SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2598

KVANTIZACIJA DUBOKIH KONVOLUCIJSKIH MODELA

Filip Oreč

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2598

KVANTIZACIJA DUBOKIH KONVOLUCIJSKIH MODELA

Filip Oreč

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

DIPLOMSKI ZADATAK br. 2598

Pristupnik:	**Filip Oreč (0036501416)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	prof. dr. sc. Siniša Šegvić

Zadatak: **Kvantizacija dubokih konvolucijskih modela**

Opis zadatka:

Duboki konvolucijski modeli danas su metoda izbora za mnoge zadatke računalnog vida. 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 istražiti postojeće pristupe za kvantiziranje težina dubokih modela. Naučiti i vrednovati model SwiftNet na javno dostupnim skupovima za semantičku segmentaciju. Modificirati postupak učenja tako da na izlazu dobijemo kvantizirani model. Validirati hiperparametre, prikazati i ocijeniti ostvarene rezultate te provesti usporedbu s rezultatima iz literature. Predložiti pravce budućeg razvoja. Radu priložiti izvorni kod razvijenih postupaka uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 28. lipnja 2021.

SADRŽAJ

1. Uvod

U posljednje vrijeme dolazi do sve većeg razvoja sustava poput autonomnih vozila, robota i drugih. Kako bi se takvi sustavi snalazili u prostoru sve više se razvijaju i novi pristupi u računalnom vidu koji to omogućuju. Računalni vid je područje znanosti koje se bavi načinom na koji računala mogu steći razumijevanje na visokoj razini pomoću digitalnih slika ili videozapisa. Danas najčešće korišteni pristupi dolaze iz područja dubokog učenja (*engl.* Deep learning).

U dubokom učenju konvolucijske neuronske mreže (*engl.* Convolutional neural networks) su razred dubokih neuronskih mreža koje se najčešće primjenjuju za rješavanje problema računalnog vida. U zadnje vrijeme postale su jako popularan pristup za rješavanje problema klasifikacije i semantičke segmentacije slika. Ali takvi modeli troše jako puno resursa. Naime, oni ne samo da zahtijevaju jako puno računske snage za izvršavanje operacija, nego im je potrebno i jako puno memorije. Kvantizacija je tehnika koja koristi 8-bitne cijele brojeve za izvršavanje operacija kako bi se smanjilo vrijeme potrebno za zaključivanje, te za pohranu parametara modela kako bi se smanjio memorijski otisak modela. Kvantizirani model izvršava neke ili sve operacije nad tenzorima s cijelim brojevima (*engl.* Integers), a ne s decimalnim brojevima (*engl.* floating point values). To omogućuje kompaktniji prikaz modela i upotrebu vektoriziranih operacija visokih performansi na mnogim hardverskim platformama.

U ovom radu detaljnije će se obraditi kvantizacija dubokih konvolucijskih modela, te usporediti kvantizirani modeli s običnim modelima. Evaluacija i usporedba će se odraditi na modelu SwiftNet na problemu semantičke segmentacije. Korišten je javno dostupan skup podataka *Cityscapes* koji sadrži semantičke oznake uličnih scena. Kvantizacija je provedena za uređaj NVIDIA Jetson AGX Xavier koristeći programski okvir *TensorRT*.

2. Kvantizacija

U matematici i digitalnoj obradi signala kvantizacija se odnosi na postupak preslikavanja ulaznih vrijednosti iz velikog skupa (često kontinuiranog skupa) u izlazne vrijednosti u (prebrojivom) manjem skupu, često s konačnim brojem elemenata. U računarstvu kvantizacija se češće odnosi na tehnike izvođenja računskih operacija i pristupa memoriji s podacima niže preciznosti, obično INT8. Takav pristup povećava brzinu prijenosa podataka i brzinu izvođenja računskih operacija.

	Dinamički raspon	Min. pozitivna vrijednost
FP32	$-3.4 \times 10^{38} \sim +3.4 \times 10^{38}$	1.4×10^{-45}
FP16	$-65504 \sim +65504$	5.96×10^{-8}
INT8	$-128 \sim 128$	1

Tablica 2.1: Prikaz raspona vrijednosti koje se mogu pohraniti s različitim preciznostima.

Kao što je vidljivo u tablici 2.1 INT8 ima značajno nižu preciznost i dinamički raspon u odnosu na FP32. Zbog toga kvantizacija zahtijeva više od jednostavne pretvorbe tipa iz FP32 u INT8.

2.1. Osnove kvantizacije

Ovaj rad se fokusira na uniformnu cjelobrojnu (*engl.* integer) kvantizaciju, jer omogućuje izračunavanje operacija matričnog množenja i konvolucija u domeni cijelih brojeva, omogućujući upotrebu cjelobrojnih matematičkih cjevovoda (*engl.* pipes) velike propusnosti. Uniformna kvantizacija se može podijeliti u dva koraka:

1. Odabir raspona realnih brojeva koji će se kvantizirati. Sve vrijednosti izvan tog raspona se pritežu na maksimalnu, odnosno minimalnu odabranu vrijednost.

2. Preslikavanje realnih vrijednosti na cijele brojeve koji se mogu prikazati odabranom širinom bita za kvantizirani prikaz. Svaka preslikana realna vrijednost se zaokružuje na najbližu cjelobrojnu vrijednost.

Realni brojevi predstavljeni su formatom veće preciznosti s pomičnim zarezom (*engl.* floating-point format) s 32 bita (FP32).

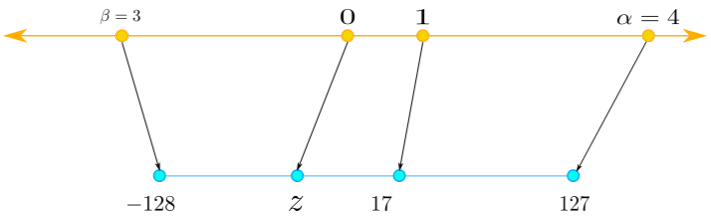
Slika 2.1: Prikaz pritezanja vrijednosti koje su izvan odabranog raspona $[\beta, \alpha]$ na najbližu granicu. Vrijednosti na narančastom pravcu koje su izvan odabranog raspona ispunjene su crvenom bojom. Kako se te vrijednosti nalaze lijevo od vrijednosti β one se pritežu upravo na tu vrijednosti, tj. na najbližu ogradu.

Neka je $[\beta, \alpha]$ interval koji predstavlja odabrani raspon realnih brojeva koji će se kvantizirati i neka je b širina bita koja se koristi za kvantizirani prikaz. Uniformna kvantizacija preslikava ulaznu realnu vrijednost $x \in [\beta, \alpha]$ u cjelobrojnu vrijednost iz intervala $[-2^{b-1}, 2^{b-1} - 1]$, pri čemu se ulazne vrijednosti izvan odabranog raspona pritežu na najbližu granicu. Slika 2.1 prikazuje pritezanje vrijednosti izvan odabranog raspona $[\beta, \alpha]$ na najbližu ogradu. Preslikavanje se izvršava na jedan od sljedećih načina:

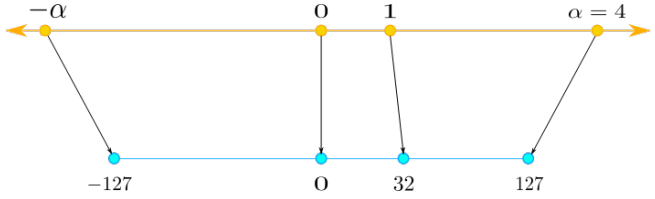
$$f(x) = s \cdot x + z \tag{2.1}$$

$$f(x) = s \cdot x \tag{2.2}$$

gdje su $x, s, z \in \mathbb{R}$. Jednadžba 2.1 predstavlja afinu kvantizaciju, a jednadžba 2.2 predstavlja kvantizaciju skaliranjem.



(a) Afina kvantizacija.



(b) Kvantizacija skaliranjem.

Slika 2.2: Preslikavanje realnih vrijednosti na cijele brojeve sa širinom bita jednako 8 (INT8).

2.1.1. Afina kvantizacija

Afina kvantizacija preslikava realnu vrijednost $x \in \mathbb{R}$ u b-bitni cijeli broj $x_q \in \{-2^{b-1}, -2^{b-1}+1, \ldots, 2^{b-1}-1\}$. Jednadžbe 2.3 i 2.4 definiraju funkciju afine kvantizacije.

$$s = \frac{2^b - 1}{\alpha - \beta} \tag{2.3}$$

$$z = -round(\beta \cdot s) - 2^{b-1} \tag{2.4}$$

gdje je s faktor skaliranja (*engl.* scale factor), a z je cjelobrojna vrijednost u koju je realna vrijednost nula preslikana (*engl.* zero-point). Za 8-bitni slučaj, $s = \frac{255}{\alpha-\beta}$ i $z = -round(\beta \cdot s) - 2^{b-1}$. Nakon određenog preslikavanja potrebno je zaokružiti rezultat preslikavanja na najbliži cijeli broj. Potpuni proces afine kvantizacije definiran je na sljedeći način:

$$clip(x, a, b) = \begin{cases} a, & x < a \\ x, & a \leq x \leq b \\ b, & x > b \end{cases} \tag{2.5}$$

$$x_q = clip(round(s \cdot x + z), -2^{b-1}, 2^{b-1} - 1) \tag{2.6}$$

Potrebno je primijetiti kako se realna vrijednost nula ne preslikava u cjelobrojnu vrijednost nula. Slika 2.2a prikazuje preslikavanja afine kvantizacije.

Jednadžba 2.7 prikazuje obrnuti proces afine kvantizacije, tj. preslikavanje nazad u realni prostor. Izračunava se aproksimacija stvarne realne vrijednosti koja je kvantizirana, $\hat{x} \approx x$.

$$\hat{x} = \frac{1}{s}(x_q - z) \tag{2.7}$$

2.1.2. Kvantizacija skaliranjem

Kvantizacija skaliranjem preslikava realnu vrijednost $x \in \mathbb{R}$ u b-bitni cijeli broj $x_q \in \{-2^{b-1}, -2^{b-1}+1, \ldots, 2^{b-1}-1\}$ koristeći samo transformaciju skaliranja. Ovaj rad bavi se simetričnom kvantizacijom skaliranjem, gdje su ulazni raspon $[\beta, \alpha]$ i cjelobrojni raspon simetrični oko nule, tj. $|\alpha| = |\beta|$, pa je ulazni raspon $[-\alpha, \alpha]$. Za 8-bitni slučaj koristi se cjelobrojni raspon $[-127, 127]$, gdje se ne uzima vrijednost -128 za donju granicu kako bi se postigla simetričnost oko nule. Jednadžba 2.8 definira funkciju kvantizacije skaliranjem za ulazni raspon $[-\alpha, \alpha]$.

$$s = \frac{2^{b-1} - 1}{\alpha} \tag{2.8}$$

Potpuni proces simetrične kvantizacije skaliranjem definiran je na sljedeći način:

$$x_q = clip(round(s \cdot x), -2^{b-1} + 1, 2^{b-1} - 1) \tag{2.9}$$

, gdje je funkcija $clip$ definirana jednadžbom 2.5. Slika 2.2b prikazuje preslikavanje simetrične kvantizacije skaliranjem.

