izraza koji se diferencira automatski), a prvi korak inicijalizacije pridruženog člana na vrednost 1 je preskočen. Zbog toga imamo razliku u računanju $\delta$ za izlazni sloj i za skrivene slojeve.

- Računanje izvoda na linijama 7-9 je množenje pridruženog člana izvodom lokalnog operatora po parametru težine (u kom slučaju operator je vektorski proizvod) ili parametru praga (gde je operator sabiranje).

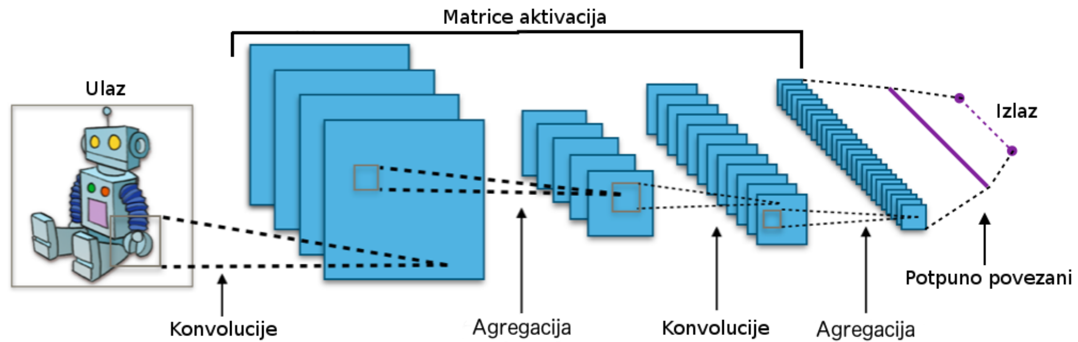
Zbog toga je u implementaciji biblioteke koja je opisana u ovom radu, za izvođenje propagacije grešaka podržan metod automatskog diferenciranja.

## 3.4 Konvolutivne neuronske mreže

Kod nekih vrsta podataka, kao što su slike, postoje prostorne zavisnosti i strukture između susednih komponenti u vektoru. Potpuno povezani slojevi neurona trenirani nad takvim podacima, ne koriste efikasno postojanje tih struktura, te moraju da ih nauče iznova na svakoj poziciji u vektoru. Takođe, potpuno povezani slojevi se ne skaliraju dobro na slikama, jer čak i slike male rezolucije imaju približno kvadratni broj piksela u odnosu na njihove dimenzije, koji kada se pomnoži sa brojem neurona u prvom sloju daje veliki broj težina koje treba optimizovati i čuvati u memoriji.

Na ideji da detekcija lokalnih struktura u vektoru može biti korisna na svim pozicijama u vektoru, zasnivaju se konvolutivne mreže [32]. One su slične višeslojnom perceptronu po tome što njihovi neuroni takođe rade vektorski proizvod težina i ulaza koje sabiraju sa pragom i rezultat propuštaju kroz funkciju aktivacije ($a = \sigma(WX + b)$). Jedna od razlika je u tome što svaki neuron u konvolutivnoj mreži nema težine za sve komponente vektora ulaza, već samo ograničeni lokalni podniz, čije su dimenzije zadate i jednake za sve neurone u sloju. Taj podniz fiksnih dimenzija se naziva i kernel (eng. *receptive field*). Druga razlika je što se računanje aktivacije svakog neurona izvršava na svim pozicijama u vektoru, pomerajući kenrel po njemu. U slučaju primene na dvodimenzionalne ulazne podatke, izlaz svakog neurona na kraju te operacije je matrica izračunatih aktivacija, a izlaz celog sloja je trodimenzionalni niz. Takođe, dimenzije tog izlaznog niza se dodatno mogu smanjiti korakom agregacije.

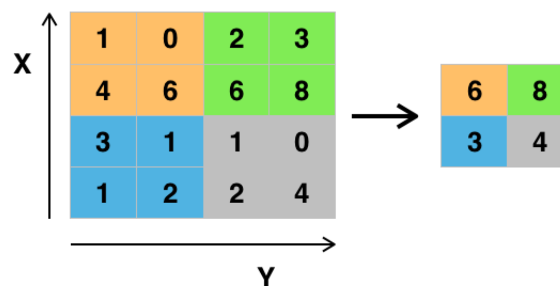
U terminima obrade signala, svaki neuron sa svojim težinama definiše jedan filter, a primena tog filtera na ulazne podatke se naziva konvolucija. Slika 3.4 daje prikaz tipične strukture slojeva konvolutivne mreže. Na ulazu se dovodi slika kao trodimenzionalni niz – dve dimenzije za širinu i visinu slike, i treća za kanale boja. Konvolucija se uvek vrši samo nad prostornim dimenzijama, a svaki filter ima treću dimenziju koja je ista kao broj kanala na ulazu. Parametri za operaciju konvolucije su širina i visina filtera, pomeraj po svakoj prostornoj dimenziji (eng. *stride*) i način konvolucije na margini ulaza. Pri konvoluciji blizu ivica ulaza, gde filter prelazi granice slike, moguće dodati marginu popunjenu nulama ili ne vršiti konvoluciju na marginama. Zajedno sa brojem




SLIKA 3.4: Konvolutivna neuronska mreža[4].

filtera, tj. sa brojem neurona u sloju, ovi parametri definišu dimenzije izlaza sloja. Broj dimenzija izlaza je isti kao broj dimenzija ulaza, ako primetimo da se trodimenzionalna slika može posmatrati kao četvorodimenzioni niz gde je poslednja dimenzija broj slika, koji može biti i veći od 1 ako se obrada vrši u paketima (eng. *batches*).

Radi smanjenja prostorne veličine izlaza, moguće je koristiti veći pomeraj pri konvoluciji, ili dodatni korak agregacije (eng. *pooling*). Agregacija poput konvolucije uzima lokalne podnizove ulaza i nad njima primenjuje neku operaciju (na primer $max()$) i zamenjuje ih u izlazu jednim skalarom. Parametri ove operacije su dimenzije lokalnog polja nad kojim se izvršava, i pomeraj po svim prostornim dimenzijama ulaza. Na slici 3.5 je prikazano agregiranje operacijom traženja maksimalne vrednosti, sa dimenzijama polja $2 \times 2$ i pomerajem po x i po y osama jednakim 2.




SLIKA 3.5: Agregacija maksimumom[4].

Ako dimenzije ulazne slike označimo sa $D_1 \times H_1 \times W_1$, gde je $D_1$ dubina ili broj kanala, $H_1$ visina a $W_1$ širina ulazne slike, dimenzije izlaza date su sledećim formulama:

$$W_2 = (W_1 - F + 2P)/S + 1 \tag{3.10}$$

$$H_2 = (H_2 - F + 2P)/S + 1 \tag{3.11}$$

$$D_2 = K \tag{3.12}$$

[4] Slika preuzeta sa Vikipedije.

gde su $D_2$, $H_2$ i $W_2$ dimenzije izlaza, $F$ je širina ili visina filtera (pretpostavljajući da je filter kvadratnog oblika), $P$ je veličina okvira nula koji se dodaje na marginama (eng. *zero padding*), $S$ je pomeraj filtera po prostornim dimenzijama ulaza a $K$ je broj filtera.

