SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 013

Kvantizacija konvolucijskih modela za raspoznavanje slika

Marko Minđek

Zagreb, srpanj 2021.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Zahvaljujem mojoj obitelji na bezuvjetnoj potpori kroz cijeli proces obrazovanja.

Zahvaljujem mentoru prof. dr. sc. Siniši Šegviću na trudu, vremenu, brojnim stručnim savjetima i prenesenom znanju.

SADRŽAJ

1. Uvod

Računalni vid područje je umjetne inteligencije kojem je cilj oponašati i shvatiti ljudski vid. Koliko je taj zadatak zapravo težak govori informacija da se u jezgri ljudskog mozga zaduženoj za vid nalazi otprilike 140 milijuna neurona koji su međusobno povezani pomoću više milijardi veza. Razvoj računalnog vida započeo je na američkom MIT-u 60-ih godina prošlog stoljeća, a doživio je procvat pojavom koncepta dubokog učenja. Duboki modeli dovoljno su dobri da ih se koristi u industriji za razne rutinske zadatke koje je prije obavljao čovjek. Čitanje registracijskih oznaka automobila, prepoznavanje lica, čitanje ljudskog rukopisa i detekcija objekata u prometu samo su neki od primjera gdje se može iskoristiti računalni vid.

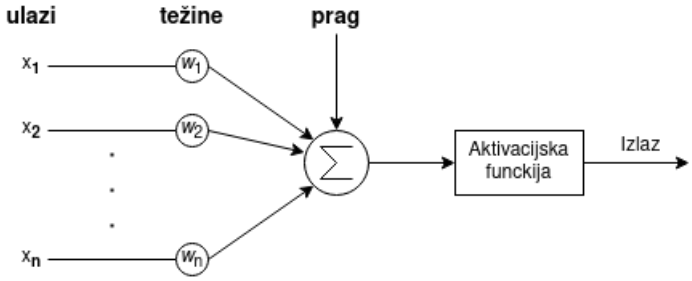
Računalni vid ponekad zahtjeva velik količine računalne snage kako bi uspio obaviti potrebne proračune nad podacima. Tu se posebno ističu *real-time* sustavi koji moraju imati odziv reda veličina milisekundi. Primjer takvog sustava bio bi autonomni automobil, gdje je velika brzina nužan preduvjet za funkcioniranje u stvarnom okruženju.

Dvije tehničke grane koje računalni vid obuhvaća su rekonstrukcija i raspoznavanje slika. Ovaj rad bavi se raspoznavanjem, tj. klasifikacijom slika, a stanje tehnike (engl. *state-of-the-art*) u tom području trenutno se postiže dubokim neuronskim mrežama. Rad neuronskih mreža možemo ubrzati na razne načine, a fokus ovog rada je na ubrzanju postupkom kvantizacije. Poznato je da su operacije nad decimalnim brojevima vrlo skupe, ali postupkom kvantizacije možemo ih pretvoriti u prirodne brojeve, a da pritom minimalno gubimo na točnosti mreže. Na taj ćemo način smanjiti latenciju mreže prilikom postupka zaključivanja. Kvantizaciju ćemo najprije provesti na procesoru opće namjene, a zatim i na grafičkom procesoru.

2. Umjetne neuronske mreže

Umjetne neuronske mreže[1] (engl. *artificial neural networks*) nastale su kao rezultat konektivističkog pristupa umjetnoj inteligenciji sredinom 20. stoljeća. Takav pristup nastoji izgraditi arhitekturu sličnu mozgu pa možemo reći da su umjetne neuronske mreže biološki inspirirane. Duboki modeli odstupaju od ovog pristupa te je njihov cilj simulirati univerzalni aproksimator funkcija. Umjetnu neuronsku mrežu prije eksploatacije moramo najprije naučiti, tj. pokazati što se od nje očekuje. Osnovna ideja učenja je predočiti mreži primjere ulaznih podataka, ali i primjere izlaznih vrijednosti koje očekujemo od mreže za te ulaze. Takav način učenja, kada mrežu učimo na temelju ulaza i željenih izlaza nazivamo nadzirano učenje (engl. *supervised learning*). Spomenimo da postoje i druge vrste učenja koje nećemo obrađivati u sklopu ovog rada.

Elementi umjetnih neuronskih mreža nekada su bili umjetni neuroni. Razvijeno je više vrsta umjetnih neurona, no najčešće se koristo *McCulloch-Pitts-ov* model (slika 2.1) koji su Warren S. McCalloch i Walter Pitts predložili 1943. godine.



Slika 2.1: Model umjetnog neurona

Zadaća umjetnog neurona bila je generirati jedinstveni izlaz y na temelju ulaza $x_1,...,x_n$. Osim o konkretnim ulazima, primjetite da izlaz ovisi i o težini (engl. *weight*) w_i svakog ulaza, pragu $-\Theta$ (engl. *threshold*) i aktivacijskoj funkciji f. Izraz ovako

[1] U ovom radu često ćemo koristiti pokratu "neuronske mreže" ili "mreže".

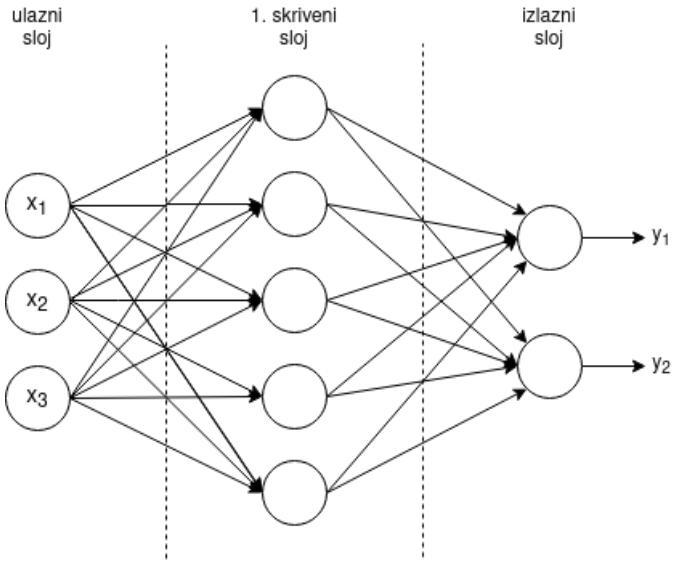
određenog neurona definiran je izrazom:

$$y = f(\sum_{i=1}^{\infty} w_i * x_i - \Theta). \tag{2.1}$$

Kako bi pojednostavili zapis potrebno je definirati veličinu suprotnu pragu, a to je *pomak* ili *pristranost*(engl. *bias*) $b = -\Theta$. Pomak je, uz težine neurona, jedan od parametara koji će se mijenjati kako mreža bude učila. Možemo ju zamisliti kao pomak u linearnoj funkciji. Nakon što smo definirali, pomak b možemo promatrati kao dodatan ulaz x_0 konstantne vrijednosti 1 i promjenjive vrijednosti w_0. Sada uz $x_0 = 1$ i $w_0 = b$ možemo pisati:

$$\begin{aligned} y &= f(\sum_{i=0}^{\infty} w_i * x_i) \\ &= f(\vec{w}^T * \vec{x}). \end{aligned} \tag{2.2}$$

Danas moderne neuronske mreže više ne promatramo kao skup neurona već kao kompoziciju diferencijabilnih nelinearnih funkcija sa slobodnim parametrima. Programski se ostvaruju usmjerenim računskim grafovima. Svaki čvor grafa predstavlja varijablu koja može biti tenzor bilo kojeg reda. Veze grafa predstavljaju težine povezanosti i određenu računsku operaciju. Računski graf možemo prema dubini čvorova podijeliti u slojeve. Primjer takvog grafa nalazi se na slici 2.2. Izvedba pomoću povezanog grafa omogućuje učinkovito računanje derivacija potrebnih za učenje mreže što je detaljnije opisano u poglavlju 3.3. U ovom radu spominjemo samo mreže koje su acikličke (bez povratnih i lateralnih veza), tj. unaprijedne (engl. *feed-forward*).



Slika 2.2: Model slojevite umjetne neuronske mreže

Prvi sloj je ulazni sloj koji zapravo čine ulazni podaci. Zadnji sloj je izlazni sloj gdje očitavamo vrijednosti i na temelju njih očitujemo rezulate obrade. Sve između ulaznog i izlaznog sloja možemo apstrahirati kroz tzv. skrivene slojeve (engl. *hidden layers*). Primjetite da su svi izlazi iz nekog sloja spojeni na sve ulaze iz sljedećeg sloja. Ovako povezane slojeve nazivamo potpuno povezani slojevi (engl. *fully connected layers*). Umjetne neuronske mreže možemo matematički promatrati kao funkcije koje pretvaraju tenzor n-tog reda u izlazni tenzor m-tog reda. ($\vec{f} : \mathbb{R}^n \to \mathbb{R}^m$). Kod klasifikacije slika, ulazni tenzor uglavnom je trećeg reda (promatramo RGB komponente), dok je izlazni tenzor prvog reda pa ga možemo nazvati i izlaznim vektorom.

Umjetne neuronske mreže danas se primjenjuju za širok spektar problema. Najviše se koriste u svrhu klasifikacije i segmentacije slika, ali rješavaju čak i kompleksnije probleme poput upravljanja kompleksnim sustavima (svemirske letjelice, autonomna vozila) ili obradom ljudskog jezika (prevođenje u stvarnom vremenu, autentifikacija na temelju govora). Jedna od glavnih mana neuronskih mreža jest ta što je znanje mreže ljudima neinterpretabilno što ograničava primjenu ljudske intuicije u njihovom razvoju.

2.1. Aktivacijske funkcije

Promotrimo sljedeću funkciju:

$$step(x) = \begin{cases} 0, & x < 0 \\ 1, & y \geq 0. \end{cases} \tag{2.3}$$

Ovakva funkcija korištena je kao aktivacijska funkcija u početnim modelima neuronskim mreža. Svojstva ove funkcije nisu pogodna za primjenu u modernim neuronskim mrežama koje koriste algoritme učenja koji se temelje na derivaciji funkcije aktivacije. Naime, funckija *step* nije diferencijabilna u $x = 0$, a u svim ostalim točkama njezina derivacija iznosi 0, stoga bi učenje bilo nemoguće. Također, ako od mreže očekujemo dobru generalizaciju, aktivacijska funkcija za male promjene ulaza ne smije davati velike promjene na izlazu.

Problemi koje rješavamo neuronskim mrežama često zahtjevaju nelinearnost aktivacijske funkcije. Prva funkcija koja se počela koristiti u svrhu nelinearnost bila je sigmoidalna funkcija[2] σ:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.4}$$

[2]Često je još naziva i logistička funkcija.

