Razvoj, optimizacija i implementacija dubokih neuronskih mreža na FPGA

Miličević, Davorin

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij automobilskog računarstva i komunikacija

Razvoj, optimizacija i implementacija dubokih neuronskih mreža na FPGA

Diplomski rad

Davorin Miličević

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za ocjenu diplomskog rada na sveučilišnom diplomskom studiju

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	Davorin Miličević
Studij, smjer:	Sveučilišni diplomski studij Automobilsko računarstvo i
Mat. br. pristupnika, god.	D-70 ARK, 07.10.2022.
JMBAG:	0165083320
Mentor:	doc. dr. sc. Ivan Vidović
Sumentor:	
Sumentor iz tvrtke:	Filip Novoselnik
Predsjednik Povjerenstva:	izv. prof. dr. sc. Tomislav Matić
Član Povjerenstva 1:	doc. dr. sc. Ivan Vidović
Član Povjerenstva 2:	izv. prof. dr. sc. Ivan Aleksi
Naslov diplomskog rada:	Razvoj, optimizacija i implementacija dubokih neuronskih mreža na FPGA
Znanstvena grana diplomskog rada:	**Procesno računarstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U ovom radu potrebno je istražiti mogućnosti implementacije modela neuronskih mreža na FPGA integrirane sklopove. Nakon istraživanja, potrebno je realizirati vlastiti pristup za izgradnju modela i njegovu prilagodbu za implementaciju na FPGA integrirane sklopove. Potrebno je staviti naglasak na optimizaciju modela dubokih neuronskih mreža za dobivanje što boljih performansi u pogledu zauzeća memorije i potrošnje energije. Razvijeni i implementirani model potrebno je evaluirati na problemima kao što su detekcija i klasifikacija objekata na slici
Datum ocjene pismenog dijela diplomskog rada od strane mentora:	02.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	12.9.2024.
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	16.09.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 16.09.2024.

Ime i prezime Pristupnika:	Davorin Miličević
Studij:	Sveučilišni diplomski studij Automobilsko računarstvo i komunikacije
Mat. br. Pristupnika, godina upisa:	D-70 ARK, 07.10.2022.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Razvoj, optimizacija i implementacija dubokih neuronskih mreža na FPGA**

izrađen pod vodstvom mentora doc. dr. sc. Ivan Vidović

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

1. UVOD

U suvremenom svijetu tehnološka dostignuća sve više ovise o dubokim neuronskim mrežama (engl. *Deep Neural Networks* - DNN), koje su temelj inovacija poput autonomnih vozila te analize slika i zvuka. Ove mreže zahtijevaju sofisticirane računalne platforme, a programirljivi logički sklopovi (engl. *Field Programmable Gate Arrays* - FPGA) izdvajaju se kao efektivno rješenje, osobito za konvolucijske neuronske mreže (engl. *Convolution Neural Networks* - CNN) koje su ključne pri implementaciji postupaka za detekciju, segmentaciju i klasifikaciju. Izazovi implementacije i optimizacije CNN-a na FPGA platformama odražavaju potrebu za visokoučinkovitim računalnim rješenjima koja moraju uskladiti ograničene hardverske resurse s potrebom za visokom računalnom efikasnošću. Zato su potrebni alati poput Brevitas-a za optimizaciju modela unutar PyTorch okruženja i FINN kompajlera (engl. *Flexile Inference of Neural Networks*) za prilagodbu modela na FPGA, uz podršku Vitis, Vivado i Vitis HLS (engl. *High Level Synthesis*) alata za implementaciju. Ovakav pristup usmjeren je na razvoj metoda koje omogućavaju efikasno korištenje CNN-a na FPGA, s posebnim fokusom na primjenjivost na Xilinx PYNQ-Z2 platformi, gdje je bit same izvedbe pojednostavljeni i automatizirani pristup konačnim rješenjima. Analiza potencijala FPGA otkriva mogućnosti za poboljšanje DNN modela, kao što su očuvanje točnosti i optimizacija potrošnje energije. Napredne metode optimizacije i kvantizacije ključne su za usklađivanje modela s ograničenjima FPGA tehnologije, omogućujući njihovu širu primjenu i razvoj energetski efikasnih platformi za složene aplikacije [1, 2]. Ove tehnike, prikazane kroz primjere na PYNQ-Z2 platformi u [1], pružaju osnovu za temeljitu analizu i evaluaciju ističući ključne aspekte koji utječu na performanse i učinkovitost CNN-a na FPGA.

