SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 688

Prepoznavanje gesti na GPU

Marijan Šuflaj

Zagreb, lipanj 2014.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 7. ožujka 2014.

DIPLOMSKI ZADATAK br. 688

Pristupnik: **Marijan Šuflaj (0036452901)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Prepoznavanja gesti na GPU**

Opis zadatka:

Proučiti algoritme strojnog učenja u kontekstu prepoznavanja znakova te konturama zadanih objekata. Proučiti postupke implementacije algoritama na grafičkoj procesnoj jedinici, posebice upotrebom biblioteke OpenCL. Razmotriti mogućnosti proučenih algoritama posebno u kontekstu paralelizacije odnosno pogodnosti implementacije na GPU. Načiniti programsku implementaciju razrađenih algoritama i načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate primjenom implementiranih postupaka i usporediti performanse na CPU odnosno GPU. Diskutirati upotrebljivost ostvarene aplikacije.
Izraditi odgovarajući programski proizvod. Koristiti programski jezik Java, odnosno biblioteku OpenCL. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 14. ožujka 2014.
Rok za predaju rada: 30. lipnja 2014.

Mentor:
Prof.dr.sc. Željka Mihajlović

Predsjednik odbora za diplomski rad profila:
Prof.dr.sc. Siniša Srbljić

Djelovođa:
Doc.dr.sc. Tomislav Hrkać

Zahvaljujem se svojoj metorici prof. dr. sc. Željki Mihajlović kao i dr. sc. Marku Čupiću na ukazanoj stručnoj pomoći prilikom izrade ovog rada. Posebno se želim zahvaliti svojoj obitelji koja me je podupirala tijekom studija u ostvarivanju svojih želja i ciljeva.

Sadržaj

Popis algoritama

Popis slika

1. Uvod

U današnje, računalno doba, u kojem se sve više stvari automatizira i dalje postoji par segmenata gdje računala nisu dobra kao ljudi. Primjerice, napisati program koji će na predočenoj slici označiti lica svih ljudi koji se na toj slici nalaze nije lagan posao. Iz tog se razloga razvila posebna grana računarske znanosti zvana umjetna inteligencija koja istražuje algoritme i metode rješavanja problema u kojima su ljudi još uvijek puno djelotvorniji.

Problem za koji ne postoji efikasan deterministički[1] algoritam nastoji se riješiti algoritmima umjetne inteligencije. Primjeri takvih problema jesu prepoznavanje kompleksnijih objekata na slikama poput ljudskog lica ili izvlačenje znanja iz velikog skupa podataka.

Valja napomenti da se prepoznavanje odvija temeljem značajki koje opisuju jedan uzorak koje sustav primi. Sustav potom vraća razred u koji je svrstao predočeni uzorak. Također je bitno napomenuti da je proces učenja puno kompleksniji i računalno zahtjevniji od samog procesa prepoznavanja.

Može se povući i analogija s ljudima. Poznato je da je potrebno poprilično više vremena da se nauče svi prometni znakovi i to je proces koji zahjeva veći napor mozga. No jednom kada je taj problem savladan i naučeni su svi prometni znakovi predočavanjem nekog prometnog znaka prepoznavanje je gotovo trenutno.

Primjena prepoznavanja gesti ima sve veću primjenu razvojem pametnih telefona. Njihovi zasloni osjetljivi su na dodir i time pružaju mogućnost unošenja gesti u dvije dimenzije. Također, razvijaju se uređaji koji omogućavaju uzorkovanje gesti u trodimenzionalnom prostoru te se time sam postupak učenja i prepoznavanja dodatno otežava.

Dodatno, geste mogu biti razlomljene, primjerice kada više prstiju izvodi jednu gestu, što još više otežava problem prepoznavanja gesti.

Ovaj rad organiziran je kako slijedi. U poglavlju 2 opisan je problem prepoznavanja gesti. Nakon toga, u poglavlju 3 dan je uvod u model umjetnog neurona

[1] svaki je korak točno određen, nema slučajnih događaja koji mogu utjecati na stanje sustava

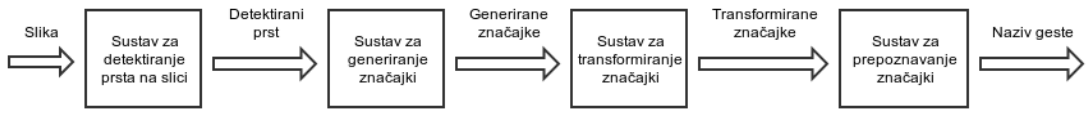
kao i model umjetne neuronske mreže te je u poglavlju 4 opisan jedan od načina učenja tih umjetnih neuronskih mreža. Zatim je u poglavlju 5 opisana arhitektura umjetne neuronske mreže koja se uči kao i načini obrade uzoraka nad kojima se ta mreža uči. U poglavlju 6 opisani su načini ubrzavanja postupka učenja neuronske mreže. Poglavlje 7 daje uvod u biblioteku OpenCL dok je u poglavlju 8 opisana implementacija postupka učenja umjetne neuronske mreže potpomognuta bibliotekom OpenCL. U poglavlju 9 dana su mjerenja ostvarena u sklopu ovog rada kao i smjernice za daljnja poboljšanja implementacije za postupak učenja umjetne neuronske mreže.

2. Problem prepoznavanja gesti

Problem prepoznavanja gesti puno je kompleksniji nego što zvuči. Jedan je primjer takvog prepoznavanja i prepoznavanje geste ostvarene vrhom prsta pred kamerom. Kamera kao izlaz daje niz sličica od kojih svaka prikazuje vrh prsta u drugom vremenskom trenutku. Odmah prepoznati napravljenu gestu nije moguće, već se problem treba razložiti na manje, jednostavnije probleme. Tako će, primjerice, prvo postojati jedan dio sustava koji će na slici ostvarenoj kamerom detektirati prst.

Detektiranje kompleksnijih objekata na slici složen je problem sam za sebe i neće biti razmatran u ovom radu. Nakon toga postoji drugi dio sustava koji temeljem informacija prvoga sustava gradi listu koordinata u ovisnosti o vremenu gdje se vrh prsta nalazi i to predstavlja skup značajki dok je cijela gesta jedan uzorak. Ni taj dio sustava neće biti razmatran u ovom radu.

Na kraju, dolaze dva dijela sustava koji obavljaju prepoznavanje gesti. Prvi je od njih sustav koji će izvorni skup značajki pretvoriti u neki drugi skup značajki, pogodniji za prepoznavanje i to je ulaz u drugi gdje će se obaviti samo prepoznavanje. Kao izlaz iz sustava dobije se ime geste koja je napravljena prstom. Dijagram tih sustava vidljiv je na slici 2.1.



Slika 2.1: Dijagram sustava za prepoznavanje gesti

U ovom će radu biti razmotrena upravo zadnja dva dijela sustava pa će tako biti obrađena dva algoritama i tri načina pretvaranja značajki.

2.1. Ispitni podatci

Kako je rečeno ranije, prepoznavanje vrha prsta na slici dobivenoj iz kamere kao i praćenje istoga ne razmatra se u radu pa je bilo potrebno na neki način prikupiti ispitne podatke. U tu je svrhu napravljena mala internetska aplikacija koja je posjetiteljima ponudila mogućnost ucrtavanja četiri simbola korištenjem miša: α, β, γ i Ω. Navedena aplikacija svakih 50 ms očitava poziciju miša na ekranu i sprema vrijednosti u polje. To se polje potom šalje poslužitelju i sprema u bazu podataka za daljnju obradu. Na taj se način dobivaju koordinate točaka u ovisnosti o vremenu.

Ukupno je prikupljeno 1088 uzoraka gesti i udjeli su vidljivi u tablici 2.1.

Tablica 2.1: Udjeli uzoraka

Simbol	Broj uzoraka	Postatak uzorka
α	278	25.55%
β	249	22.89%
γ	276	25.37%
Ω	285	26.19%

Kako je veći broj ljudi unosio podatke velika je raznolikost unesenih podataka. Tako, primjerice, dvije različite osobe poprilično različito crtaju simbol α, što je i vidljivo iz slike 2.2a i slike 2.2b.

(a) Prvi simbol α **(b)** Drugi simbol α

Slika 2.2: Simbol α unesen od dvije različite osobe

Što je skup raznolikiji to će učenje biti zahtjevnije (teže dostizanje manjih pogrešaka), no naučeni sustav za prepoznavanje bit će učinkovitiji i otporniji na varijacije uzoraka koje predstavljaju isti simbol.

3. Neuronske mreže

Umjetna neuronska mreža predstavlja pojednostavljeni model koji opisuje rad ljudskoga mozga. Ljudski mozak sastoji se od 10^{11} (Eluyode i Akomolafe, 2008) malih jednostavnih procesnih elemenata nazvanih neuroni. Jedan neuron tako je povezan s otprilike još 10^4 neurona. Smatra se da iz tako velike interakcije malih procesnih elemenata proizlazi inteligencija bioloških organizama. Tako, primjerice, čovjek kojem je predočena slika majke prepoznat će svoju majku unutar 0.1 sekunde.

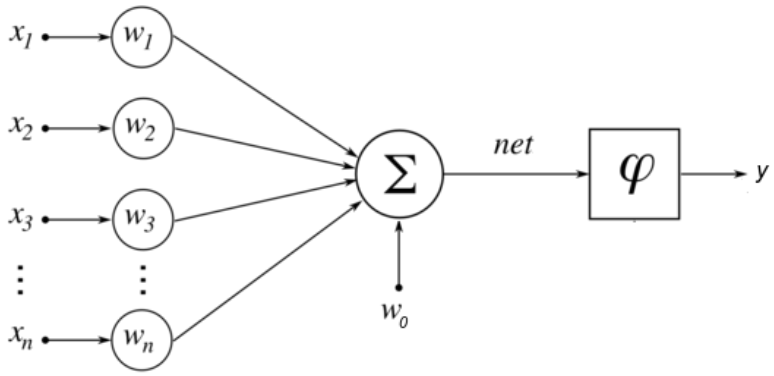
Uzevši u obzir da prolazak impulsa kroz neuron traje 1 ms (Čupić et al., 2013) vidljivo je da je već kroz manji broj slojeva neurona donesen zaključak. To pokazuje koliko je obrada podataka u ljudskom mozgu masovno paralelna i pogodna za implementaciju na grafičkom procesoru (GPU[1]).

3.1. Osnovni model umjetnog neurona

Promatranjem biološkoga neurona, kakav se nalazi u ljudskom mozgu, vidljivo je da se on sastoji od dendrita, tijela neurona i aksona. Neuron je preko dendrita povezan s velikim brojem ostalih neurona koji mu preko njih šalju električne impulse. Ti se naboji akumuliraju i u jednom trenutku, kada je dosegnut određen prag, neuron *pali* – sav taj naboj šalje kroz akson i time pobuđuje druge neurone.

Razmatranjem primjera biološkoga neurona moguće je definirati jednostavan model umjetnoga neurona koji se može koristiti na računalu. Primjer takvoga modela neurona vidljiv je na slici 3.1, a sastoji se od ulaza x_1 do x_n koji predstavljaju veze s preostalim neuronima, težina w_1 do w_n koje određuju u kojoj mjeri ulazi pobuđuju neuron, tijela neurona koje akumulira pobudu net i prijenosne funkcije φ koja predstavlja akson gdje se akumulirana pobuda obrađuje prijenosnom funkcijom te potom prosljeđuje na izlaz neurona (neu).

[1](engl. *General processing unit*) – predstavlja centralnu jedinicu za upravljanje i provedbu matematičkih operacija na grafičkoj kartici



Slika 3.1: Osnovni model umjetnog neurona

Time se odmah dolazi i do izraza za računanje akumuliranja pobuda (izraz (3.1)) i samog izlaza neurona (izraz (3.2)).

$$\begin{aligned} net &= w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n \\ &= w_0 + \sum_1^n w_i \cdot x_i \\ &= [w_0 \ w_1 \ \ldots \ w_n] \cdot [1 \ x_1 \ \ldots \ x_n] \\ &= \vec{w} \cdot \vec{x} \end{aligned} \tag{3.1}$$

$$y = \varphi(net) \tag{3.2}$$

Varijabla w_0 predstavlja negativni iznos praga koji kod biološkog neurona postoji prije otpuštanja pobude. Vektor $\vec{w} = (w_0, w_1, \ldots, w_n)$ predstavlja $(n+1)$-dimenzijski vektor težina dok $\vec{x} = (1, x_1, x_2, \ldots, x_n)$ predstavlja $(n+1)$-dimenzijski vektor ulaza (Dalbelo Bašić et al., 2008).

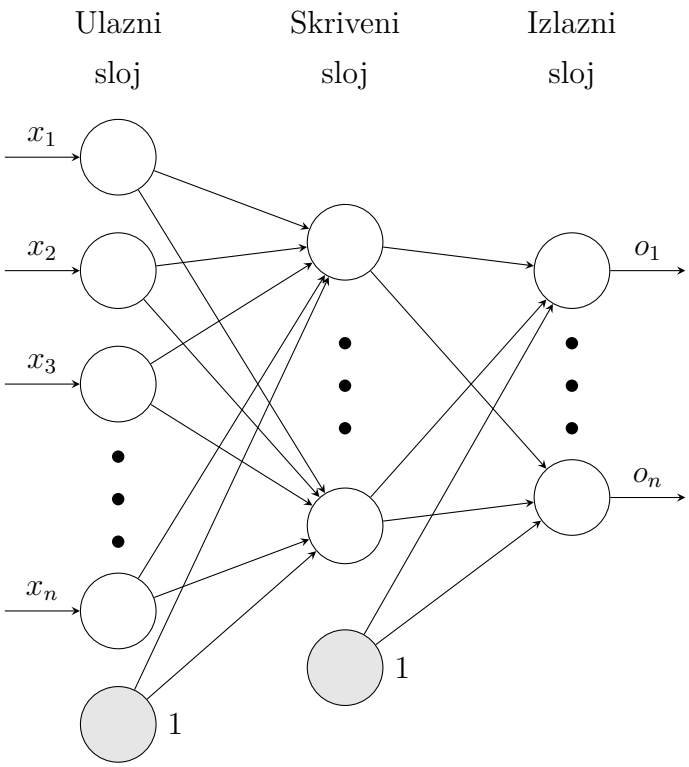
Funkcija φ predstavlja prijenosnu funkciju. Često se kao prijenosna funkcija koriste sigmoidalna[2] funkcija ili funkcija tangens hiperbolni[3] ($tanh$).