Sigmoidalna funkcija je derivabilna za svaki $x \in \mathbb{R}$ i zbog toga je pogodna za algoritme učenje koji koriste gradijentnu optimizaciju. U poglavlju 3.3 opisat ćemo algoritam propagacije pogreške unatrag koji koristi tehniku gradijentnog spusta. Derivacije sigmoidalne funkcije iznosi:

$$\begin{aligned} \frac{d\sigma(x)}{dx} &= \frac{d\frac{1}{1+e^{-x}}}{dx} \\ &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} * \frac{1-1+e^{-x}}{1+e^{-x}} \\ &= \sigma(x) * (1-\sigma(x)). \end{aligned} \tag{2.5}$$

Primijetite da se derivacija sigmoidalne funkcije može izraziti preko nje same. Ovo zgodno svojstvo može nam omogućiti efikasno računanje derivacije korištenjem predmemoriranja (engl. *caching*) vrijednosti $\sigma(x)$. Međutim, ova derivacija ima nezgodno svojstvo prigušivanja derivacije jer vrijedi: $|\frac{d\sigma(x)}{dx}| \leq \frac{1}{4}$, tj. vrijednost derivacije sigmoidalne funkcije gornje je ograničena vrijednošću $\frac{1}{4}$. Glavni problem je u tome što jednadžba 2.5 može poprimiti eksponencijalno malenu vrijednost. To može jako usporiti gradijentne postupke učenja.

Konačno, promotrimo sada funkciju zglobnicu (poluvalno ispravljenu linearnu funkciju (engl. *Rectified Linear Unit, ReLU*) R koja djelomično rješava problem prigušenja gradijenta. R je definirana sljedećim izrazom:

$$R(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0. \end{cases} \tag{2.6}$$

Primijetite da je derivacija ove funkcije trivijalna, a iznosi 0 za negativne i 1 za pozitivne vrijednosti. Ova funkcija ne prigušuje gradijent i minimalno je računski zahtjevna kod izračuna aktivacije, ali i derivacije. Problem funkcije zglobnice su negativne vrijednosti, za koje funkcija daje izlaz 0, ali i derivaciju 0. Ukoliko se neke od težina postave na negativne vrijednosti moglo bi doći do pojave tzv. *mrtvih aktivacija*. Aktivacije nazivamo *mrtvima* ako ih je jako teško *natjerati* da rezultiraju vrijednošću > 0. Također, prilikom postupka učenja gradijent će se u nekim čvorovima evaluirati kao 0 i neće imati daljnjih efekata na mrežu. ReLU se najčešće koristi kao aktivacijska funkcija u dubokim modelima.

2.2. Duboki modeli

Jednoslojne mreže funkcioniraju dobro na podatcima male dimenzionalnosti. Problem koji nastaje povećanjem dimenzionalnosti jest eksponencijalan rast broja parametara potrebnih za uspješno zaključivanje mreže. Rješavanju tog problema pristupit ćemo umjetnim neuronskim mrežama sa više od jednog skrivenog sloja, tj. dubokim neuronskim mrežama. Duboki model možemo promatrati kao slijed nelinearnih transformacija.

Ovaj pristup u prošlosti nije bio popularan, jer za razliku od jednoslojnih mreža ne daje garanciju konvergencije učenja, a učenje mreže ponekad zahtjeva milijune označenih ulaznih podataka. Također, u dubokim modelima javljaju se problemi nestajućeg i eksplodirajućeg gradijenta. Spomenimo kako postoje mnogobrojne tehnike rješavanja ili ublažavanja učinaka ovih problema koje samim time pospješuju konvergenciju učenja. Zbog problema nestajućeg gradijenta uglavnom ne koristimo sigmoidalnu funkciju u dubokim modelima. Trenutno popularni gusto povezani duboki modeli uspješno rješavaju problem nestajućeg gradijenta, a biti će opisani u poglavlju 4. Eksplodirajući gradijent česta je pojava kad se koristi ReLU aktivacijska funkcije jer ona nije gornje ograničena i gradijent može poprimiti eksponencijalo velike vrijednosti.

Duboki modeli postali su popularni dijelom zahvaljujući i velikoj procesnoj moći. Grafički procesori (GPU) u kombinaciji sa optimiziranom programskom podrškom za duboko učenje (npr. CuDNN, TensorRT) omogućili su treniranje ovakvih mreža u razumnom vremenu. Google je 2016. godine razvio tenzorske procesore (TPU) čija je namjena izvođenje operacija dubokog učenja maksimalno brzo i efikasno. Prema [21] koristeći TPU verzije 3 možemo postići prosecnu moć do 420 TFLOPS-a što je gotovo trostruko više od popularnog grafičkog procesora NVidia V100. Ovo ubrzanje može znatno ubrzati treniranje mreže uzimajući u obzir da učenje može trajati tjednima ili mjesecima. Google trenutno nudi uslugu korištenja njihovih tenzorskih procesora u oblaku (engl. *cloud*).

2.3. Duboki konvolucijski modeli

Dugo vremena glavne sastavnice neuronskih mreža bili su potpuno povezani slojevi. Problemi potpuno povezanih slojeva su neskalabilnost, veliki zahtjevi za memorijom i neefikasnost otkrivanja značajki podataka. Za primjer uzmimo mrežu koja klasificira slike veličine 224×224 piksela. U praksi, slike na ulazu u klasifikaciju rastavimo prema RGB vrijednostima u 3 kanala. Dakle, ulaz ima $M = 224 * 224 * 3 = 150528$

dimenzija. Recimo da sljedeći sloj sadrži samo $N = 1024$ dimenzija. Tada bi samo težine tog potpuno povezanog sloja zauzimale bi $N * M * 4$ bajtova ($\approx 0.57GB$). Prema 2.2 vrijednosti izlaza možemo matrično izraziti kao množenje transponirane matrice težina dimenzije $[N \times 150528]$ i vektora ulaza dimenzije $[150528 \times 1]$, gdje N predstavlja broj dimenzija u sloju. Ova operacija je zapravo istovjetna provođenju skalarnog produkta N puta. Shodno tome, kompleksnost ove operacije je O(N*M), gdje je M broj dimenzija prethodnog sloja. Skalarni produkt moguće je računati vrlo efikasno pomoću posebnog skupa vektorskih instrukcija, ali kod dubokih modela to nam nije dovoljno.

Promjena u arhitekturi dubokih modela za klasifikaciju slika dogodila se uvođenjem konvolucijskih slojeva. Ideja je umjesto povezivanja piksela slike *svaki sa svakim*, povezati piksel slike samo s njegovim lokalnim okruženjem proizvoljne veličine. Na taj način smanjujemo broj potrebnih parametara mreže, a omogućavamo slojevima da preciznije detektiraju lokalne značajke ulaznih podataka. Konvolucijski slojevi najprije su se koristili za problem klasifikacije slika, ali trenutno se koriste i za razne druge probleme iz područja strojnog učenja.

Definirajmo operaciju dvodimenzionalne matrične konvolucije na jednostavnom primjeru. Ulaz u model je matrica I dimenzija $[W \times H]$. Lokalnu povezanost ostvarujemo filtrima[3] F_i, naučenim matricama dimenzija $[F_W \times F_H]$ koje modeliraju težine povezanosti između dva sloja. Uglavnom su filtri kvadratne matrice pa možemo zapisati dimenziju filtra kao $[F \times F]$. Rezultat operacije je matrica O dimenzija $[W - F + 1 \times H - F + 1]$ definirana $\forall x \in [0, W - F], \forall y \in [0, H - F]$ izrazom:

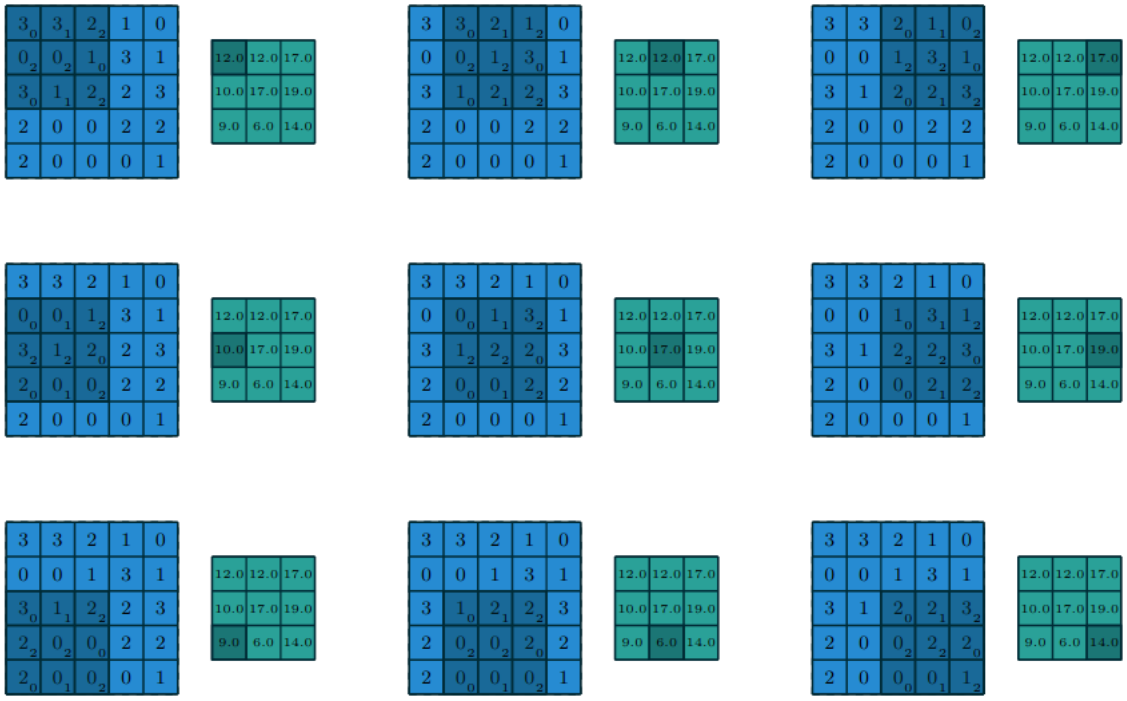
$$O_{x,y} = \sum_{i=x}^{x+F-1} \sum_{j=y}^{y+F-1} I_{i,j} \cdot F_{i-x,j-y} \tag{2.7}$$

Grafički primjer konvolucije dimenzije ulaza $[5 \times 5]$ uz filtar dimenzije $[3 \times 3]$:

0	1	2
2	2	0
0	1	2

Slika 2.3: Primjer filtra $[3 \times 3]$ [8]

[3]U stranoj literaturi često se za *filtar* koristi izraz *jezgra* (engl. *kernel*)



Slika 2.4: Demonstracija konvolucije [8]

Na slici 2.4 prikazan je primjer konvolucije opisan formulom 2.7. Filtar sa konkretnim vrijednostima prikazan je na slici 2.3.