Kontinuirani napredak i inovacije u području dubokog učenja na FPGA ključni su za otvaranje novih mogućnosti i pružanje robustnih, skalabilnih i energetski efikasnih platformi za složene aplikacije dubokog učenja. Razvoj, optimizacija i implementacija DNN-a na FPGA platformama predstavljaju vitalna područja istraživanja koja će nastaviti oblikovati budućnost tehnologije i umjetne inteligencije [3]. Ovo područje ne samo da pruža temelje za tehnološke inovacije nego također postavlja okvir za buduća istraživanja.

U konačnici integracija dubokih neuronskih mreža s FPGA tehnologijom ne predstavlja samo tehnički izazov već i priliku za stvaranje novih paradigmi u računarstvu. Drugim riječima, pružaju se nove mogućnosti za inovacije, posebno u scenarijima poput detekcije i klasifikacije objekata snimljenih dronom [4]. Dinamičko rekonfiguriranje modela omogućuje prilagodbu sustava specifičnim zahtjevima, što dovodi do razvoja efikasnijih i pouzdanijih rješenja za primjene u

realno-vremenskim zahtjevima. Ova fleksibilnost omogućuje sustavu da koristi različite modele detekcije ovisno o vanjskim uvjetima, primjerice, promjeni visine drona. Na taj način FPGA platforme omogućuju optimizaciju performansi i energetske učinkovitosti kroz dinamičko prilagođavanje modela, što je ključno za sustave koji moraju reagirati na promjenjive uvjete u stvarnom vremenu.

Cilj ovog diplomskog rada je razviti i evaluirati metode za efikasnu implementaciju dubokih neuronskih mreža na FPGA uređajima, koristeći alate kao što su Brevitas za optimizaciju modela unutar PyTorch okruženja i FINN kompajler za njihovu adaptaciju na FPGA. Kroz primjenu ovih metoda, u radu se provodi istraživanje o dinamičkom rekonfiguriranju DNN modela na Xilinx PYNQ-Z2 platformi, s ciljem optimizacije performansi i smanjenja potrošnje energije u realnim aplikacijama, kao što su detekcija i klasifikacija objekata snimljenih dronom. Ovaj pristup omogućuje detaljno razumijevanje kako integracija naprednih alata i sustava može unaprijediti primjenu DNN-a na FPGA, pružajući temelj za buduće inovacije u području umjetne inteligencije.

Diplomski rad podijeljen je u šest poglavlja koja su ukratko opisana u nastavku. *Uvod* pruža osnovni pregled problematike dubokih neuronskih mreža i njihove implementacije na FPGA platformama, s naglaskom na optimizaciju i efikasnost. U poglavlju *Neuronske mreže na FPGA* razmatraju se DNN, upotreba FPGA za DNN, alati Brevitas i FINN te implementacija neuronskih mreža na FPGA. Postupak implementacije konvolucijskih neuronskih mreža, klasifikacija i detekcija objekata te prilagodba i implementacija modela na FPGA opisani su u poglavlju *Metodologija.* U poglavlju *Rezultati implementacije i inferencije neuronskih mreža* analiziraju se rezultati implementacije DNN modela na FPGA, uključujući točnost, brzinu inferencije i potrošnju resursa. Poglavlje *Dinamičko rekonfiguriranje FPGA sustava za prilagođenu detekciju* bavi se implementacijom dinamičke rekonfiguracije na FPGA uređajima, omogućujući prilagodbu detekcijskih modela u realnom vremenu. U zaključnom poglavlju sažimaju se postignuti rezultati i daju prijedlozi za buduća istraživanja.

2. NEURONSKE MREŽE NA FPGA

U ovom poglavlju naveden je pregled inovacija i pristupa u optimizaciji i implementaciji DNN-a na FPGA istražujući kako alati poput Brevitas i FINN omogućavaju efikasnu adaptaciju i primjenu modela dubokog učenja u različitim aplikacijskim domenama.