3.2. Unaprijedna slojevita potpuno povezana umjetna neuronska mreža

Definiranjem umjetnoga neurona mogu se početi koristiti te male građevne jedinice, koje same po sebi ne mogu rješavati kompleksnije probleme, kako bi se

[2] definirana kao $\frac{1}{1+e^{-x}}$

[3] definiran kao $\frac{sinh(x)}{cosh(x)}$



Slika 3.2: Unaprijedna slojevita potpuno povezana umjetna neuronska mreža s jednim skrivenim slojem

ostvarila jedna veća cijelina koja može obavljati neku funkcionalnost. U ovome radu korištena je unaprijedna slojevita potpuno povezana umjetna neuronska mreža. Primjer takve mreže prikazan je na slici 3.2. Mreža se sastoji od $k + 1$ slojeva od kojih je prvi sloj ulazni sloj, zadnji sloj predstavlja izlazni sloj, a ostalih $k - 1$ slojeva predstavljaju skrivene slojeve (ann).

Kako težina w_0 ne bi bila izuzetak u odnosu na sve ostale težine koje se nalaze između umjetnih neurona ona se povezuje s virtualnim umjetnim neuronom koji na izlazu uvijek daje vrijednost 1 i nema ulaza. Na taj je način implementacija jednostavnija budući da su sada sve težine povezane s neuronima i može ih se tretirati na isti način. Na slici 3.2 ti su neuroni predstavljeni sivim kružićem.

3.2.1. Ulazni sloj

Ulazni sloj predstavlja neurone koji ne obavljaju nikakvu funkcionalnost, već im je jedini zadatak da ulaz preslikaju na izlaz. Tako, primjerice, ako uzorci problema koji se rješava imaju tri značajke, postojat će ukupno tri ulazna neurona. Prijenosna funkcija definirana im je kao $\varphi(net) = net$.

3.2.2. Skriveni sloj

Skriveni sloj predstavljaju neuroni koji prethodne ulaze transformiraju do nečega što će u konačnici izlazni sloj moći koristiti. Oni imaju konkretne ulaze od neurona prethodnih slojeva i sa svakim neuronom iz prethodnog sloja povezani su jednom vezom. Izlaz neurona i utječe na ulaz neurona j težinom $w_{i,j}$.

3.2.3. Izlazni sloj

Izlazni sloj je zadnji sloj i svoje izlaze ne spaja na druge neurone već ti izlazi predstavljaju izlaze umjetne neuronske mreže.

4. Algoritmi učenja umjetnih neuronskih mreža

Mreža sama po sebi ne predstavlja ništa korisno za upotrebu ako težine nisu dobro podešene. U težinama je implicitno pohranjeno znanje o problemu. Prilagođavanje vrijednosti tih težina zove se učenje neuronske mreže. Naime, potrebno je težine prilagoditi tako da mreža radi što manju pogrešku. Kao i kod čovjeka, učenjem i predočavanjem istih uzoraka više puta neke veze će ojačati dok će neke oslabiti. Princip učenja umjetne neuronske mreže slijedi princip učenja kod bioloških organizama. Algoritmima se neke veze jačaju dok se neke druge slabe.

4.1. Algoritam propagiranja greške unazad

Najpoznatiji algoritam za učenje je algoritam propagiranja greške unazad (engl. *back-propagation*), koji temeljem pogreške neurona prilagođava težine do ostalih neurona. Izvod izraza koji se koriste u algoritmu vidljiv je u dodatku A. U nastavku će biti dan samo uvid u ideju i izraze koji se koriste dalje u algoritmu.

4.1.1. Ideja algoritma

Algoritam se temelji na gradijentnom spustu[1]. Greška koja se optimira definirana je izrazom

$$E(\vec{w}, \{(\vec{x}_s, \vec{t}_s)\}) = \frac{1}{2N} \sum_{s=1}^{N} \sum_{l=1}^{m} \left(t_{s,l} - y_{s,l}^{(k+1)} \right)^2$$

gdje je $t_{s,l}$ očekivana vrijednost izlaznoga neurona l za uzorak s, $y_{s,l}^{(k+1)}$ je stvarna vrijednost neurona l za uzorak s, N je broj uzoraka nad kojima se provodi učenje

[1] Optimizacijski algoritam prvoga reda. Poboljšava trenutni vektor rješenja tako da ga umanji za gradijent pomnožen s nekim pozitivnim korekcijskim faktorom

i m je broj izlaznih neurona. Vektor svih težina predstavljen je kao $\vec{w}$ dok par $(\vec{x}, \vec{t_s})$ predstavlja ulaze u umjetnu neuronsku mrežu i očekivane izlaze za te ulaze za uzorak s (Čupić et al., 2013).

To postavlja ograničenje na prijenosnu funkciju umjetne neuronske mreže, tj. to znači da sve prijenosne funkcije moraju biti derivabilne nad domenom vrijednosti koje će se pojaviti u neuronskoj mreži.

Ideja je svaku težinu umanjiti za iznos parcijalne derivacije funkcije pogreške po toj težini pomnožen nekom malom pozitivnom konstantom tako da njezin utjecaj bude takav da se ukupna greška smanjuje. Formalno je to ažuriranje težine $w_{i,j}^{(k)}$, gdje (k) predstavlja k-ti sloj. Ažuriranje se tada može zapisati kao

$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \psi \cdot \frac{\partial E}{\partial w_{i,j}^{(k)}}$$

gdje je ψ upravo ta mala pozitivna konstanta.

Razlog postojanja male konstante je u tome što algoritam mora konvergirati, a ne divergirati. Efikasno učenje mreže svodi se na odabir dobroga parametra ψ te odabira dobre konfiguracije mreže.

Sada je moguće definirati algoritam propagiranja greške unazad čiji je pseudokod vidljivi u algoritmu 1.

Algoritam 1 Algoritam propagiranja greške unazad

```
A – dvodimenzionalno polje sa značajkama za pojedini uzorak
B – dvodimenzionalno polje sa očekivanim izlazima za pojedini uzorak
C – broj uzoraka
N – mreža koja se uči
K – maksimalan broj epoha
eps – minimalna željena greška
iter := 0 – trenutna epoha
ponavljaj
  za (i := 0; i < C; i++) radi
    akumulirajGradijente(A[i], B[i], N)
  kraj
  ažurirajTežine(N)
  e := izračunajUkupnuPogrešku(A, B, N)
  iter++
dok (iter ≤ K ∧ e > eps)
```

Kako se računaju parcijalne derivacije također je vidljivo u dodatku A, dok je način ažuriranja težina opisan u algoritmu 1. Bitno je za napomenuti da se procjene gradijenta akumuliraju za sve uzorke (svaki je uzorak umjetnoj neuronskoj mreži predočen jednom tijekom jedne epohe učenja) i taj se način rada algoritma zove grupno učenje (engl. *batch learning*). Postoji i izvedba pojedinačnog[2] učenja (engl. *online learning*). To je način rada koji ovdje neće biti razmatran, no često pokazuje bolje rezultate od grupnoga učenja.

4.2. Algoritam rprop

Pokazalo se kako algoritam rprop često nudi bržu i bolju konvergenciju ka manjim pogreškama. Sve je isto kao i kod algoritma propagiranja greške unatrag s grupnim učenjem, osim što se težine ažuriraju na drugi način. Naime, ovdje se vrijednost gradijenta koristi samo kao informacija u kojem smjeru treba ići. Svaka težina ima svoje vlastite korekcijske faktore i zbog toga je konvergencija često puno bolja i brža (Riedmiller, 1994).

Algoritam je definiran sljedećim parametrima:

$\triangle_0$ - inicijalni iznos korekcije

$\triangle_{max}$ - maksimalan iznos korekcije

$\triangle_{min}$ - minimalan iznos korekcije

κ^+ - uvećanje iznosa korekcije

κ^- - umanjenjenje iznosa korekcije

Način ažuriranja vidljiv je u algoritmu 2 uz početne parametre $\frac{\partial E}{\partial w_{i,j}}(-1) = 0$ i $\triangle_{i,j}(-1) = \triangle_0$.

[2]Težine se ažuriraju nakon svakog predočavanja uzorka a ne nakon što su svi uzorci predočeni

Algoritam 2 Algoritam rprop za ažuriranje težina

$\triangle_{i,j}(t-1)$ – vrijednost korekcijskog faktora u prethodnoj epohi

if $\frac{\partial E}{\partial w_{i,j}}(t-1) \cdot \frac{\partial E}{\partial w_{i,j}}(t) > 0$ **then**

$\triangle_{i,j}(t) = min(\kappa^+ \cdot \triangle_{i,j}(t-1), \triangle_{max})$

else if $\frac{\partial E}{\partial w_{i,j}}(t-1) \cdot \frac{\partial E}{\partial w_{i,j}}(t) < 0$ **then**

$\triangle_{i,j}(t) = max(\kappa^- \cdot \triangle_{i,j}(t-1), \triangle_{min})$

else

$\triangle_{i,j}(t) = \triangle_{i,j}(t-1)$

end if

$w_{i,j}(t+1) = w_{i,j}(t) - sgn(\frac{\partial E}{\partial w_{i,j}}(t)) \cdot \triangle_{i,j}(t)$

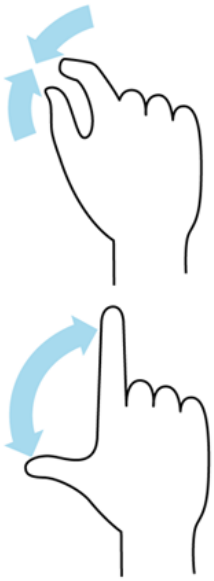
5. Odabir arhitekture umjetne neuronske mreže i pripremanje uzoraka

Pogleda li se neuronsku mrežu sa slike 3.2 vidljivo je da ona prima jednodimenzionalno polje značajki koje su statične, tj. ne ovise o vremenu. S druge strane, problem prepoznavanja gesti sadrži vremensku komponentu i potrebno je naći način da mreža uči rješavati i takve probleme.

Postoje arhitekture umjetnih neuronskih mreža s povratnim vezama koje se koriste kod rješavanja problema koje uključuju vremensku komponentu jer one prirodno iziskuju vremenski promjenjivo ponašanje. Ipak imaju nekih poteškoća poput traženja derivacija izlaza pojedinoga neurona budući da u sam izračun izlaza ulazi i vrijednost izlaza tog neurona u prethodnoj epohi te je stoga učenje takvih mreža puno kompleksnije od učenja mreža bez povratnih veza.

Ideja je koristiti jednostavnu unaprijednu slojevitu potpuno povezanu mrežu uz mogućnost učenja kompleksnih uzoraka poput onih ovisnih o vremenu. Netko bi pomislio da je moguće uzeti konačnu putanju koju stvori vrh prsta (*trag*) kod praćene geste i pokušati prepoznati tu sliku. Problem prepoznavanja kompleksnih uzoraka na slikama poprilično je dobro riješen uz učinkovitost čak od 99.65% koristeći neuronsku mrežu od 7 slojeva dimenzija $784 \times 2500 \times 2000 \times 1500 \times 1000 \times 500 \times 10$ (LeCun et al.).

Pokazuje se da taj način ipak nije dobar, a evo i jedan primjer koji to ilustrira. Neka postoje dvije geste koje su različite no mogu generirati identičan trag praćenja po svom završetku. Primjer takve geste vidljiv je na slici 5.1. Vidljivo je kako su to dvije potpuno različite geste no u konačnici će rezultirati istom statičnom slikom. Pusti li se neuronska mreža da prepoznaje takve geste, ona će reći da je to jedna od te dvije geste, no neće znati točno odrediti koja jer joj nedostaje informacija iz vremenske domene, tj. nedostaje jedna dimenzija značajki.



Slika 5.1: Dvije geste koje rezultiraju istim tragom

Ovdje se dolazi do problema smanjenja dimenzionalnosti uz očuvanje informacije.

5.1. Obrada značajki za učenje neuronske mreže

Budući da u računalu nije moguće pohraniti trag geste u beskonačnoj preciznosti, podatci su uzorkovani već od strane računala. Koordinata miša očitana je svakih 50 ms. Tako jedan podatak (uzorak) za učenje predstavljaja jedan očitani trag geste urađene mišem. Trag se sastoji od parova (D_i, t_i) gdje D_i predstavlja koordinatu (dvodimenzionalni vektor) i-te očitane točke, a t_i vremenski trenutak u kojem je točka očitana. Vremenski trenutak relativan je u odnosu na prvu očitanu točku, tj. prva je točka očitana u trenutku $t_0 = 0$.

Iz tog je skupa uzoraka potrebno generirati značajke koje opisuju uzorak i koje će se koristiti za učenje. Jedan je od načina provesti novo uzorkovanje nad svakim tragom. Uzorkovanjem će se također provesti i normalizacija[1] skupa uzoraka. Normalizacija je potrebna budući da ne traje svaka gesta jednako dugo. Tako će neki tragovi imati više dok će neki imati manje parova očitanih točaka i vremenskih trenutaka.

Neka gesta traje 600 ms i neka se želi tu gestu normalizirati na 10 točaka. To znači da je svakih 60 ms potrebno očitati vrijednosti koordinata. Izvorni su podatci uzorkovani svakih 50 ms. Jedan je od načina uzeti vrijednosti u najbližem

[1]Skaliranje podataka na određeni interval tako da su sve vrijednosti iz novog intervala

vremenskom trenutku što bi u ovom slučaju bile vrijednosti očitane u pedesetoj milisekundi. Drugi je način provoditi interpolaciju[2], što daje puno bolje rezultate, stoga se u radu koristi jednostavna linearna interpolacija.

5.1.1. Koordinatno uzorkovanje

U trenutku kada postoje uzorci u diskretnim intervalima, potrebno ih je nekako prosljediti u umjetnu neuronsku mrežu. Najjednostavniji je način imati $2 \cdot d$ ulaznih neurona, po dva za svaki uzorak. Neuroni bi u paru predstavljali x i y koordinatu za d diskretnih trenutaka.

5.1.2. Mrežasto uzorkovanje

Druga je ideja, koja proizlazi iz načina kako se matrice spremaju u memoriju računala, reducirati problem za jednu dimenziju i koristiti samo d ulaznih neurona. Slika na ekranu nije ništa drugo nego matrica D dimenzija $w \times h$, gdje je w broj točkica u širinu i h broj točkica u visinu. Poznato je da je memorija u računalu jednodimenzionalna, no računalo može pohraniti dvodimenzionalno polje. Način na koji to radi jest da pohranjuje redove jedan iza drugoga. Tako je primjerice redni broj elementa $D_{1,2} = 1 \cdot w + 2$ (brojanje je 0-indeksirano[3]).