Dimenzija filtra jedan je od hiperparametara konvolucijskog sloja, a uz nju moguće je definirati i korak (engl. *stride*) konvolucije kao i nadopunjavanje nulama (engl. *zero-padding*). Hiperparametri konvolucije utječu na dimenzije rezultantne matrice.

Korak S konvolucije govori nam za koliko mjesta *kližemo* filtar u horizontalnom ili vertikalnom smjeru. Moguće je definirati različite vrijednosti za pojedini smjer pomaka, ali u praksi se najčešće koriste jednake vrijednosti pomaka. Primijetite da će veći korak rezultirati manjim dimenzijama rezultata konvolucije.

Nadopunjavanje P konvolucije vrši se dodavanjem dodatnih elemenata vrijednosti 0. Ovaj postupak u praksi se najčešće koristi za manipulaciju veličine rezultantne matrice. Primjena P na matricu dimenzija $[x \times y]$ rezultirat će proširenom matricom dimenzija $[x + 2P \times y + 2P]$ čijih će prvih i zadnjih P redova i stupaca biti popunjeno nulama. Ukoliko želimo očuvati dimenziju ulaza, tj. da vrijedi $dim(I) = dim(O)$ tada postavljamo vrijednost P tako da vrijedi:

$$\begin{cases} \frac{W+2P-F}{S} = 1 \\ \frac{H+2P-F}{S} = 1 \end{cases} \tag{2.8}$$

ako je $S > 1$, odnosno tako da vrijedi:

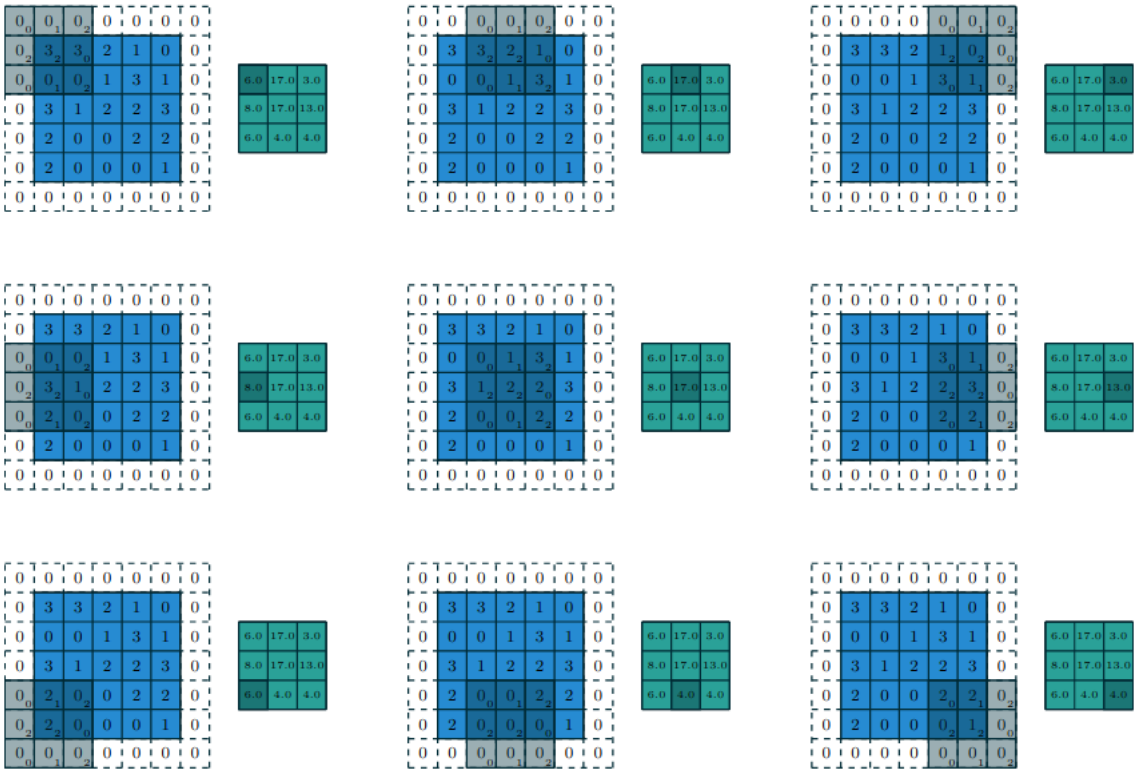
$$\begin{cases} 2P - F + 1 = 0 \\ 2P - F + 1 = 0 \end{cases} \tag{2.9}$$

ako je $S = 1$.

Općenitije, ukoliko želimo osigurati da korak i proširenje konvolucije ne preskoče određene redove i stupce, dovoljno je generalizirati formulu 2.8, tj. osigurati da vrijedi:

$$\begin{cases} (W + 2P - F)\%S = 0 \\ (H + 2P - F)\%S = 0. \end{cases} \tag{2.10}$$

Ovaj uvjet osigurava da su svi redovi i stupci ulazne matrice uključeni u izlaz. Na sljedećoj slici prikazana je konvolucija istovjetna primjeru 2.4 uz korak $S = 2$ i proširenje $P = 1$:



Slika 2.5: Demonstracija konvolucije ($S \neq 1, P \neq 0$) [8]

U ResNet arhitekuri koristi se dvodimenzionalna konvolucija koja operira nad tenzorom četvrtog reda dimenzija (B,C,H,W), gdje je B veličina grupe, C broj kanala, a H i W su visina, odnosno širina ulazne slike. Da pojasnimo, u ovom se slučaju koristi dvodimenzionalni filtri na niz trodimenzionalnih podataka. Zapravo, isti dvodimenzinalni filtar je *dijeljen* između svih kanala istovremeno. Postoji i opcija korištenja

trodimenzionalne konvolucije, ali primijetite da će se tada broj parametara konvolucijskog sloja povećati za faktor C.

3. Postupak učenja

U ovom poglavlju opisani su osnovni koncepti učenja neuronskih mreža, algoritam propagacije pogreške unatrag (engl. *error backpropagation*) i hiperparametri postupka učenja.

Učenje neuronske mreže podrazumijeva postupak kojim mreža mijenja vrijednosti težina i pomaka neurona kako bi mogla obaviti neki zadatak. Ako neuronske mreže promatramo iz perspektive teorije strojnog učenja, onda je jasno da težine i pomaci odgovaraju parametrima modela. Učenje neuronskih mreža je vrlo kompleksna tema i zato ćemo se ograničiti na učenje mreža za klasifikaciju slika koristeći nadzirano učenje.

Ideja učenja je prilagoditi parametre mreže na način da funkcija gubitka bude minimalna. Učenje se sastoji od više epoha. Tijekom jedne epohe mreža uči na čitavom skupu podataka za učenje. Svaka se epoha sastoji od iteracija. Nakon svake iteracije računa se iznos funkcije gubitka i na temelju nje se prilagođavaju parametri modela.

Svaka iteracija predaje mreži grupu (engl. *batch*) slika na obradu. Između slika u jednoj grupi ne radi se postupak prilagođavanja parametara, već se samo računa izlaz mreže s obzirom na slike. Odabir veličine grupe jedan je od hiperparametara mreže i direktno utječe na vrijeme potrebno za učenje, ali i na točnost modela.

3.1. Funkcija gubitka

Uloga funkcije gubitka jest odrediti pogrešku predikcije mreže. Pogreška se računa na temelju dobivenog i željenog izlaza (oznake). Funkcija gubitka računa se za vrijeme postupka učenja nakon svake predikcije, a akumulira se tijekom predikcije jedne grupe. Nakon grupne predikcije računa se prosječna pogreška s obzirom na varijable izlaza.

Odabir funkcije gubitka kojeg ćemo koristiti može utjecati na konačan ishod učenja. Za potrebe treniranja mreža u ovom radu koristi se gubitak unakrsne entropije (engl. *cross entropy loss*). Gubitak unaktrsne entropije računa se prema 3.1. Da bismo razumjeli ovu jednadžbu potrebno je uvesti oznaku M koja predstavlja naš mo-

del, oznaku C koja predstavlja broj mogućih klasa za problem klasifikacije, oznaku D koji predstavlja skup označenih podataka (engl. *dataset*) oblika: $[(x_0, y_0), ..., (x_n, y_n)]$, gdje x_i predstavlja ulazni podatak, a y_i pripadajuću ispravnu oznaku. U modelima za klasifikaciju ulazi su slike predstavljene kao tenzor bitova, a oznake predstavljaju C-dimenzionalni vektori gdje je i-ti element postavljen na vrijednost 1, a svi ostali elementi su 0. Upravo taj i-ti element predstavlja označenu klasu. Ovakav način kodiranja vrijednosti u stranoj literaturi naziva se *one-hot encoding*. Konačno, N koji označava broj podataka u D.

$$\text{CE}(M, D) = \frac{1}{N} \sum_{i=0}^{N} -\log\left(P(Y = y_i|x_i)\right) \tag{3.1}$$

Izraz koji sumiramo zapravo predstavlja pogrešku predikcije jedne slike. Izraz $P(Y = y_i|x_i)$ je aposteriorna vjerojatnost da se ulaz x_i ispravno klasificirao u klasu y_i. Da bismo dobili ove vjerojatnosti potrebno je razmotriti izlaz modela.

Označimo slovom O izlazni vektor modela. To je C-dimenzionalni vektor i sadrži tzv. klasifikacijske bodove pojedinih razreda. Ovaj vektor možemo pretvoriti u vjerojatnost koristeći funkciju *softmax* koja je definirana kao:

$$softmax(O, i) = \frac{e^{-O_i}}{\sum_{i=0}^{C-1} e^{-O_i}}, \tag{3.2}$$

$\forall i \in [0, C-1]$. Primijetite da će ova funkcija uvijek biti > 0, ali nikada neće prelaziti vrijednost 1 jer je brojnik dio sume u nazivniku. Također, trivijalno je dokazati da vrijedi $\sum_{i=1}^{C} softmax(O, i) = 1$ i $P(y_{i=k} \cup y_{i\neq k}) = P(y_{i=k}) \cup P(y_{i\neq k})$ jer su klase međusobno disjunktne. Zadovoljeni su aksiomi vjerojatnosti, a to znači da je izlazni vektor funkcije 3.2 vjerojatnost.