2.1. Osnove dubokih neuronskih mreža

Duboke neuronske mreže su temelj suvremene umjetne inteligencije, omogućujući napredak u brojnim područjima kao što su računalni vid, obrada ljudskog govora i autonomna vožnja. DNN mreže sastoje se od višeslojnih perceptrona, gdje svaki sloj ima sposobnost učenja specifičnih značajki iz ulaznih podataka. Ova hijerarhijska struktura omogućuje da se iz ulaznih podataka izvlače i uče apstraktne značajke, čime se postiže efikasnost u interpretaciji složenih podatkovnih skupova [5].

Jedan od ključnih aspekata DNN-a jest algoritam povratne propagacije, koji omogućuje mreži da se prilagodi i poboljša na temelju pogrešaka u predviđanju. Kroz iterativni proces, mreža postupno optimizira svoje težine i pristranosti (engl. *bias*) kako bi se smanjila greška, čime se poboljšava njena sposobnost predviđanja ili klasifikacije. DNN mreže su inspirirane strukturom i funkcijom ljudskog mozga, što se odražava u njihovoj sposobnosti da obrade velike količine podataka i izvedu složene zaključke na temelju naučenih značajki [5]. Ova analogija s mozgom nije samo metaforička; istraživanja u području neuroznanosti i računalne znanosti sve više se isprepliću kako bi se bolje razumjeli temeljni procesi učenja i odlučivanja. Unatoč svojoj moći implementacija i treniranje DNN-a dolaze s određenim izazovima, uključujući potrebu za velikim količinama podataka za učenje i visokim računalnim resursima za procesiranje tih podataka [6]. Međutim, razvoj specijaliziranih hardverskih rješenja, kao što su grafičke procesorske jedinice (engl. *graphics processing unit* - GPU) i neuronske procesne jedinice (engl. neural processing unit - NPU), omogućio je brže i efikasnije treniranje mreža, otvarajući put za širu primjenu DNN-a u praksi. Kao što se ističe u [1, 5], duboko učenje predstavlja revolucionaran napredak u području strojnog učenja, pružajući temelje za razvoj inteligentnih sustava sposobnih za samostalno učenje i adaptaciju. Zahvaljujući ovim naprecima, duboke neuronske mreže postaju sve relevantnije u razvoju naprednih sustava koji su sposobni za autonomno donošenje odluka i inteligentnu obradu podataka.

2.2. FPGA u kontekstu dubokih neuronskih mreža

U prethodnom poglavlju istaknuto je kako su razvoj i primjena dubokih neuronskih mreža značajno oblikovali suvremenu tehnologiju, uključujući autonomna vozila, sustave za obradu govora i slika, medicinske dijagnostičke alate te napredne robotske sustave. Paralelno s ovim napretkom, FPGA tehnologija ističe se kao ključna komponenta za efikasno izvršavanje DNN algoritama, pružajući jedinstvenu kombinaciju fleksibilnosti, brzine i energetske učinkovitosti. U kontekstu DNN-a, FPGA platforme omogućavaju značajne prednosti u brzini obrade i adaptabilnosti, što ih čini idealnim za implementaciju složenih modela dubokog učenja u različitim aplikacijama. Glavna prednost FPGA u odnosu na tradicionalne platforme je njihova sposobnost paralelizacije operacija, što omogućuje učinkovito izvršavanje slojevitih struktura DNN-a. Ova sposobnost omogućava FPGA uređajima da uz malo kašnjenje (engl. *latency*) brzo obrađuju velike količine podataka, smanjujući vrijeme potrebno za treniranje i izvršavanje modela. Osim toga, njihova rekonfigurabilnost pruža dodatnu fleksibilnost, omogućujući prilagodbe hardvera specifičnim potrebama algoritma dubokog učenja bez potrebe za fizičkom zamjenom komponenti. U literaturi, radovi kao što su [1, 2] istražuju različite strategije i metodologije za optimizaciju i implementaciju DNN-a na FPGA, naglašavajući kako se kroz inovativne pristupe može postići visoka točnost i efikasnost. Ovi radovi pružaju dubinski uvid u mogućnosti i izazove povezane s korištenjem FPGA tehnologije u domeni dubokog učenja, pokazujući put k razvoju učinkovitijih i energetski efikasnijih rješenja.