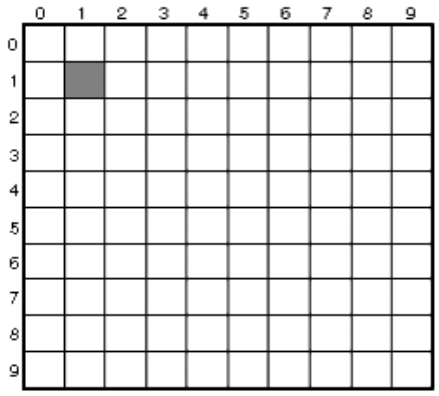
Isti se algoritam može iskoristiti za uzorkovanje. Izvorne su slike dimenzija 250×250, koje se prvo reduciraju na dimenziju 10×10. To znači da će raspon rednih brojeva elemenata biti od 0 do 99. Neka je skaliranjem dobiven element na poziciji $D_{1,1}$ kao što je to vidljivo na slici 5.2. Pretvaranjem pozicije u redni broj dobije se da je redni broj tog elementa 11.

Ponovi li se to za svih d parova koordinata iz d diskretnih trenutaka uzorkovanja problem je reduciran za jednu dimenziju i sada je dovoljno samo d ulaznih neurona.

Problem do kojega se nailazi jest slučaj za bliske koordinate. Vidljivo je da je točka $D_{0,1}$ vrlo bliska točci $D_{1,1}$, no provedbom mrežastog uzorkovanja dobivaju se vrijednosti dosta odmaknute jedna od druge.

[2]Metoda dobivanja novih točaka iz skupa postojećih diskretnih točaka

[3]Indeksiranje kreće od vrijednosti 0



Slika 5.2: Skalirani koordinatni sustav

5.1.3. Uzorkovanje sin-cos

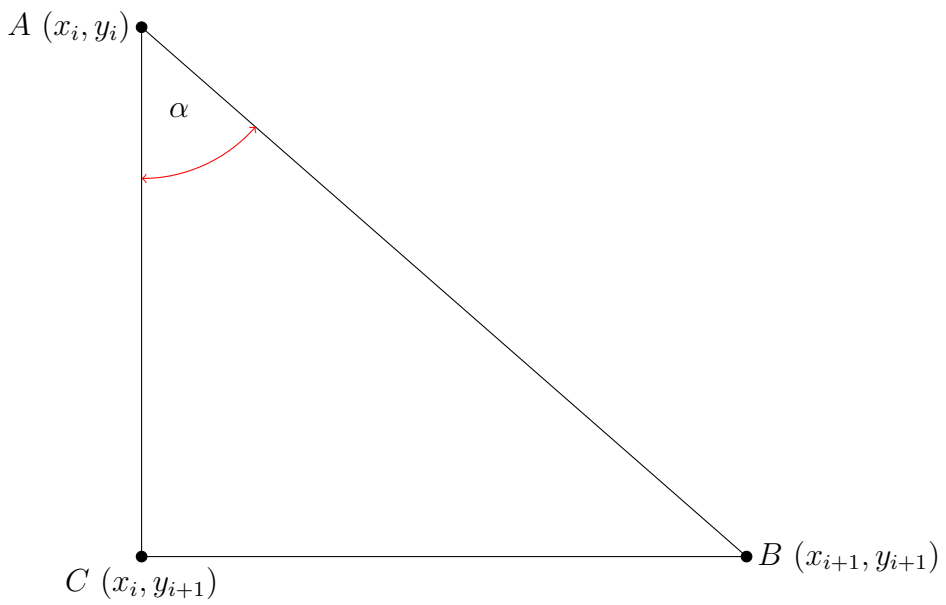
Mrežasto uzorkovanje ima jedan dodatan problem koji se može ilustrirati na sljedećem primjeru. Neka se prepoznaje trag koji ostavlja gesta crtanja simbola 0. Isto je krene li se simbol crtati od vrha ili uz neki odmak. Prethodno uzorkovanje te uzorke tretira kao potpuno različite. Tako se primjerice za istu gestu i različite početne točke može dobiti slijed točaka $(14, 37, \ldots, 23)$, dok u drugom slučaju točke zarotirane za 3 elementa u desno.

Potrebno je promijeniti način kako se generiraju značajke uzorka i. Jedan je način definirati tri točke $A = (x_i, y_i)$, $B = (x_{i+1}, y_{i+1})$ i $C = (x_i, y_{i+1})$ te trokut ABC s pravim kutem u vrhu C. Primjer takvog trokuta vidljiv je na slici 5.3. Sada se kao značajke mogu uzeti sinus i kosinus kuta α. Razlog zašto su uzeti i sinus i kosinus kuta jest to što samo jedna trigonometrijska vrijednost ne daje informaciju za sva četiri kvadranta koordinatnoga sustava. Tako primjerice odlasci iz točke $(0, 0)$ u točku $(1, -1)$ ili točku $(1, 1)$ imaju istu vrijednost kosinusa kuta.

Kod uzorkovanja sin-cos i d diskretnih trenutaka potrebno je $2 \cdot (d-1)$ uzoraka zbog toga što se za $d - 1$ očitanih točaka računaju i sinus i kosinus kuta α.

5.2. Kodiranje razreda

Zadaća neuronske mreže jest da temeljem značajki koje primi na ulaznim neuronima vrati brojčane vrijednosti koje su dobivenem propagiranjem tih značajki kroz mrežu. Razredi su predstavljeni simbolom dok neuronska mreža obrađuje i generira brojeve te je stoga potrebno obaviti *kodiranje*. To je postupak u ko-



Slika 5.3: Primjer uzorkovanja sin-cos i kuta α

jem se svakom razredu pridjeljuje jednoznačni[4] kôd (uređena N-torka brojeva). Obrnuti postupak kojim se iz kôda dobije razred kojem taj kôd pripada zove se dekodiranje.

Kodiranjem se određuje koji su očekivani izlazi neuronske mreže za pojedine vrijednosti ulaznih značajki te su uz razne konfiguracije mreža razmotrena dva načina kodiranja koji su objašnjeni u nastavku.

5.2.1. Kodiranje jedan-od-N

Kodiranje jedan-od-N (engl. *One-of-N*) predstavlja način kodiranja gdje je svaki razred predstavljen uređenom N-torkom brojeva u kojoj samo jedan broj ima vrijednost 1 dok svi ostali vrijednost 0. Na taj se način kodira svih N uzoraka i dobije se da je svakom razredu pridjeljen neki kôd. Tako bi primjerice simbol α bio kodiran kao $(0, 0, 0, 1)$, β kao $(0, 0, 1, 0)$, γ kao $(0, 1, 0, 0)$ i Ω kao $(1, 0, 0, 0)$.

Neuronska mreža sada treba imati N izlaznih neurona budući da svaki neuron odgovara jednome elementu u N-torci. Postupak dekodiranja radi se tako da najveća vrijednost od izlaznih vrijednosti izlaznih neurona, koje općenito mogu biti realni brojevi iz intervala $[0, 1]$ postaje 1, a sve ostale 0.

Neka su izlazi neurona $(0.011, 0.188, 0.988, 0.451)$, tada bi se ti izlazi dekodirali

[4]Jedna N-torka može predstavljati samo jedan simbol i jedan simbol smije biti predstavljen samo jednom N-torkom

u simbol β jer su transformirane izlazne vrijednosti $(0, 0, 1, 0)$.

5.2.2. Ekvilateralno kodiranje

Glavni problem kodiranja jedan-od-N jest način na koji se rasprostire pogreška. Neka se radi klasificiranje simbola α koji je kodiran s $(0, 0, 0, 1)$ i neka je kao izlaz dobivena N-torka $(0.11, 0.188, 0.988, 0.451)$. Promatranjem izlaza neuronske mreže vidljivo je da zadnja dva neurona imaju krive vrijednosti i prilikom učenja ta su dva neurona najviše "krivi" za pogrešnu klasifikaciju. Pogreška po neuronima koja je napravljena iznosi:

$$(0, 0, 0, 1) - (0.11, 0.188, 0.988, 0.451) = (-0.11, -0.188, -0.988, 0.549)$$

i vidljivo je će zadnja dva neurona imati veći utjecaj na ažuriranje težina budući da im je pogreška veća od pogreške prva dva neurona. Na taj način sporije će se ažurirati preostale težine koje nisu imale toliki utjecaj na tu pogrešku.

Ideja je stoga napraviti takvo kodiranje koje će ravnomjernije rasporediti očekivane vrijednosti izlaza i time pokušati raspršiti pogreške neurona. Jedan je od načina naći takve vrijednosti da svaki razred ima jednaku euklidsku udaljenost kôda do kôdova preostalih razreda (Heaton, 2011).

Za dvije točke, $\mathbf{p} = (p_1, p_2, \ldots, p_N)$ i $\mathbf{q} = (q_1, q_2, \ldots, q_N)$, u N dimenzijskom prostoru, euklidska udaljenost l između te dvije točke definirana je kao

$$\begin{aligned} l(\mathbf{p}, \mathbf{q}) = l(\mathbf{q}, \mathbf{p}) &= \sqrt{(\mathbf{p} - \mathbf{q}) \cdot (\mathbf{p} - \mathbf{q})} \\ &= \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_N - p_N)^2} \\ &= \sqrt{\sum_{i=1}^{N} (q_i - p_i)^2} \end{aligned}$$

Kod ovog načina kodiranja, za N različitih razreda potrebno je $N-1$ izlaznih neurona budući da se N točaka može u minimalno $N-1$ dimenzijskom prostoru razmjestiti na jednake udaljenosti. Za 2 točke to je linija duljine l i ona je u jednodimenzionalnom prostoru, za 3 točke to je jednakostranični trokut u dvodimenzionalnom prostoru sa stranicama duljine l i tako dalje.

Tablica kodiranja 5.1 prikazuje razrede i pridružene kôdove na primjeru 4 razreda.

Tablica 5.1: Udjeli uzoraka

Simbol	Kôd
α	$(0, 0, 0)$
β	$(1, 0, 0)$
γ	$(0.5, 0.866, 0)$
Ω	$(0.5, 0.289, 0.816)$

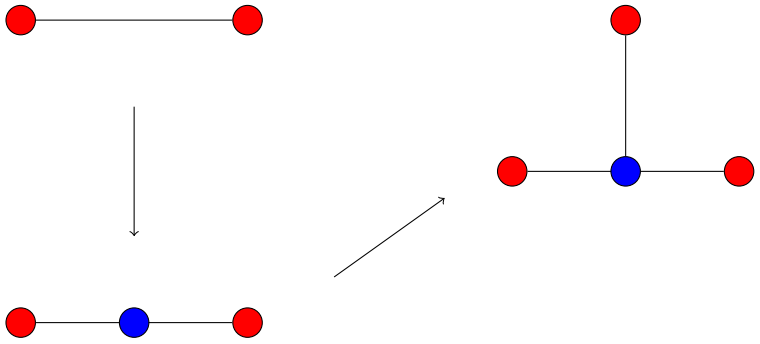
Algoritam 3 prikazuje algoritam provođenja kodiranja i biti će objašnjen na primjeru $N = 3$ kako bi bio lakše shvatljiv. Također, neka je željena udaljenost l jednaka 1. Već inicijalno stanje algoritma kaže da su prva dva kôda udaljena za vrijednost 1 i njihove vrijednosti su $(0, 0)$ i $(1, 0)$. Preostali kôd ima vrijednost $(0, 0)$. Potrebno je još dakle preurediti vrijednost zadnjeg elementa preostalog kôda tako da udaljenost do svih preostalih već izračunatih kôdova bude 1.

Prvo se računaju centri svih točaka po elementima i dobije se N-torka vrijednosti $(0.5, 0)$.

Algoritam 3 Algoritam ekvilateralnoga kodiranja

```
l - željena euklidska udaljenost između kôdova razreda
kodovi - polje dimenzija N × (N − 1) za pohranu kôdova
kodovi[1][0] = l
za (i = 2; i < N; i++) radi
    centar = kodovi[i] – trenutna točka koju računamo
    za (j = 0; j < i - 1; j++) – računanje centra za trenutne uzorke radi
        za (k = 0; k < i; k++) radi
            centar[j] = centar[j] + kodovi[k][j]
        kraj
        centar[j] = centar[j] / i
    kraj
    d=udaljenost(centar, map[0]) – računanje trenutne udaljenosti
    centar[i - 1] = √(l² − d²) – dodavanje vrijednosti u element koji nedostaje tako
    da udaljnost bude l
kraj
```

Nakon toga računa se udaljenost te N-torke do nekog drugog, već ranije izračunatog kôda (najlakše je uzeti prvi kôd). Dobije se da je udaljenost 0.5. Sada se pomoću pitagorinog poučka može preurediti zadnji element preostalog kôda tako



Slika 5.4: Konstruiranje treće kodne točke koristeći pravi kut i dvije poznate točke

da se zadnji element postavi na vrijednost $\sqrt{1-0.5^2}$ i time se dobiju konačne vrijednosti $(0.5, \frac{\sqrt{3}}{2})$.

Vizualno se to može zamisliti kao konstruiranje jednakostraničnoga trokuta iz poznata dva kôda te konstruiranje jednakokračne piramide za poznata tri kôda. Na slici 5.4 vidljivo je kako se konstruira treći kôd iz dva poznata kôda.

Također, valja napomenuti da nema smisla raditi ekvilateralno kodiranje za manje od 3 razreda. Za dva razreda dovoljno je kodiranje jedan-od-N budući da će pogreška sigurno bit raspršena po svim neuronima.

Dekodiranje će se provoditi na način da izlazne vrijednosti neurona predstavljaju razred kojem je euklidska udaljenost kodiranih vrijednosti do vrijednosti dobivenih na izlazima iz mreže najmanja.

5.3. Normalizacija podataka

Pogledaju li se izlazne vrijednosti mrežastoga uzorkovanja, vidljivo je da je raspon tih brojeva velik i teže je učiti neuronsku mrežu jer ulazi imaju puno veći utjecaj na ukupnu sumu. Želja je stoga te vrijednosti svesti na neki manji interval kako bi se postigli manji i ravnomjerniji utjecaji značajki. Taj se postupak zove normaliziranje. Uzorkovanje sin-cos nije potrebno normalizirati budući da su kodomene funkcija sinus i kosinus intervali $[-1, 1]$ što je već poprilično malo, dok će mrežasto uzorkovanje biti normalizirano na interval $[0, 1]$.

Normalizacija se provodi tako da se nađu maksimalna i minimalna vrijednost koju je moguće poprimiti. U primjeru mrežastoga uzorkovanja minimalna vrijednost je 0 dok je maksimalna 99. Želja je interval duljine 99 svesti na interval duljine 1 i to je moguće postići jednostavnim omjerima gdje vrijedi $\frac{novaVrijednost}{1} = \frac{staraVrijednost}{99}$ iz čega proizlazi da je $novaVrijednost = \frac{staraVrijednost}{99}$.