Posle nekoliko konvolutivnih slojeva dodaju se potpuno povezani slojevi koji su povezani sa svim aktivacijama u poslednjem konvolutivnom sloju. Njihova svojstva su ista kao u višeslojnom perceptronu. U ovakvoj arhitekturi, većina parametara mreže se nalazi u ovim potpuno povezanim slojevima, i njihov broj može biti reda veličine $10^6$. Uopšteno u mašinskom učenju kada model ima mnogo parametara, javlja se problem preprilagođavanja, koji se ogleda u tome da model nauči suvišne specifičnosti prisutne u podacima za trening. Te naučene specifičnosti smanjuju generalizaciju modela, tj. performanse na podacima koji nisu viđeni u treningu. Da bi se to izbeglo, koriste se regularizacione tehnike [3] koje ograničavaju parametre modela (u ovom slučaju mreže) tako da se umanji učenje nepotrebnih detalja iz trening podataka.

U radu Križevskog [1, 33], u potpuno povezanim slojevima uvedena je operacija *Dropout* koja se sastoji u tome da se prilikom treniranja mreže, slučajno izabran deo neurona u sloju ugasi (pomnoži nulom), kako bi se sprečilo interno preprilagođavanje između slojeva. Ova tehnika regularizacije je postigla značajno poboljšanje generalizacije i model predstavljen u tom radu, nazvan Alexnet, je 2012. ostvario najbolji rezultat na ILSVRC-2012 takmičenju u klasifikaciji slika, sa greškom od 15.3% što je bio veliki pomak u odnosu na drugoplasiranu metodu sa 26.2%.

Na kraju, izlazi poslednjeg sloja se dovode na softmax funkciju, data u jednačini 3.13, koja vraća vektor verovatnoća pripadnosti ulaza svakoj od klasa.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \text{ for } j \in [1, k] \tag{3.13}$$

Način efikasne implementacije konvolutivnih operacija u slojevima neuronske mreže je opisan u odeljku 4.4. Propagacija greške kroz konvolutivne slojeve je slična propagaciji greške u potpuno povezanim slojevima a detaljniji prikaz se može naći u odeljku 4.5.

# Glava 4

# Implementacija bibilioteke za razvoj neuronskih mreža na OpenCL-u

Implementaciji neuronskih mreža, može se pristupiti na dva načina. Jedan je da se one implementiraju kao graf objekata, gde je svaki objekat pojedinačni neuron. Ovaj "objektno orijentisani" pristup nije pogodan za brzo izvršavanje na mašinama optimizovanim za vektorske operacije, kakve su grafičke kartice a i sve veći broj modernih procesora. U ovom radu izabran je pristup u kome se mreža predstavlja kao skup linearnih i nelinearnih operacija nad višedimenzionim nizovima. Po uzoru na slične biblioteke [16–18] koje rade na CUDA platformi, ova biblioteka ne implementira određeni skup konkretnih arhitektura neuronskih mreža, već operatore i atome od kojih se mogu konstruisati izrazi koji definišu neuronsku mrežu. To daje veću fleksibilnost u eksperimentisanju sa arhitekturama mreža. Svaki od tih operatora računanje svoje vrednosti izvršava na grafičkoj kartici. Za olakšanje implementacije treniranja mreža, izrazi definisani pomoću apstraktne sintakse koju biblioteka pruža, podržavaju automatsko diferenciranje, što se koristi u propagaciji grešaka unazad opisanoj u odeljku 3.3. Radni naziv biblioteke razvijene u ovom radu je *shaderrider*, a ime dolazi od starog naziva za procesorska jezgra na grafičkim karticama – *shaders*.

Za implementaciju biblioteke izabran je programski jezik Python (verzije 2.7), kao jedan od popularnijih u mašinskom učenju. Takođe, Python je pogodan zbog efikasne interakcije sa nativnim kodom[1], što je iskorišćeno za pozivanje brzih operacija za množenje matrica i vektora iz clBLAS[2] biblioteke. Nativni modul za interakciju sa clBLAS-om

[1]Python je interpretirani jezik što znači da interpreter izoluje python kod od specifičnosti mašine i operativnog sistema na kome se izvršava. Nativni kod u ovom kontekstu znači kod koji nije pisan u pythonu i koji se kompilira i direktno izvršava na platformi na kojoj radi i pythonov interpreter. To omogućava veću brzinu izvršavanja i dublju integraciju sa sistemom.

[2]clBLAS je verzija biblioteke BLAS (*Basic Linear Algebra Subprograms*) koja implementira operacije nad vektorima i matricama u OpenCL-u.

je razvijen u Cython-u [34], jeziku koji je proširenje Python-a i prevodilac koji generiše nativni kod i može da se upotrebi za pisanje nativnih proširenja za Python.

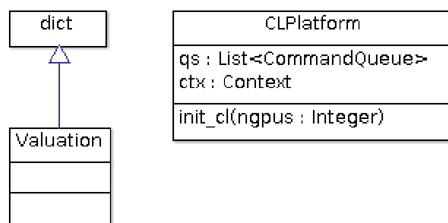
Biblioteka se sastoji od sledećih celina:

- modula za upravljanje OpenCL platformom,
- apstraktne sintakse za definisanje simboličkih izraza uz podršku za automatsko diferenciranje,
- funkcija za rad sa PyOpencl višedimenzionim nizovima na grafičkoj kartici.

Generalni pristup korišćenja biblioteke počinje tako se pomoću modula za upravljanje OpenCL platformom inicijalizuje uređaj na kome se želi izvršavati računanje (CPU ili GPU). Zatim se pomoću atoma i operatora konstruišu izrazi koji definišu neuronsku mrežu. Za tu mrežu se takođe definiše i funkcija greške koja će biti optimizovana tokom treninga. Zatim se definiše valuacija sa konkretnim vrednostima promenljivih koje učestvuju u izrazima neuronske mreže i funkcije greške. Tada se mogu izvršavati koraci optimizacije koju korisnik biblioteke treba da implementira, uz korišćenje automatskog računanja gradijenta za funkciju greške, koje mu biblioteka pruža. Ažuriranje parametara mreže, koji se tokom optimizacije nalaze u memoriji GPU-a, se odvija korišćenjem odgovarajućih izraza za ažuriranje vrednosti, takođe definisanih pomoću operaatora iz biblioteke. Uopšteno, računanje vrednosti i diferenciranje operatora izraza su jedine funkcije koje se izvršavaju na GPU-u.