Napredak u tehnologiji FPGA također omogućava njihovu sve širu primjenu u realnim scenarijima, kao što su sustavi za detekciju objekata korištenjem dronova [4]. Ovakve aplikacije demonstriraju praktičnu vrijednost FPGA u dinamičnim okruženjima gdje su potrebne brza obrada podataka i visoka točnost. Kroz kontinuirani razvoj i istraživanje, FPGA tehnologija i duboko učenje zajedno pružaju snažan temelj za buduće tehnološke inovacije. Integracija ovih dviju tehnologija ne samo da unapređuje postojeće aplikacije već i otvara put za stvaranje novih rješenja koja će oblikovati budućnost umjetne inteligencije.

2.3. Brevitas i FINN

Funkcionalnost potrebna za rad s dubokim neuronskim mrežama te njihovu implementaciju na ugradbeni sustav je zadovoljena kroz ključne alate Brevitas i FINN. Brevitas je alat razvijen unutar PyTorch okruženja, dizajniran za preciznu kvantizaciju DNN modela [7]. Kvantizacija, postupak smanjenja preciznosti težina unutar mreže, ključna je za prilagodbu modela

prema ograničenim resursima FPGA uređaja. Kroz Brevitas, se može podešavanjem kvantizacijskih parametara tijekom treninga značajno smanjiti potreba za memorijom i poboljšati brzina izvršavanja, uz minimalan utjecaj na točnost modela. Ovakva optimizacija je neophodna za implementaciju složenih DNN modela u realnom vremenu na FPGA uređajima, gdje su prostor i energetska učinkovitost od ključne važnosti [2].

FINN, alat razvijen od strane Xilinx-a, služi za brzu adaptaciju i implementaciju kvantiziranih neuronskih mreža na FPGA [8]. Kroz automatizirani proces, FINN omogućava generiranje sintetiziranih hardverskih datoteka iz modela napisanih u visoko razinskom jeziku. To uključuje podršku za specifične operacije i aktivacije prilagođene kvantiziranim modelima, omogućavajući istovremeno maksimalne performanse i energetsku učinkovitost na željenim FPGA platformama. Ovaj pristup olakšava konverziju i optimizaciju DNN modela za FPGA, pružajući jedinstvenu priliku za efikasno korištenje takve tehnologije u dubokom učenju [9].

Kombinacija Brevitas-a za optimizaciju modela i FINN-a za njihovu implementaciju omogućuje iskorištavanje potencijala FPGA tehnologije [8]. Razvoj modela na jeziku visoke razine olakšan je te nije potrebno detaljno poznavanje hardvera. Ova sinergija pruža moćan alat za trenutne potrebe dubokog učenja i postavlja temelje za buduće inovacije, omogućujući razvoj naprednijih i energetski efikasnijih DNN rješenja. Kombinacija ovih alata predstavlja učinkovit put prema pokretanju modela na FPGA.

2.4. Implementacija neuronskih mreža na FPGA

Rasprava u različitim radovima u kontekstu dubokih neuronskih mreža na FPGA platformama razotkriva širok spektar pristupa i inovacija koje su oblikovale trenutno stanje rada i istraživanja u području implementacije DNN-a na FPGA platforme. Od binariziranih neuronskih mreža do sofisticiranih alata za optimizaciju i akceleraciju, otkrivene su različite metode za povećanje efikasnosti i smanjenje resursnih zahtjeva DNN-a na FPGA.

Neki radovi, poput onog od Lianga [1] , bave se binariziranim neuronskim mrežama (engl. *Binarized Neural Networks* - BNN) implementiranim na FPGA. Demonstrira se mogućnost značajnog smanjivanja potrošnje hardverskih resursa uz zadržavanje prihvatljive razine točnosti [1]. Ovaj pristup omogućava brzu i efikasnu realno-vremensku obradu podataka s ograničenim hardverskim kapacitetom.