6. Ubrzavanje postupka učenja neuronske mreže

Jedna od odlika neuronskih mreža jest velika paralelnost u samoj arhitekturi mreže. Dok se vrednovanje mreže može jako lijepo implementirati paralelno, ispostavlja se da učenje neuronske mreže i nije toliko paralelno koliko bi se očekivalo.

6.1. Paralelizacija na razini neurona

Paralelizacija evaluacije je jednostavna. Neka je na raspolaganju centralna procesna jedinica (CPU[1]) s 4 jezgre. Moguće je imati ukupno 4 dretve koje će predstavljati jedinicu posla od čega će jedna obrađivati jedan neuron i tako će se redom davati poslovi dretvama. Nakon što se cijeli jedan sloj evaluira, sinkronizira se sve kako bi algoritam bio siguran da idući sloj ima sve potrebne podatke. To proizlazi iz činjenice da je to unaprijedna slojevita mreža što znači da svaki sloj ovisi samo o prethodnom sloju.

Kako sinkronizacije i stavljanje poslova u red ipak iziskuju neko procesno vrijeme koje nije zanemarivo potrebno je imati dovoljan broj neurona unutar jednog sloja kako bi vrijeme procesiranja bilo veće od vremena sinkronizacije i stavljanja poslova u red. Što je veći broj konekcija, to ubrzanje više teži prema broju jezgri procesne jedinice budući da ima sve više posla za obaviti.

Paralelizacija učenja na razini neurona već je malo teža budući da bi bilo potrebno imati puno više sinkronizacija. Razlog tome je što učenje poprilično intenzivno koristi zapisivanje u memoriju. I dok je evaluacija više orijentirana na čitanje, a čitanje s iste memorijske lokacije dvije iste dretve u istom trenutku neće narušiti stanje programa, upisivanje u memoriju u istu lokaciju u istom trenutku

[1](engl. *Central processing unit*) – predstavlja centralnu procesnu jedinicu računala koja je zapravo "mozak" cijelog računala

može dovesti do nekonzistentnoga stanja programa. Doskočiti tome moguće je tako da se napravi puno veće polje za spremanje podataka te na samom kraju, nakon što su sve operacije izvršene, to se polje sumira i dobije se konačan rezultat.

6.2. Paralelizacija na razini mreže

Drugi način paralelizacije, koji je implementiran kroz ovaj rad, jest paralelizacija na razini mreži. U stvarnosti, broj uzoraka koji su na raspolaganju za učenje često je puno veći od samih dimenzija mreže, tj. mreža često nije previše kompleksna.

Kod ove paralelizacije jedinica posla je evaluacija i računanje procjene gradijenta za jedan ulazni primjerak. Kao i kod prethodnoga načina paralelizacije i dalje je potrebno imati sinkronizacije kod sumiranja procjene gradijenta, što se opet može riješiti većim poljem gdje će svaka jedinica posla spremati svoju procjenu gradijenta. Nakon što su svi primjerci predočeni, dolazi do sumiranja svih procjena gradijenta i ažuriranja težina te to završava jednu epohu.

6.3. Grafička kartica

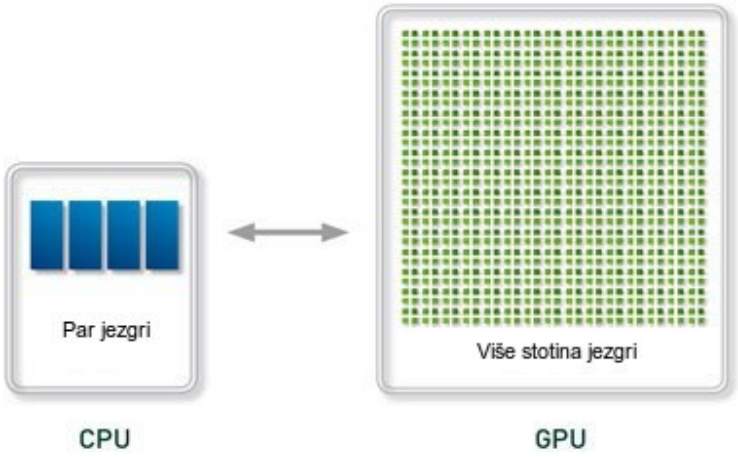
Kako je učenje implementirano na grafičkoj kartici i potrebno je više vremena za dodavanje posla u red poslova nego što je to inače potrebno za implementaciju koja koristi centralnu procesnu jedinicu, odabran je drugi način paralelizacije.

Grafička kartica ima svoju vlastitu memoriju, dok centralna procesna jedinica komunicira s glavnom memorijom računala. Obje memorije su memorije s nasumičnim pristupom (RAM[2]). Prijenos između tih dviju memorija kao i pristup samoj memoriji vremenski je skupa operacija u odnosu na izvođenje naredbi koje manipuliraju nad tim podatcima.

6.3.1. Razlike u arhitekturi centralnog i grafičkog procesora

Primarna namjena centralnog procesora jest izvršavanje operacija široke primjene. Jedna od čestih naredbi koje centralni procesor izvršava jest naredba skoka te stoga postoji jedan poseban sklop koji je zadužen za predviđanje hoće li do skoka doći ili ne. Razlog tome je taj što se sljedeća naredba mora dohvatiti iz

[2](engl. *Random access memory*) – jako brza memorija u kojoj se nalaze podatci za vrijeme izvršavanja programa



Slika 6.1: Usporedba arhitekture jezgri grafičkog i centralnog procesora

glavne memorije računala i u slučaju da se dohvati kriva naredba, potrebno je čekati dok se ne dohvati nova naredba.

Također, kako se dobar dio naredbi poput naredbi za aritmetičko-logičku jedinicu (ALU[3]), naredbi za dohvaćanje iz memorije i slično, može paralelizirati, centralni procesor ima razvijen veliki sustav cjevovoda[4] koji pružaju podršku upravo za to (Palacios i Triska, 2011).

S druge strane, primarna namjena grafičkog procesora jest obrađivanje grafičkih elemenata. Kako su te operacije često jednostavne i neovisne jedna o drugoj, to pruža mogućnost za velike paralelizacije (način paralelizacije SIMT[5]). Tako su procesni elementi grafičke procesne jedinice vrlo jednostavne građe. Zbog toga nisu tako dobri u grananjima kao centralni procesor, no ima ih velik broj (reda veličine 10^3 u skupljim grafičkim karticama) i time se postiže ubrzanje.

Sama frekvencija na kojima rade jezgre grafičkog procesora otprilike je $2 - 3$ puta manja od onih na kojoj rade jezgre centralnoga procesora. No, zbog velikoga broja procesnih elemenata i dalje je moguće postići veliko ubrzanje. Primjer usporedbe arhitekture jezgri grafičkoga i centralnoga procesora vidljiv je na slici 6.1.

Grafički procesor ima velik broj dretvi koje mogu paralelno postojati. Dok

[3](engl. *Arithmetic logic unit*) – sklop u procesoru zaslužan za obavljanje matematičkih i logičkih operacija

[4](engl. *Pipeline*) – skup procesnih elemenata povezanih u seriju koji omogućavaju paralelizaciju

[5](engl. *Single instruction multiple thread*) – ista operacija izvodi se nad više različitih podataka u istom vremenskom trenutku na više različitih dretvi

jedna čeka pristup memoriji uzima se neka od sljedećih koja se može izvršavati. Na taj se način doskače problemu latencije memorije. Centralni procesor pokušava taj problem riješiti s više razina pričuvne memorije (najčešće su $L1$ i $L2$). Te su pričuvne memorije izrazito brze, no malih kapaciteta u usporedbi s glavnom memorijom računala. U slučaju da traženog podatka nema u pričuvnoj memoriji, dretva mora stati dok se podatak ne učita u pričuvnu memoriju.

7. Biblioteka OpenCL

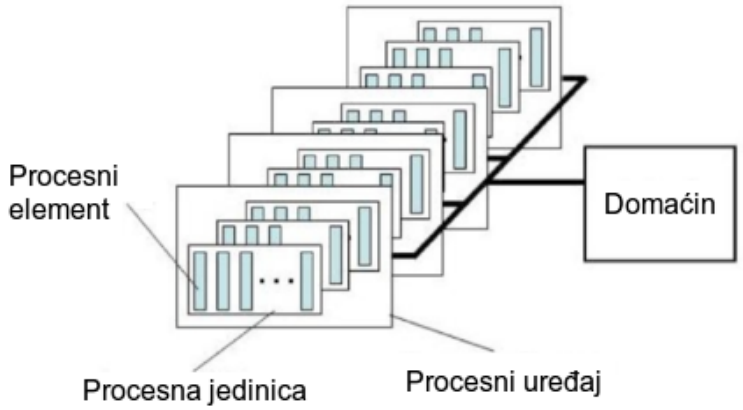
Izravno programiranje grafičke kartice nije jednostavno, stoga su razvijene razne biblioteke koje taj problem olakšavaju. Jedna je od takvih biblioteka i OpenCL koja pruža mogućnost pisanje programa koji se mogu izvršavati na heterogenim sustavima (CPU, GPU, ...). U sklopu razvojnoga okruženja dolazi i programski jezik baziran na standardu C99,[1] koji služi za pisanje OpenCL jezgri (engl. *kernel*). Jezgre predstavljaju programski kôd koji će se izvršavati. Jedna od glavnih primjena je korištenje grafičke kartice za programiranje opće namjene, a ne samo za procesiranje grafičkih elemenata (ope, 2010).

7.1. Organizacijski model

Bilo da se programira za grafički procesor, centralni procesor ili nešto drugo, model je isti. OpenCL apstrahira pristup memoriji i općenito uređaju za koji se programira te na taj način nudi unificirano sučelje za komunikaciju. Uređaj koji pokreće izvorni kod i upravlja ostalim uređajima zove se domaćin (engl. *host*). Zadaća domaćina jest upravljanje memorijom, zadavanje posla ostalim uređajima poput grafičkog procesora i određivanje koliko će se radnih stavki stvoriti. Domaćin šalje OpenCL-jezgru na izvršavanje i sama jezgra predstavlja najmanju jedinicu paralelizacije te se izvršava unutar radne stavke (engl. *work item*), tj. radna stavka je primjerak jezgre.

Svaka radna stavka ima svoj identifikator koji se može dohvatiti iz jezgre i predstavlja jednu dretvu, najmanju jedinicu posla koja se može izvršavati. Te radne stavke dalje su organizirane u radne grupe (engl. *work group*) kako bi se omogućila međusobna komunikacija između radnih grupa. Organizacija je moguća u 1, 2 ili 3 dimenzije. Radne grupe dalje su organizirane u ND-raspon (engl. *N-Dimensional range*), koji predstavlja najveću oragnizacijsku strukturu i također može biti u 1, 2 ili 3 dimenzije (Munshi, 2011).

[1] Standard jezika C



Slika 7.1: Model platforme unutar biblioteke OpenCL

Nadalje, jedan domaćin može komunicirati s više uređaja. Jedan uređaj unutar sebe ima barem jednu procesnu jedinicu, a svaka procesna jedinica ima barem jedan procesni element, kako je to prikazano na slici 7.1.

Organizacija u radne stavke je logička, dok je organizacija u procesne elemente fizička i u jednom trenutku na jednom procesnom elementu može se izvršavati samo jedna radna stavka. Broj radnih stavki može biti puno veći od ukupnoga broja procesnih elemenata.

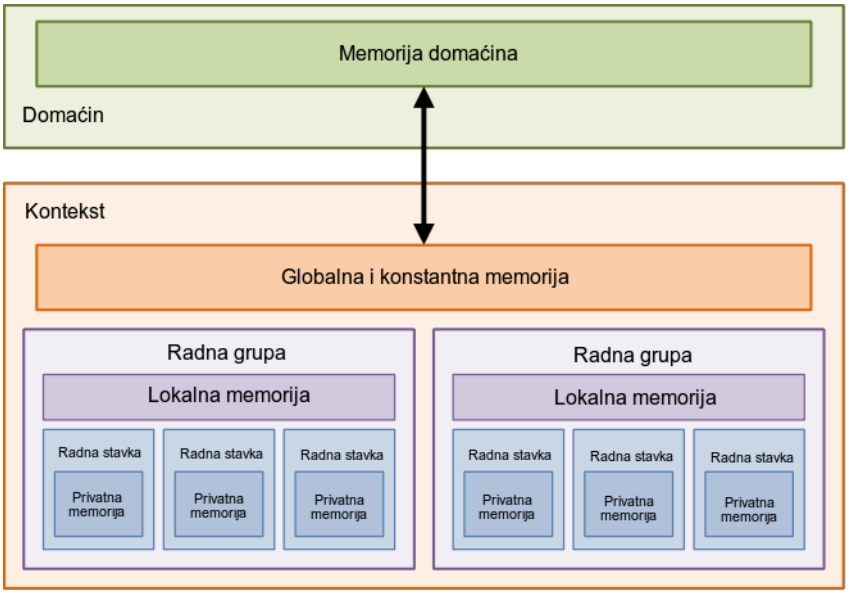
Na primjeru grafičke kartice koja ima 20 procesnih jedinica, od kojih svaka ima 80 procesnih elemenata (jedna procesna jedinica ima 16 jezgri i svaka jezgra ima 5 procesnih elemenata), može se izračunati kako u jednom trenutku grafički procesor može obrađivati maksimalno 1600 radnih stavki paralelno.

Jedna procesna jedinica može obrađivati samo jednu radnu grupu u jednome trenutku.

7.2. Memorijski model

Memorija i red poslova koji čekaju na izvršavanje ovise o kontekstu. Unutar jednoga konteksta postoje globalni i konstantni segment memorije. Globalna memorija predstavlja najveću i najsporiju memoriju iz koje svi mogu čitati i pisati. Konstatnom segmentu memorije svi mogu pristupiti.

Nadalje, postoji i lokalna memorija koja je dostupna svim radnim stavkama unutar iste radne grupe. Ta je memorija nešto brža od globalne memorije. Na kraju postoji privatna memorija koja je poprilično mala (reda veličine $64kb$) i



Slika 7.2: Memorijski model biblioteke OpenCL

kojoj može pristupiti radna stavka koja je vlasnik te privatne memorije. Svaka radna stavka ima svoju vlastitu privatnu memoriju. Privatna se memorija koristi za spremanje međurezultata i pristup toj memoriji jako je brz. Slika 7.2 zornije prikazuje memorijski model.

Cjelokupnu memoriju, osim privatne memorije, zauzima i oslobađa domaćin. Privatna memorija zauzima se prilikom pokretanja pripadne jezgre.

7.3. Primjer korištenja biblioteke OpenCL

U nastavku je dan primjer koji korištenjem biblioteke OpenCL množi dva polja tako da se pomnože vrijednosti na istim indeksima. Implementacija biblioteke OpenCL u programskom jeziku Java koja se koristi je biblioteka JOCL verzije 0.1.9.