Jednadžba 2.10 prikazuje obrnuti proces afine kvantizacije, tj. preslikavanje nazad u realni prostor. Izračunava se aproksimacija stvarne realne vrijednosti koja je kvantizirana, $\hat{x} \approx x$.

$$\hat{x} = \frac{1}{s} x_q \tag{2.10}$$

2.1.3. Odabir raspona

Nameće se pitanje kako odabrati raspon, tj. interval $[\beta, \alpha]$, realnih brojeva koji će se kvantizirati. Radi jednostavnosti ovaj rad razmatra samo raspon za simetričnu kvantizaciju opisanu u poglavlju 2.1.2, tj. interval $[-\alpha, \alpha]$.

Najjednostavnija i najintuitivnija metoda je odabrati maksimalnu apsolutnu vrijednost među vrijednostima koje se kvantiziraju.

Kako obični model i kvantizirani model zapravo sadrže istu informaciju, može se pokušati minimizirati gubitak te informacije prilikom kvantiziranja. Gubitak informacije se mjeri s Kullback-Leibler divergencijom. Odabiru se oni α i β za koji je gubitak informacije minimalan. Za diskretne distribucije P i Q definirane na istom prostoru vjerojatnosti $\mathcal{X}$, relativna entropija od Q do P ili Kullback-Leibler divergencija definirana je na sljedeći način:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) log(\frac{P(x)}{Q(x)}) \tag{2.11}$$

2.1.4. Kvantizacija tenzora

Postoji više načina kako se tenzori u dubokom modelu mogu kvantizirati, a ovisi o načinu na koji se dijele parametri kvantizacije između elemenata tenzora. Ovaj izbor se naziva granularnost kvantizacije (*engl*. quantization granularity). U najgrubljem slučaju koristi se granularnost po tenzoru (*engl*. per-tensor granularity), gdje svi elementi tenzora dijele iste parametre kvantizacije. U finijim slučajevima parametri se ne djele između elemenata, nego svaki element tenzora ima svoje parametre. Isto tako moguće je dijeljenje parametara preko različitih dimenzija, npr. dijeljenje parametra po stupcu ili po retku za 2D tenzore ili po kanalima za 3D tenzore. Dijeljenje parametra po kanalima je najčešći izbor ako se radi sa slikama. Prilikom odabira granularnosti kvantizacije potrebno je uzeti u obzir utjecaj na točnost modela i na trošak izračunavanja operacija, tj. vrijeme zaključivanja modela.

2.2. Kvantizacija dubokih modela

Kvantizacija omogućuje poboljšanje performansi u nekoliko važnih područja. Za slučaj 8-bitne kvantizacije: 4 puta smanjenje veličine modela, 2 do 4 puta smanjenje propusnosti memorije, 2 do 4 puta brže zaključivanje zbog uštede na propusnosti memorije i bržih računanja s aritmetikom INT8 (točno ubrzanje ovisi o hardveru i modelu). Međutim, kvantizacija ne dolazi bez dodatnih troškova. U osnovi kvantizacija znači uvođenje aproksimacija, a rezultirajući modeli imaju nešto lošiju točnost u odnosu na nekvantizirane modele. Cilj kvantizacije konvolucijskih modela je minimizirati gubitak informacija prilikom kvantiziranja njihovih težina i izračuna aktivacija.

Postoje tri načina kako se provodi kvantizacija dubokih modela:

1. Kvantizacija nakon treninga (*engl*. Post training quantization) - PTQ
   (a) Dinamička kvantizacija
   (b) Statička kvantizacija
2. Trening svjestan kvantizacije (*engl*. Quantization aware training) - QAT

2.2.1. Dinamička kvantizacija

Ovo je najlakša i najjednostavnija metoda kvantizacije, a provodi se nakon treninga modela, zato i spada u grupu kvantizacije nakon treninga. Kako se radi na naučenom modelu kvantizacija težina nije problem jer se ne mijenjaju i ne ovise o ulazima, te se lako može naći raspon vrijednosti, tj. interval $[\beta, \alpha]$ koji će se kvantizirati. Dakle, težine se kvantiziraju prije vremena i spremaju se u kvantiziranom obliku. Problem je kvantizirati aktivacije, jer ovise o ulazu i za različite ulaze potrebni su različiti parametri kvantizacije. Ova metoda kvantizira aktivacije prilikom zaključivanja. Raspon vrijednosti, tj. interval $[\beta, \alpha]$ se izračuna dinamički, na temelju raspona vrijednosti aktivacija, te se zatim te vrijednosti kvantiziraju. Stoga se izračuni provode pomoću učinkovitih kvantiziranih cjelobrojnih operacija. Međutim, aktivacije se čitaju i zapisuju u memoriju u FP32 formatu, jer se interval $[\beta, \alpha]$ ne može odrediti prije vremena. Zbog stalnog pretvaranja aktivacija iz FP32 u INT8 i obratno, brzina zaključivanja se smanjuje i učinkovite implementacije kvantiziranog matričnog množenja ne dolaze do izražaja. Ova tehnika se koristi u situacijama kada vremenom izvođenja modela dominira učitavanje težina iz memorije, a ne izračunavanje matričnog množenja. Ovo je istina za LSTM modele i za transformator modele (npr. BERT) s malom veličinom grupe na ulazu.

Iz tablice 2.2 vidi se smanjenje vremena potrebno za zaključivanja kad se koristi kvantizirani model. Poboljšanje u performansi je 85.6%. Za točnost je uzeta F1 mjera, te se vidi kako kvantizirani model ima jako mali pad u točnosti [3].

Model	F1 FP32	F1 INT8	Inf. time FP32	Inf. time INT8	Device
BERT	0.902	0.895	581 ms	313 ms	Xeon-D2191 (1.6Ghz)

Tablica 2.2: Prikaz rezultata dinamičke kvantizacije BERT modela na GLUE benchmark-u za MRPC. Korištena veličina grupe je 1 i maksimalna duljina sekvence je 128 [3].

2.2.2. Statička kvantizacija

Ova metoda se također provodi nakon treninga modela kao i dinamička kvantizacija. Težine modela se također kvantiziraju prije vremene iz istog razloga kao kod dinamičke kvantizacija. Razlika u odnosu na dinamičku kvantizaciju je način na koji se kvantiziraju aktivacije. Kako bi se uklonio nedostatak dinamičke kvantizacije, gdje se aktivacije moraju pretvarati iz FP32 u kvantizirani oblik, ova metoda provodi dodatni korak zvan kalibracija.

Kalibracija je proces određivanja intervala $[\beta, \alpha]$ za aktivacije modela prije izvođenja. Cilj je za svaki sloj modela odrediti α i β kako bi se aktivacije mogle spremati u kvantiziranom obliku i time izbaciti dodatno opterećenje dinamičkog računanja α i β parametara i stalnog pretvaranja iz FP32 u kvantizirani oblik i obrnuto. Kalibracija se izvodi tako da se kalibracijski skup podataka prikaže modelu i provuče unaprijednim prolazom kroz sve slojeve modela. Prilikom prolaza kalibracijskog skupa podataka svaki sloj izračunava stvari bitne za određivanje kvantizacijskih parametara ovisno o metodi određivanja raspona koja se koristi. Ako se koristi metoda maksimalne apsolutne vrijednosti, samo se pamti maksimalna apsolutna vrijednost. Ako se koristi metoda minimizacije gubitka informacije onda se izračunava histogram aktivacija. Kalibracijski skup podataka mora biti reprezentativan, raznolik, idealno podskup validacijskog skupa podataka, oko 1000 podataka (zavisi do modela i problema koji se rješava). Ovaj proces omogućava učinkovito iskorištavanje kvantiziranih operacija, jer se više parametri kvantizacije ne određuju dinamički, nego su određeni unaprijed (statički), čime se značajno smanjuje vrijeme zaključivanja u odnosu na dinamičku kvantizaciju.

Ova metoda se obično koristi kada su važni i propusnost memorije i ušteda na izračunavanju operacija. Duboke konvolucijske neuronske mreže su tipičan primjer takvih arhitektura, stoga je ova metoda posebno bitna za računalni vid, te će se u eksperimentima ovog rada razmatrati samo ova metoda.

Iz tablice 2.3 može se vidjeti kako se za ResNet-50 model dobije 2 puta ubrzanje korištenjem statičke kvantizacije, dok je gubitak točnosti minimalan (0.2%). Za model ResNet-18 gubitak točnosti je također minimalan (0.4%).

Model	Top-1 Acc. FP32	Top-1 Acc. INT8	Inf. time FP32	Inf. time INT8
ResNet-50	76.1	75.9	214 ms	103 ms
ResNet-18	69.8	69.4	-	-

Tablica 2.3: Prikaz rezultata statičke kvantizacije ResNet modela na Imagentu-u [3]. Vrijeme zaključivanja izmjereno je na uređaju Xeon-D2191 (1.6Ghz).

2.2.3. Trening svjestan kvantizacije

Trening svjestan kvantizacije (QAT) je treća tehnika kvantizacije dubokih modela i ona ne spada u tehnike kvantizacije poslije treninga. Ideja ove metode ja da se prilikom učenja dubokog modela njegovi parametri i aktivacije "lažno kvantiziraju", te na taj način model "bude svjestan" kako će prilikom zaključivanja biti kvantiziran. Pod "lažnom kvantizacijom" misli se kako se FP32 vrijednosti zaokružuju kako bi predstavljale INT8 vrijednosti, ali se i dalje drže u FP32 formatu i sva se izračunavanja i dalje rade s tim formatom. Dakle, sva ažuriranja parametara modela tijekom treninga vrše se dok su "svjesni" činjenice da će model u konačnici biti kvantiziran.