Sada se funkcija 3.1 može izračunati jer su nam poznate aposteriorne vjerojatnosti pojedinih klasa na za svaki izlaz modela. Primijetite da će unakrsna entropija iznositi 0 ako su svi primjeri iz grupe ispravno predviđeni.

3.2. Gradijentni spust

Cilj stohastičkog gradijentnog spusta je minimizirati funkciju gubitka. Funkcija gubitka je funkcija svih parametara mreže, pa je shodno tome gradijent funkcije gubitka vektor parcijalnih derivacija po svim parametrima. Linearne pomake modela promatrati ćemo kao težine. Tada su parametri modela samo težine w_i. Broj težina označiti ćemo slovom N. Gradijent funkcije gubitka L sada možemo izraziti kao:

$$\nabla L = [\frac{\partial L}{\partial w_1}, ..., \frac{\partial L}{\partial w_N}]^T. \tag{3.3}$$

Promjenu vrijednosti funkcije gubitka sada možemo aproksimirati kao:

$$\begin{aligned} \Delta L &\approx \sum_{i=0}^{N} \frac{\partial L}{\partial w_i} \cdot \Delta w_i \\ &\approx \Delta w \cdot \nabla L, \end{aligned} \tag{3.4}$$

gdje je Δw vektor pomaka pojedinih težina.

Znamo da gradijent funkcije pokazuje smjer u kojem ta funkcija najbrže raste. Shodno tome, mi želimo mijenjati težine u smjeru u kojem će funkcija najbrže padati, tj. suprotno od gradijenta. Jasnije rečeno, želimo da izraz 3.4 poprima negativnu vrijednost za svaku našu promjenu težina. Vektor pomaka težina tada je prikladno definirati na sljedeći način:

$$\Delta w = -\eta \nabla L. \tag{3.5}$$

Nadalje, sada aproksimacija promjene vrijednosti funkcije gubitka iznosi:

$$\Delta L \approx -\eta \|\nabla L\|^2. \tag{3.6}$$

Znamo da je norma vektora uvijek pozitivna i stoga ćemo uz pozitivnu konstantu η osigurati da se vrijednost funkcije L uvijek smanjuje.

Konstantu η nazivamo *stopa učenja* jer upravo ovaj hiperparametar modela određuje iznos prilagođavanja težina. Bitno je da bude postavljen na prikladne vrijednosti. Ukoliko je η prevelika, postupak učenja ne mora konvergirati i moguće je da model uopće neće učiti. Primijetite, ako je konstanta η prevelika više neće vrijediti tvrdnja da će se funkcija L uvijek smanjivati naprosto zbog loše aproksimacije. S druge strane, ako je stopa učenja premala tada će učenje trajati predugo. Stopa učenja obično je mali pozitivni broj reda veličine 10^{-2}.

Konačno, dolazimo do zaključka da ako težine modela mijenjamo prema izrazu 3.5 tada bi uz prikladnu konstantu η vrijednost funkcije gubitka L trebala konvergirati prema 0. Svaka težina modela prilikom gradijentnog spusta promijeniti će se prema izrazu:

$$\vec{w}' = \vec{w} - \eta \nabla L. \tag{3.7}$$

Spomenimo da je ovo temeljni algoritam gradijentnog spusta, a danas postoje mnoge tehnike kojima se ovaj postupak može poboljšati.

3.3. Algoritam propagacije pogreške unatrag

Propagacija pogreške unatrag je algoritam za učenje neuronskih mreža koji koristi gradijentni spust za podešavanje parametara modela.

Prema izrazu 3.3 vidimo da je gradijent funkcije gubitka zapravo funkcija svih parametara modela. Za primjenu u slojevitim neuronskim mrežama potrebno je parametre podijeliti po slojevima. Redni broj sloja u mreži označit ćemo sa l, a redni broj aktivacije u tom sloju označit ćemo sa i. Također, slovom j označit ćemo redni broj aktivacije u sloju $l-1$ s kojom je povezana i-ta aktivacija u l-tom sloju. Formalno, takve težine zapisujemo u obliku $w_{i,j}^{(l)}$. Također, u vektorskom obliku ovo možemo zapisati kao $\vec{w}_i^{(l)}$. Tako zapisan vektor je j-dimenzionalan, gdje j predstavlja broj aktivacija u prošlom spoju koje težina povezuje sa i-tom aktivacijom l-tog sloja.

Definirajmo $z_j^{(l)}$ i $y_j^{(l)}$ kao izlaz j-tog elementa l-tog sloja prije aktivacije, odnosno kao vrijednost aktivacijske funkcije na $z_j^{(l)}$. Sada izraz 2.2 možemo poopćiti i izlaz j-te aktivacije u l-tom sloju zapisati na sljedeći način:

$$y_j^{(l)} = \mathrm{f}(z_j^{(l)}), \tag{3.8}$$

gdje f predstavlja aktivacijsku funkciju, a $z_j^{(l)}$ predstavlja umnožak vektora težina j-tog elementa sloja l i vektora izlaza povezanih elemenata sloja $l-1$.

Primijetite da aktivacije sloja l ovise o svim prijašnjim parametrima s kojima su direktno ili indirektno povezani. Dakle, imamo situaciju u kojoj trebamo izračunati parcijalnu derivaciju funkcije gubitka u ovisnosti o parametru $w_{i,j}^{(l)}$, koji ovisi o parametrima $\vec{w}_i^{(l-1)}$ s kojima je povezan, koji opet ovise o parametrima ranijeg sloja i tako sve do ulaznog sloja. Naivni pristup računanju ovakvih izraza bio bi neučinkovit jer bi lanci derivacija bili veliki, a veliki broj operacija bi se ponavljao. Algoritam propagiranja pogreške unatrag omogućuje nam da se računanje potrebnih parcijalnih derivacija izvršava u linearnom vremenu.

Za provedbu algoritma prvo je potrebno odrediti pogreške izlaznog sloja prema izrazu:

$$\begin{aligned} \delta_j^{(l)} &= \frac{\partial L}{\partial z_j^{(l)}} \\ &= \frac{\partial L}{\partial y_j^{(l)}} \cdot \frac{\partial y_j^{(l)}}{\partial z_j^{(l)}} \\ &= \frac{\partial L}{\partial y_j^{(l)}} \cdot \mathrm{f}'(z_j^{(l)}). \end{aligned} \tag{3.9}$$

Pogrešku i-tog neurona u l-tom sloju mreže računamo kao umnožak derivacije aktivacijske funkcije tog neurona i težinske sume pogrešaka neurona kojima on šalje svoj izlaz [22]. Dakle, pogreške nekog sloja možemo izraziti pomoću pogrešaka sljedećeg sloja. Sada možemo zapisati preoblikovati izraz 3.9 u:

$$\delta_j^{(l)} = \mathrm{f}'(z_j^{(l)}) \odot (w^{l+1})^T \delta^{l+1}, \tag{3.10}$$

gdje $\odot$ predstavlja hadamardov produkt nad dva vektora ($(a \odot b)_j = (a_j \cdot b_j)$). Primijetite da ćemo prilikom računanja pogrešaka slojeva morati slojeve obilaziti od kraja prema početku.

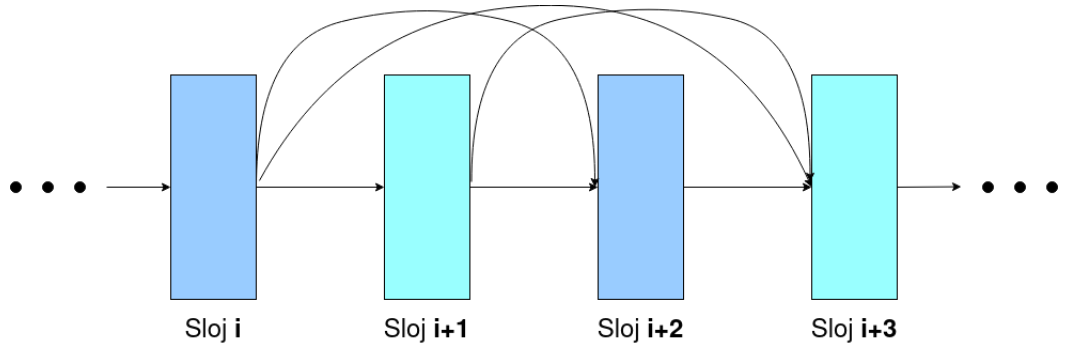
Konačno, sada možemo izračunati nove vrijednosti parametara modela prema izrazu:

$$\begin{aligned} w_{i,j}^{(l)\prime} &= w_{i,j}^{(l)} - \eta \cdot \frac{\partial L}{\partial w_{i,j}^{(l+1)}} \\ &= w_{i,j}^{(l)} - \eta \cdot y_i^{(l)} \cdot \delta_j^{(l+1)}. \end{aligned} \tag{3.11}$$

Vrijednosti $z_i^{(l)}$ i $y_i^{(l)}$ već su izračunate tijekom evaluacije modela rezultata i te se vrijednosti privremeno spremaju kako bi se ponovno iskoristile za računanje gradijenta.

4. Gusto povezani konvolucijski modeli

Gusto povezani duboki konvolucijski modeli trenutno postižu *state-of-the-art* rezultate u području klasifikacije slika. Ovakvi modeli uvode novi način povezivanja slojeva koristeći tzv. preskočne veze (engl. *skip-connections*). Na ovaj način, izlaz nekog sloja mreže moguće je povezati na nekoliko različitih slojeva. Zabranjene su povratne veze (iz sloja l u sloj $k, k < l$). Ovime smo osigurali da ne izađemo iz domene unaprijednih neuronskih mreža. Konceptualna shema gusto povezanih modela na vrlo visokoj razini apstrakcije može izgledati ovako:



Slika 4.1: Shema gusto povezanog modela.