OpenCL-jezgru treba promatrati kao jednu jedinicu posla koja se može izvoditi paralelno. Primjerice, neka postoji kôd koji zbraja dva polja na sljedeći način:

```
private void zbroji(int[] a, int[] b, int[] c) {
    for (int i = 0; i < a.length; i++) {
        c[i] = a[i] + b[i];
    }
}
```

gdje je rezultat množenja spremljen u polju označenom varijablom `c`.

Vidljivo je kako je zapisivanje rezultata razdjeljeno po indeksima što ovaj kôd čini pogodnim za paralelnu implementaciju. Točnije,

```
c[i] = a[i] + b[i];
```

može predstavljati jednu jedinicu posla. Kako jedna OpenCL-jezgra predstavlja jednu jedinicu posla moguće je napisati OpenCL-jezgru na sljedeći način:

```
kernel void mnozi(
    global const float *a,
    global const float *b,
    global float *c
) {
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
```

gdje ključna riječ `kernel` kaže kako se radi o funkciji koja predstavlja OpenCL-jezgru. Sve funkcije koje nemaju ključnu riječ `kernel` predstavljaju standardne funkcije koje je moguće koristiti unutar same OpenCL-jezgre.

Ta jezgra prima tri parametra koji predstavljaju pokazivače na polja u globalnoj memoriji. Budući da se iz prva dva polja vrijednosti samo čitaju moguće ih je definirati kao konstantne (ključna riječ `const`) kako bi sam grafički procesor mogao optimirati pristup memoriji. Unutar jezgre potrebno je dohvatiti identifikator primjerka jezgre te se ta vrijednost koristi kao indeks za pristupanje polju.

Pisanje kôda domaćina započinje ubacivanjem potrebnih razreda biblioteke koji će se koristiti.

```
import org.jocl.*;
import static org.jocl.CL.*;
```

Nakon toga potrebno je definirati polja, popuniti ih i stvoriti pokazivače na ta polja. Kako Java nema standardne pokazivače, potrebno je naša polja enkapsulirati u objekte koji predstavljaju pokazivače.

```
float prvoPolje[] = new float[N];
float drugoPolje[] = new float[N];
float odredisnoPolje[] = new float[N];

for (int i = 0; i < N; i++) {
```

```
    prvoPolje[i] = i;
    drugoPolje[i] = i;
}

Pointer pokazivacNaPrvoPolje = Pointer.to(prvoPolje);
Pointer pokazivacNaDrugoPolje = Pointer.to(drugoPolje);
Pointer pokazivacNaOdredisnoPolje
  = Pointer.to(odredisnoPolje);
```

Programski jezik Java podržava iznimke kao način obrade grešaka te je stoga dobra praksa omogućiti prijavu iznimki u biblioteci OpenCL.

```
CL.setExceptionsEnabled(true);
```

Potom je potrebno detektirati uređaje i platforme. Prvo se dohvaća broj platformi. U većini slučajeva postoji samo jedna platforma dok u slučaju kada je na istom računalu instalirano više implementacija biblioteka OpenCL može biti i više platformi. Nakon što je poznat broj platformi, dohvaća se prva platforma. Ta se platforma dalje koristi za dohvaćanje broja uređaja tipa grafički procesor i dohvaćanje prvog takvog uređaja. Potom se taj uređaj dalje koristi za stvaranje konteksta unutar kojeg će se izvoditi OpenCL-jezgre.

```
int poljeBrojaPlatformi[] = new int[1];
clGetPlatformIDs(0, null, poljeBrojaPlatformi);
int brojPlatformi = poljeBrojaPlatformi[0];

cl_platform_id poljePlatformi[]
  = new cl_platform_id[brojPlatformi];
clGetPlatformIDs(
  poljePlatformi.length, poljePlatformi, null
);
cl_platform_id platforma = poljePlatformi[0];

int poljeBrojaUredaja[] = new int[1];
clGetDeviceIDs(
    platforma, CL_DEVICE_TYPE_GPU, 0,
    null, poljeBrojaUredaja
);
int brojUredaja = poljeBrojaUredaja[0];
```

```
cl_device_id poljeUredaja[]
  = new cl_device_id[brojUredaja];
clGetDeviceIDs(
    platforma, CL_DEVICE_TYPE_GPU,
    brojUredaja, poljeUredaja, null
);
cl_device_id uredaj = poljeUredaja[0];

cl_context_properties postavkeKonteksta
  = new cl_context_properties();
postavkeKonteksta.addProperty(
  CL_CONTEXT_PLATFORM, platforma
);
```

Nakon što je stvoren kontekst moguće je stvoriti red poslova i memorijske spremnike. Memorijski spremnici alociraju se na uređaju i moguće je odabrati automatsko kopiranje podataka s domaćina na uređaj. To se postiže konstantom `CL_MEM_COPY_HOST_PTR`, inače se pretpostavlja samo alociranje memorije određene veličine. Prilikom stvaranja memorijskih spremnika potrebno je i poslati broj bajtova koji određuje veličinu spremnika koji se alocira. U slučaju kada je odabrano ručno popunjavanje memorije, nije potrebno slati pokazivač na podatke kao što je to slučaj kod stvaranja spremnika za rezultate množenja.

```
cl_command_queue redPoslova = clCreateCommandQueue(
    kontekst, uredaj, 0, null
);

cl_mem memorijskiObjekti[] = new cl_mem[3];
memorijskiObjekti[0] = clCreateBuffer(
    kontekst,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    Sizeof.cl_float * N,
    pokazivacNaPrvoPolje, null
);
memorijskiObjekti[1] = clCreateBuffer(
    kontekst,
```

```
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    Sizeof.cl_float * N,
    pokazivacNaDrugoPolje, null
);
memorijskiObjekti[2] = clCreateBuffer(
    kontekst,
    CL_MEM_READ_WRITE,
    Sizeof.cl_float * N,
    null, null
);
```

Idući korak koji je potrebno napraviti je stvoriti program iz izvornog kôda. Jednom kada je program stvoren, moguće je stvoriti objekt koji predstavlja OpenCL-jezgru. Jezgri se potom postave ulazni parametri. Iz izvornog kôda OpenCL-jezgre vidljivo je kako ta jezgra prima tri parametra pa je stoga potrebno i objektu unutar domaćina pridjeliti tri parametra.

```
cl_program program = clCreateProgramWithSource(
    kontekst, 1, new String[]{
        izvorniKodOpenCLPrograma
    }, null, null
);

clBuildProgram(
    program, 0, null, null, null, null
);

cl_kernel jezgra = clCreateKernel(
    program, "mnozi", null
);

clSetKernelArg(
    jezgra, 0, Sizeof.cl_mem,
    Pointer.to(memorijskiObjekti[0])
);
clSetKernelArg(
    jezgra, 1, Sizeof.cl_mem,
```

```
        Pointer.to(memorijskiObjekti[1])
);
clSetKernelArg(
        jezgra, 2, Sizeof.cl_mem,
        Pointer.to(memorijskiObjekti[2])
);
```

Sada je još potrebno pozvati OpenCL-jezgru. Prilikom poziva potrebno je odrediti veličine radnih grupa. Za potrebe ovog primjera dovoljno je da je lokalna veličina radne grupe 1 dok je globalna veličina radne grupe veličina jednog polja koje se množi. Nakon što je poziv jezgre stavljen u red poslova potrebno je još u red poslova staviti čitanje iz odredišnog spremnika. Taj je poziv blokirajući zbog parametra kojem se kao vrijednost daje vrijednost konstante `CL_TRUE` i program domaćina će čekati dok se čitanje ne obavi u potpunosti. Nakon što je čitanje obavljeno u potpunosti, odredišno polje u memoriji domaćina sadrži rezultate izvođenja OpenCL-jezgre.

```
clEnqueueNDRangeKernel(
        redPoslova, jezgra, 1, null,
        new long[] {N}, new long[] {1}, 0, null, null
);

clEnqueueReadBuffer(
        redPoslova, memorijskiObjekti[2], CL_TRUE, 0,
        N * Sizeof.cl_float, pokazivacNaOdredisnoPolje,
        0, null, null
);
```

Nakon što je čitanje obavljeno i nema više potrebe za korištenjem resursa grafičkog procesora potrebno je sve zauzete resurse otpustiti.

```
clReleaseMemObject(memorijskiObjekti[0]);
clReleaseMemObject(memorijskiObjekti[1]);
clReleaseMemObject(memorijskiObjekti[2]);

clReleaseKernel(jezgra);
clReleaseProgram(program);
clReleaseCommandQueue(redPoslova);
clReleaseContext(kontekst);
```

Time je korištenje biblioteke OpenCL završeno i moguće je pristupiti rezultatima u varijabli `odredisnoPolje`.

8. Implementacija

Napravljene su četiri različite implementacije algoritma. Dvije implementacije koje nisu paralelizirane i dvije implementacije koje jesu paralelizirane. Paralelizacija je napravljena pomoću razvojnoga okruženja OpenCL i moguće je odabrati koristi li se kao uređaj-radnik centralni procesor ili grafički procesor. Algoritmi koji su implementirani za učenje jesu algoritam rprop i algoritam propagiranja greške unatrag. Kao prijenosna funkcija korištena je funkcija tangens hiperbolni.

Korišten je programski jezik Java i kao implementacija OpenCL standarda korištena je biblioteka JOCL verzije 0.1.9. Korištena paralelizacija je paralelizacija na razini mreže.

Postoji ukupno 7 jezgri od kojih dvije imaju isto ime i predstavljaju ažuriranje težina koje je različito ovisno o tome koja se inačica algoritma za učenje koristi.

8.1. Korištene OpenCL-jezgre

U nastavku je dan popis korištenih OpenCL jezgri kao i njihov opis.

Jezgra `evaluate` evaluira mrežu za jedan uzorak. Kako je rečeno ranije, jedna radna stavka nam predstavlja jedan primjerak mreže. Tako će za skup uzoraka veličine od 10^3 elemenata postojati ukupno 10^3 radnih stavki. Prilikom evaluiranja spremaju se izlazi svih neurona u zasebno polje kao i njihovi "sirovi" izlazi (izlazi prije primjene prijenosne funkcije). Veličina polja je $brojNeurona \times brojUzoraka$.

Jezgra `calculateLayerError` računa kolika je pogreška pojedinoga neurona za predočeni uzorak. Pogreška svakoga neurona također se sprema u zasebno polje dimenzija $brojNeurona \times brojUzoraka$.

Jezgra `calculateGradients` temeljem izlaza i grešaka neurona računa procjene gradijente za jedan primjerak mrežu. Rezultate sprema u zasebno polje dimenzija $brojTezina \times brojUzoraka$.

Jezgra `sumGradients` koja nakon što su sve procjene gradijenata izračunate sumira te procjene gradijenta za iste težine i sprema ih u zasebno polje dimenzija $brojTezina$. Ova je jezgra paralelizirana po broju težina što znači da ako postoji 100 težina postojat će i 100 radnih stavki.

Jezgra `sumErrors` temeljem izlaza neuronske mreze računa vrijednost funkcije pogreške koja je definirana ranije (vidi izraz 3.1).

Jezgra `updateWeights` jest jezgra koja je specifična algoritmu koji se koristi i temeljem sume procjena gradijenta ažurira težine. Ova jezgra također je paralelizirana po broju težina, a ne po broju uzoraka.

8.2. Algoritam upravljanja OpenCL-jezgrama

Dok je serijska implementacija jednostavna i ne zahtjeva previše pažnje, implementacija učenja koja koristi OpenCL ponešto je kompleksnija i u osnovnom obliku prikazuje ju algoritam 4.

Algoritam 4 Implementacija učenja pomoću OpenCL biblioteke

```
C - maksimalni broj epoha
odaberiUređaj()
stvoriKontekst()
inicijalizirajMemorijskeObjekte()
prevediKod()
inicijalizirajJezgre()
pozoviJezgru(evaluate)
za (i := 0; i < C; i++) radi
    pozoviJezgru(calculateLayerError)
    pozoviJezgru(calculateGradients)
    pozoviJezgru(sumGradients)
    pozoviJezgru(updateWeights)
    pozoviJezgru(evaluate)
kraj
pozoviJezgru(sumErrors)
pogreška = učitajGrešku()
vrati pogreška
```

8.3. Programska podrška

Implementirana su dva odvojena računalna programa, jedan za učenje neuronske mreže i drugi za demonstraciju naučene neuronske mreže. Objašnjenja konfiguracijskih parametara moguće je vidjeti u dodatku B.

8.3.1. Program za učenje neuronske mreže

Implementacija programa za učenje neuronske mreže napravljena je tako da ne ovisi previše o specifičnom problemu koji se rješava, već je moguće proizvoljno konfigurirati sustav da uči kompleksnije mreže. Tako je moguće promijeniti dimenzije mreže, izmijeniti skup podataka nad kojim se uči i izmijeniti parametre algoritama učenja te sam algoritam učenja. Program je napisan kao konzolna aplikacija.

Neke segmente poput načina kodiranja (ekvilateralno) i prijenosnih funkcija (tangens hiperbolni) nije moguće mijenjati.

Potrebe za grafičkim sučeljem nema budući da zbog implementacije pomoću bilbioteke OpenCL nije moguće prikazati neke zanimljivije podatke poput kretanje pogreške nad skupom za učenje kroz vrijeme.

Program se pokreće sljedećom naredbom u konzoli:

```
java -cp Diplomski.jar com.msuflaj.Train
```

te je moguće pozivanje programa s opcionalnim argumentom `c` koji određuje putanju do konfiguracijske datoteke (vidi dodatak B) i tada je naredba za pokretanje:

```
java -cp Diplomski.jar com.msuflaj.Train -c putanja/do/config.properties.
```

Pretpostavljena je vrijednost `conf.properties`.

Jednom kada sustav završi s učenjem, vrijednosti težina spremljene su u datoteci koja je definirana u konfiguraciji te je potom moguće pokrenuti demonstracijski program.

8.3.2. Demonstracijski program

Pokretanjem demonstracijskog programa moguće je mišem unositi geste koje će se iscrtati na ekranu, a program će na dnu ispisati u koji je razred klasificirao unijetu gestu mišem.

Konfiguracijska datoteka također je objašnjena u dodatku B, a sam program pokreće se koristeći naredbu:

Slika 8.1: Demonstracijski program

`java -cp Diplomski.jar com.msuflaj.Display`
u konzoli te je također program moguće pozvati s opcionalnim argumentom `c` koji određuje putanju do konfiguracijske datoteke i tada je naredba za pokretanje:
`java -cp Diplomski.jar com.msuflaj.Display -c putanja/do/config.properties`.
Pretpostavljena je vrijednost `conf.properties`.