Ova metoda se obično koristi kada metoda statičke kvantizacije ne daje dovoljno dobru točnost. Ako statička kvantizacija rezultira minimalnim padom točnosti, kao u tablici 2.3, onda nema potrebe za ovom metodom, jer obično neće rezultirati boljim rezultatom ili će rezultirati minimalno boljim rezultatom [3].

Model	Top-1 Acc. FP32	Top-1 Acc. INT8	Inf. time FP32	Inf. time INT8
MobileNet-V2	71.9	71.6	97 ms	17 ms

Tablica 2.4: Prikaz rezultata MobileNet-V2 modela koji je kvantiziran metodom kvantizacije poslje treninga na Imagentu-u [3]. Vrijeme zaključivanja izmjereno je na uređaju Samsung S9.

S arhitekturom MobileNet-V2 nisu se uspjeli dobiti dovoljno dobri rezultati na Imagentu-u primjenom statičke kvantizacije, te se zato ova arhitektura kvantizirala metodom kvantizacije nakon treninga [3]. Iz tablice 2.4 može se vidjeti kako ova arhitektura primjenom metode kvantizacije nakon treninga ima minimalan pad u točnosti (0.3%). Također, može se vidjeti jako veliko ubrzanje prilikom zaključivanja. Kvantizirani model ima 5.7 puta manje vrijeme zaključivanja u odnosu na FP32 model.

3. Korištene tehnologije

U ovom poglavlju opisane su tehnologije korištene za izvedbu eksperimenata, te implementacija programskog dijela. Kako bi se usporedile točnosti i performanse kvantiziranog i FP modela rješavan je problem semantičke segmentacije na skupu podataka Cityscapes [7]. Model nad kojim je prevedena kvantizacija i usporedba kvantiziranog modela, modela s FP16 preciznosti i modela s FP32 preciznosti je SwiftNet [10]. Hardver na kojem su provedeni svi eksperimenti je NVIDIA Jetson AGX Xavier. Za kvantizaciju modela i pokretanje zaključivanja modela korišten je programski okvir TensorRT [5]. Za definiranje modela korišten je programski okvir PyTorch, te za prebacivanje modela definiranog u PyTorch-u u programski okvir TensorRT korišten je format ONNX [2].

3.1. Model

Model koji je korišten za provedbu eksperimenata je SwiftNet [3]. To je model koji je napravljen za rješavanje problema semantičke segmentacije. Model se sastoji od tri glavna gradivna bloka. To su enkoder za raspoznavanje, dekoder za povećanje dimenzija mape značajki i modul za povećanje receptivnog polja.

Za enkoder se obično koristi neki od ResNet modela (ResNet-18 i ResNet-34) ili MobileNet-V2. Parametri enkodera mogu biti prethodno istrenirani na Imagent skupu podataka. Kako enkoder smanjuje prostorne dimenzije ulazne slike, njegovi izlazi se dalje prosljeđuju dekoderu.

Svrha dekodera je prikazati značajke u rezoluciji ulazne slike. Dekoder je građen od modula koji imaju dva ulaza. Prvi ulaz je mapa značajki niže rezolucije kojoj se treba povećati rezolucija, a drugi ulaz su lateralne značajke dobivene iz ranijih slojeva enkodera. Značajke niže rezolucije prvo se povećavaju bilinearnim naduzorkovanjem na rezoluciju lateralnih značajki. Te dvije mape značajki prvo se zbroje, što je moguće jer su dovedene na jednake dimenzije, pa se zatim izmiješaju konvolucijom 3×3.

Rad [10] predlaže dva pristupa za modul za povećanje receptivnog polja. To su prostorno piramidalno sažimanje (*engl.* spatial pyramid pooling) i piramidalna fuzija (*engl.* pyramid fusion). Model koji koristi prostorno piramidalno sažimanje u ovom radu se naziva SwiftNet

single scale model i njegova struktura prikazana je na slici 3.1. Model koji koristi piramidalnu fuziju u ovom radu naziva se SwiftNet pyramid i prikazan je na slici 3.2. Prostorno piramidalno sažimanje skuplja značajke iz enkodera na različitim razinama sloja sažimanja i tako nastaje prikaz s različitom razinom detalja. Piramidalna fuzija stvara reprezentacije na više razina koje se trebaju pažljivo stopiti unutar dekodera kako bi se izbjegla prenaučenost modela.

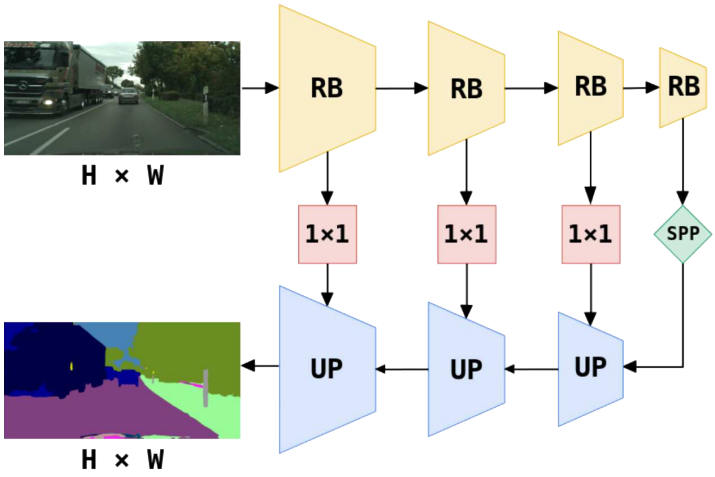
3.1.1. SwiftNet single scale

SwiftNet single scale model ulaznu sliku prostornih dimenzija $H \times W$ pretvara u semantičke oznake jednakih prostornih dimenzija kao i ulazna slika. Ulazna slika se prvo prosljeđuje enkoderu koji smanjuje prostorne dimenzije mapa značajki, a zatim se ta mapa značajki prosljeđuje dekoderu koji povećava prostornu rezoluciju mapa značajki. Slika 3.1 prikazuje strukturu modela. Žuti trapezi na slici predstavljaju konvolucijske grupe. Te konvolucijske grupe su dio enkodera za prepoznavanje i djeluju na jednakim prostornim dimenzijama. Enkoder koristi četiri takve konvolucijske grupe i svaka konvolucijska grupa smanjuje prostorne dimenzije mapa značajki četiri puta. Izlaz iz zadnje konvolucijske grupe ima prostorne dimenzije veličine $H/32 \times W/32$. Te mape značajki prosljeđuju se sloju za prostorno piramidalno sažimanje koje povećava receptivno polje modela. Na slici 3.1 je taj sloj prikazan zelenim rombom. Dalje se izlazi iz tog sloja prosljeđuju dekoderu. Dekoder je građen od modula za povećanje rezolucije koji su na slici prikazani plavim trapezima. Značajkama enkodera se povećava broj kanala kako se približavaju kraju, dok dekoder radi s jednakim brojem kanala cijelo vrijeme. Kako bi se lateralne značajke enkodera mogle zbrojiti s značajkama niže rezolucije koriste se konvolucije 1×1 koje lateralne značajke dovode na jednaku dimenzionalnost značajki niže rezolucije. Na slici je ta operacija prikazana crvenim kvadratima. Modul za povećanje dimenzija prvo bilinearnim naduzorkovanjem poveća prostorne dimenzije značajki niže rezolucije, zatim dobivenu reprezentaciju zbraja s lateralnim značajkama, te dobiveni zbroj izmiješa konvolucijom 3×3.

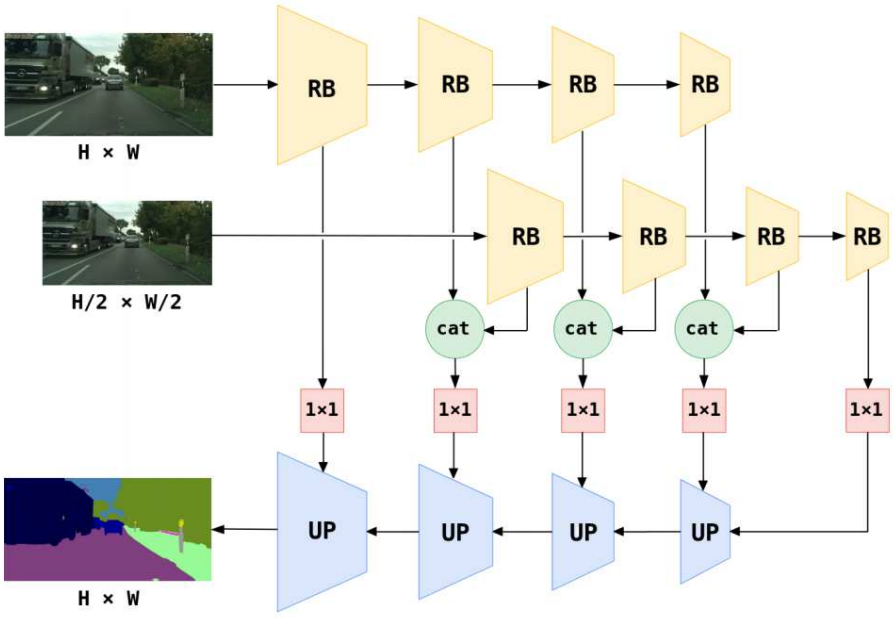
3.1.2. SwiftNet pyramid

Zbog korištenja kompaktnog enkodera dolazi do smanjenja receptivnog polja i manjeg kapaciteta modela u odnosu na konvolucijske modela opće svrhe za vizualno prepoznavanje. SwiftNet pyramid modela umanjuje te nedostatke, korištenjem piramide za povećanje receptivnog polja, te se tako i smanjuju potrebe za kapacitetom modela. Struktura modela prikazana je na slici 3.2. Žutom bojom su prikazane dvije instance enkodera koje na ulaz primaju ulazne slike na različitim rezolucijama piramide. Instance enkodera dijele parametre međusobno, što omogućava prepoznavanje objekata različitih veličina sa zajedničkim

skupom parametara, čime se smanjuje potreba za većim kapacitetom modela. Također se dodaju lateralne veze koje poboljšavaju protok gradijenata. Lateralne veze stvaraju tako da se povezuju značajke iz različitih enkodera susjednih razina. To povezivanje je na slici 3.2 prikazano zelenim krugovima. Dalje dekoder funkcionira jednako kao i kod SwiftNet single scale modela.