## 4.1 Upravaljanje OpenCL platformom

Modul za upravljanje OpenCL platformom (`shaderrider.clplatf`), sadrži funkciju za inicijalizaciju OpenCL uređaja, konteksta i reda za izrvšavanje, zvanu `init_cl(ngpus=0)`. Njen jedini parametar označava broj grafičkih kartica na kojima će se izvršavati, a kada je na podrazumevanoj vrednosti 0 to znači da se za izvršavanje koristi CPU. U klasnom dijagramu na slici 4.1, ova funkcija je prikazana kao metoda klase CLPlatform, ali ta klasa odgovara modulu. Osim inicijalizacije platforme u ovom modulu se nalazi i klasa `Valuation` koja predstavlja valuaciju za izraze, tj. mapiranje konretnih vrednosti na imena promenljivih koje se koriste u izrazima definisanim pomoću klasa apstraktne sintakse. Ova klasa takođe upravlja transferom podataka između sistemske memorije i memorije GPU-a na kome se izrazi izvršavaju. Klasa proširuje standardni Python rečnik, a kada se nekom ključu (imenu promenljive) dodeli vrednost tipa Numpy višedimenzionog niza (NDArray) ona se prebacuje u PyOpencl Array tip koji podatke drži na GPU memoriji. Sam memorijski transfer se dodaje u red za izvršavanje na odgovarajućem GPU-u i




SLIKA 4.1: Klase modula za upravljanje OpenCL platformom.

prema trenutnoj konfiguraciji reda, dešava se kada se u izvršavanju prvi put referencira dati niz.

## 4.2 Apstraktna sintaksa i automatsko diferenciranje

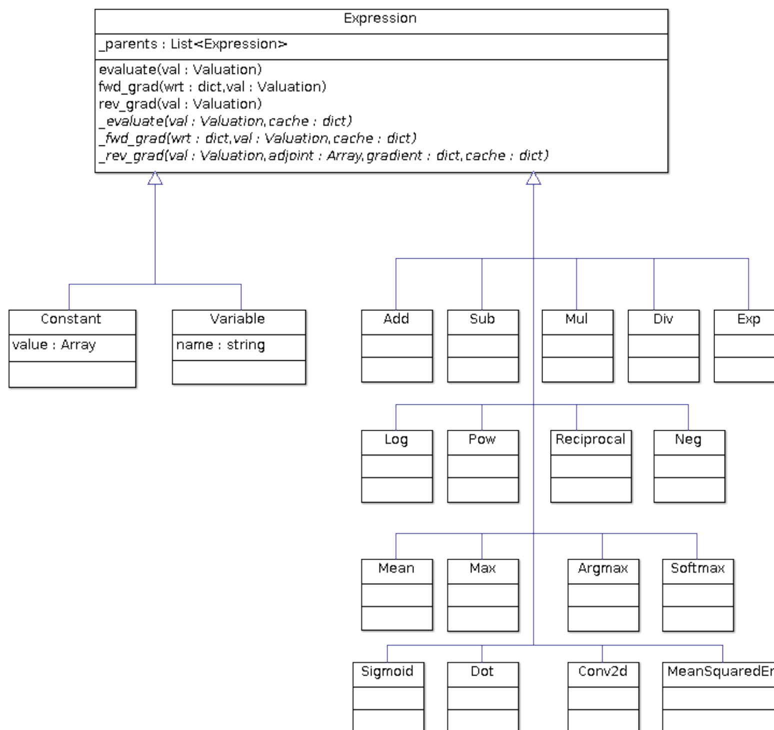
Ključna osobina biblioteke za podršku lakšem programiranju neuronskih mreža, je automatsko računanje gradijenta funkcija definisanih pomoću objekata apstraktne sintakse koju biblioteka implementira. Ti objekti obuhvataju atome, tj. promenljive i konstante, i operatore, kao što su sabiranje, množenje, deljenje, kao i operatore specifične za neke neuronske mreže, kao dropout, maxpool i slično. Izrazi definisani pomoću tih objekata podržavaju računanje vrednosti i automatsko diferenciranje[3], unapred i unazad.

Na slici 4.2 prikazan je klasni dijagram apstraktne sintakse i pomoćnih objekata za definisanje valuacije i inicijalizaciju OpenCL platforme. Glavni objekat apstraktne sintakse je `Expression` koji definiše izraz. U njemu se nalaze javne metode za pozivanje evaluacije i računanja izvoda unapred i unazad. Takođe on definiše apstraktne zaštićene metode koje implementiraju svi konkretni tipovi izraza.

Svaki izraz se gradi atoma koji su konstante (`Constant`) ili promenljive (`Variable`) pomoću operatora koji su veznici. Da bi se izračunala vrednost simboličkog izraza, definisanog pomoću `Expression` objekata, poziva se `evaluate(valuation)` metoda koja kao parametar uzima objekat valuacije u kome očekuje da se nađu vrednosti za sve promenljive izraza. Prilikom računanja u `Expression` objektu se održava keš svih izračunatih vrednosti podizraza koji se kasnije koristi pri računanju gradijenta unazad. Izračunata vrednost se vraća pozivaocu kao PyOpencl niz čiji se podaci mogu po potrebi prebaciti sa memorije GPU-a na sistemsku.

Automatsko diferenciranje unapred se izvršava pozivanjem metode `fwd_grad(wrt, valuation)`. Prvi parametar, `wrt`, je Python mapa u kojoj su ključevi imena promenljivih a vrednosti

[3]Za pojedine nediferencijabilne operatore, definisani su specijalni slučajevi.




SLIKA 4.2: Klasni dijagram apstraktne sintakse biblioteke.

0, osim za jednu promenljivu po kojoj se izvod računa, za koju vrednost iznosi 1. Ovaj metod se ne koristi često u implementaciji neuronskih mreža, ali je uključen zbog kompletnosti i relativno lake implementacije u odnosu na diferenciranje unazad. Ova metoda takođe vraća rezultat kao PyOpencl niz.

Automatsko diferenciranje unazad, koje koristi propagacija grešaka unazad, se nalazi u metodi **`rev_grad(valuation)`**. Ova metoda računa gradijent izraza u tački zadatoj valuacijom. Gradijent se vraća kao Python mapa u kojoj su ključevi imena promenljivih, a vrednosti PyOpencl nizovi koji predstavljaju vrednost parcijalnog izvoda po datoj promenljivoj, u tački definisanoj valuacijom. Ova mapa se kasnije u metodi optimizacije koristi za ažuriranje vrednosti parametara mreže.

Ova tri metoda predstavljaju javni interfejs izraza. Sledeća tri metoda koji se vide na dijagramu i počinju karakterom '_' su privatne metode[4] koje izvršavaju rekurzivne pozive kroz graf izraza.