Ovaj program ponešto je ograničeniji po pitanju mogućnosti pa su tako moguća samo dva uzorkovanja, sin-cos i mrežasto. Primjer demonstracijskoga programa vidljiv je na slici 8.1.

8.3.3. Demonstracijska internet aplikacija

Demonstracijski program naveden ranije dostupan je u svom pojednostavljenom obliku kao internet aplikacija na adresi:
`http://demos.php4every1.com/diplomski/`.

Otvaranjem navedene adrese u internet pregledniku prikazuje se polje za unos geste u koje je moguće unijeti gestu. Otpuštanjem miša obrazac s unešenom gestom šalje se poslužitelju na obradu. Nakon što je zahtjev obrađen ispisati će se rezultat prepoznavanja.

Aplikacija je ograničena na uzorkovanje sin-cos i ekvilateralno kodiranje.

9. Mjerenja

U nastavku će biti dana mjerenja ostvarena u sklopu ovog rada. Mjerenja su provedena varirajući veličinu skupa za učenje, broj neurona po skrivenom sloju i broj skrivenih slojeva. Posljednje mjerenje prikazuje uspješnost učenja ovisno u broju epoha, odabranom algoritmu učenja i načinu uzorkovanja.

Mjerenja su provedena na manjim dimenzijama mreže i ne tako velikim skupovima za učenje zbog ograničenosti vremenom, no i iz tih je mjerenja vidljivo da kako mreže postaju kompleksnije i uzoraka ima sve više, ubrzanje je sve više i više vidljivo.

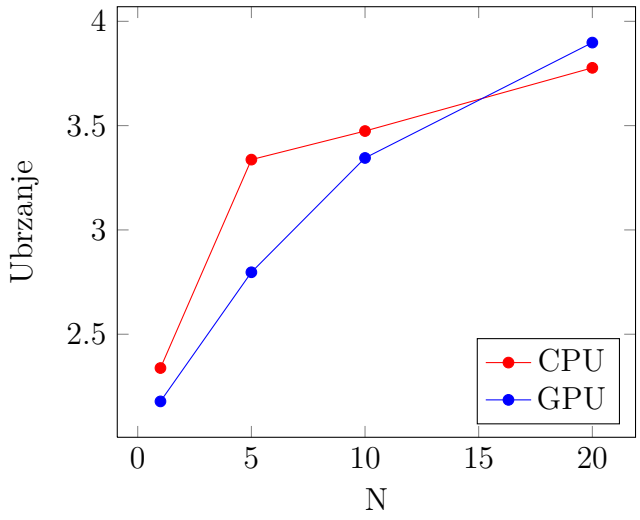
Ubrzanje je bilo manje prije određenih optimizacijskih postupaka poput uklanjanja uvjeta. Neka za potrebe primjera postoji odsječak kôda `if (a > 0) { b += 10; } else { b -= 10; }`. Uzme li se u obzir da je jezik OpenCL baziran na jeziku C koji kao istina/laž vrijednosti koristi brojeve i rezultati logičkih operacija su 0 ili 1, uvjet se može pretvoriti u čiste operacije aritmetičko-logičke jedinice bez potrebe za naredbama grananja. Jedan je od načina odsječak kôda napisati kao `b += 10 * (a > 0) - 10 * (a <= 0);`.

U nastavku poglavlja bit će dana razna mjerenja napravljena koristeći centralni procesor i grafički procesor kao uređaje kojima upravlja OpenCL.

9.1. Variranje veličine skupa za učenje

Prvo mjerenje jest mjerenje u kojem je varirana veličina skupa za učenje. Vrijednosti za veličinu uzorka u grafu 9.1 izražene su u tisućama (1000). Mreža korištena za testiranje je dimenzija $18 \times 24 \times 12 \times 3$.

Kako je implementacija dizajnirana upravo s ciljem da ubrzanje bude najveće povećanjem skupa za učenje, za očekivati je da će upravo ovdje ubrzanje biti najveće. No, to nije slučaj. Vjerojatno se povećanjem broja uzoraka previše vremena utroši na pristupanje globalnoj memoriji. Same jezgre nemaju puno programskih granjanja, no imaju dosta petlji koje također uzrokuju dotične probleme. Problem



Slika 9.1: Ubrzanje u ovisnosti o veličina skupa za učenje

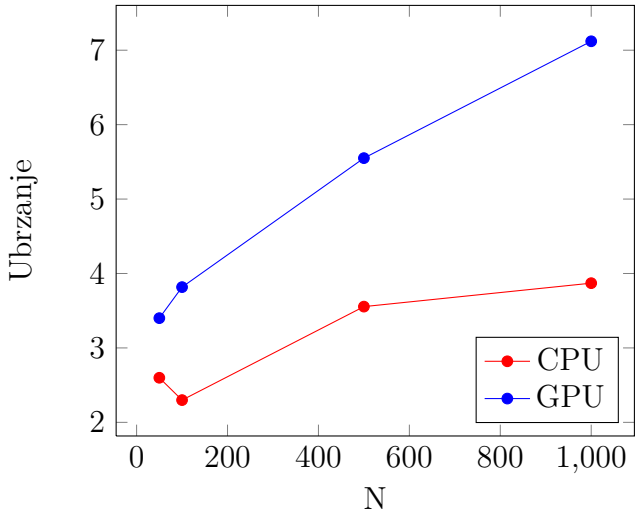
bi se mogao riješiti tako da se koristi raspetljavanjem petlji (engl. *loop unwinding*) budući da u trenutku kada se učenje pokrene, arhitektura i dimenzije mreže su poznati i ne mijenjaju se tijekom procesa učenja.

9.2. Variranje broja neurona po skrivenom sloju

Vjerojatno su najveće iznenađenje rezultati variranja broja neurona po skrivenom sloju. Testiranje je provedeno tako da je uzeta mreža dimenzija $18 \times M \times 3$ gdje M predstavlja broj neurona u skrivenom sloju. Broj uzoraka za učenje je 1000. Rezultati mjerenja prikazani su u grafu 9.2.

Iako je paralelizacija napravljena na način da jedna radna stavka predstavlja cijelu mrežu, ažuriranja težina i sumiranje procjena gradijenta paralelizirani su po broju težina. To znači da za mrežu dimenzija $3 \times 7 \times 2$ i skup uzoraka za učenje veličine 1000 kod poziva jezgri sumiranja procjena gradijenta i ažuriranja težina postojat će ukupno $(3+1) \cdot 7 + (7+1) \cdot 2 = 44$ radne stavke. Za preostale pozive jezgri postojat će 1000 radnih stavki budući da je to veličina skupa uzoraka za učenje. Što je veći broj neurona po sloju to je broj težina veći i tu se postiže ubrzanje.

Zanimljivo je da je to ubrzanje daleko veće od samog ubrzanja koje se dobije evaluacijom mreže i računanjem procjena gradijenata. Također je vidljivo da što ima više neurona po sloju, to se ubrzanje brže približava maksimalnome ubrzanju.



Slika 9.2: Ubrzanje u ovisnosti o broju neurona po skrivenom sloju

9.3. Variranje broja skrivenih slojeva

Posljednje mjerenje koje je provedeno gleda utjecaj povećanja broja skrivenih slojeva na ukupno ubrzanje. Mreža ima 18 ulaznih neurona i 3 izlazna neurona. Svi skriveni slojevi imaju po 24 neurona i varira se broj tih skrivenih slojeva. Broj uzoraka za učenje je 1000. Rezultati su vidljivi u grafu 9.3.

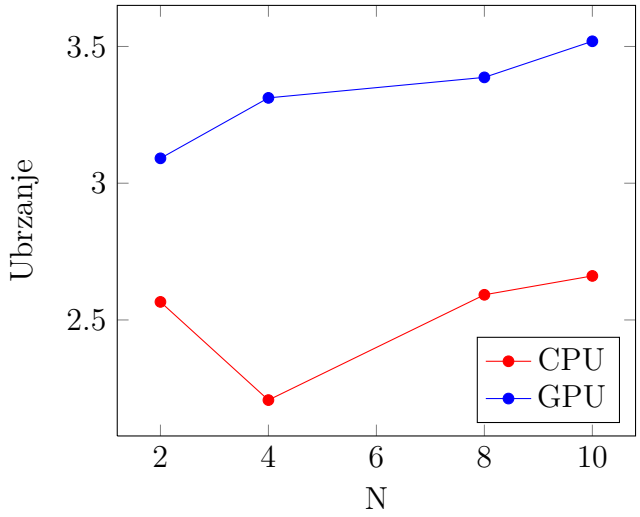
Pogledaju li se rezultati mjerenja u odjeljku 9.1 moguće je zaključiti kako se puno više vremena utroši na zadavanje posla budući da su slojevi relativno jednostavni i vrijeme utrošeno na koristan rad računanja nije dovoljno da sakrije vrijeme utrošeno na zadavanje posla.

9.4. Pogreška učenja kod različitih uzorkovanja i metoda učenja

Konačno valja vidjeti kako se različiti algoritmi učenja ponašaju u kombinaciji s različitim načinima uzorkovanja. Testiranja su provodena za sljedeće kombinacije: algoritam rprop i uzorkovanje sin-cos, algoritam rprop i mrežasto uzorkovanje, algoritam propagiranja greške unazad i uzorkovanje sin-cos te algoritam propagiranja greške unazad i mrežasto uzorkovanje.

Svuda je korišteno ekvilateralno kodiranje i testiranje je napravljeno koristeći svih 1088 uzoraka.

Za uzorkovanje sin-cos dimenzije mreže jesu $18 \times 24 \times 12 \times 3$ dok su dimenzije



Slika 9.3: Ubrzanje u ovisnosti o broju skrivenih slojeva

za mrežasto uzorkovanje $10 \times 22 \times 11 \times 3$. Te su se dimenzije mreža pokazale kao dovoljno jednostavne uz dovoljno malu pogrešku prilikom provođenja postupka učenja. Parametar koji se pokazao kao najuspješniji za algoritam propagiranja greške unatrag jest $\rho = 0.0001$ (vidi dodatak A) za uzorkovanje sin-cos i $\rho = 0.0028$ za mrežasto uzorkovanje.

Kod algoritma rprop najbolji parametri za oba načina uzorkovanja jesu:

$$\triangle_0 = 0.1$$

$$\triangle_{min} = 10^{-9}$$

$$\triangle_{max} = 10$$

$$\kappa^- = 0.5$$

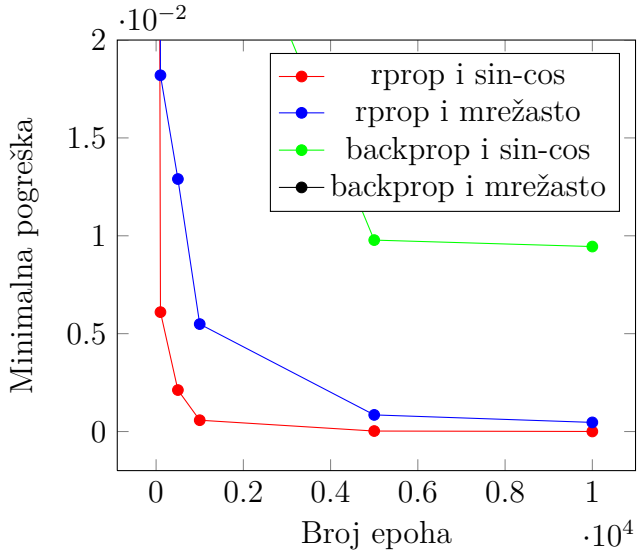
$$\kappa^+ = 1.2$$

Rezultati mjerenja pokazali su kako kombinacija algoritma rprop i uzorkovanja sin-cos konvergira najbrže i dostiže najmanje vrijednosti pogreške.

Također se pokazalo kako je algoritam rprop uz mrežasto uzorkovanje puno bolji od klasičnog algoritma propagiranja greške unatrag uz uzorkovanje sin-cos.

Graf 9.4 prikazuje ponašanje minimalne pogreške nad skupom za učenje uz dane kombinacije algoritama učenja i načina uzorkovanja.

Iako su već mreže s dva skrivena sloja pokazale dovoljno zadovoljavajuće rezultate (minimalna pogreška od 3.89×10^{-6}) istraženo je još i ponašanje mreža s tri



Slika 9.4: Minimalna pogreška u ovisnosti o broju epoha, algoritmu učenja i načinu uzorkovanja

skrivena sloja. Dimenzije korištenih mreža su $18 \times 28 \times 15 \times 9 \times 3$ za uzorkovanje sin-cos i $10 \times 26 \times 14 \times 7 \times 3$ za mrežasto uzorkovanje. Učenje je također provedeno kroz 10000 epoha i kako je vidljivo u tablici 9.1, minimalna pogreška je manja no učenje je trajalo nešto duže zbog više težina koje je potrebno naučiti.

Tablica 9.1: Minimalne ostvarene pogreške kod učenja umjetne neuronske mreže s tri skrivena sloja

Algoritam učenja	Način uzorkovanja	Minimalna pogreška
backprop	sin-cos	2.64×10^{-4}
backprop	mrežasto	6.57×10^{-3}
rprop	sin-cos	9.57×10^{-7}
rprop	mrežasto	8.37×10^{-5}

9.4.1. Ukupan broj pogrešnih klasifikacija

Sama vrijednost pogreške ne daje iznos pogrešno klasificiranih primjera budući da se ona računa preko očekivanih vrijednosti. Iz tog je razloga napravljeno mjerenje pogrešno klasificiranih primjera. Za testiranje su korištene prethodno naučene mreže s dva skrivena sloja i najmanjim pogreškama učenja. Rezultati su poprilično zanimljivi.

Mreža učena nad značajkama koje su dobivene uzorkovanjem sin-cos pogrešno je klasificirala samo dva od 1088 primjera što je 0.18% od ukupnog broja primjera. Primjeri simbola koji su pogrešno klasificirani vidljivi su na slici 9.5. Razlozi krive klasifikacije poprilično su očiti. Slika 9.5a prikazuje simbol β krivo klasificiran kao simbol α zbog zavijutka na samom dnu koji može predstavljati dio simbola α.

(a) Simbol β krivo klasificiran kao simbol α

(b) Simbol γ krivo klasificiran kao simbol β

Slika 9.5: Primjeri krivo klasificiranih simbola kod mreže učene nad podatcima uzorkovanim sin-cos uzorkovanjem

Slika 9.5b s druge strane prikazuje simbol γ krivo klasificiran kao simbol β i nije toliko očit razlog krive klasifikacije no vjerojatno je zbog toga što gesta nije napravljena ravnomjernom brzinom pa su kod uzorkovanja negdje točke gušće dok su negdje rijeđe.