Slika 3.1: Prikaz modela SwiftNet single scale. Žuti trapezi označavaju konvolucijske grupe unutar enkodera. Zeleni romb predstavlja *spatial pyramid pooling* sloj. Crveni kvadrati predstavljaju usko grlo, tj. konvolucije s jezgrom 1×1. Plavi trapezi predstavljaju module za naduzorkovanje. Logiti se naduzorkuju do orginalne rezolucije korištenjem bilinearnog naduzorkovanja [10].



Slika 3.2: Prikaz modela SwiftNet pyramid. Parametri enkodera (žuti trapezi) su dijeljeni preko svih razina piramide. Značajke iste rezolucije se konkateniraju (zeleni krugovi), predaju se konvolucijama 1×1 uskog grla, te se stapaju unutar dekodera (plavi trapezi) [10].

3.2. Programski okvir TensorRT

TensorRT je programski okvir za zaključivanje dubokih modela visokih performansi. Uključuje optimizator za zaključivanje dubokih modela koji omogućava veliku propusnost i malo kašnjenje prilikom zaključivanja. Također, pruža i sučelja visoke razine (APIs) i parsere za učitavanje naučenih modela iz svih značajnijih programskih okvira za duboko učenje. U trenutku pisanja ovog rada podržani su parseri za Caffe2, UFF (TensorFlow) i ONNX format za definiranje modela. Osim parsiranja modela definiranih u prethodno spomenutim formatima TensorRT omogućuje i ručno definiranje i unos slojeva te parametara slojeva. Nakon što je model definiran, TensorRT gradi stroj za izvođenje (*engl.* inference engine) koji omogućuje optimalno izvođenje zaključivanja. Prilikom građenja stroja za izvođenje provode se optimizacije specifične za hardver na kojem se izvodi program. Provodi se 6 glavnih optimizacija:

1. Korištenje mješovite preciznosti (*engl*. mixed precision) - maksimizira se propusnost kvantiziranjem modela ili korištenjem FP16 preciznosti.

2. Stapanje slojeva i tenzora (*engl.* layer and tensor fusion) - optimizira se korištenje GPU memorije i propusnost stapanjem čvorova u jezgru.

3. Automatsko podešavanje jezgre (*engl*. kernel auto-tuning) - odabiru se najbolji podatkovni slojevi i algoritmi na temelju ciljne GPU platforme

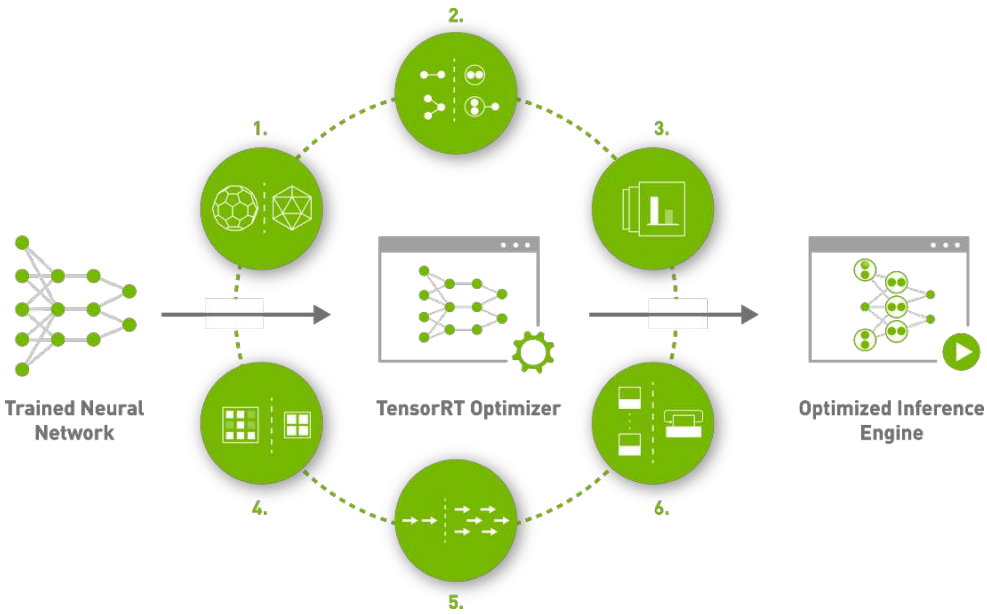
4. Dinamička memorija tenzora (*engl*. dynamic tensor memory) - smanjuje se memorijski otisak i učinkovito se ponovo koristi memorija za tenzore

5. Izvođenje više tokova (*engl*. multi-stream execution) - koristi se skalabilni dizajn za paralelnu obradu više ulaznih tokova.

6. Stapanje vremena (*engl*. time fusion) - optimiziraju se povratne neuronske mreže tijekom vremenskih koraka s dinamički generiranim jezgrama.

Bitno svojstvo za ovaj rad je podržavanje kvantizacije modela i korištenje FP16 preciznosti prilikom zaključivanja.

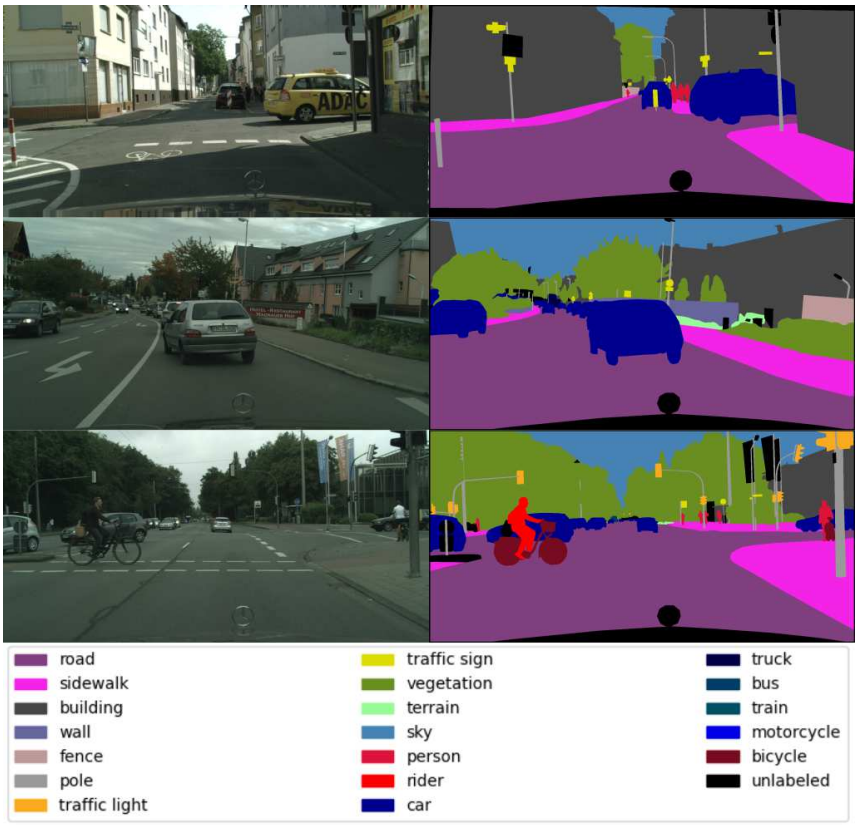
TensorRT podržava programske jezike C++ i Python. Izgrađen je na CUDA-i, NVIDIA-noj platformi i programskom modelu za paralelno izračunavanje na grafičkim procesnim jedinicama (GPU). Napravljen je tako da može izvršavati optimizacije i postaviti aplikacije na podatkovne centre (*engl*. data center), kao i na ugradbene uređaje.



Slika 3.3: Prikaz građenja stroja za optimizaciju programskog okvira TensorRT. Kao ulaz se prima naučen duboki modela nad kojim TensorRT izvodi 6 glavnih optimizacija specifičnih za ciljani GPU. Rezultat izvođenja optimizacija je stroj za zaključivanje koji učinkovito provodi zaključivanje dubokog modela koji se primio na ulazu [6].

3.3. Skup podataka Cityscapes

Skup podataka Cityscapes [7] je skup podataka koji sadrži slike visoke rezolucije snimljene u uličnim scenama iz vozačeve perspektive iz 50 različitih gradova tijekom dana i lijepog vremena. Posjeduje 5000 slika s visoko kvalitetnim oznakama razreda na razini piksela i dodatnih 20000 slika s niže kvalitetnim oznakama razreda na razini piksela. Zbog tako označenih podataka ovaj skup se koristio za rješavanje problema semantičke segmentacije u ovom radu. Od 5000 slika s oznakama visoke kvalitete 2975 slika pripada skupu podataka za treniranje, 500 pripada skupu podataka za validaciju i 1525 slika pripada skupu podataka za testiranje. Dodatnih 20000 slika nije korišteno u eksperimentima. Rezolucija slika je 1024×2048 piksela. Oznake slika sadrže 19 različitih razreda: cesta, pločnik, osoba, vozač, automobil, kamion, autobus, vozilo na tračnicama, motocikl, bicikl, građevina, zid, ograda, stup, prometni znak, semafor, vegetacija, teren i nebo.



Slika 3.4: Nekoliko primjera ulaznih podataka i odgovarajućih točnih oznaka u skupu podataka Cityscapes. Slike su rezolucije 1024×2048.

4. Implementacija

U ovom poglavlju opisani su implementacijski detalji ovog rada. Korišten je programski okvir TensorRT u programskom jeziku C++, te programski okvir PyTorch u programskom jeziku Python. U programskom kodu 4.1 prikazana je struktura glavnog razreda za provedbu zaključivanja SwiftNet modela u TensorRT-u. Glavna metoda je metoda `build` koja je zadužena za parsiranje definicije modela iz ONNX formata, te stvaranje stroja za zaključivanje. Ostale metode i parametri će biti objašnjeni u nastavku.

```
struct SwiftNetParams : public samplesCommon::OnnxSampleParams {
    int nbCalBatches;           // The nuber of bathces for calibration
    int calBatchSize;           // The calibration batch size
    std::string networkName;
};

class SwiftNetEngine {
    template<typename T>
    using SNUniquePtr = std::unique_ptr<T, samplesCommon::InferDeleter>;

    public:
        SwiftNetEngine(const SwiftNetParams& params)
            : mParams(params)
            , mEngine(nullptr)
        {
            initLibNvInferPlugins(&sample::gLogger.getTRTLogger(), "");
        }

        bool build(DataType dataType);
        bool teardown();
        bool saveEngine(const char* filepath);
        bool loadEngine(const char* filepath);
        bool performanceTest();
        bool accuracyTest();

    private:
        SwiftNetParams mParams;
        nvinfer1::Dims mInputDims;
```

```
        nvinfer1::Dims mOutputDims;
        std::shared_ptr<nvinfer1::ICudaEngine> mEngine;
        ...
};
```

Programski kod 4.1: Struktura razreda koji je zadužen za parsiranje definicije modela stvaranje stroja za zaključivanje i izvedbu zaključivanja SwiftNet modela u porgramskom okviru TensorRT.