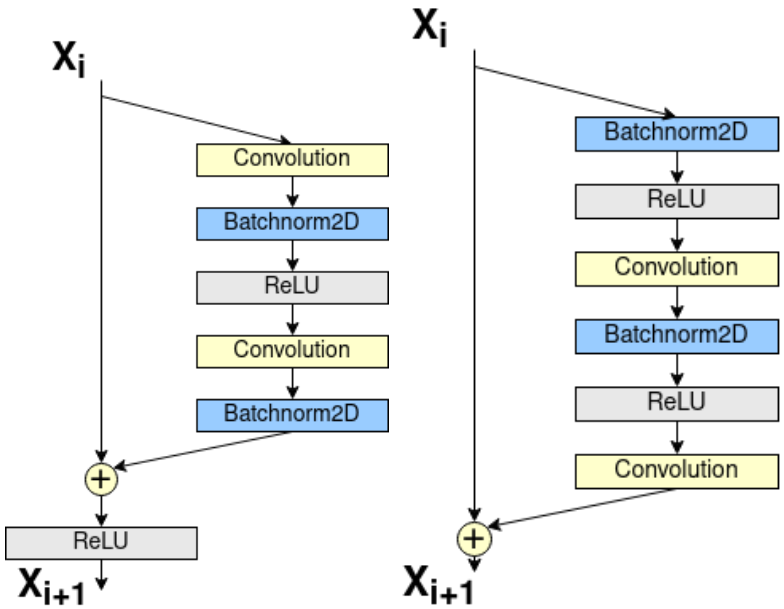
Ovakve arhitekture pospješuju protok gradijenta tijekom učenja. Prema [12] mreža koja se sastojala od ≈ 19.4 milijuna parametara u 1202 sloja uspješno je naučila klasificirati na skupu podataka CIFAR-10 sa pogreškom od 7.93% na skupu za testiranje.

Primijetite da neki sloj i ima jednak broj težina kao i u običnoj mreži. Razlog tome jest taj da izlazi prethodnih slojeva ne ulaze u i-ti sloj svakih zasebno već se spajaju na određeni način. Način spajanja izlaza slojeva koji su povezani na sloj i ovisi o konkretnoj arhitekturi.

Među popularnijim gusto povezanim arhitekturama danas su DenseNet i ResNet. Obje arhitekture sastoje se od povezanih blokova koji se slažu u dubinu do željenog broja slojeva. Razlika između ove dvije dvije mreže upravo su sastavni blokovi, tj. međusobna povezanost slojeva u takvim blokovima.

DenseNet definira blokove koji se interno sastoje od L slojeva. Sloj jednog bloka zapravo su dvije ulančane konvolucije dimenzija $[1 \times 1]$ i $[3 \times 3]$. Svaki se sloj povezuje sa svim slojevima unutar istog bloka koji su ispred njega. Na taj način, u svakom bloku nastaje $\frac{L \cdot (L+1)}{2}$ veza između slojeva. Spomenimo kako se spajanje izlaza slojeva vrši konkatenacijom istih u dubinu.

ResNet svoje blokove definira nešto jednostavnije. Svaki blok sastoji se od konačnog broja rezidualnih jedinica. Razvijene su dvije vrste rezidualnih jedinica. Prvu vrstu (slika 4.2a) nazivamo *originalnom* rezidualnom jedinicom jer se predložena zajedno sa samom ResNet arhitekturom u [12]. Drugu vrstu (slika 4.2b) nazivamo rezidualnom jedinicom identiteta jer dobiveni ulaz ne provlači kroz aktivacijsku funkciju, već se direktno prenosi na izlazno zbrajalo. Prema [13] ResNet1001 mreža je koristeći rezidualnu jedinicu identiteta smanjila pogrešku na skupu za testiranje za 2.69 postotnih bodova. Spajanje izlaza je u ovom slučaju trivijalno jer imamo samo 2 sloja koji se spajaju zbrajanjem. Pritom je bitno da slojevi preko kojih se proteže preskočna veza zadrže dimenzionalnost ulaza kako bi se na kraju mogli zbrojiti.



(a) Originalna rezidualna jedinica **(b)** Rezidualna jedinica identiteta

Slika 4.2: Rezidualna jedinice.

5. Kvantizacija

Kvantizacija je matematički postupak kojim pretvaramo realne vrijednosti u cjelobrojne ili u vrijednosti ograničene preciznosti. U ovom poglavlju opisana je motivacija primjene ovog postupka, kvantizacijska funkcija i pogreška kvantizacije.

5.1. Zašto kvantizacija?

Kvantizaciju u računarstvu općenito koristimo za pohranu numeričkih vrijednosti. Za spremanje decimalnih brojeva u računalnu memoriju koristi se međunarodni standard IEEE-754.[1] Ovim je standardom specificiran postupak pretvorbe cijelih brojeva u realne i obrnuto, kao i odgovarajuće računske operacije. Spomenute postupke pretvorbe možemo također promatrati kao kvantizaciju. Također, ovaj postupak je nužan za pohranu iracionalnih brojeva (npr. π, e).

Neuronske mreže izrazito su zahtjevne u vidu memorije i računanja. Modeli koji predstavljaju stanje tehnike koristili su nekoliko desetaka milijuna parametara, dok je za zaključivanje bilo potrebno nekoliko desetaka milijardi FLOP-a (engl. *floating point operation*)[5]. Potreba za većim modelima u stalnom je porastu i to nas dovodi do korištenja postupka kvantizacije u neuronskim mrežama. Ukratko, ideja je zamijeniti decimalne brojeve prirodnim brojevima kako bi se smanjila potreba za memorijom i omogućile učinkovitije računske operacije. Pritom je potrebno na odgovarajući način provesti samu kvantizaciju kako bi mreža izgubila što manje točnosti.

5.2. Kvantizacija u radnom okviru PyTorch

PyTorch je radni okvir otvorenog koda (engl. *open-source framework*) koji se koristi za razvoj modela strojnog učenja. Njegove glavne funkcionalnosti su automatsko diferenciranje i transparentno izvođenje na NVidia i AMD grafičkim karticama. Postoje

[1] https://ieeexplore.ieee.org/document/4610935

službene verzije radnog okvira za jezike C++, Java i za Python koji će se koristiti u ovom radu. Glavna prednost PyTorcha je izvrsna organizacija koja pogoduje brzom prototipiranju.

Za potrebe eksperimentalne provedbe koristiti ćemo modificirani izvorni kod mrežne arhitekture ResNet. Kao ulazne podatke koristiti ćemo skup slika CIFAR-10 koji sadrži 60000 ispravno označenih slika dimenzija 32×32 piksela podijeljenih u 10 klasa. Skup podataka javno je dostupan[2], a za preuzimanje direktno iz Python skripte moguće je koristiti i gotov modul *datasets* iz paketa *torchvision*.

Sve mrežne arhitekure u okviru PyTorch koriste tip FP32 za pohranu decimalnih vrijednosti. Pomoću kvantizacije ćemo sve ili barem neke od operanada prebaciti u INT8 oblik koji koristi 4 puta manje memorije, a procjenjuje se da su operacije nad tenzorima cijelih brojeva okvirno 2 do 4 puta brže od operacija nad tenzorima decimalnih brojeva. Stvarno ubrzanje ovisit će o uređaju na kojem izvodimo zaključivanje.

Kvanitizacija u PyTorch-u u trenutku pisanja još je u beta fazi razvoja i u ovom trenutku podržava samo kvantizaciju na ARM i x86, tj. dinamička i statička kvantizacija nisu moguće na grafičkim procesorima, dok se modeli koji tijekom učenja simuliraju kvantizaciju mogu prebaciti na grafički procesor koristeći CUDA API. Dinamička i statička kvantizacija mogu se prebaciti na grafički procesor koristeći TensorRT API.

U ovom radu razmotrit će se dinamička, statička i naučena kvantizacija (engl. *Quantization Aware Training, QAT*). Sva tri oblika kvantiziranja obavljaju se nakon treninga mreže.

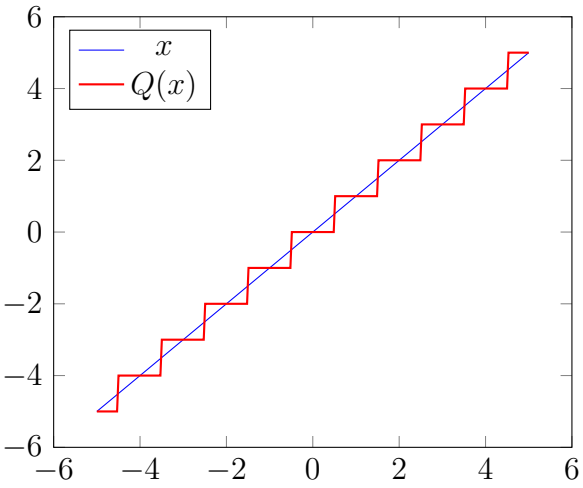
5.2.1. Kvantizacijska funkcija

Za potrebe primjera definirajmo jednostavnu kvantizacijsku funkciju $Q : \mathbb{R} \to \mathbb{Z}$ na sljedeći način:

$$Q = \left\lfloor x + \frac{1}{2} \right\rfloor \tag{5.1}$$

Ovako definirana funkcija zapravo je funkcija zaokruživanja brojeva na najbliže cijelo. Grafički prikaz izgleda ovako:

[2]https://www.cs.toronto.edu/ kriz/cifar.html

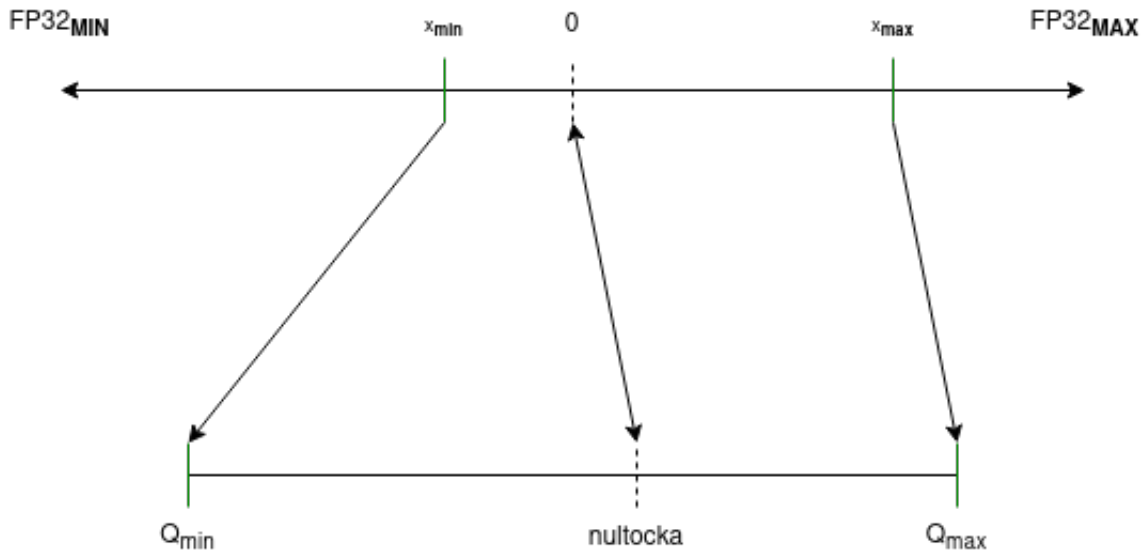


Slika 5.1: Grafički prikaz funkcije. 5.1

Iz grafičkog prikaza jednostavno možemo utvrditi da Q nije injektivna. Budući da funkcija za više različitih ulaza može dati isti izlaz, provođenje kvantizacije rezultirati će gubitkom dijela informacije o stvarnoj vrijednosti naučenih parametara modela. Gubitak informacije konkretno nastaje zaokruživanjem decimalne vrijednosti, a rezultirat će tzv. kvantizacijskom pogreškom (koristi se još i naziv kvantizacijski šum) koja iznosi upravo onoliko koliko smo oduzeli zaokruživanjem.