Puno su zanimljiviji rezultati pogrešnog klasificiranja kod mreže učene nad značajkama koje su dobivene mrežastim uzorkovanjem. Ukupno je 48 primjera krivo klasificirano što je 4.41% ukupnog broja primjera. To je 23 puta više krivo klasificiranih primjera nego kod značajki dobivenih uzorkovanjem sin-cos.

Tablica 9.2: Broj krivih klasifikacija po simbolu

Simbol	Broj krivih klasifikacija
α	6
β	6
γ	33
Ω	2

(a) Simbol γ krivo klasificiran kao simbol α

(b) Simbol γ krivo klasificiran kao simbol Ω

Slika 9.6: Primjeri krivo klasificiranih simbola kod mreže učene nad podatcima uzorkovanim mrežastim uzorkovanjem

Kako je vidljivo iz tablice 9.2, najviše je pogrešnih klasifikacija simbola γ. Od toga je 15 pogrešnih klasifikacija u simbol α i 15 pogrešnih klasifikacija u simbol Ω. Primjer jedne takve pogrešne klasifikacije vidljiv je na slici 9.6. Slika 9.6a prikazuje simbol γ krivo klasificiran kao simbol α zbog toga što se uzorkovanjem izgubio dio informacija o tragu geste i dobivene značajke više nalikuju malo zarotiranoj varijaciji simbola α.

Slika 9.6b prikazuje krivo klasificirani simbol γ u simbol Ω. Razlog tome je također u neravnomjernoj brzini stvaranja geste. Lijevi je dio geste napravljen puno brže nego desni dio i zbog toga se uzorkovanjem dobije da značajke bolje predstavljaju simbol Ω nego simbol γ.

9.5. Daljnji radovi

Trenutna implementacija nije savršena. Naprotiv, daleko je od toga no pokazuje da ima velikog potencijala imati stvarnu primjenu uz neka poboljšanja. Također moguće je proučiti dodatne načine uzorkovanja i istražiti mogućnosti prepoznavanja gesti u tri dimenzije.

9.5.1. Optimizacije

Prvo bi trebalo poboljšati način pristupanja memoriji. Trenutna implementacija u velikoj mjeri koristi globalnu memoriju koja je daleko sporija od lokalne ili privatne. Puno se bolja ubrzanja mogu postići pametnijim upravljanjem memo-

rijom.

Kako je rečeno ranije, centralni procesor oslanja se na priručnu memoriju kako bi sakrio latenciju prema radnoj memoriji. Grafički procesor nema tu mogućnost, već se oslanja na to da bude zaposlen nečim drugim dok čeka rezultate iz memorije. Dok neka dretva čeka na podatak iz memorije, druga dretva koja ima sve potrebne podatke može se izvršavati. Što više posla ima za obradu, to je manja latencija memorije. Dretvama upravlja sklop unutar grafičkog procesora pa je izvršavanje par tisuća dretvi u paraleli jako jeftino.

Potrebno je dakle pomnije paziti na sljedeće:

- jezgre trebaju biti pokrenute u što većem broju kako bi se povećala mogućnost sakrivanja latencije memorije,
- jezgre bi trebale koristiti što manje memorije jer će biti jako puno jezgri pokrenuto paralelno,
- način pristupanja memoriji treba biti dizajniran tako da grafički procesor može što bolje optimirati pristupe memoriji budući da on nema pričuvnu memoriju.

Također, potrebno je istražiti mogućnosti da se ne nalaze svi podatci odmah u memoriji budući da su memorije grafičkih kartica manje od glavnih memorija računala. Kompleksne mreže i puno podataka za učenje mogu brzo prepuniti tu memoriju.

9.5.2. Paralelizacija na razini neurona

U radu je obrađena paralelizacija na razini mreže gdje jedna radna stavka predstavlja jedan primjerak mreže za dani uzorak. Što ima više uzoraka, to je više primjeraka i više se posla odradi paralelno. Drugi način paralelizacije koji bi se moglo istražiti jest pralelizacija na razini neurona gdje jedna radna stavka predstavlja jedan neuron.

Kako grafički procesor zahtjeva puno posla kako bi se što bolje sakrila latencija pristupa memoriji, mreže nad kojima bi se koristila takva implementacija trebale bi imati jako velik broj neurona po slojevima.

9.5.3. Prepoznavanje gesti u trodimenzionalnom prostoru

Sam problem prepoznavanja gesti u tri dimenzije više je vezan uz način uzorkovanja. Potrebno je istražiti načine uzorkovanja kao i dimenzije mreže uz koje bi

učenje bilo najučinkovitije. Svi se načini uzorkovanja dani u ovom radu mogu lako poopćiti na tri dimenzije.

Kod koordinatnoga uzorkovanja postojala bi za svaki uzorak još jedna dodatna dimenzija koja bi predstavljala treću dimenziju u prostoru.

U mrežastome uzorkovanju koordinatni se sustav može gledati kao kocka gdje se plohe promatraju kao dvodimenzionalna polja i tako redom slažu. Primjerice, za kocku dimenzija $2 \times 2 \times 2$ koordinate spremljene u memoriju računala redom bi bile $(0,0,0)$, $(0,1,0)$, $(1,0,0)$, $(1,1,0)$, $(0,0,1)$, $(0,1,1)$, $(1,0,1)$ i $(1,1,1)$.

Za uzorkovanje sin-cos potrebno bi bilo dodati još 4 dodatne dimenzije značajki budući da bi tada postojalo više kuteva. Tako bi bilo potrebno imati još kuteve između osi y i z te osi x i z.

9.5.4. Ostali tipovi mreža i algoritmi učenja

Moguće je da bi mreže s povratnim vezama mogle biti od pomoći kod rješavanja ovog problema pa stoga nije loše istražiti arhitekture mreža za koje nije potrebno dodatno uzorkovati podatke i transformirati probleme u probleme manjih dimenzionalnosti.

Konačno, algoritam propagiranja greške unatrag i algoritam rprop nisu jedini algoritmi koji se koriste za učenje mreža. Potrebno bi bilo istražiti mogućnosti paralelizacije drugih algoritama kao i njihovu učinkovitost kod učenja.

10. Zaključak

Problem prepoznavanja gesti postaje sve zanimljiviji i kompleksniji. Što se bolje želi naučiti sustav za prepoznavanje određenih gesti ili proširiti bazu gesti koje je moguće prepoznati, potrebno je imati kompleksniju mrežu kao i više podataka za učenje. Obje te stavke produžuju vrijeme učenja i stoga je potrebno istražiti mogućnosti ubrzanja.

Implementacija algoritama na grafičkom procesoru može donijeti ubrzanje kao što je to u radu i pokazano, no s trenutnom implementacijom to ubrzanje vidljivije je tek kod velikih mreža s velikim brojem uzoraka za učenje. Sam problem prepoznavanja gesti može se poopćiti na prepoznavanje gesti u tri dimenzije. Te je geste moguće uzorkovati raznim senzorima poput uređaja Kinect[1] (Microsoft).

Kako bi se dobilo što bolje ubrzanje, potrebno je stoga bolje optimirati pristupe memoriji budući da se dobar dio vremena troši upravo na to. Ideja je koristiti što više lokalnu memoriju te je stoga potrebno drugačije napraviti paralelizaciju kako bi bilo što manje sinkronizacija.

Kroz ovaj rad istražene su mogućnosti učenja pomoću dva algoritma, algoritma propagirnja greške unazad i algoritma rprop. Također su istražena tri načina generiranja značajki iz izvornih tragova gesti. Umjetne neuronske mreže obrađuju i generiraju brojeve dok su razredi predstavljeni simbolom te su stoga istražena dva načina kodiranja razreda. Nakon toga je dan uvod u biblioteku OpenCL te su dani i implementacijski detalji paralelizacije pomoću biblioteke OpenCL. Na kraju su izložena razna mjerenja napravljena u sklopu ovoga rada kao i smjernice za daljnja poboljšanja implementacije.

[1] Microsoftov senzor pokreta

Literatura

Artificial neural network. `http://en.wikipedia.org/wiki/Artificial_neural_network`. Pristupljeno: 4.3.2014.

Artificial neuron. `http://en.wikipedia.org/wiki/Artificial_neuron`. Pristupljeno: 4.3.2014.

Introduction to OpenCL Programming, 2010.

B. Dalbelo Bašić, M. Čupić, i J. Šnajder. *Umjetne neuronske mreže*. 2008.

O.S. Eluyode i T.D. Akomolafe. Comparative study of biological and artificial neural networks. 2008.

J. Heaton. Equilateral. `http://www.heatonresearch.com/wiki/Equilateral`, 2011. Pristupljeno: 19.3.2014.

Y. LeCun, C. Cortes, i C J.C. Burges. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. Pristupljeno: 8.3.2014.

Microsoft. Kinect. `http://www.microsoft.com/en-us/kinectforwindows/`. Pristupljeno: 11.5.2014.

A. Munshi. *The OpenCL Specification*, 2011.

J. Palacios i J. Triska. A comparison of modern gpu and cpu architectures: And the common convergence of both. 2011.

M. Riedmiller. *Rprop - Description nad Implementation Details*. University of Karlsruhe, 1994.

M. Čupić, B. Dalbelo Bašić, i M. Golub. *Neizrazito evolucijsko i neuroračunarstvo*. 2013.

Dodatci

A. Izvod algoritma propagiranja greške unazad

Algoritam se izvodi za mrežu s m izlaza učenu nad N uzoraka. Funkcija pogreške koja se optimira definirana je kao:

$$E = \frac{1}{2N} \sum_{s=1}^{N} \sum_{l=1}^{m} \left(t_{s,l} - y_{s,l}^{(k+1)} \right)^2$$

gdje $t_{s,l}$ predstavlja očekivani izlaz na neuronu l za predočeni uzorak s. $y_{s,l}^{(k+1)}$ predstavlja izlaz neurona l u $k+1$-om sloju za predočeni uzorak s i to je vrijednost dobivena primjenom prijenosne funkcije na sumu težina i pobuda iz sloja prije.

Težina koja s-ti neuron u k-tom sloju povezuje s l-tim neuronom u $k+1$-om sloju označava se kao $w_{i,j}^{(k)}$. Koristeći ideju gradijentnog spusta ažuriranje težine moguće je pisati kao:

$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \psi \frac{\partial E}{\partial w_{i,j}^{(k)}}$$

gdje će ψ biti mala pozitivna konstanta.

A.1. Izlazni sloj

Izvod se razmatra kao dva posebna slučaja. Prvi slučaj je za izlazni sloj gdje se može pisati:

$$\begin{aligned}\frac{\partial E}{\partial w_{i,j}^{(k)}} &= \frac{\partial}{\partial w_{i,j}^{(k)}}\left[\frac{1}{2N}\sum_{s=1}^{N}\sum_{l=1}^{m}\left(t_{s,l}-y_{s,l}^{(k+1)}\right)^2\right] \\ &= \frac{1}{2N}\sum_{s=1}^{N}\sum_{l=1}^{m}2\cdot\left(t_{s,l}-y_{s,l}^{(k+1)}\right)\cdot(-1)\cdot\frac{\partial y_{s,l}^{(k+1)}}{\partial w_{i,j}^{(k)}} \\ &= -\frac{1}{N}\sum_{s=1}^{N}\sum_{l=1}^{m}\left(t_{s,l}-y_{s,l}^{(k+1)}\right)\cdot\frac{\partial y_{s,l}^{(k+1)}}{\partial w_{i,j}^{(k)}}.\end{aligned}$$

Vidljivo je da težina $w_{i,j}^{(k)}$ utječe samo na neuron j u $k+1$-vom sloju. Iz tog je razloga samo parcijalna derivacija za $l=j$ različita od 0. Sve ostale nestaju pa vrijedi:

$$\sum_{l=1}^{m}\left(t_{s,l}+y_{s,l}^{(k+1)}\right)\cdot\frac{\partial y_{s,l}^{(k+1)}}{\partial w_{i,j}^{(k)}} = \left(t_{s,j}-y_{s,j}^{(k+1)}\right)\cdot\frac{\partial y_{s,j}^{(k+1)}}{\partial w_{i,j}^{(k)}}$$

pa se dalje može pisati:

$$\frac{\partial E}{\partial w_{i,j}^{(k)}} = -\frac{1}{N}\sum_{s=1}^{N}\left(t_{s,j}+y_{s,j}^{(k+1)}\right)\cdot\frac{\partial y_{s,j}^{(k+1)}}{\partial w_{i,j}^{(k)}}.$$

Preostaje još izračunati parcijalnu derivaciju gdje se može koristiti pravilo ulančavanja derivacija. Tako vrijedi:

$$\frac{\partial y_{s,j}^{(k+1)}}{\partial w_{i,j}^{(k)}} = \frac{\partial y_{s,j}^{(k+1)}}{\partial net_{s,j}^{(k+1)}}\cdot\frac{\partial net_{s,j}^{(k+1)}}{\partial w_{i,j}^{(k)}}.$$

Budući da se radi o slojevitoj mreži gdje nema cikličkih i lateralnih veza sve vrijednosti koje nisu uz traženu težinu predstavljaju konstante. Tako je tražena parcijalna derivacija:

$$\frac{\partial net_{s,j}^{(k+1)}}{\partial w_{i,j}^{(k)}} = y_{s,i}^{(k)}.$$

Uvrštavanjem dobivenih vrijednosti proizlazi da je parcijalna derivacija definirana kao:

$$\begin{aligned}\frac{\partial E}{\partial w_{i,j}^{(k)}} &= -\frac{1}{N}\sum_{s=1}^{N}\left(t_{s,j}-y_{s,j}^{(k+1)}\right)\cdot\frac{\partial y_{s,j}^{(k+1)}}{\partial net_{s,j}^{(k+1)}}\cdot y_{s,i}^{(k)} \\ &= -\frac{1}{N}\sum_{s=1}^{N}\omega_{s,j}^{(k+1)}\cdot y_{s,i}^{(k)}\end{aligned}$$

gdje vrijedi:

$$\omega_{s,j}^{(k+1)} = \left(t_{s,j} - y_{s,j}^{(k+1)}\right) \cdot \frac{\partial y_{s,j}^{(k+1)}}{\partial net_{s,j}^{(k+1)}}.$$

Vrijednost $\omega_{s,j}^{(k+1)}$ predstavlja pogrešku j-tog izlaznog neurona za s-ti predočeni uzorak. Kako se izvod provodi za proizvoljnu prijenosnu funkciju, $\frac{\partial y_{s,j}^{(k+1)}}{\partial net_{s,j}^{(k+1)}}$ predstavlja vrijednost derivacije te prijenosne funkcije za trenutnu pobudu.