4.1. Prebacivanje PyTorch modela u ONNX format

Kako je model s kojim se radi definiran u PyTorch-u, potrebno je iz PyTorch-a učitati taj model u TensorRT. TensorRT trenutno ne podržava parser za PyTorch modele, ali podržava parser za modele zapisane u ONNX formatu. Zbog toga potrebno je prvo prebaciti model definiran u PyTorch-u u ONNX format, te zatim iz ONNX formata učitati model u TensorRT. U programskom isječku 4.2 prikazan je način prebacivanja PyTorch modela u ONNX format.

```
import torch
import torch.onnx as onnx

params = '/path/to/model/params'
model = create_model() # creating model instance
model.load_state_dict(torch.load(params)) # loading trained parameters
model.to('cuda') # transfering model to GPU
model.eval() # setting model to eval mode

# creating input tensor
input_ = torch.ones((1, 3, 1024, 2048)).to('cuda')
model.forward(input_) # pushing tensor trough model

# exporting model to onnx format
onnx.export(
    model,
    input_,
    'filename.onnx',
    opset_version=11,
    verbose=True,
    do_constant_folding=True,
    export_params=True,
    input_names=["input"],
    output_names=["pred"]
)
```

Programski kod 4.2: Primjer prebacivanja PyTorch modela u ONNX format.

PyTorch ima modul `torch.onnx` koji nudi metodu `torch.onnx.export` za izvoz modela u ONNX format. Prvo je potrebno stvoriti instancu PyTorch modela, učitati naučene parametre, prebaciti model na GPU i u evaluacijski način rada. U programskom kodu 4.2 se to može vidjeti na linijama 4 do 8. Nadalje, kako bi se modela mogao uspješno izvesti potrebno je provući "lažni" ulazni tenzor kroz unaprijedni prolaz modela. To je bitno zbog načina na koji se model prebacuje u ONNX format. Naime, prebacivanje funkcionira na način da se provede jedan unaprijedni prolaz modela prilikom čega se pamte sve obavljene računske operacije, te se iz toga rekonstruira mreža. Zbog toga ne smije postojati ovisnost izlaza sloja o njegovom ulazu, tj. slojevi (operacije) se ne smiju dinamički odabirati s obzirom na ulaz. Pozivanjem metode `torch.onnx.export` model se zapisuje u ONNX format. Metoda kao prvi argument prima instancu PyTorch modela, kao drugi argument prima ulazni "lažni" tenzor koji će se provući kroz unaprijedni prolaz, te kao treći argument prima putanju do izlazne datoteke, tj. zapisa modela u ONNX formatu. Metoda također prima i dodatne argumente. Argument `opset_version` definira verziju seta operacija definiranih za ONNX format, `do_constant_folding` primjenjuje optimizaciju za definiranje konstati u računskom grafu ONNX formata, `export_params` izvozi i parametre modela zajedno s računskim operacijama, `input_names` i `output_names` definiraju imena ulaznih i izlaznih tenzora, što može biti korisno za rad u TensorRT-u. Primjer pozivanja metode `torch.onnx.export` može se vidjeti u programskom kodu 4.2 na linijama 15 do 25. Bitno je primijetiti kako je potrebno definirati ulazne dimenzije prilikom prebacivanja. Nakon što je model prebačen u ONNX format on radi samo s tim ulaznim dimenzijama, te dinamičko definiranje ulaznih dimenzija nije moguće.

Operacija `adaptive_avg_pool2d` iz modula `torch.nn.functional` se pokazala problematičnom prilikom prebacivanja modela SwiftNet single scale u ONNX format. Ta operacija nije definirana u ONNX formatu pa se ne može ni prebaciti u taj format, te zbog toga prebacivanje cijelog modela u ONNX format nije moguće. Kako bi se taj problem zaobišao prostorne dimenzije ulazne slike moraju biti višekratnik broja 256, jer se tad operacija `adaptive_avg_pool2d` pretvara u običnu operaciju sažimanja koja je definirana u ONNX formatu.

4.2. Učitavanje modela iz ONNX formata u TensorRT

Nakon što je model prebačen u ONNX format potrebno ga je učitati u programski okvir TensorRT kako bi se model mogao optimizirati, izgraditi stroj za zaključivanje te provesti zaključivanje nad modelom. To je moguće jer TensorRT nudi sučelje prema parseru za ONNX format koji prevodi ONNX operacije u TensorRT operacije nad kojima TensorRT zna provesti optimizacije.

```
1  template<typename T>
2  using SNUniquePtr = std::unique_ptr<T, samplesCommon::InferDeleter>;
3
4  auto builder = SNUniquePtr<nvinfer1::IBuilder>(nvinfer1::
       createInferBuilder(sample::gLogger.getTRTLogger()));
5
6  const auto explicit_batch = 1U << static_cast<uint32_t>(
       NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
7  auto network = SNUniquePtr<nvinfer1::INetworkDefinition>(builder->
       createNetworkV2(explicit_batch));
8
9  auto parser = SNUniquePtr<nvonnxparser::IParser>(nvonnxparser::
       createParser(*network, sample::gLogger.getTRTLogger()));
10
11 auto parsed = parser->parseFromFile(locateFile(mParams.onnxFileName,
       mParams.dataDirs).c_str(), static_cast<int>(sample::gLogger.
       getReportableSeverity()));
```

Programski kod 4.3: Stvaranje modela u TensorRT-u i parsiranje modela zapisanog u ONNX formatu. Ovaj dio koda se nalazi unutar metode build razreda SwiftNetEngine.

Prije samog parsiranja ONNX datoteke, potrebno je prvo stvoriti objekte definirane sučeljima `IBuilder` i `INetworkDefinition`. Objekt predstavljen sučeljem `IBuilder` zadužen je za stvaranje stroja za zaključivanje (inference engine). TensorRT nudi metodu `nvinfer1::createInferBuilder` koja stvara `IBuilder` objekt, a kao argument prima `logger` objekt, koji je zadužen za zapisivanje svih stvari koje se događaju prilikom rada TensorRT aplikacije. Može se iskoristiti gotov `logger` definiran u modulu `sample` koji dolazi uz TensorRT. Stvaranje `IBuilder` objekta prikazano je u programskom kodu 4.3 na liniji 4.

`IBuilder` objekt se koristi kako bi se stvorio `INetworkDefinition` objekt koji sadrži sve informacije vezane za definiciju dubokog modela. To je prikazano na linijama 6 i 7. Nad `IBuilder` objektom se poziva `createNetworkV2` metoda koja prima dodatnu zastavicu kojom se definira kako će veličina skupine (*engl.* batch size) biti eksplicitno definirana. Trenutno `INetworkDefinition` objekt ne sadrži nikakve informacije jer mu nisu dodani nikakvi slojevi niti je parsirana definicija modela u njega.

Nakon toga se definira parser koji je predstavljen sučeljem `IParser`. Parser objekt se stvara pomoću metode `nvonnxparser::createParser` koja prima objekt `INetworkDefinition` koji je prethodno stvoren, te `logger` objekt. Potom se nad parser objektom poziva metoda `parseFromFile` koja prima putanju do ONNX datoteke. Nakon uspješnog parsiranja objekt `INetworkDefinition` sadrži sve informacije o definiciji dubokog modela te nudi metode za dohvaćanje tih informacija.

4.3. Stvaranje stroja za zaključivanje

Nakon uspješnog parsiranja definicije modela potrebno je stvoriti stroj za zaključivanje (inference engine). Prilikom stvaranja stroja za zaključivanje TensorRT provodi sve optimizacije nad modelom, te se stvoreni stroj koristi prilikom zaključivanja. Kako bi se stvorio stroj za zaključivanje potrebno je predati model te sve bitne konfiguracijske parametre. Kako je jedna od optimizacija i kvantizacije u ovom koraku se postavljaju i parametri kvantizacije.

```
1  auto config = SNUniquePtr<nvinfer1::IBuilderConfig>(builder->
       createBuilderConfig());
2  config->setMaxWorkspaceSize(20_GiB);
3
4  if (mParams.fp16) {
5      config->setFlag(nvinfer1::BuilderFlag::kFP16);
6  }
7
8  std::unique_ptr<nvinfer1::IInt8Calibrator> calibrator;
9
10 if (mParams.int8) {
11     config->setFlag(nvinfer1::BuilderFlag::kINT8);
12     CityScapeBatchStream calibration_stream(1, 100, "/path/to/data", "cal
       ");
13     calibrator.reset(new Int8EntropyCalibrator2<CityScapeBatchStream>(
14                 calibration_stream, 0, "SwiftNet", mParams.
       inputTensorNames[0].c_str()));
15     config->setInt8Calibrator(calibrator.get());
16 }
17
18 mEngine = std::shared_ptr<nvinfer1::ICudaEngine>(
19         builder->buildEngineWithConfig(*network, *config), samplesCommon
       ::InferDeleter()
20         );
```

Programski kod 4.4: Stvaranje stroja za zaključivanje (inference engine). Ovaj dio koda se nalazi unutar metode build razreda SwiftNetEngine.

Konfiguracije za stvaranje stroja za zaključivanje su modelirane `IBuilderConfig` sučeljem. Stoga je prvo potrebno stvoriti taj objekt pozivom `createBuilderConfig` metode nad objektom `IBuilder` koji je prethodno stvoren. Jedan od bitnijih parametara za konfiguraciju je veličina memorije koju TensorRT smije koristiti za izvršavanje optimizacija. Preporuča se staviti maksimalnu dostupnu memoriju, jer neke optimizacije se neće uspješno izvesti ako nema dovoljno radne memorije. TensorRT neće iskoristiti cijelu maksimalno dostupnu memoriju ako nije potrebno. Veličina memorije se predaje `IBuilderConfig` objektu kao što je vidljivo u programskom kodu 4.4 na liniji 2.