Za sada biblioteka podržava 22 operatora nabrojana u tabeli 4.1. Ovaj skup operatora je podržan jer je dovoljan za implementaciju testova izvršenih za potrebe ovog rada. Svi operatori podržavaju računanje vrednosti i automatsko diferenciranje, prema interfejsu

[4]Python nema zaštitu pristupa na metodama i promenljivima klasa, ali postji konvencija da sve što počinje sa '_' karakterom treba da bude tretirano kao privatno.

| | | | |
|---|---|---|---|
| `Add` | sabiranje | `Max` | maksimum |
| `Sub` | oduzimanje | `Dot` | vektorski proizvod ili množenje matrica |
| `Mul` | množenje | `Conv2d` | konvolucija po dve prostorne dimenzije |
| `Div` | deljenje | `Dropout` | dropout regularizacija |
| `Pow` | stepenovanje | `ReLU` | ispravljačka linearna jedinica |
| `Exp` | eksponencijalna f-ja | `Softmax` | softmax funkcija |
| `Log` | logaritam | `MaxPool` | agregacija maksimumom |
| `Reciprocal` | recipročna vrednost | `Mean` | srednja vrednost |
| `Sigmoid` | sigmoidalna f-ja | `MeanSquaredErr` | usrednjena kvadratna greška |
| `Neg` | negacija | `NotEq` | nejednakost |
| `Argmax` | pozicija maksimalnog argumenta | `Reshape` | menjanje oblika niza |

Tabela 4.1: Podržani operatori.

iz klase `Expression`. Operatori se nalaze u modulu `shaderrider.operators`, a apstraktna klasa `Expression`, kao i `Variable` i `Constant` se nalaze u modulu `shaderrider.expr`.

## 4.3 Višedimenzioni nizovi

Za reprezentaciju višedimenzionih nizova, potrebno je podržati neograničen broj dimenzija, kao i nekontinualnost nizova u memoriji. Pod nekontinualnošću niza ovde podrazumevamo da susedni elementi niza ne moraju biti na susednim lokacijama u memoriji. To omogućava da se neke operacije nad nizom mogu obaviti bez kopiranja podataka u memoriji. Struktura koja podržava takav višedimenzioni niz je prikazana na listingu 4.3. Numerički podaci u njoj se čuvaju u jednodimenzionom nizu *data*, broj dimenzija u polju *nd*, same dimenzije u nizu *dims*, kao i niz koraka tj. rastojanja u memoriji između dva susedna elementa po svakoj dimenziji (*strides*). Sa takvom reprezentacijom moguće je

efikasnije implemetnirati operacije indeksiranja, izdvajanja podnizova, transponovanja i druge.

```
1 struct ndarray {
2         int nd;
3         int *dims;
4         int *strides;
5         int element_size;
6         void *data;
7 }
```

Listing 4.1: Struktura višedimenzionog niza.

U Python-u svi podaci, uključujući i numeričke, posmatraju se kao objekti, zbog čega direktno računanje sa nizovima ne bi efikasno koristilo resurse. Za efikasna numerička izračunavanja u Python-u se koristi biblioteka Numpy koja se oslanja na nativne numeričke tipove podržane na platformi na kojoj se izvršava. Numpy takođe implementira višedimenzioni niz u klasi NDArray koji odgovara strukturi datoj u listingu 4.3. Međutim, Numpy ne može da pristupi radnoj memoriji na grafičkoj kartici, kao ni da izvršava instrukcije na njoj.

Zbog toga postoje i druge biblioteke koje implementiraju neki podskup funkcionalnosti Numpy i podržavaju rad na GPU. Prema stepenu podrške OpenCL-u i dostupnosti dokumentacije, izdvaja se biblioteka PyOpencl [35], koja uz Python omotač oko OpenCL api-a, sadrži i klasu višedimenzionog niza. Ograničenje višedimenzionog niza iz PyOpencl-a u odnosu na Numpy je nedostatak podrške za napredno indeksiranje niza kao i uklapanja kompatibilnih[5] dimenzija operanada, u Numpy terminologiji zvano *broadcasting*.

U našoj biblioteci iskorišćen je višedimenzioni niz iz PyOpencl-a, a funkcije nižeg nivoa su razvijene tako da nadomeste njegova ograničenja implementacijom specifičnih kernela za svaki operator koji od njih zavisi. U odeljku 4.2 je bilo reči o metodama izraza za računanje vrednosti i diferenciranje. Ove metode pozivaju operatore i metode PyOpencl-a ako ekvivalenti postoje (kao u slučaju operatora `Add` za sabiranje). Kada odgovarajuća operacija nije podržana u PyOpencl-u, kao što je slučaj za konvoluciju (operator `Conv2d`) ili BLAS operacije (operator `Dot`), implementirane su nedostajuće operacije.

## 4.4 Konvolucija kao množenje matrica

Zbog arhitekture GPU kartica, ubrzanje paralelnog koda zavisi od primene algoritama prilagođenih specifičnostima hijerarhije memorije i kontrole toka [13]. Za ubrzanje konvolucije koja se koristi u neuronskim mrežama, osim direktne implementacije i oslanjanja

[5]Dimenzije dva operanda su kompatibilne ako su jednake ili ako je jedna od njih 1, pa se niz njenog operanda može iskopirati onoliko puta koliko iznosi odgovarajuća dimenzija drugog operanda. Zahvaljujući *strides* polju u strukturi višedimenzionog niza, samo kopiranje podataka niza se može izbeći tako što se za pomeraj (*stride*) po toj dimenziji stavi 0.

na FFT [36], postoji i način da se konvolucija prevede u množenje matrica [37]. Poslednja tehnika daje dobre rezultate jer proizvođači hardvera održavaju optimizovane biblioteke za linearnu algebru (cuBLAS, clBLAS, MKL i sl.), koje pružaju brze implementacije množenja matrica. Ova tehnika ima dobar odnos između dobitka performansi i komplikovanosti implementacije, te je ona iskorišćena u ovom radu.

Algoritam se sastoji od dve faze – konverzije ulaznog niza i filtera u 2-d matrice, i samog množenja matrica. Množenje matrica se implementira koristeći operator vektorskog proizvoda koji poziva GEMM (*general matrix-matrix*) funkciju BLAS biblioteke. Korak konverzije u matrice se naziva *im2col* i na slici 4.3 je dat prikaz pomeranja podataka u nizovima koje on izvodi. Na levoj strani slike (4.3a), vidi se konverzija ulazne matrice u kolone a na desnoj (4.3b) koverzija matrica filtera u redove.

(A) transformacija ulaza (B) transformacija filtera

Slika 4.3: im2col operacije nad ulaznim vektorom i filterima u konvoluciji.

Konverzija slike se odvija tako što se svaki kernel svakog filtera, na svakoj poziciji u slici razvije u jednu kolonu izlazne matrice. Filteri se razvijaju u redove matrice koja je drugi operand proizvoda. Dužina kolona u izlaznoj matrici biće jednaka veličini filtera, tj. ukupnom zbiru broja elemenata po svim osama filtera. Broj kolona, tj. širina izlazne matrice je broj pozicija na kojima se dati filter primenjuje na ulaznom nizu, tj proizvodu širine i visine izlaza konvolucije, date u jednačinama 3.10 i 3.11.