Također, potrebno je izdvojiti i rad od Sharma [2]. Fokus istraživanja se očitava u načinu kako se iz visokog opisa modela mogu automatski generirati sintetizabilni akceleratori za DNN na FPGA uređajima, što pojednostavljuje postizanje visoke performanse i energetske efikasnosti [2]. Ova

studija naglašava potencijal FPGA tehnologije kao ključnog faktora u budućnosti dubokog učenja. S druge strane, kroz rad od Mittal [10] se pruža opsežan pregled FPGA baziranih akceleratora za konvolucijske neuronske mreže, kategorizirajući različite pristupe i tehnike implementacije. U ovom radu se ističe važnost FPGA u premošćivanju razlike između teorijskih mogućnosti dubokog učenja i njegove praktične primjenjivosti [10]. Tehnike za akceleraciju dubokih mreža na FPGA su obrađene od strane Shawahna [11], s fokusom na ključne značajke koje koriste različite tehnike za poboljšanje performansi. Ovi pregledi su ključni za razumijevanje kako se teorijski koncepti dubokog učenja mogu uspješno primijeniti u praktičnim hardverskim rješenjima.

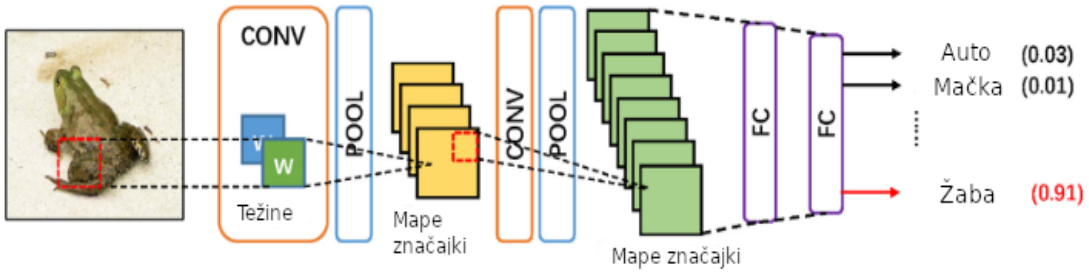
U konačnici, od strane Zhanga [3] razvijena je Caffeine, hardversko/softversko ko-dizajnirana biblioteka namijenjena efikasnom ubrzanju CNN-a i potpuno povezanih mreža (engl. *Fully Connected Networks* - FCN) na FPGA, s naglaskom na uniformnu reprezentaciju i optimizaciju mikroarhitekture akceleratora [3]. Ovaj rad ilustrira kako prilagođeni hardverski dizajni mogu značajno poboljšati izvođenje dubokih mreža, čineći FPGA privlačnim izborom za buduće istraživanje i razvoj u području umjetne inteligencije. Kroz ove studije, jasno je da kontinuirani napredak u tehnologiji FPGA i metodologijama dubokog učenja omogućava razvoj učinkovitijih i energetski efikasnijih rješenja.

3. METODOLOGIJA

Kroz ovo poglavlje detaljno se objašnjava koncept, matematički princip i arhitektura konvolucijskih neuronskih mreža, čime se pruža temelj za razumijevanje njihove primjene i efikasnosti pri obradi slika tijekom implementacije na FPGA.

3.1. Konvolucijske Neuronske Mreže (CNN)

Konvolucijske neuronske mreže su specijalizirani tip dubokih neuronskih mreža koje su posebno dizajnirane za obradu podataka koji imaju poznatu rešetkastu strukturu, kao što su slike. CNN koristi seriju konvolucijskih slojeva koji automatski i adaptivno uče prostorne hijerarhije značajki iz slika [12]. Ove značajke variraju od jednostavnih rubova i tekstura do složenih objekata u višim slojevima mreže. Konvolucijska neuronska mreža sastavljena je od više slojeva kroz koje određeni podaci prolaze. Primjer jedne tipične mreže je prikazan na slici 3.1.



Slika 3.1. Primjer konvolucijske neuronske mreže [1].