Konfiguracijskom objektu se također predaje i preciznost u kojoj će se zaključivanje izvoditi. U programskom kodu 4.4 prikazano je postavljanje FP16 ili INT8 preciznosti. Postavljanje FP16 preciznosti je dosta jednostavno. Potrebno je samo konfiguracijskom objektu poslati odgovarajuću zastavicu. Postavljanje INT8 preciznosti je složenije jer je potrebno provesti postupak kalibracije. Bitno je primijetiti kako se na ovaj način primjenjuje postupak statičke kvantizacije. Postupak kalibracije bit će opisan u nastavku.

Nakon definiranja konfiguracijskog objekta stvara se stroj za zaključivanje koji je modeliran sučeljem `ICudaEngine`. Stvaranje stroja za zaključivanje obavlja se pozivom metode `buildEngineWithConfig` nad objektom `IBuilder` koji je stvoren ranije. Metodi se kao argumenti šalju definicija modela modelirana sučeljem `INetworkDefinition`, te konfiguracijski objekt modeliran sučeljem `IBuilderConfig`. Ovaj postupak je vidljiv u programskom kodu 4.4 na linijama 18, 19 i 20.

TensorRT također nudi mogućnost serijalizacije i deserijalizacije stvorenog stroja za zaključivanje što omogućava spremanje jednom stvorenog stroja, te njegovo ponovno učitavanje. Ova funkcionalnost može biti jako korisna jer stvaranje stroja za zaključivanje može trajati jako dugo ovisno o hardveru i veličini modela. Serijalizacija i deserijalizacija stroja za zaključivanje implementirane su u metodama `saveEngine` i `loadEngine` razreda `SwiftNetEngine`.

```
1  auto layer1 = network->getLayer(i);
2  auto layer1_output = layer->getOutput(j);
3
4  auto layer2 = network->getLayer(i + 1);
5  auto layer2_output = layer->getOutput(j + 1);
6
7  layer1->setPrecision(DataType::kINT8);
8  layer->setOutputType(j, DataType::kINT8);
9
10 layer1_output->setType(DataType::kINT8);
11
12 layer2->setPrecision(DataType::kHALF);
13 layer2->setOutputType(j, DataType::kHALF);
14
15 layer2_output->setType(DataType::kHALF);
```

Programski kod 4.5: Primjer postavljanja preciznosti za željeni sloj i željenu aktivaciju

Također moguće je definirati preciznost za svaki pojedinačni sloj i aktivaciju kako je prikazano u primjeru 4.5. Pojedinim slojevima i aktivacijama se pristupa preko objekta `INetworkDefinition` u koji je spremljena definicija parsiranog modela. S obzirom na ovu funkcionalnost TensorRT-a moguće je kvantizirati samo određene dijelove modela. Kalibracijski postupak će biti primjenjen samo na one slojeve koji koriste INT8 preciznost.

4.4. Kalibracijski postupak u TensorRT-u

Kako se za kvantizaciju SwiftNet modela u ovom radu koristi postupak statičke kvantizacije potrebno je implementirati postupak kalibracije. Za taj postupak potrebno je kroz unaprijedni prolaz provući sve slike iz kalibracijskog skupa podataka, pa je potrebno implementirati i učitavanje i iteriranje po slikama iz skupa podataka.

Učitavanje podataka i iteriranje po njima modelirano je `CityScapeBatchStream` razredom. Najbitnije metode ovog razreda su metoda `next` koja se koristi za iteriranje, te mteoda `getBatch` koja se koristi za dohvaćanje grupe podataka. Ulazni podaci, tj. slike se učitavaju kao pokazivač na float. Za čitanje slika i odgovarajućih oznaka koristi se *header-only* biblioteka *stbi*.

```
class CityScapeBatchStream {
    public:
        CityScapeBatchStream(int batch_size, int max_batches, const std::
    string& root, const std::string subset);
        void reset(int first_batch);
        int getBatchSize() const;
        bool next();
        void skip(int skip_count);
        [[nodiscard]] float* getBatch();
        [[nodiscard]] uint8_t* getLabel();
        nvinfer1::Dims getDims() const;

    private:
        ...
        float* read_data_file();
        uint8_t* read_label_file();
        ...
```

Programski kod 4.6: Prikaz razreda kojim je modelirano učitavanje i iteriranje po skupu podataka Cityscapes

Kako se ulazna slika dimenzija $H \times W \times 3$ (RGB slika) sprema u 1D polje bitno je na koji način se vrijednosti slike redaju u to polje. Normalno čitanje slike rezultira redanjem vrijednosti u 1D polje na način da se prvo uzimaju sve tri RGB vrijednosti prvog piksela i stave u polje, zatim se pomjera na drugi piksel i uzimaju RGB vrijednosti drugog piksela i stave u polje i tako dalje. Model očekuje poredak na način da se prvo uzme samo R vrijednost piksela i stavi polje, zatim se pomjera na drugi piksel i uzme se R vrijednost drugog piksela i stavi u polje i tako dok se ne uzmu sve R vrijednosti, zatim G vrijednosti i na kraju B vrijednosti. Ta transformacija prikazana je u kodu 4.7 na linijama 12 do 16.

```
float* read_data_file() {
    const int num_elements = samplesCommon::volume(m_input_dims);
```

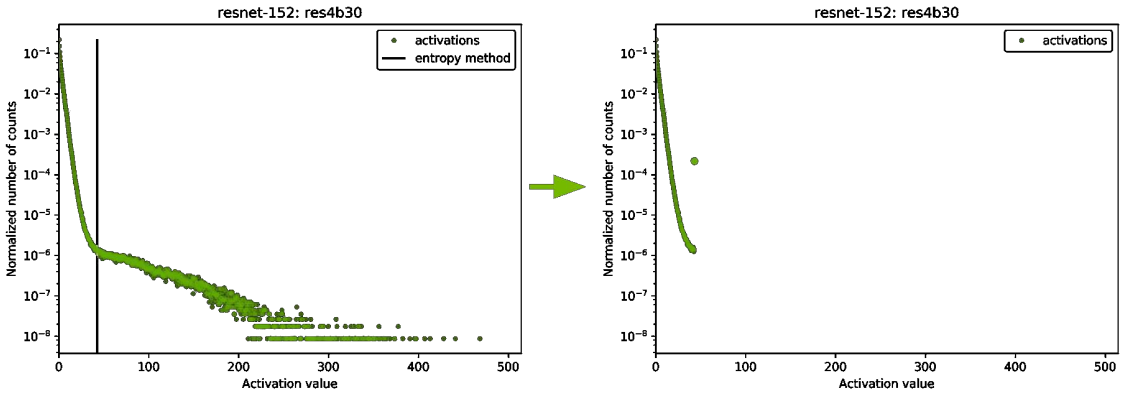
```
    int C, W, H;

    uint8_t* raw_data = stbi_load(m_image_file_names[m_curr_batch_idx].
    c_str(), &W, &H, &C, STBI_rgb);

    float* data = new float[num_elements];
    int n = W * H * C;
    int cnt = 0;

    // Change shape from HxWxC to CxWxH and normalize output with mean
    and std
    for(int c = 0; c < C; ++c) {
        for(int i = 0; i < n; i += 3) {
                data[cnt++] = (((float)raw_data[i + c]) - m_mean[c]) /
    m_std[c];
        }
    }
    delete[] raw_data;
    return data;
}
```

Programski kod 4.7: Prikaz učitavanja slike i redanja vrijedosti slike u 1D polje.



Slika 4.1: Prikaz distribucije aktivacija jednog sloja modela ResNet-152. Na lijevoj strani je prikazana distribucija normalnih aktivacija. Crna okomita crta predstavlja odabranu granicu raspona. Nakon primjene te granice dobiva se distribucija na desnoj strani. Bitno je primjeniti kako se sve vrijednosti desno od granice pritežu na vrijednost granice. Zbog toga je na desnom grafu nastala nova točka na granici [5].

Nakon što je implementirano iteriranje po podacima, potrebno je stvoriti kalibracijski objekt koji je predstavljen sučeljem `IInt8Calibrator`. U svim eksperimentima koristio se postupak kalibracije zasnovan na minimizaciji gubitka informacije sa simetričnim rasponom vrijednosti. Taj postupak implementiran je u `Int8EntropyCalibrator2` razredu

koji implementira `IInt8Calibrator` sučelje. Gubitak informacije mjeri se Kullback-Leibler divergencijom. Kako bi se mogla primjeniti KL divergencija potrebno je prvo za svaki sloj napraviti histogram aktivacija. Nakon prikupljanja histograma aktivacija potrebno je odabrati različite raspone kako bi se dobile nove distribucije koje će se kvantizirati. Odabire se onaj raspon za koji je gubitak informacije između normalne distribucije i distribucije formirane s tim rasponom minimalan. Na slici 4.1 može se vidjeti ovaj postupak. Za odabrani raspon KL divergencija je minimalna, tj. za bilo koji drugi raspon povećava se gubitak informacije.

4.5. Zaključivanje u TensorRT-u

Jednom kad je stvoren stroj za zaključivanje moguće je provesti zaključivanje nad dubokim modelom i dobiti semantičke predikcije modela. Prvo se iz stroja za zaključivanja stvara kontekst izvođenja (*engl.* execution context) koji provodi zaključivanje. Kontekst izvođenja predstavljen je sučeljem `IExecutionContext` i stvara se pozivom metode `createExecutionContext` nad strojem za zaključivanje.