Na primer, ako imamo ulaznu matricu dimenzija $7 \times 7$, i dva kernela dimenzija $3 \times 3$ sa pomerajem 1 po obe ose, *im2col* operacija nad ulaznom slikom će vratiti matricu visine 9 i širine 25. Ako su ulazni nizovi i filteri dimenzija većih od 2, i onda se na isti način slika razvija u kolone po svim filterima. To znači da prva kolona sadrži komponente koje ulaze u prvu poziciju kernela svih kanala filtera, redom. Ovako razvijene ulazne matrice i filteri, kada se pomnože daju isti rezultat kao konvolucija.

## 4.5 Propagacija greške unazad u konvolutivnoj mreži

Propagacija greške unazad u konvolutivnim slojevima mreže se može postići tako što se izvrši potpuna konvolucija signala greške iz prethodnog[6] sloja $\delta^{l+1}$ i filtera prethodnog sloja, rotiranih za 180 stepeni, kao što je prikazano u jednačini 4.1. Detaljno izvođenje ovog postupka se može naći u članku [38].

$$\delta^l = f'(z^l) \cdot full_conv(\delta^{l+1}, rot180(W^{l+1})) \tag{4.1}$$

Pošto je u ovom radu iskorišćen pristup sa razvijanjem konvolucije u množenje matrica, jednostavniji način da se izračuna gradijent jeste korišćenjem standardnog pravila proizvoda (jednačina 4.2).

$$(f \cdot g)' = f' \cdot g + f \cdot g' \tag{4.2}$$

Međutim, samo pravilo proizvoda nije dovoljno, nego je potrebno primeniti transformaciju obrnutu[7] u odnosu na *im2col* iz odeljka 4.4, koja se obično naziva *col2im*. Kako se u izlazu *im2col* funkcije, jedna komponenta ulaznog niza može pojaviti jednom ili više puta, u prolazu unazad, signali greške koji se nalaze na pozicijama ponovljenih elemenata se sabiraju u jedinstveni signal greške za taj element.

Drugi način da se razmišlja o propagaciji greške u konvoluciji je da se neuron konvolutivnog sloja posmatra kao potpuno povezani sloj u kome svi neuroni dele jedan skup težina, i pri ažuriranju, sve promene se dešavaju na istom skupu deljenih težina.

## 4.6 Primer upotrebe za definisanje višeslojnog perceptrona

Primer upotrebe biblioteke za definisanje višeslojnog perceptrona je dat u listingu 4.2. Na početku se uvozi biblioteka Numpy i objekat višedimenzionog niza i PyOpencl biblioteke. Zatim su uvezeni moduli shaderrider biblioteke, clplatf za inicijalizaciju platforme

[6] Pošto se propagacija greške odvija unazad, prethodni sloj ovde ima obrnuto značenje u odnosu na ono u evaluaciji ili konstrukciji mreže, pa je $l+1$ prethodni sloj sloja $l$.

[7] *col2im* operacija nije pravi inverz od *im2col*.

i objekat valuacije, operators za operatore i expr jer se u njemu nalaze definicije klasa `Variable` i `Constant`. Modul linalg sadrži funkcije za rad sa clBLAS bibliotekom i uvezen je zbog funkcije vektorskog proizvoda koja na liniji 55 iskorišćena dase zaobiđe nepostojanje sumiranja po određenoj osi u PyOpencl nizovima. Ovo je privremeno rešenje koje bi trebalo zameniti operatorom sumiranja.

Posle uvoženja modula, prikazana je definicija potpuno povezanog sloja neurona od 7. do 28. linije. Konstruktor sloja uzima Numpy generator slučajnih brojeva rng, ceo broj koji označava broj sloja li, promenljivu koja predstavlja ulazni niz input, i dimenzije ulaza i izlaza sloja nin i nout. Zatim je moguće proslediti i težine W i pragove b ako su već poznati[8] i funkciju aktivacije, za koju je podrazumevana sigmoidalna funkcija.

Na liniji 11 se dohvata red za izvršavanje koji je potreban kasnijekao parametar za kreiranje PyOpencl nizova koji će sadržati W i b. Na linijama 22 i 23 se nalazi kreiranje simboličkih promenljivih koje predstavljaju težine i pragove. Konstruktoru klase `Variable` se prosleđuje ime promenljive u kojem učestvuje broj sloja `li`.

Zatim je definisan višeslojni perceptron od linije 30, koji sadrži dva potpuno povezana sloja, i metode za treniranje i test. Konstruktor uzima generator slučajnih brojeva koji će slojevi koristiti za inicijalizaciju težina i pragova, i dimenzije ulaza, skrivenog sloja i izlaza. Počinje sa konstrukcijom simboličkog izraza uz pomoć prethodno definisane klase `FullyConnectedLayer`. Kao funkcija greške koju trening minimizuje, na 38. liniji je definisana usrednjena kvadratna razlika između izlaza mreže i pravih klasa ulaza ($Y$).

```
1  import numpy as np
2  from pyopencl import array as clarray
3  from shaderrider import clplatf as platform, operators as op, expr
4  from shaderrider import linalg
5
6
7  class FullyConnectedLayer(object):
8      def __init__(self, rng, li, input, nin, nout, W=None, b=None,
9                   activation_fn=op.Sigmoid):
10         self.input = input
11         q = platform.qs[0]
12         if W is None:
13             nw = np.asarray(rng.uniform(low=-0.01, high=0.01, size=(nin, nout)),
14                             dtype=np.float32)
15             W = clarray.to_device(q, nw)
16         if b is None:
17             b = clarray.zeros(q, (nout,), np.float32)
18
19         self.W = W
20         self.b = b
21
22         vW = expr.Variable('W'+li)
23         vb = expr.Variable('b'+li)
24         lin_out = op.Add(op.Dot(self.input, vW), vb)
25         self.output = lin_out if activation_fn is None \
26                             else activation_fn(lin_out)
27         self.params = [(vW.name, self.W), (vb.name, self.b)]
```

[8]Mogu biti unapred poznati ako se prethodno istreninrana mreža učitava iz fajla.

```
28
29
30 class MLP(object):
31     def __init__(self, rng, nin, nhid, nout):
32         X = expr.Variable('X')
33         Y = expr.Variable('Y')
34         self.layer1 = FullyConnectedLayer(rng, '1', X, nin, nhid)
35         self.layer2 = FullyConnectedLayer(rng, '2', self.layer1.output,
36                                           nhid, nout)
37         self.y_pred = op.Argmax(self.layer2.output, -1)
38         self.cost = op.MeanSquaredErr(self.layer2.output, Y)
39         self.errors = op.Mean(op.NotEq(self.y_pred, Y))
40         self.params = self.layer1.params + self.layer2.params
41
42     def train(self, X, Y, learning_rate=0.01):
43         val = platform.valuation()
44         val['X'] = X
45         val['Y'] = Y
46         for name, value in self.params:
47             val[name] = value
48
49         grad = self.cost.rev_grad(val)
50
51         debatch_help_vector = clarray.zeros(platform.qs[0], (Y.shape[0], 1),
52                                             dtype=np.float32) + 1
53         for name, value in self.params:
54             if name.startswith('b'):
55                 dbh = linalg.dot(platform.qs[0], grad[name],
56                                  debatch_help_vector, transA=True)
57                 value -= learning_rate*dbh.ravel()
58             else:
59                 value -= learning_rate*grad[name]
60
61     def test(self, X, Y):
62         val = platform.valuation()
63         val['X'] = X
64         val['Y'] = Y
65         for param, value in self.params:
66             val[param] = value
67         err = self.errors.evaluate(val)
68         return err
```

Listing 4.2: Primer višeslojnog perceptrona sa dva potpuno povezana sloja, koji koriste logističku funkciju aktivacije, i usrednjeno kvadratno odstupanje kao funkciju greške.