Funkcije poput funkcije 5.1 često nam nisu od velike koristi u praksi. Kako bi izvukli maksimalnu dobit od postupka kvantizacije potrebno je ograničiti raspon brojeva koji čine kodomenu kvantizacijske funkcije. U tom slučaju moramo također ograničiti i ulazne vrijednosti.

Pokušajmo sada konstruirati funkciju koja ima opisana svojstva. Mapiranje sa ograničenim rasponom i proizvoljnim pomakom grafički je prikazano na slici 5.2.



Slika 5.2: Model kvantizacijske funkcije.

Promotrimo sliku 5.2. Vrijednosti x_{min} i x_{max} predstavljaju minimalnu odnosno maksimalnu vrijednost u tenzoru kojeg kvantiziramo. Q_{min} i Q_{max} predstavljaju raspon kodomene kvantizacijske funkcije koji će iznositi $[-2^{B-1}, 2^{B-1} - 1]$ za predznačene (engl. *signed*) tipove, odnosno $[0, 2^B - 1]$ za tipove bez predznaka (engl. *unsigned*). Definirajmo konstantu $B = 8$ koja predstavlja broj bitova potreban za prikaz kodomene kvantizacijske funkcije. Za pohranu cjelobrojnih vrijednosti možemo koristiti tipove sa i bez predznaka. Konačno, definirajmo sada varijablu $faktor$:

$$faktor = \frac{Q_{max} - Q_{min}}{x_{max} - x_{min}}. \tag{5.2}$$

Vrijedi spomenuti kako je poželjno da raspon ulaznih vrijednosti $x_{max} - x_{min}$ bude čim manji kako bi i sama kvantizacijska greška bila manja. Zapišimo sada kvantizacijsku funkciju Q^1 kao:

$$Q^1(x) = \text{zaokruzi}(x \cdot faktor). \tag{5.3}$$

Međutim, ovakva funkcija daje željeni rezultat samo ako je raspon ulaznih vrijednosti simetričan, što često nije slučaj. Kako bismo riješili ovaj problem, u funkciju dodajemo linearni pomak $nultocka$:

$$nultocka = Q_{min} - \text{zaokruzi}(x_{min} * factor) \tag{5.4}$$

$$Q^2(x) = \text{zaokruzi}(x * factor + nultocka). \tag{5.5}$$

Primijetite da je tako definiran linearni pomak uvijek cjelobrojan i govori nam u koju će se vrijednost preslikati 0. Zbog toga ne nastaje kvantizacijska greška prilikom kvantiziranja broja 0:

$$\begin{aligned} Q(0) &= \text{zaokruzi}(0 * faktor + nultocka) \\ &= \text{zaokruzi}(nultocka) \\ &= nultocka. \end{aligned} \tag{5.6}$$

Kvantizacijsku pogrešku želimo izbjeći specifično za broj 0 zato što se u konvolucijskim slojevima često koristi operacija dopunjavanja (engl. *padding*) matrica upravo brojem 0. Ukoliko bi se kvantizacijska pogreška dogodila prilikom kvantiziranja nadopune, semantika modela mogla bi biti izmijenjena.

Na kraju, potrebno je definirati i ponašanje funkcije kada se na ulazu u kvantizacijsku funkciju pojavi x za kojeg vrijedi $x < x_{min} || x > x_{max}$. Ovaj slučaj je nemoguć kod dinamičke kvantizacije jer se ulazi u dinamički kvantizirani sloj kvantiziraju prilikom izvođenja pa je ulazni vektor upravo onaj prema kojemo određujemo vrijednosti x_{min} i x_{min}. Zbog toga, u dinamičkoj kvantizaciji dovoljno je koristiti funkciju 5.5. U slučaju statičke i naučene kvantizacije ovo teoretski može biti slučaj jer se parametri određuju na temelju vrijednosti iz skupa za treniranje. Ulazne vrijednosti u tom slučaju nisu unaprijed poznate i moramo uzeti u obzir moguća odstupanja od skupa za treniranje. Primijetite, ukoliko bi se na ulazu pojavila vrijednost koja je izvan definiranog raspona, kvantizacijska funkcija dala bi rezultat koji nije moguće prikazati sa B bitova i dogodio bi se preljev (engl. *overflow*). U svrhu rješavanja tog problema definirat ćemo sljedeću funkciju:

$$\text{ogranici}(x, min, max) = \begin{cases} min, & x < min \\ max, & x > max \\ x, & min \leq x \leq max \end{cases} \tag{5.7}$$

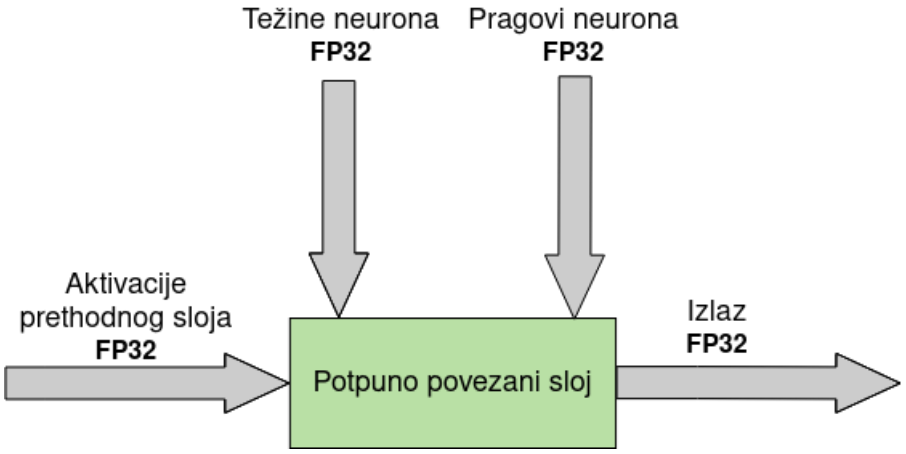
Kvantiziranje vrijednosti izvan definiranog raspona rezultirati će nešto većom greškom, ali to je još uvijek puno bolje nego da se dogodi preljev. Završno, našu kvantizacijsku funkciju $Q : \mathbb{R} \to [Q_{min}, Q_{max}]$ možemo zapisati kao:

$$Q(x) = \text{ogranici}(\text{zaokruzi}(x * factor + nultocka), Q_{min}, Q_{max}) \tag{5.8}$$

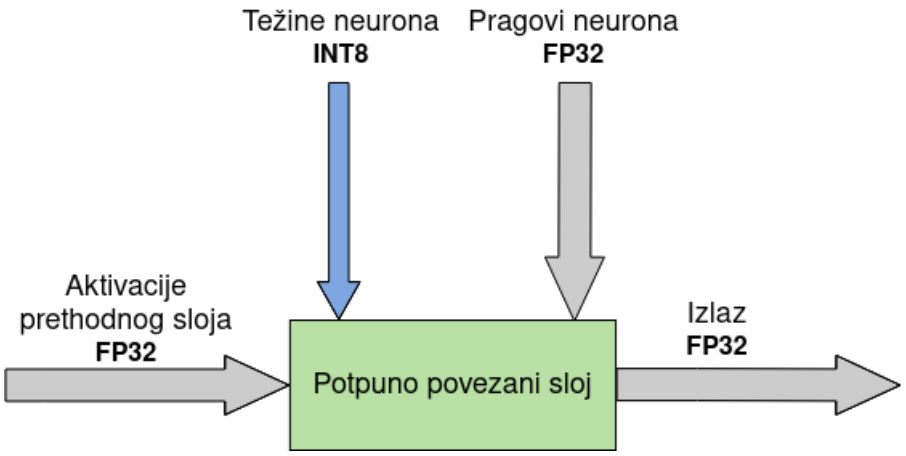
Ovakva se funkcija koristi za kvantiziranje tenzora u radnom okviru PyTorch. Spomenimo kako PyTorch kod višekanalnih ulaza podržava kvantiziranje svih dimenzija istim parametrima ((engl. *per-tensor*)) i kvantiziranje svake dimenzije zasebnim parametrima ((engl. *per-channel*)) koje obično rezultira boljom točnošću modela.

5.2.2. Dinamička kvantizacija

Najjednostavniji oblik kvantizacije u PyTorch-u je dinamička kvantizacija. Ideja je kvantizirati težine modela prije, a aktivacije tijekom vremena izvođenja. Memorijsko zauzeće težina svakog dinamički kvantiziranog sloja smanjit će se $\times 4$. Potrebno je za svaki sloj koji želimo dinamički kvantizirati odrediti parametre kvantizacije $faktor$ i $nultocka$ i na kraju kvantizirati težine tog sloja. Pogledajmo razliku između izvornog i dinamički kvantiziranog potpuno povezanog sloja:



(a) Prije

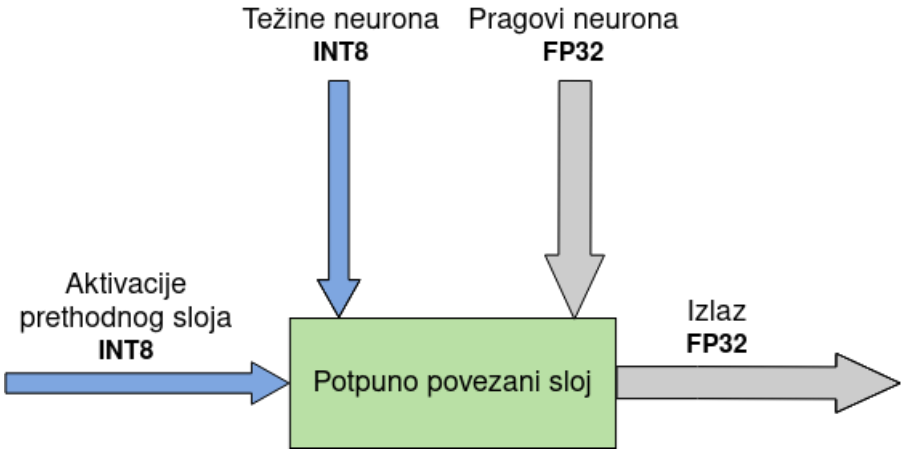


(b) Poslije

Slika 5.3: Učinak dinamičke kvantizacije.