Prije poziva zaključivanje potrebno je zauzeti memoriju na GPU za ulazni i izlazni tenzor, te stvoriti CUDA stream. Pozivom `cudaStreamCreate` metode obavalja se stvaranje CUDA stream-a. Zauzimanje memorije se obavlja pomoću dvostrukog pokazivača na void i pozivom funkcije `cudaMalloc`, kojoj se predaje broj byte-ova. U kodu 4.8 na liniji 10 zauzima se memorija na GPU za ulazni tenzor, a na liniji 11 se zauzima memorija za izlazni tenzor. Bitno je primijetiti kako je tip ulaznog tenzora `float`, a tip izlaznog tenzora je `int`. Kako u slučaju SwiftNet modela postoji samo jedan ulaz i jedan izlaz potrebno je stvoriti polje od dva pokazivača (jedan za ulazni tenzor, jedan za izlazni). Koristi se pokazivač na void jer ulazni i izlazni tip ne moraju biti jednaki. Nakon zauzimanja memorije potrebno je popuniti zauzetu memoriju. Pozivom metode `cudaMemcpyAsync` kopiraju se ulazne vrijednosti s CPU-a na GPU. Metodi je potrebno predati pokazivač na memoriju na GPU, pokazivač na memoriju na CPU, veličinu memorije u byte-ovima, CUDA stream, te smjer kopiranja. Ovaj postupak može se vidjeti u programskom kodu 4.8 na liniji 13. Parametar `cudaMemcpyHostToDevice` definira smjer kopiranja s CPU na GPU.

```
1 void* buffers[2];
2 float* input;
3
4 CityScapeBatchStream batch_stream(1, 1, "/path/to/data", "val");
5 auto context = SNUniquePtr<nvinfer1::IExecutionContext>(mEngine->
     createExecutionContext());
6
7 input = batch_stream.getBatch();
```

```
8
9  cudaStream_t stream;
10 CHECK(cudaStreamCreate(&stream));
11 CHECK(cudaMalloc(&buffers[0], 1024*2048*3*sizeof(float)));
12 CHECK(cudaMalloc(&buffers[1], 1024*2048*sizeof(int)));
13 CHECK(cudaMemcpyAsync(buffers[0], input, 1024 * 2048 * 3 * sizeof(float),
       cudaMemcpyHostToDevice, stream));
14
15 context->enqueue(1, buffers, stream, nullptr);
16
17 int* output = new int[19*1024*2048];
18 CHECK(cudaMemcpyAsync(buffers[1], output, 1024 * 2048 * sizeof(int),
      cudaMemcpyDeviceToHost, stream));
19 cudaStreamSynchronize(stream);
20 cudaStreamDestroy(stream);
21 CHECK(cudaFree(buffers[0]));
22 CHECK(cudaFree(buffers[1]));
23
24 delete[] input;
25 delete[] output;
```

Programski kod 4.8: Primjer izvođenja zaključivanja u TensorRT-u.

Nakon alociranja memorije i prebacivanja ulaznog tenzora na GPU moguće je izvršiti zaključivanje pozivom metode `enqueue` nad objektom konteksta izvođenja. Ta metode kao argumente prima dvostruki pokazivač na zauzete memorije na GPU, te CUDA stream. Kao rezultat zaključivanja dubokog modela popunjava se memorija na GPU koja je zauzeta za izlazni tenzor. Kako bi mogli spremiti rezultate izvođenja ili ih dalje obrađivati potrebno je prebaciti te vrijednosti nazad na CPU. To se također obavlja `cudaMemcpyAsync` metodom, samo se kao parametar smjera šalje `cudaMemcpyDeviceToHost` koji definira smjer kopiranja s GPU na CPU. Time je postupak zaključivanja završen.

Vrijeme zaključivanja računa se kako je prikazano u programskom kodu 4.9. Koristi se pomoćni razred za upravljanjem memorijom. Prvo se napravi 200 unaprijedni prolaza koji ne ulaze u izračun zatim se provede 100 unaprijedni prolaza i za svaki se mjeri vrijeme izvođenja. Na kraju se uzima prosječno vrijeme.

```
1 for(int i = 0; i < 200; ++i) {
2     buffers.copyInputToDevice();
3     context->executeV2(buffers.getDeviceBindings().data());
4 }
5
6 float time_avg = 0.0f;
7 for(int i = 0; i < 100; ++i) {
8     buffers.copyInputToDevice();
```

```
    auto data = buffers.getDeviceBindings().data();
    auto t1 = std::chrono::high_resolution_clock::now();
    context->executeV2(data);
    auto t2 = std::chrono::high_resolution_clock::now();
    float inference_time = ((float) std::chrono::duration_cast<std::
    chrono::milliseconds>(t2 - t1).count());
    buffers.copyOutputToHost();
    time_avg += inference_time;
    printf("Inference time: %.3f ms.\n", inference_time);
}
sample::gLogInfo << "Avg inference: " << time_avg / 100 << " ms." << std
    ::endl;
```

Programski kod 4.9: Prikaz računanja vremena zaključivanja.

5. Eksperimenti

U ovom poglavlju opisani su svi provedeni eksperimenti, te su analizirani njihovi rezultati. Eksperimenti su provedeni koristeći statičku kvantizaciju nakon učenja na skupu podataka *Cityscapes*.

5.1. SwiftNet single scale

Eksperimenti u ovom poglavlju provedeni su nad SwiftNet single scale arhitekturom s ResNet18 i ResNet34 arhitekturom kao *backbone*. Mixed model označava model koji ima *backbone* u INT8 preciznosti, a *spp* i *upsampling* dijelove u FP16 preciznosti.

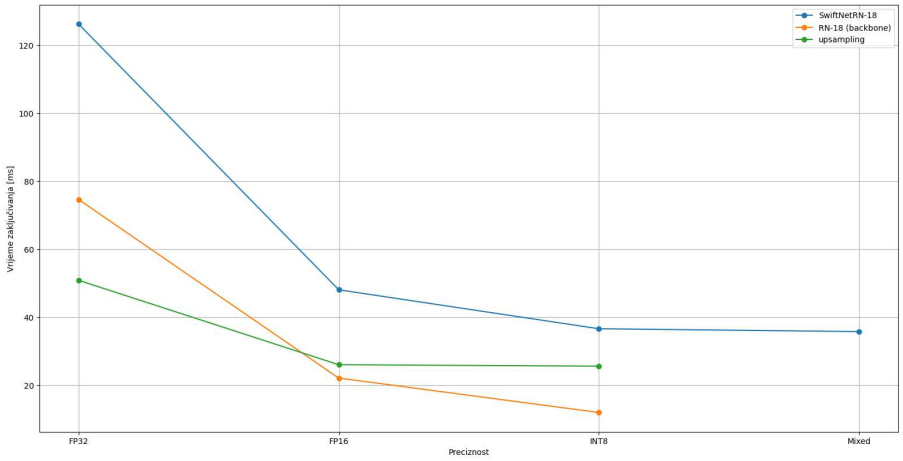
	mIoU	Accuracy	Inference time	FPS
FP32	75.52%	95.68%	126.23 ms	7.92
FP16	75.52%	95.68%	48.09 ms	20.79
INT8	75.04%	95.60%	36.63 ms	27.30
Mixed	75.07%	95.60%	35.80 ms	27.93

Tablica 5.1: Prikaz rezultata SwiftNet single scale arhitekture s ResNet18 arhitekturom kao *backbone*.

Inference time [ms]	FP32	FP16	INT8
SwiftNetRN-18	126.23 ms	48.09 ms	36.63 ms
RN-18 (backbone)	74.64 ms	22.13 ms	12.00 ms
RN-18 (backbone) + spp	75.35 ms	22.03 ms	11.00 ms
SwiftNetRN-18 - RN-18 (backbone)	51.59 ms	25.96 ms	24.63 ms

Tablica 5.2: Prikaz brzine izvođenja raznih dijelova SwiftNet single scale arhitekture s ResNet18 arhitekturom kao *backbone*.

Iz tablice 5.1 može se vidjeti kako INT8 model ima jako mal pad u točnosti u odnosu na FP32 model (0.48% mIoU i 0.08% accuracy), a ima dosta veliko ubrzanje. Ubrzanje u



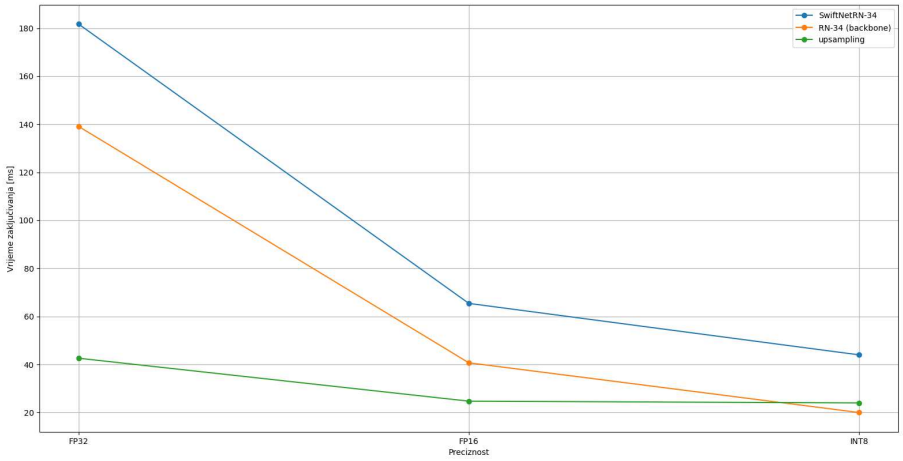
Slika 5.1: Vrijeme zaključivanja za razne dijelove arhitekture SwiftNetRN-18 single scale za različite preciznosti. Vrijeme zaključivanja za RN18 (backbone) je manje za INT8 preciznost dok je upsampling dio skoro pa jednak za FP16 i INT8 preciznosti.

odnosu na FP32 model iznosi 89.6 ms, a u odnosu na FP16 model iznosi 11.46 ms što je poboljšanje od 31.3% uz minimalan gubitak točnosti. Iz tablice 5.2 može se vidjeti kako RN-18 (backbone) INT8 ima skoro pa duplo ubrzanje u odnosu na isti FP16 model, dok *upsampling* dio traje skoro pa jednako. Ako se s tim usporedi rezultat *mixed* modela iz tablice 5.1 dobije se isti zaključak. *Mixed* model ćak ima i manje vrijeme zaključivanja u odnosu na INT8 model, što upućuje na to da *upsampling* dio arhitekture ne radi brže pod INT8 u odnosu na FP16.

Inference time [ms]	FP32	FP16	INT8
SwiftNetRN-34	181.74 ms	65.43 ms	44.00 ms
RN-34 (backbone)	139.13 ms	40.70 ms	20.00 ms
SwiftNetRN-34 - RN-34 (backbone)	42.61 ms	24.73 ms	24.00 ms

Tablica 5.3: Prikaz brzine izvođenja raznih dijelova SwiftNet single scale arhitekture s ResNet34 arhitekturom kao *backbone*.

U tablici 5.3 vidljivi su rezultati brzine izvođenja SwiftNet single scale arhitekture s ResNet34 arhitekturom kao *backbone*. Brzina izvođenja cijelog modela pod FP16 je 65.43 ms, dok je pod INT8 44.00 ms. Dakle INT8 zaključivanje daje ubrzanje od 21.43 ms, tj. poboljšanje od 48.7%. Nad arhitekturom RN-34 (*backbone*) dobije se skoro duplo ubrzanje pod INT8 u odnosu na FP16. Ako se usporede rezultati s tablicom 5.2 vidi se da i kod RN-34 *backbone*-a *upsampling* dio arhitekture traje skoro pa jednako pod FP16 i INT8 preciznosti.