Sama optimizacija mreže nije deo biblioteke, te je korisniku ostavljen izbor metode koju će implementirati, a gradijent mreže može dobiti pozivanjem `rev_grad()` metode. U primerima u ovom radu korišćen je stohastički gradijentni spust, čiji je korak promene parametara dat u jednačinama 3.8 i 3.9, a može se videti i u metodi `train()`. Ona počinje prebacivanjem ulaznih parametara u objekat valuacije koji ih po potrebi prebacuje iz sistemske memorije na grafičku karticu. Zatim se poziva automatsko računanje gradijenta funkcije greške definisane u konstruktoru kao usrednjeno kvadratno rastojanje. Vektor `debatch_help_vector` je vektor jedinica dužine kao broj primera koji se koristi da zaobiđe ograničenje operatora sumiranja koji u ovoj verziji ne može da sumira višedimenzione nizove po proizvoljnoj dimenziji. Zbog toga se sumiranje gradijenata za pragove ($b$) računa vektorskim proizvodom sa tim nizom jedinica na liniji 55. Metoda `test` izračunava vrednosti mreže za ulaze $X$ i poredi ih sa zadatim izlazima $Y$, vraćajući grešku kao tačnost klasifikacije, tj. procenat loše predviđenih klasa.

U listingu 4.3 je prikazano korišćenje objekata definisanih u istingu 4.2. Na početku se inicijalizuje OpenCL platforma i generator slučajnih brojeva (iz Numpy bibliteke) koji će biti korišćen za slučajnu inicijalizaciju nizova težina i pragova u mreži. Kreiran je višeslojni perceptron sa dva sloja, koji ima ulazne vektore dužine 784 (jer su MNIST slike dimenzija 28x28), u prvom skrivenom sloju ima 32 neurona, a u izlaznom sloju ih ima 10 (za svaku klasu, tj. cifru koju može da prepozna).

Zatim se učitavaju podaci i dele na trening, validacione i test podskupove. U ovom primeru je prikazan gradijentni spust koji se odvija u paketima od po 512 (*n_batches*) u 100 ponavljanja (*n_epochs*). U svakom prolazu, za poslednji podskup podataka se računa i ispisuje greška validacije.

```
1   platform.init_cl(1)
2   rng = np.random.RandomState(1234)
3   mlp = MLP(rng, 784, 32, 10)
4
5   X, Y, testX, testY = get_mnist_data()
6   vsplit = int(0.9 * X.shape[0])
7   trainX = X[:vsplit, :]
8   validX = X[vsplit:, :]
9   trainY = Y[:vsplit, :]
10  validY = Y[vsplit:]
11
12  n_epochs = 100
13  batch_size = 512
14  n_train_batches = trainY.shape[0] / batch_size
15
16  epoch = 0
17  while epoch < n_epochs:
18      epoch += 1
19      for minibatch_index in xrange(n_train_batches):
20          mlp.train(trainX[minibatch_index*batch_size:
21                           (minibatch_index+1)*batch_size, :],
22                    trainY[minibatch_index*batch_size:
23                           (minibatch_index+1)*batch_size, :])
24
25          if (minibatch_index == n_train_batches-1) and minibatch_index > 0:
26              err = mlp.test(validX, validY)
27              print 'validation error:', err, 'batch:',
28              print minibatch_index, '/', n_train_batches
29
30  err = mlp.test(testX, testY)
31  print 'TEST ERROR:', err
```

LISTING 4.3: Primer korišćenja objekta višeslojnog perceptrona za prepoznavanje MNIST cifara.

# Glava 5

# Eksperimentalni rezultati

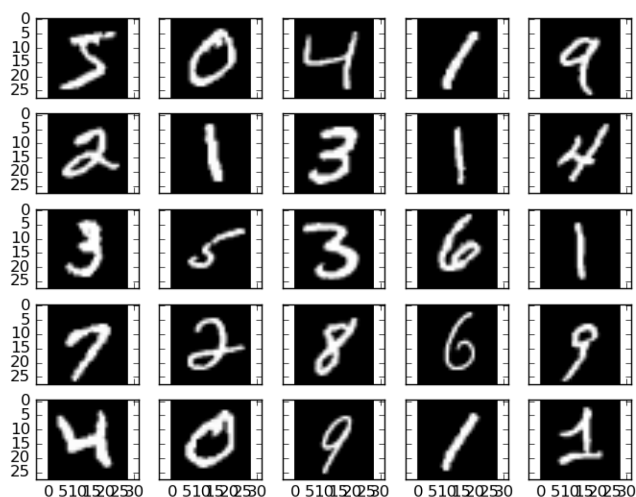
Da bi se ispitalo kako razvijeni sistem radi, testovi su izvršeni nad, za današnje standarde, manjim skupovima podataka, MNIST i CIFAR. Veći skupovi podataka kao što su ImageNet [39] i MS COCO [40] su za sada van domašaja testova koje biblioteka može da izvrši za manje od nekoliko sati, a kraći testovi su sasvim dovoljni da se isprobaju sve implementirane operacije.

Svi testovi su izvršavani na AMD Radeon R9 390x grafičkoj kartici sa 8 GB radne memorije, i procesoru Intel i7-4770 sa 32 GB radne memorije. Grafička kartica sadrži 44 procesorske jedinice sa ukupnim brojem od 2816 jezgara i radi na taktu od oko 1 GHz, tok procesor sadrži 4 jezgra i radi na brzini od 3.9 GHz.

## 5.1 MNIST

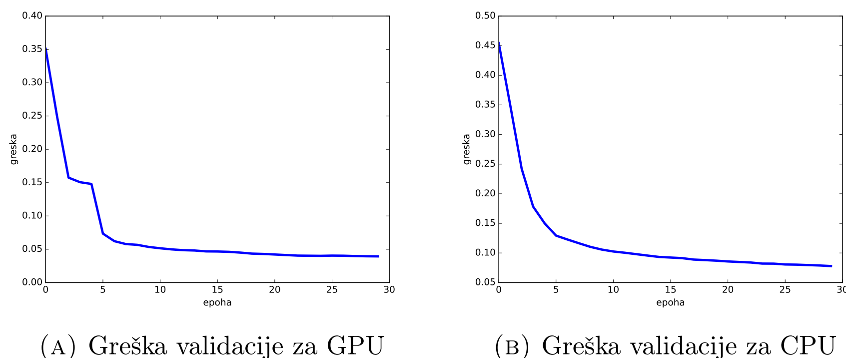
MNIST [41] baza je skup crno-belih slika rukom napisanih cifara. Sadrži 60000 primera za treniranje i 10000 primera za testiranje, zajedno sa tačnim klasama za svaki od njih. Svaka slika je u rezoluciji $28 \times 28$ piksela, gde jedan bajt predstavlja jedan piksel. Primer slučajno izabranih slika iz ove baze je prikazan na slici 5.1. Ovaj skup podataka i problem nisu toliko teški za današnje tehnike, ali se često koristi za rano i brzo testiranje modela.