Na slici 5.3a prikazan je model običnog nekvantiziranog sloja kakvog susrećemo u praksi. Vidljivo je da su parametri modela pohranjeni u FP32 preciznosti. Na slici 5.3b plavom bojom označene su težine sloja koje su dinamički kvantizirale u INT8 format. Primijetite da su kvantizirane samo težine sloja što nam još uvijek ne omogućuje efikasno u INT8 preciznosti. Dakle, kako bi se postiglo ubrzanje potrebno je kvantizirati

i ulaze pojedinih slojeva, a to jedino možemo učiniti prilikom izvođenja, jer za razliku od težina, ulazi u mrežu nisu unaprijed poznati. U dinamički kvantiziranim mrežama kvantiziraju se one aktivacije slojeva iza kojih neposredno slijedi dinamički kvantiziran sloj. Kvantizirani ulazi u sloj omogućuju množenje matrica u INT8 aritmetici, što će, ovisno o arhitekturi i podsustavu za matričnu aritmetiku, značajno ubrzati matrično množenje. Na slici 5.4 prikazan je dinamički kvantiziran potpuno povezani sloj neposredno prije računanja njegovog izlaza. Uočite da je tada omogućena INT8 aritmetika za množenje težina neurona i aktivacija prethodnog sloja.



Slika 5.4: Dinamički kvantizirani sloj u izvođenju

Važno je primijetiti kako kvantizacija ulaza dinamičkih slojeva troši određeno vrijeme. Naime, potrebno je za svaki ulazni tenzor koji se kvantizira pronaći minimalnu i maksimalnu vrijednost (kompleksnost O(n)) i izračunati *faktor* i *nultocka* što može biti poprilično skupa operacija. Na kraju, kvantizaciju treba i izvesti te definitivno ne možemo zanemariti vremenski trag kvantiziranja za vrijeme izvođenja.

5.2.3. Statička kvantizacija

Statička kvantizacija uz težine kvantizira i aktivacije slojeva. Kao što je već spomenuto, aktivacije modela ovise o njegovom ulazu i nisu nam unaprijed poznate. Kako onda možemo kvantizirati nešto što nam je nepoznato? Odgovor leži u postupku kalibracije mreže. Naime, pretpostavljamo da će skup za treniranje mreže sadržavati podatke koji su vrlo slični onima na kojima će mreža provoditi zaključivanje. Nakon treninga mreže u računski graf dodaju se promatrači (engl. *observers*) koji će tijekom faze kalibracije promatrati vrijednosti koje se pojavljuju na izlazima pojedinih aktivacija. U najjednostavnijem slučaju promatrači će tražiti samo minimalne i maksimalne

vrijednosti pojedinih slojeva, ali postoje i složeniji promatrači koji određuju parametre na drugačiji način. Nakon što smo u model dodali promatrače kroz mrežu jednom propustimo cijeli skup za treniranje kako bi promatrači izračunali potrebne parametre za kvantizaciju. Nakon kalibracije, model pretvaramo u kvantizirani model na način da su sve težine kvantizirane, a aktivacije se kvantiziraju tijekom izvođenja ali uz prethodno poznate parametre *faktor* i *nultočka* koje su promatrači spremili u svaki od slojeva.

Statička kvantizacija ne provodi kvantizaciju tijekom zaključivanja što cjelokupni postupak čini znatno bržim. Također, za razliku od dinamičke kvantizacije gdje je bilo moguće kvantizirati samo potpuno povezane slojeve, kod statičke kvantizacije nemamo ograničenja na vrstu podržanih slojeva, tj. svi slojevi imaju podržanu vlastitu INT8 inačicu. Primijetite da će se kvantizirati sve težine pa će se veličina modela smanjiti $\times 4$.

Glavni problem statičke kvantizacije je taj da moramo raditi intervenciju u programskom kodu mreže koju kvantiziramo. Točnije, potrebno je umetnuti po jedan pomoćni sloj za kvantiziranje na ulazu i dekvantiziranje na izlazu mreže. Također, kao nedostatak ove metode možemo smatrati i potrebnu kalibraciju na skupu za treniranje.

5.2.4. Naučena kvantizacija

Naučena kvantizacija funkcionira slično statičkoj kvantizaciji, osim što ne zahtjeva posebnu kalibraciju i u praksi obično rezultira manjim gubitkom točnosti. Ova metoda podrazumijeva dodavanje "promatrača" u model prije postupka treniranja. Ideja je simulirati učinke kvantizacije, ali operacije izvršavati u FP32 aritmetici kako bi zadržala točnost za vrijeme treniranja. Ovo možemo simulirati na način da svaku aktivaciju (osim izlazne) provućemo kroz lažnu kvantizaciju (engl. *fake quantization*). Za potrebe pojašnjenja postupka lažne kvantizacije definirajmo funkciju dekvantizacije $D(x)$ na sljedeći način:

$$D(X) = \frac{x - nultocka}{faktor} \tag{5.9}$$

Lažna kvantizacija kvantizirat će aktivaciju, obavijestiti o tome određene promatrače dekvantizirati vrijednost natrag u FP32 oblik, tj. poziva se funkcija $D(Q(x))$. Primijetite da ne vrijedi sljedeća relacija: $\forall x(D(Q(x)) = x)$, tj. da funkcija 5.9 ne vraća točnu vrijednost koju smo imali na ulazu u lažnu kvantizaciju. Ovim postupkom modelu pružamo priliku da se prilagodi na kvantizacijski šum. Glavna prednost naučene kvantizacije je da ne zahtjeva zasebnu kalibraciju modela, već se ona vrši samim treningom. Ukoliko se prilikom treninga koriste transformacije za uvećanje ulaznog skupa podataka tada ova metoda kvantizacije može rezultirati boljom točnošću od statičke

kvantizacije jer se kalibracija vrši na većem skupu ulaznih vrijednosti.

Nakon treninga model se pretvara u kvantizirani model i koristi se na identičan način kao i statički kvantizirani model.

5.3. Kvantizacija na grafičkom procesoru

PyTorch trenutno podržava samo izvođenje modela naučene kvantizacije na grafičkom procesoru koristeći CUDA API, a ne podržava dinamičku i statičku kvantizaciju.

5.3.1. ONNX

Open Neural Network Exchange je standardni format modela strojnog učenja koji omogućuje interoperabilnost između brojnih radnih okvira za razvoj neuronskih mreža. Uz tako definiran format potrebno je svakom od podržanih radnih okvira definirati postupak pretvorbe modela u ONNX format, ali i postupak pretvorbe iz ONNX-a u model kompatibilan sa radnim okvirom. Na taj su način razvojni inženjeri lišeni okova korištenja jednog radnog okvira i moguće je kombinirati mogućnosti različitih radnih okvira koje ONNX podržava. ONNX API dio je PyTorch radnog okvira (modul torch.onnx) stoga nije potrebna posebna instalacija.

Naš model neuronske mreže pretvorit ćemo u ONNX format kako bismo zaključivanje mreže mogli pokrenuti u TensorRT izvršnom okruženju.

5.3.2. TensorRT

TensorRT je skup razvojnih alata (engl. *software development kit, SDK*) koji su fokusirani na optimiranje postupka zaključivanja dubokih neuronskih mreža. Cilj je učitati ONNX model, optimizirati računski graf i pokrenuti zaključivanje u izvršnom okruženju TensorRT. Optimiranje računskog grafa TensorRT provodi prilikom gradnje izvršnog okruženja, a u ovom postupku provodi se fuzija slojeva i odbacivanje, reorganizacija ili kombiniranje računskih operacija. Također, provodi se i prijevremena evaluacija konstanti (analogno *constexpr* direktivi iz C++11). Izrada optimizacijskog plana traje i do nekoliko minuta, ali je omogućena njegova serijalizacija bi ga mogli koristiti prilikom višestrukog pokretanja izvršnog okruženja. Spomenimo kako optimizacijski plan specifičan s obzirom na platformu, modela grafičkih procesora i verzije TensorRT-a, tj. nije portabilan.

Izvršno okruženje TensorRT možemo pokrenuti u koristeći njegov C++ ili Python API, ali i zasebno iz komandne linije što ga čini pogodnim za izvedbu u praksi.

6. Eksperimentalni rezultati

Cilj eksperimenta je usporediti brzinu i točnost kvantiziranih modela u odnosu na referentni FP32 model. Također, izvođenjem u TensorRT izvršnom okruženju pokazuje se učinak optimizacije računskog grafa. Za provedbu koristi se mreža ResNet18 koja rješava problem klasifikacije slika na skupu podataka CIFAR-10.

Postupak učenja mreže proveli smo za referentni model te smo naučene parametre spremili kako bismo ih koristili prilikom instanciranja modela ostalih modela. Model naučene kvantizacije trenirali smo ispočetka kako bi promatrači čim točnije izračunali potrebne parametre.

Za potrebe kvantizacije koristili smo *MinMax* promatrač koji računa *faktor* i *nul-tocka* prema 5.2, odnosno 5.6. Trenutno je podržano kvantiziranje aktivacija u tip *torch.quint8* i kvantiziranje težina u *torch.qint8*.

U mjerenjima smo simulirali situaciju iz prakse postavljanjem veličine grupe na 1 (svaka slika evaluira se zasebno). Prilikom mjerenja koristimo tehniku *zagrijavanja* mreže kako bi se proveli eventualni inicijalizacijski postupci na GPU koji znaju potrajati i do nekoliko sekundi. Te postupke potrebno je obaviti kad je GPU u stanju niske potrošnje energije. Ovakve postupke ne želimo mjeriti te zatim prije svakog mjerenja kroz mrežu provućemo nekoliko nasumičnih ulaza kako bismo simulirali opterećenje.

Svako mjerenje izvedeno je u 10000 iteracija, a kao rezultat uzimamo prosječno vrijeme izvođenja i ukupnu akumuliranu točnost. Rezultati svih mreža dobivenu su na temelju jednakih slika kako bismo mogli međusobno uspoređivati točnosti.