Slika 5.2: Vrijeme zaključivanja za razne dijelove arhitekture SwiftNetRN-34 single scale za različite preciznosti. Vrijeme zaključivanja za RN-34 (backbone) je manje za INT8 preciznost dok je upsampling dio skoro pa jednak za FP16 i INT8 preciznosti.

5.2. SwiftNet pyramid

Eksperimenti u ovom poglavlju provedeni su nad SwiftNet pyramid arhitekturom s ResNet18 i ResNet34 arhitekturom kao *backbone*.

	mIoU	Accuracy	Inference time	FPS
FP32	76.57%	95.79%	141.15 ms	7.08
FP16	76.57%	95.79%	57.08 ms	17.52
INT8	76.31%	95.75%	43.02 ms	23.25

Tablica 5.4: Prikaz rezultata SwiftNet pyramid arhitekture s ResNet18 arhitekturom kao *backbone*.

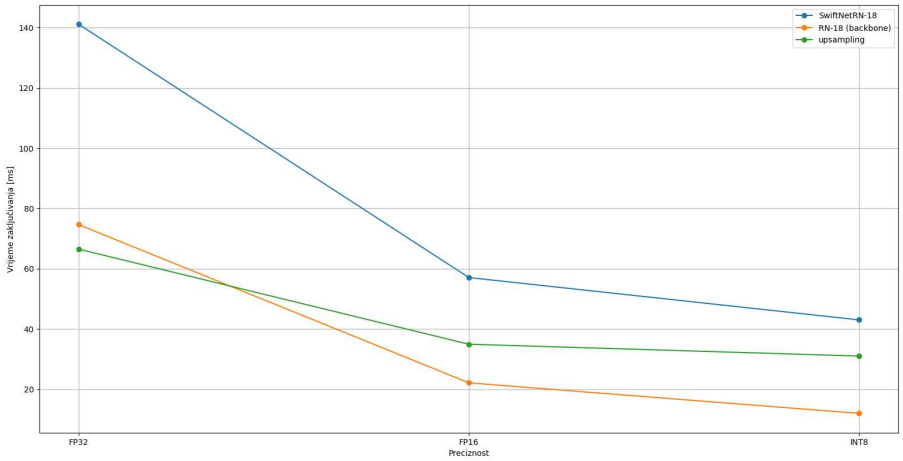
Iz tablice 5.4 može se vidjeti kako SwiftNetRN-18 pyramid arhitektura, kao i SwiftNetRN-18 single scale arhitektura, s INT8 preciznosti ima minimalan gubitak u mIoU rezultatu i točnosti u odnosu na model s FP32 preciznosti, a ima dosta veliko ubrzanje. Gubitak u mIoU rezultatu iznosi 0.26%, dok gubitak u točnosti iznosi 0.04%, a ubrzanje u vremenu zaključivanja iznosi 98.02 ms. Ako se usporedi INT8 model s FP16 modelom, gubitak u točnosti i mIoU rezultatu je jednak kao i kod FP32 modela, dok ubrzanje iznosi 14.02 ms, što je poboljšanje od 32,7%.

Ako se usporede rezultati tablice 5.4 i tablice 5.1 vidi se kako SwiftNetRN-18 pyramid FP32 arhitektura ima za 1.05% bolji mIoU rezultat, te za 0.11% bolju točnost u odnosu na SwiftNetRN-18 single scale FP32 arhitekturu. No to povećanje u mIoU rezultatu i točnosti

dolazi uz cijenu povećanja vremena zaključivanja. Obični FP32 model ima 14,92 ms veće vrijeme zaključivanja, dok FP16 model ima 8,99 ms, a INT8 model ima 6,39 ms veće vrijeme zaključivanja.

Inference time [ms]	FP32	FP16	INT8
SwiftNetRN-18	141.15 ms	57.08 ms	43.02 ms
RN-18 (backbone)	74.64 ms	22.13 ms	12.00 ms
SwiftNetRN-18 - RN-18 (backbone)	98.54 ms	34.95 ms	31.02 ms

Tablica 5.5: Prikaz brzine izvođenja raznih dijelova SwiftNet pyramid arhitekture s ResNet18 arhitekturom kao *backbone*.



Slika 5.3: Vrijeme zaključivanja za razne dijelove arhitekture SwiftNetRN-18 pyramid za različite preciznosti. *Upsampling* dio s INT8 preciznosti traje 3.93 ms kraće nego s FP16 preciznosti.

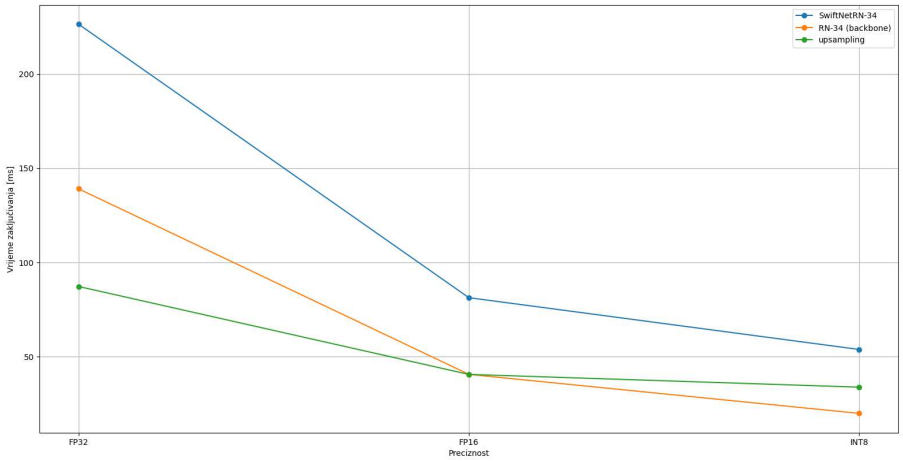
Iz tablice 5.5 može se vidjeti kako i kod SwiftNet pyramid arhitekture *upsampling* za FP16 i INT8 preciznost traje skoro pa jednako, kao kod SwiftNet single scale arhitekture. Ali *upsampling* dio kod SwiftNet pyramid arhitekture traje duže nego kod single scale arhitekture. Zbog toga vrijeme zaključivanja cijelog SwiftNet pyramid modela traje duže u odnosu na SwiftNet single scale model s obzirom kako je *backbone* dio arhitekture jednak.

U tablici 5.6 vidljivi su rezultati brzine izvođenja SwiftNet pyramid arhitekture s ResNet34 arhitekturom kao *backbone*. Brzina izvođenja cijelog modela s FP16 preciznosti je 81.38 ms, dok je s INT8 preciznosti 53.91 ms. Dakle, zaključivanje s INT8 preciznosti daje ubrzanje od 27.47 ms, tj. poboljšanje od 50.9%. Ako se usporede rezultati s tablicom 5.4 vidi se da *upsampling* dio, kad se koristi RN-34 arhitektura kao *backbone*, traje nešto duže

Inference time [ms]	FP32	FP16	INT8
SwiftNetRN-34	226.46 ms	81.38 ms	53.91 ms
RN-34 (backbone)	139.13 ms	40.70 ms	20.00 ms
SwiftNetRN-34 - RN-34 (backbone)	87.33 ms	40.68 ms	33.91 ms

Tablica 5.6: Prikaz brzine izvođenja raznih dijelova SwiftNet pyramid arhitekture s ResNet34 arhitekturom kao *backbone*.

kod modela s FP16 preciznosti u odnosu na model s INT8 preciznosti. Mogući razlog za to je način na koji *TensorRT* izvodi optimizaciju neuronske mreže.



Slika 5.4: Vrijeme zaključivanja za razne dijelove arhitekture SwiftNetRN-34 pyramid za različite preciznosti. *Upsampling* dio s INT8 preciznosti traje 6.77 ms kraće nego s FP16 preciznosti. *Backbone* dio s FP16 preciznosti traje jednako kao i *upsampling* dio.

6. Zaključak

Zaključak.

LITERATURA

[1] Hello AI World guide to deploying deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson. https://github.com/dusty-nv/jetson-inference. Accessed: 21-06-2021.

[2] ONNX | Open standard for machine learning interoperability. https://github.com/onnx/onnx. Accessed: 21-06-2021.

[3] Introduction to Quantization on PyTorch | PyTorch. https://pytorch.org/blog/introduction-to-quantization-on-pytorch/. Accessed: 21-06-2021.

[4] PyTorch documentation - PyTorch 1.9.0 documentation. https://pytorch.org/docs/stable/index.html. Accessed: 21-06-2021.

[5] NVIDIA TensorRT documentation | NVIDIA Developer. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html, . Accessed: 21-06-2021.

[6] NVIDIA TensorRT | NVIDIA Developer. https://developer.nvidia.com/tensorrt, . Accessed: 21-06-2021.

[7] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, i Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. U *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

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

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

[10] Marin Oršić, Ivan Krešo, Petra Bevandić, i Siniša Šegvić. In defense of pre-trained imagenet architectures for real-time semantic segmentation of road-driving images. *CoRR*, abs/1903.08469, 2019. URL http://arxiv.org/abs/1903.08469.

[11] 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.

Kvantizacija dubokih konvolucijskih modela

Sažetak

Ovaj rad se bavi kvantiziranjem dubokih konvolucijskih modela. Svrha kvantizacije je provođenje izračuna u nižoj preciznosti kako bi se smanjilo vrijeme zaključivanja dubokog modela. Kvantiziran je model SwiftNet na problemu semantičke segmentacije. Za implementaciju korišten je okvir TensorRT, te je model kvantiziran za uređaj NVIDIA Jetson AGX Xavier. Pokazalo se kako kvantizirani modela ima puno manje vrijeme zaključivanja u odnosu na FP32 model uz minimalan gubitak točnosti, te nešto manje vrijeme zaključivanja nego FP16 model također uz minimalan gubitak točnosti.

Ključne riječi: Kvantizacija, konvolucijski modeli, duboko učenje, TensorRT

Quantization of deep convolutional models

Abstract

The goal of this thesis is quantization of deep convolutional models. The goal of quantization is to use lower precision for computation which lowers inference time of the deep model. Model which is quantized in this thesis is SwiftNet model on the problem of semantic segmentation. For implementation TensorRT framework was used and models was quantized for the NVIDIA Jetson AGX Xavier embedded hardware. It has been shown that quantized model has much lower inference time than FP32 model with minimal accuracy loss and little lower inference time than FP16 model also with minimal accuracy loss.

Keywords: Quantization, convolutional models, deep learning, TensorRT