Za model na ovom testu izabran je višeslojni perceptron, sa jednim skrivenim slojem od 32 neurona, i izlaznim slojem sa 10 neurona za 10 mogućih klasa cifara koje predstavlja tako što izlazni vektor ima jedinicu na mestu predviđene cifre, a nulu na svim ostalim (eng. *one-hot vector*). Funkcija aktivacije je logistička funkcija. Za optimizaciju je korišćen stohastički gradijentni spust u grupama (eng. *batched stochastic gradient descent*). Kao funkcija greške korišćena je usrednjenog kvadratnog rastojanja (3.7). Regularizacije u



Slika 5.1: MNIST primeri podataka

funkciji greške, kao što su L1 i L2, nisu korišćene, a brzina učenja (eng. *learning rate*) je fiksna i postavljena na vrednost 0.01.




(A) Greška validacije za GPU (B) Greška validacije za CPU

Slika 5.2: Opadanje greške validacije u treningu višeslojnog perceptrona

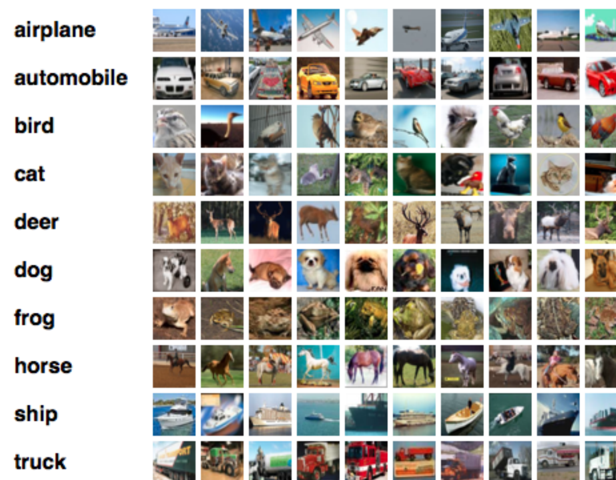
Rezultati treniranja su prikazani na slici 5.2 gde je prikazana greška validacije kroz 30 epoha treniranja, kako opada do konačnog rezultata od 5% (95% tačnosti) za GPU (OpenCL) verziju i 7.8% (92.2% tačnosti) za CPU (Numpy). Bolji rezultati sa dva reda veličine manjom greškom se mogu postići naprednijom arhitekturom mreže [32], ali cilj ovog testa je bio da se uspostavi da osnovna funkcionalnost biblioteke radi i da daje bilo kakvo ubrzanje u odnosu na izvršavanje na procesoru. Razlika u brzini treninga se može videti u tabeli 5.1.

| | 1 epoha | ukupno |
|---|---|---|
| CPU | 6.9s | 207s |
| GPU | 2.11s | 63.4s |

TABELA 5.1: Vreme treninga višeslojnog perceptrona na MNIST podacima, koristeći procesor (CPU) i grafičku karticu (GPU).

## 5.2 CIFAR

Ovaj skup podataka je obeleženi skup od 60000 slika dimenzija $32 \times 32$ sa 3 kanala za boju, koje su raspoređene u 10 klasa. Model koji je primenjen za klasifikaciju u ovom eksperimentu je smanjena verzija Alexnet [1] mreže, koja se sastoji od tri konvolutivna sloja, dva potpuno povezana skrivena sloja i softmax izlaznog sloja. Praćen je pristup u kome konvolutivne operacije čuvaju veličinu slika, a pooling operacije je uvek smanjuju.




SLIKA 5.3: CIFAR skup podataka

Arhitektura slojeva je sledeća:

- Ulazne slike dolaze u grupama od po $m$ i predstavljene su četvorodimenzionim tenzorom dimenzija $m \times 3 \times 32 \times 32$.
- Prvi konvolutivni sloj ima 32 filtera sa 3 kanala i širinom i visinom od $5 \times 5$. Pomeraj filtera je 1, a margina za dodavanje nula je 2.
- Izlaz prvog konvolutivnog sloja prolazi kroz operaciju agregacije gde se bira maksimalna vrednost iz polja veličine $2 \times 2$ sa pomerajem 2 u obe prostorne dimenzije. Ovo smanjuje prostorne dimenzije izlaza, pa su njegove nove dimenzije $m \times 32 \times 16 \times 16$.

- U drugom konvolutivnom sloju, tenzor koji predstavlja filtere ima dimenzije $32 \times 32 \times 5 \times 5$, što znači da ima 32 filtera sa 32 kanala (jer je prethodni sloj imao 32 filtera), i kernel dimenzija $5 \times 5$. I u ovom sloju, pomeraj po obe dimenzije je 1, i dodavanje nula je na margini širine 2, što čini da izlaz iz ovog sloja bude istih prostornih dimenzija kao ulaz.
- Izlaz drugog sloja se takođe propušta kroz agregaciju max operacijom, na poljima $2 \times 2$, sa pomerajem 2, što dimenzije izlaza transformiše u $m \times 32 \times 8 \times 8$.
- Treći konvolutivni sloj filtere drži u tenzoru dimenzija $64 \times 32 \times 5 \times 5$, i ima pomeraj 1 po obe prostorne dimenzije, kao i dodavanje nula na margini širine 2.
- Max agregacija je takođe primenjena i na ovaj sloj sa istim parametrima kao u prethodna dva slučaja. Dimenzije izlaza su $m \times 64 \times 4 \times 4$.
- Prvi potpuno povezani sloj ima 64 neurona i veličinu ulaza 1024 (što je jednako broju izlaza za pojedinačnu sliku u prethodnom sloju, $64 \times 4 \times 4$)
- Drugi potpuno povezani sloj ima 10 neurona.
- Izlazni sloj je softmax funkcija koja ima 10 izlaza u slučaju CIFAR-10 skupa.