Na slici 3.1. su vidljivi slojevi *CONV, POOL, FC* koji su okosnica mreže. Svaki sloj je zaslužan za dobivanje konačne predikcije, odnosno odluke koju mreža može donijeti. Kao takva, CNN se koristi za klasifikaciju slika, pri čemu mreža identificira prisutne kategorije objekata, a osim toga i za detekciju objekata, pri čemu CNN lokalizira i klasificira objekte unutar slike.
Stoga jezgru konvolucijske neuronske mreže čine slojevi opisani u nastavku.:

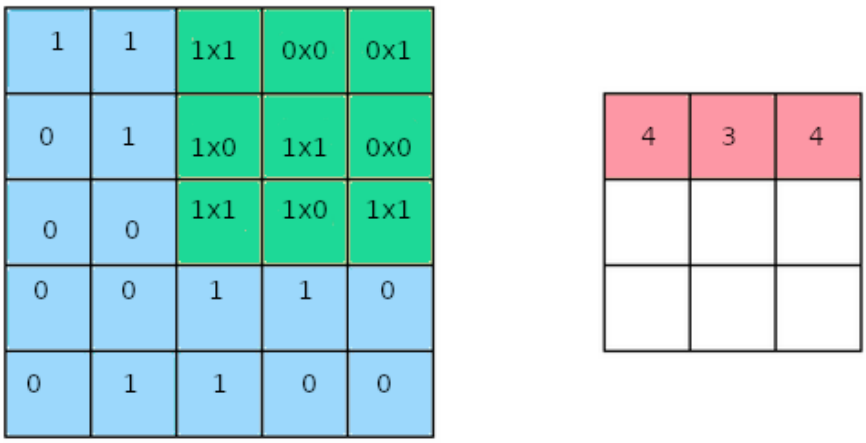
- **Konvolucijski sloj (*CONV*)** - za svoj rad koristi skup filtera (jezgri) koji se primjenjuju na ulazne podatke kroz operaciju konvolucije. Kao rezultat primjene filtera na ulazne podatke dobivaju se aktivacijske mape, što omogućava izdvajanje važnih značajki iz ulaznih slika. Za *n*-ti filter na *i, j* lokaciji, aktivacijsku mapu moguće je izračunati korištenjem jednadžbe (3-1) [1].

$$A_n^{(l)}(i,j) = B^{(l)}(n) + \sum_{m=1}^{N_{in}^{(l)}} W^{(l)}(m,n) * I_m^{(l)}(i,j) \qquad (3\text{-}1)$$

U jednadžbi (3-1) $A_n(i,j)$ označava aktivacijsku mapu, $B^{(l)}(n)$ je *bias* za *n*-ti filter, $W^{(l)}(m,n)$ su težine, a $I^{(l)}{}_m(i,j)$ ulazna slika ili prethodna aktivacijska mapa. Oznaka l predstavlja trenutni sloj u kojem se vrši operacija, bilo da se radi o konvolucijskom sloju, sloju sažimanja, ili potpuno povezanom sloju. U svakoj jednadžbi, l označava sloj u kojem se odvija konkretna operacija, čime se olakšava praćenje procesa kroz različite slojeve neuronske mreže. Operacija $*$ označava konvoluciju koja je opisana jednadžbom (3-2) [1].

$$(X * Y)(i,j) = \sum_{m=1}^{K} \sum_{n=1}^{K} X(i+m, j+n) \cdot Y(m,n) \qquad (3\text{-}2)$$

Prema navedenoj jednadžbi (3-2) $X * Y$ predstavlja operaciju konvolucije između dvije matrice, gdje X označava ulaznu sliku ili značajku, a Y predstavlja filter (jezgru) konvolucije. Indeksi (i, j) označavaju pozicije u izlaznoj matrici, poznatoj kao aktivacijska mapa. K predstavlja veličinu jezgre, odnosno dimenzije filtera. Operacija konvolucije uključuje preklapanje filtera Y na svaku moguću lokaciju ulazne matrice X. Za svaku takvu poziciju, vrši se množenje odgovarajućih elemenata matrice X i jezgre Y te se zatim sumiraju dobiveni produkti kako bi se formirao element (i, j) aktivacijske mape. Ova operacija se ponavlja za svaku lokaciju u matrici X, što rezultira potpuno formiranom aktivacijskom mapom koja sadrži značajke izvučene iz originalnog ulaza pomoću filtera Y [1]. Konvolucijska operacija omogućuje mreži da detektira specifične značajke kao što su rubovi, teksture ili drugi vizualni uzorci u ulaznim podacima. Osnova rada primjene filtera na ulaznu sliku i stvaranje aktivacijske mape je prikazana na slici 3.2.