Pravilo za ažuriranje težine $w_{i,j}^{(k)}$ tada glasi:

$$\begin{aligned} w_{i,j}^{(k)} &= w_{i,j}^{(k)} - \psi \frac{\partial E}{\partial w_{i,j}^{(k)}} \\ &= w_{i,j}^{(k)} - \psi \cdot \left(-\frac{1}{N} \sum_{s=1}^{N} \omega_{s,j}^{(k+1)} \cdot y_{s,i}^{(k)} \right) \\ &= w_{i,j}^{(k)} + \rho \cdot \left(\sum_{s=1}^{N} \omega_{s,j}^{(k+1)} \cdot y_{s,i}^{(k)} \right) \end{aligned}$$

gdje ρ predstavlja novu konstantu i vrijedi $\rho = \frac{\psi}{N}$.

A.2. Skriveni sloj

Potrebno je još razmotriti skriveni sloj i izračunati parcijalnu derivaciju po promatranoj težini $w_{i,j}^{(k-1)}$. Postupak deriviranja se ponovi pa tako vrijedi:

$$\begin{aligned} \frac{\partial E}{\partial w_{i,j}^{(k-1)}} &= \frac{\partial}{\partial w_{i,j}^{(k-1)}} \left[\frac{1}{2N} \sum_{s=1}^{N} \sum_{l=1}^{m} \left(t_{s,l} - y_{s,l}^{(k+1)}\right)^2 \right] \\ &= \frac{1}{2N} \sum_{s=1}^{N} \sum_{l=1}^{m} 2 \cdot \left(t_{s,l} - y_{s,l}^{(k+1)}\right) \cdot (-1) \cdot \frac{\partial y_{s,l}^{(k+1)}}{\partial w_{i,j}^{(k-1)}} \\ &= -\frac{1}{N} \sum_{s=1}^{N} \sum_{l=1}^{m} \left(t_{s,l} - y_{s,l}^{(k+1)}\right) \cdot \frac{\partial y_{s,l}^{(k+1)}}{\partial w_{i,j}^{(k-1)}} \\ &= -\frac{1}{N} \sum_{s=1}^{N} \sum_{l=1}^{m} \left(t_{s,l} - y_{s,l}^{(k+1)}\right) \cdot \frac{\partial y_{s,l}^{(k+1)}}{\partial net_{s,l}^{(k+1)}} \cdot \frac{\partial net_{s,l}^{(k+1)}}{\partial w_{i,j}^{(k-1)}} \end{aligned}$$

Ovdje također postoji parcijalna derivacija koju je potrebno riješiti. Pogleda li se ponovo kako je mreža strukturirana može se zaključiti kako na $net_{s,o}^{(k+1)}$ težina $w_{i,j}^{(k-1)}$ utječe samo preko izlaza neurona $s_{s,j}^{(k)}$. Zbog toga vrijedi:

$$\frac{\partial net_{s,l}^{(k+1)}}{\partial w_{i,j}^{(k-1)}} = w_{jl}^{(k)} \cdot \frac{\partial y_{s,j}^{(k)}}{\partial w_{i,j}^{(k-1)}}.$$

Daljnjim uvrštavanjem dolazi se do:

$$\begin{aligned}
\frac{\partial E}{\partial w_{i,j}^{(k-1)}} &= -\frac{1}{N}\sum_{s=1}^{N}\sum_{l=1}^{m}\left(t_{s,l} - y_{s,l}^{(k+1)}\right) \cdot \frac{\partial y_{s,l}^{(k+1)}}{\partial net_{s,l}^{(k+1)}} \cdot w_{jl}^{(k)} \cdot \frac{\partial y_{s,j}^{(k)}}{\partial w_{i,j}^{(k-1)}} \\
&= -\frac{1}{N}\sum_{s=1}^{N}\frac{\partial y_{s,j}^{(k)}}{\partial w_{i,j}^{(k-1)}}\left[\sum_{l=1}^{m}\left(t_{s,l} - y_{s,l}^{(k+1)}\right) \cdot \frac{\partial y_{s,l}^{(k+1)}}{\partial net_{s,l}^{(k+1)}} \cdot w_{jl}^{(k)}\right] \\
&= -\frac{1}{N}\sum_{s=1}^{N}\frac{\partial y_{s,j}^{(k)}}{\partial w_{i,j}^{(k-1)}}\left[\sum_{l=1}^{m}\omega_{s,l}^{(k+1)} \cdot w_{jl}^{(k)}\right].
\end{aligned}$$

Preostaje još izračunati zadnju parcijalnu derivaciju, no slično je već urađeno ranije. Koristeći pravilo ulančavanja vrijedi:

$$\begin{aligned}
\frac{\partial y_{s,j}^{(k)}}{\partial w_{i,j}^{(k-1)}} &= \frac{\partial y_{s,j}^{(k)}}{\partial net_{s,j}^{(k)}} \cdot \frac{\partial net_{s,j}^{(k)}}{\partial w_{i,j}^{(k-1)}} \\
&= \frac{\partial y_{s,j}^{(k)}}{\partial net_{s,j}^{(k)}} \cdot y_{s,i}^{(k-1)}.
\end{aligned}$$

Konačno, sada je moguće doći do gradijenta:

$$\begin{aligned}
\frac{\partial E}{\partial w_{i,j}^{(k-1)}} &= -\frac{1}{N}\sum_{s=1}^{N}\frac{\partial y_{s,j}^{(k)}}{\partial net_{s,j}^{(k)}} \cdot y_{s,i}^{(k-1)} \cdot \left[\sum_{l=1}^{m}\omega_{s,l}^{(k+1)} \cdot w_{jl}^{k}\right] \\
&= -\frac{1}{N}\sum_{s=1}^{N} y_{s,i}^{(k-1)} \cdot \left[\frac{\partial y_{s,j}^{(k)}}{\partial net_{s,j}^{(k)}} \cdot \left(\sum_{l=1}^{m}\omega_{s,l}^{(k+1)} \cdot w_{jl}^{k}\right)\right] \\
&= -\frac{1}{N}\sum_{s=1}^{N} y_{s,i}^{(k-1)} \cdot \omega_{s,j}^{(k)}.
\end{aligned}$$

gdje je $\omega_{s,j}^{(k)}$ definirana kao

$$\omega_{s,j}^{(k)} = \frac{\partial y_{s,j}^{(k)}}{\partial net_{s,j}^{(k)}} \cdot \left(\sum_{l=1}^{m}\omega_{s,l}^{(k+1)} \cdot w_{jl}^{k}\right)$$

i predstavlja pogrešku skrivenog neurona j u k-tom sloju za uzorak s.

Ažuriranje težine $w_{i,j}^{(k-1)}$ obavlja se kao:

$$
\begin{aligned}
w_{i,j}^{(k-1)} &= w_{i,j}^{(k)} - \psi \frac{\partial E}{\partial w_{i,j}^{(k-1)}} \\
&= w_{i,j}^{(k-1)} - \psi \cdot \left(-\frac{1}{N} \sum_{s=1}^{N} y_{s,i}^{(k-1)} \cdot \omega_{s,j}^{(k)} \right) \\
&= w_{i,j}^{(k-1)} + \rho \cdot \left(\sum_{s=1}^{N} y_{s,i}^{(k-1)} \cdot \omega_{s,j}^{(k)} \right)
\end{aligned}
$$

gdje je umjesto ψ uvedena nova konstanta $\rho =$ i vrijedi $\rho = \frac{\psi}{N}$.

Time su izvedeni izrazi za ažuriranje težina između pojedinih neurona.

B. Konfiguracijski parametri

Konfiguracijski parametri za programe treniranja i demenstracije dani su u nastavku.

`train.algorithm` - određuje koji algoritam učenja se koristi. Moguće vrijednosti su `rprop` i `backprop`. Pretpostavljena je vrijednost `rprop`.

`train.algorithm.backprop.ni` - predstavlja rijednost parametra ρ koji je dobiven u izvodu. Koristi se samo ako je kao algoritam učenja odabran `backprop`. Pretpostavljena je vrijednost 0.0001.

`train.algorithm.rprop.kMinus` - predstavlja korekcijski faktor za koji se umanjuje `k` korišen u `rpop` algoritmu kod računanja `delta` vrijednosti. Pretpostavljena je vrijednost 0.5.

`train.algorithm.rprop.kPlus` - predstavlja korekcijski faktor za koji se uvećava `k` korišen u `rpop` algoritmu kod računanja `delta` vrijednosti. Pretpostavljena je vrijednost 1.2.

`train.algorithm.rprop.deltaInitial` - predstavlja početnu vrijednost `delta` korekcije. Pretpostavljena je vrijednost 0.1.

`train.algorithm.rprop.deltaMin` - predstavlja minimalnu vrijednost koju može poprimiti `delta` korekcija. Pretpostavljena je vrijednost 10^{-9}.

`train.algorithm.rprop.deltaMax` - predstavlja maksimalnu vrijednost koju može poprimiti `delta` korekcija. Pretpostavljena je vrijednost 10.

`train.implementation` - predstavlja implementaciju koja se koristi. `regular` predstavlja klasičnu operaciju dok `opencl` predstavlja inačicu koja u pozadini koristi OpenCL. Pretpostavljena je vrijednost `opencl`.

`train.implementation.opencl.device` - predstavlja koji uređaj se koristi kao radnik. Moguće vrijednosti su 4 za GPU i 2 za CPU. Pretpostavljena je vrijednost 4.

`train.iterations` - određuje koji je broj epoha potrebno izvršiti za učenje. Pretpostavljena je vrijednost 1000

`train.logger.iterationMod` - određuje nakon koliko se epoha ispisuje informacija o stanju algoritma. Pretpostavljena je vrijednost 100.

`train.weights.low` - određuje donju granicu intervala iz kojeg će se generirati početne težine neuronske mreže. Pretpostavljena je vrijednost -1.

`train.weights.high` - određuje gornju granicu intervala iz kojeg će se generirati početne težine neuronske mreže. Pretpostavljena je vrijednost 1.

`train.seed` - određuje vrijednost sjemena koje se koristi za inicijalizaciju generatora pseoudoslučajnih brojeva. Vrijednost -1 predstavlja nasumično sjeme. Pretpostavljena je vrijednost -1.

`train.data` - putanja do datoteke gdje se nalaze uzorci za učenje. Datoteka ne smije imati zaglavlje. Prvih N stupaca predstavljaju N ulaznih podataka dok zadnji stupac predstavlja razred u koji želimo klasificirati ulazne podatke. Ukupno mora biti $N + 1$ stupac. Pretpostavljena je vrijednost `data/sin_cos.csv`.

`network.weights` - putanja do datoteke gdje se pohranjuju naučene težine. Težine su pohranjene u sirovom formatu tipa `double`. Pretpostavljena je vrijednost `data/sin_cos.net`.

`network.dimensions` - dimenzije mreže koja se uči ili koja se koristi kao demonstracijski primjer. Dimenzije su odvojene znakom `x` i obavezno moraju biti definirane barem dvije dimenzije. Dimenzija ne smije biti manja od 1. Pretpostavljena je vrijednost `18x24x12x3`.

`display.sampler` - koristi se kod demonstracijskog programa i predstalja način uzorkovanja podataka. Moguće vrijednosti su `sin_cos` za sinus-kosinus uzorkovanje i `grid` za mrežasto uzorkovanje. Pretpostavljena je vrijednost `sin_cos`.

`display.classes` - koristi se kod demonstracijskog programa i predstalja skup razreda odvojenih zarezom u koje je moguće klasificirati predočeni uzorak. Pretpostavljena je vrijednost `ALPHA,BETA,GAMA,OMEGA`.

`display.sampler.n` - definira u koliko točaka se uzorkuje gesta. Pretpostavljena je vrijednost 10.

`display.sampler.grid.n` - definira dimenziju matrice u koju se transformiraju koordinate kod mrežastog uzorkovanja. Pretpostavljena je vrijednost 12.

Prepoznavanje gesti na GPU

Sažetak

Problem prepoznavanja gesti jako je interestantan problem zbog prisutnosti vremenske dimenzije. Postoji više načina kako se problem može riješiti, a u ovom radu istražen je način rješavanja korištenjem slojevite potpuno povezane unaprijedne neuronske mreže.

Da bi to bilo moguće, potrebno je transformirati problem tako da ne ovisi o vremenskoj dimenziji. Zbog toga je istraženo više načina uzorkovanja gdje se eliminira vremenska dimenzija.

Znanje o problemu implicitno je pohranjeno u težinama između umjetnih neurona u umjetnoj neuronskoj mreži. Kako bi neuronska mreža što bolje obavljala svoj posao, potrebno je što bolje podesiti te težine i taj se postupak zove učenje neuronske mreže. Razmotrena su dva algoritma učenja neuronske mreže, algoritam propagiranja greške unatrag i algoritam rprop.

Kao i kod čovjeka, puno je više vremena potrebno utrošiti na učenje nego na prepoznavanje. To je povod za istraživanje načina ubrzavanja učenja i jedan od tih načina jest paralelizacija.

Svojom arhitekturom, grafički procesor predstavlja idealnoga kandidata za paralelizaciju učenja neuronskih mreža. U sklopu ovoga rada istražena je paralelizacija na razini mreže, tj. jedna radna stavka predstavlja jedan primjerak neuronske mreže za predočeni uzorak.

Problem prepoznavanja gesti još je neistraženo područje koje danas postaje sve zanimljivije sve većom popularnošću pametnih telefona i ekrana osjetljivih na dodir. Postoji još puno segmenata koje je potrebno istražiti i provjeriti poput drugih algoritama učenja ili samih arhitektura neuronskih mreža.

Ključne riječi: cpu, gesta, gpu, neuronska mreža, propagiranje greške unazad, rprop, opencl

Gesture recognition using GPU

Abstract

The problem of gesture recognition is a very interesting problem because of the temporal dimension that exists and has to be handled somehow. There are several ways in which the problem can be solved and in this paper way that uses fully connected feed-forward neural network has been explored.

To make this possible, it is necessary to transform the problem so it does not depend on the temporal dimension. Therefore, a few ways of sampling have been explored in order to eliminate the temporal dimension.

Knowledge about the problem is implicitly stored in the weights between the neurons in the neural network. In order for neural network to better perform its job, it is necessary to better adjust the weights and that process is called learning. Two learning algorithms have been implemented, backpropagation algorithm and rprop algorithm.

As in humans, much more time is spent on learning than on recognition itself. Because of that, methods that speed up learning had to be explored and one of those method is parallelization.

With its architecture, GPU is an ideal candidate for parallel neural network learning. As part of this study, parallel processing at the network level have been explored, ie. one work item presents one instance of neural network.

The problem of gesture recognition is still uncharted territory which is becoming more and more interesting, especially with the growing popularity of smart phones and touch screens. There are still a lot of things that need to be explored, like other learning algorithms or other architectures of neural networks.

Keywords: backpropagation, cpu, gesta, gpu, neural network, rprop, opencl