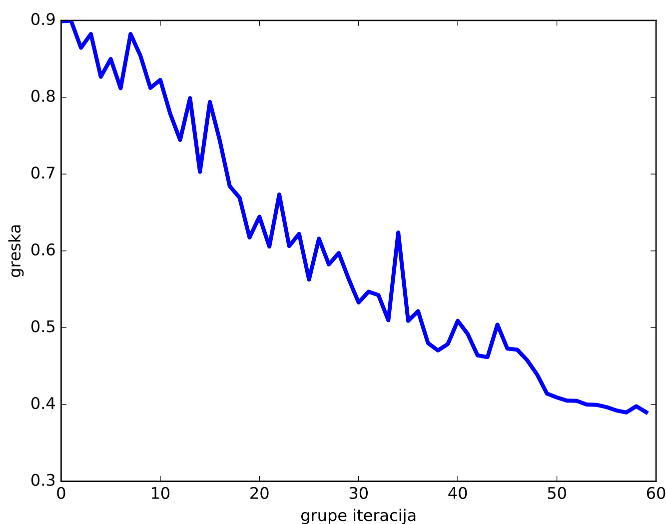
Svi pomenuti slojevi kao funkciju aktivacije koriste ispravljačku linearnu aktivaciju (3.6). Ponašanje greške na validacionom skupu podataka se može videti na slici 5.4. Posle 10 epoha treniranja, konačna greška na test skupu je iznosila 38.1% a trening je trajao 25 minuta. Može se primetiti da greška ne opada glatko što je posledica korišćenja stohastičkog gradijentnog spusta, koji pravac koraka optimizacije računa nad podskupom podataka. Ovi rezultati su lošiji od rezultata koje je postigla implementacija Aleksa Križevskog[1], koja je postigla grešku od 26% za 80 sekundi treniranja. Međutim, njegova (CUDA) implementacija je bila jako optimizovana za specifičnu arhitekturu kartice na kojoj je pokretana.

Radi bližeg poređenja, ekvivalentna mreža je implementirana i pomoću Theano biblioteke i izvršena na NVIDIA Tesla m2090 kartici[2], bez korišćenja optimizovane CuDNN[3] biblioteke. U ovim eksperimentima postignuta je greška na testu od 54% za 50 minuta treniranja mreže.

[1]Dostupno na stranici: `https://code.google.com/p/cuda-convnet/`.

[2]Eksperimenti sa Theano bibliotekom su izvršeni na PARADOX superračunaru u Laboratoriji za primenu računara u nauci, Instituta za fiziku u Beogradu, delimično finansiranom od strane Ministarstva obrazovanja, nauke i tehničkog razvoja Republike Srbije, na projektu ON171017.

[3]CuDNN nije podržan na karticama Fermi generacije, kao što je Tesla m2090.




SLIKA 5.4: Greška validacije pri treningu nad CIFAR-10 skupom podataka. Grupe iteracija predstavljaju 6 validacija po epohi, sa po 13 grupa podataka veličine 128 instanci između svake dve validacije.

## 5.3 Diskusija

Iz prikazanih rezultata se može videti da biblioteka implementirana u ovom radu, ubrzava treniranje neuronskih mreža na grafičkoj kartici u odnosu na izvršavanje na procesoru, ali da je to ubrzanje manje nego što ga nude slične biblioteke bazirane na CUDA platformi. To odstupanje se može pripisati boljoj optimizaciji parametara implementacije. Performanse na grafičkim karticama mogu mnogo da zavise od prilagođenosti velične kernela koji se računaju kao i preklapanja računanja i pomeranja podataka između memorije kartice i sistemske memorije. Da bi se vremena biblioteke razvijene u ovom radu popravila, potrebno je optimizovati takve parametre unutar operatora biblioteke i izvršiti uslovni izbor istih pri izvršavanju u zavisnoti od hardvera na kome se pokreću.

Još jedna od prednosti koje CUDA biblioteke imaju jeste postojanje CuDNN [42] bibliotke za jako optimizovane konvolutivne neuronske mreže, čiji ekvivalent trenutno ne postoji za OpenCL. Rezultat Theano biblioteke bez CuDNN optimizacija u poređenju sa efikasnom implementacijom Križevskog, ukazuje na to koliko nedostatak jako optimizovanih algoritama može da utiče na performanse.

# Glava 6

# Zaključak

U ovom radu opisana je i razvijena biblioteka za treniranje i izvršavanje neuronskih mreža na grafičkim karticama bazirana na OpenCL platformi i programskom jeziku Python. Ona podržava definisanje neuronskih mreža pomoću operacija linearne algebre nad n-dimenzionim tenzorima koji predstavljaju težine mreže. Takvi izrazi koji definišu mreže, podržavaju automatsko diferenciranje pomoću koga se automatizuje impelmentacija propagacije greške unazad u neuronskim mrežama. Takođe, biblioteka organizuje inicijalizaciju OpenCL platforme, i transfere između sistemske i memorije akceleratora.

Rezultati prikazani u poglavlju 5 ukazuju da osnovne funkcionalnosti biblioteke rade, ali da ima još dosta prostora za dalji razvoj i optimizaciju. Trenutni skup operatora koji je podržan nije dovoljan za implementaciju svih postojećih arhitektura neuronskih mreža, ali postojeće definicije daju interfejs za lako proširivanje biblioteke novim operatorima.

# Bibliografija


[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[2] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[3] Christopher M Bishop. *Pattern recognition and machine learning, vol. 1.* Springer, New York, 2006.

[4] Kevin P. Murphy. *Machine Learning, a Probabilistic Perspective.* MIT Press, 2012.

[5] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[6] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[7] Lamblin P. Popovici D. Larochelle H. Bengio, Y. Greedy layer-wise training of deep networks. page 153–160, 2006.

[8] Poultney C. Chopra S. LeCun Y. Ranzato, M. Efficient learning of sparse representations with an energy-based model. page 1137–1144, 2006.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015.

[10] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[11] Aaftab Munshi. The opencl 1.2 speciffication. *Khronos OpenCL Working Group. Khronos Grpoup. Khronos*, 2012.

[12] Nenad Mitic. *Uvod u organizaciju racunara.* Matematicki fakultet, Beograd, 2013.

[13] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach.* Newnes, 2012.

[14] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[15] Nvidia. Cuda c programming guide, 2015.

[16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

[17] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

[18] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL `http://learningsys.org/papers/LearningSys_2015_paper_33.pdf`.

[19] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[20] François Chollet. Keras: Deep learning library for theano and tensorflow, 2015.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[22] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015.

[23] Ian Goodfellow Yoshua Bengio and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016. URL `http://www.deeplearningbook.org`.

[24] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[25] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[26] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. doi: 10.1016/j.neunet.2014.09.003. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

[27] DE Rumerhart, GE Hinton, and RJ Williams. Learning representations by back-propagation errors. *Nature*, 323:533–536, 1986.

[28] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[29] Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, and Caroline Suen. Ufldl tutorial, 2013. URL `http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial`.

[30] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL `http://neuralnetworksanddeeplearning.com`.

[31] Wikipedia. Backpropagation, 2016. URL `https://en.wikipedia.org/wiki/Backpropagation`. [online; accessed 30.08.2016].

[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998.

[33] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[34] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2010.118.

[35] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012. ISSN 0167-8191. doi: 10.1016/j.parco.2011.09.001.

[36] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[37] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[38] Grzegorz Gwardys. Convolutional neural networks backpropagation: from intuition to derivation. URL `https://grzegorzgwardys.wordpress.com/2016/04/22/8/#unique-identifier`.

[39] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[40] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. URL `http://arxiv.org/abs/1405.0312`.

[41] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[42] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.