Algoritam za mjerenje vremena zadan je sljedećim pseudokodom:

Algoritam 1 Mjerenje prosječnog vremena zaključivanja i točnosti

Ulaz: T – skup podataka za testiranje.
Izlaz: t, točno – prosječno trajanje zaključivanja i točnost.
točno, t := 0
ukupno := duljina(T)
uređaj := CUDA || CPU
za svaki $(slika, oznaka) \in T$
  sinkroniziraj_uređaj()
  $t_0 \leftarrow$ očitaj_vrijeme()
  prenesi($slika$, uređaj)
  $predikcija \leftarrow$ model($slika$)
  prenesi($predikcija$, CPU)
  sinkroniziraj_uređaj()
  $t_1 \leftarrow$ očitaj_vrijeme()
  t $\leftarrow$ t+$(t_1 - t_0)$
  točno $\leftarrow$ točno + $(predikcija == oznaka)$
t $\leftarrow$ t/ukupno
točno $\leftarrow$ točno/ukupno
vrati (t,točno)

Navedeni algoritam koristi za mjerenje na CPU i GPU. Funkcija *sinkroniziraj_uređaj()* je blokirajuća funkcija koja čeka da se izvrše svi zadaci na uređaju. Ona na CPU neće imati nikakav efekt, a kod obrade na grafičkom procesoru pričekati će da završi obrada, tj. prijenos podataka s grafičke kartice na CPU. Funkcija *prenesi(podaci,uređaj)* prenosi podatke iz procesorske memorije u memoriju uređaja. Ako je uređaj sam procesor tada ova funkcija nema efekta, a kod grafičke kartice događa se prijenos na njezinu internu memoriju.

Eksperimentalni rezultati dobiveni su u okruženju opisanom u tablici 6.1. Svi provedeni eksperimenti mogli bi dati drugačije rezultate u drugim okruženjima.

Tablica 6.1: Opis izvršnog okruženja

Operacijski sustav	Ubuntu 20.04.02 LTS
Jezgra OS-a	5.8.0-55-generic
Procesor	Intel Core i5-8250U
Grafička kartica	NVidia GeForce MX150
Driver	465.19.01
Python	3.8.5
PyTorch	1.8.1
CUDA	11.1
PyCuda	2020.1
ONNX	1.8
TensorRT	7.2.3.4

Tablica 6.2: Rezultati eksperimenata

Model	Veličina (MB)	Točnost	Δ_{acc}	CPU (Intel i5-8250U)	GPU (GeForce MX150)
Referentni	≈ 44.81	82.90%	0%	10.992 ms	8.146 ms
Dinamički kvantiziran	≈ 44.79	82.93%	+0.03%	11.110 ms	-
Statički kvantiziran	≈ 11.40	80.78%	-2.12%	6.027 ms	-
Naučena kvantizacija	≈ 11.31	82.76%	-0.14%	5.937 ms	-
Referentni (TensorRT)	≈ 44.81	82.90%	0%	-	2.358 ms

U tablici 6.2 prikazani su dobiveni rezultati mjerenja. Veličina modela mjerena je veličinom svih parametara modela. Vrijednost Δ_{acc} predstavlja odstupanje točnosti od referentnog modela.

Zanimljivo je opaziti da je dinamički kvantizirani model sporiji od običnog FP32 modela. Razlog tome je spomenuto kašnjenje koje se događa prilikom dinamičkog računanja parametara kvantizacije. Jedini potpuno povezani sloj je izlazni sloj koji sadrži samo 10 elemenata pa se zato više isplatila FP32 artimetika. Također, vidljivo je da se veličina modela smanjila samo za $\approx$ 20kB što odgovara ukupnoj veličini parametara

kvantiziranog sloja, dok su parametri ostalih slojeva ostali nepromijenjeni. Bilo bi zanimljivo isprobati dinamički kvantizirani sloj sa više elemenata (npr. kod klasifikacije na ImageNet podatkovnom skupu zadnji sloj ima 1000 elemenata).

Vrijedi spomenuti kako su statička i naučena kvantizacija postigle zadovoljavajuće ubrzanje za faktor ≈ 1.83.

Ipak, najveće ubrzanje dobili smo korištenjem referentnog FP32 modela čije se zaključivanje izvodi u optimiranom TensorRT okruženju. Izvršava se gotovo 3.5 puta brže od običnog modela na grafičkoj kartici. Zapanjujuća je činjenica da je intervencija programera u ovom slučaju minimalna, tj. sve potrebne optimizacije proveo je TensorRT na temelju računskog grafa naše mreže.

Najbolje ubrzanje mogli bismo postići koristeći statičku kvantizaciju u TensorRT-u. Problem kod izvođenja statičke kvantizacije u TensorRT-u je taj što nemamo gotov mehanizam promatrača već je potrebno napisati vlastite mehanizme kalibracije slojeva za statičku kvantizaciju.

7. Zaključak

Postupak kvantizacije definitivno može ubrzati postupak zaključivanja neuronskih mreža do nekoliko puta. Brzina može igrati ulogu u sustavima koji trebaju *real-time* obradu podataka. Prema tome, ovaj postupak može se iskoristiti u industriji za probleme gdje se mogu tolerirati manji gubitci točnosti. Pokazalo se da se izvođenje na GPU može dodatno ubrzati pomoću vanjskih optimizatora.

Kao nastavak ovog rada bilo bi prikladno isprobati veće podataka i dublje mreže. Također, bilo bi zanimljivo isprobati statičku kvantizaciju u okviru TensorRT-a koja bi trebala dati maksimalne performanse.

LITERATURA

[1] Convolutional neural networks (cnns / convnets), 2020. URL https://cs231n.github.io/convolutional-networks/.

[2] Quantization, 2020. URL https://pytorch.org/docs/stable/quantization.html.

[3] Cloud tensor processing units (tpus), 2021. URL https://cloud.google.com/tpu/docs/tpus.

[4] *TensorRT Developer guide*, 2021. URL https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-723/developer-guide/index.html.

[5] Simone Bianco, Remi Cadene, Luigi Celona, i Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6: 64270–64277, 2018. ISSN 2169-3536. doi: 10.1109/access.2018.2877890. URL http://dx.doi.org/10.1109/ACCESS.2018.2877890.

[6] Pooya Davoodi, Guangda Lai, Trevor Morris, i Siddharth Sharma. High performance inference with tensorrt integration, 2019. URL https://blog.tensorflow.org/2019/06/high-performance-inference-with-TensorRT.html.

[7] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2021. Version: 1.8.

[8] Vincent Dumoulin i Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

[9] Ian Goodfellow, Yoshua Bengio, i Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[10] Chris Gottbrath. Deep dive on pytorch quantization, 2020. URL https://www.youtube.com/watch?v=c3MT2qV5f9w.

[11] Ivan Grubišić. Semantička segmentacija slika dubokim konvolucijskim mrežama, 2016. URL http://www.zemris.fer.hr/~ssegvic/project/pubs/grubisic16bs.pdf.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, i Jian Sun. Deep residual learning for image recognition, 2015.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, i Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016. URL http://arxiv.org/abs/1603.05027.

[14] T Huang. Computer Vision: Evolution And Promise. 1996. doi: 10.5170/CERN-1996-008.21. URL https://cds.cern.ch/record/400313.

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, i Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1–30, 2018. URL http://jmlr.org/papers/v18/16-456.html.

[16] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018. URL http://arxiv.org/abs/1806.08342.

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

[18] Filip Oreč. Učinak kvantizacije na duboke konvolucijske klasifikacijske modele, 2020. URL http://www.zemris.fer.hr/~ssegvic/project/pubs/orec20proj2.pdf.

[19] Marin Oršić i Siniša Šegvić. Efficient semantic segmentation with pyramidal fusion. *Pattern Recognition*, 110:107611, 2021. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2020.107611. URL https://www.sciencedirect.com/science/article/pii/S0031320320304143.

[20] Spandan Tiwari i Emma Ning. Operationalizing pytorch models using onnx and onnx runtime, 2020. URL https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/.

[21] Yu Emma Wang, Gu-Yeon Wei, i David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning, 2019.

[22] Marko Čupić. Umjetne neuronske mreže, 2018. URL http://java.zemris.fer.hr/nastava/ui/ann/ann-20180604.pdf.

[23] Marko Čupić, Jan Šnajder, i Bojana Dalbelo Bašić. Umjetne neuronske mreže, 2011. URL https://www.fer.unizg.hr/_download/repository/UI_12_UmjetneNeuronskeMreze.pdf.

[24] Marko Čupić, Marin Golub, i Bojana Dalbelo Bašić. *Neizrazito, evolucijsko i neuroračunarstvo*. 2013. URL http://java.zemris.fer.hr/nastava/nenr/knjiga-0.1.2013-08-12.pdf.

[25] Siniša Šegvić i Josip Krapac. Duboke unaprijedne mreže. URL http://www.zemris.fer.hr/~ssegvic/du/du1feedforward.pdf.

Kvantizacija konvolucijskih modela za raspoznavanje slika

Sažetak

Raspoznavanje slika važan je zadatak računalnog vida s mnogim zanimljivim primjenama. Trenutno stanje tehnike koristi duboke modele koji se uče s kraja na kraj. Nažalost, velika računska složenost tih modela isključuje mnoge zanimljive primjene. Taj problem možemo ublažiti zamjenom decimalnih računskih operacija odgovarajućim cjelobrojnim operacijama. U okviru rada, potrebno je odabrati okvir za automatsku diferencijaciju te upoznati biblioteke za rukovanje matricama i slikama. Proučiti i ukratko opisati postojeće pristupe za klasifikaciju slike. Odabrati slobodno dostupni skup slika te oblikovati podskupove za učenje, validaciju i testiranje. Uhodati postupke učenja modela u decimalnoj točnosti. Predložiti prikladnu metodologiju za prevođenje tih modela u cjelobrojnu točnost. Procijeniti mogućnost izvedbe kvantiziranog modela na grafičkom procesoru. Validirati hiperparametre, vrednovati kvantizirane modele te prikazati i ocijeniti postignutu točnost. Radu priložiti izvorni i izvršni kod razvijenih postupaka, ispitne slijedove i rezultate, uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Ključne riječi: strojno učenje, računalni vid, konvolucijske neuronske mreže, gusto povezane neuronske mreže, PyTorch, kvantizacija, klasifikacija

Quantization of convolutional models for image recognition

Abstract

Image recognition is important task with many interesting applications in computer vision. Current state of the art are deep models trained end to end. Unfortunatelly, computing complexity of those models excludes lots of interesting applications. That problem can be mitigated by replacing decimal with integer operations. Within this work it is reqired to choose a framework for automatic differentiation and get familiar with libraries for handling matrices and images. Existing approaches to image classification should be studied and described. Free public image dataset was choosen and training, validation and test subsets were constructed. Execute learining procedures in decimal precision. Propose appropriate metodology for converting those models to integer precision. Estimate possibility of executing quantized model on graphics processor. Validate hyperparameters, evaluate quantized models and present and evaluate accomplished accuracy. Submit source and executable code, test cases and results with appropriate explanation and documentation alongside paper. Cite used literature and received help.

Keywords: machine learning, computer vision, convolutional neural networks, densely connected neural networks, PyTorch, quantization, classification