Slika 3.2. Primjena 3x3 filtera na ulazni podatak 5x5.

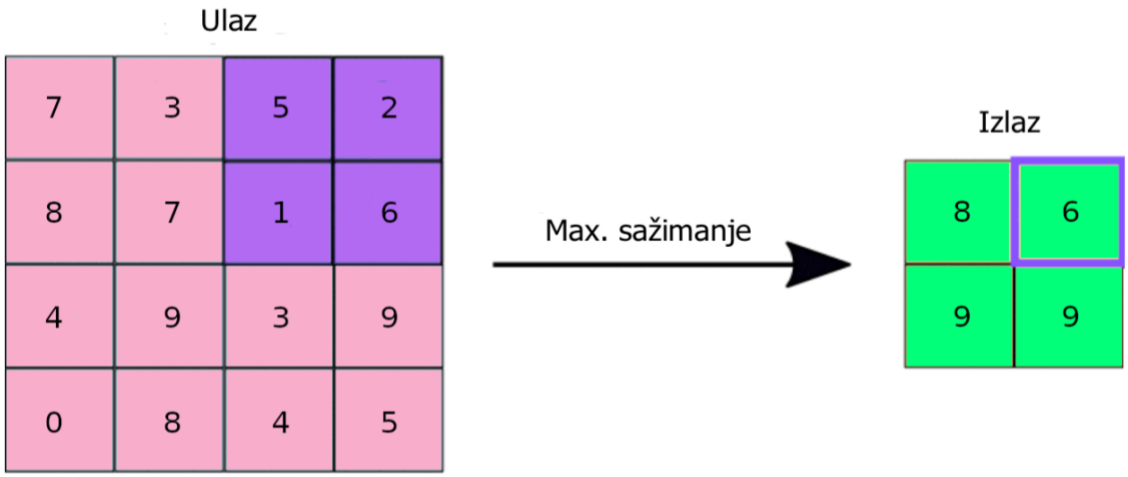
Da bi se dobila kompletna aktivacijska mapa, ovaj postupak se ponavlja za svaku moguću poziciju filtera preko ulazne matrice. Krajnji rezultat je aktivacijska mapa manje dimenzije

u odnosu na ulaznu matricu (u ovom slučaju 3x3), koja predstavlja prostorne značajke uočene filtriranjem.

- **Sloj sažimanja (*POOL)*** - sloj sažimanja smanjuje dimenzionalnost aktivacijskih mapa s ciljem smanjenja broja parametara i računske složenosti. Postoji više vrsta slojeva sažimanja poput maksimalnog, srednjeg, polusrednjeg [13], no u ovom diplomskom radu je najčešće korištena operacija maksimalnog sažimanja koja je definirana jednadžbom (3-3).

$$A^{(l)}(i,j) = max(I^{(l)}_{K\times K}(i,j)) \tag{3-3}$$

Ovaj proces odabire maksimalnu vrijednost unutar područja *K x K* za stvaranje umanjene aktivacijske mape. Vizualni primjer koji jasnije predočava proces je prikazan na slici 3.3.



Slika 3.3. Operacija maksimalnog sažimanja na ulaznom podatku 4x4 te dobivanje izlaza 2x2.

Postupak prikazan na slici 3.3. se nastavlja pomicanjem filtera vodoravno i okomito, ponavljajući postupak dok se ne popuni cijela izlazna matrica. Dobivene mape značajki, odnosno težine koje su ekstraktirane iz njih predstavljaju ulazne jedinice u potpuno povezani sloj.

- **Potpuno povezani sloj (*FC*)** - predstavlja izlazni sloj mreže koji se može koristi za klasifikaciju ili regresiju integrirajući spomenute naučene značajke iz prethodnih slojeva. Matematička funkcija sumiranja i množenja težina s ulaznim značajkama definirana je jednadžbom (3-4) [1].

$$A^{(l)}(n) = B^{(l)}(n) + \sum_{m=1}^{N_{in}^{(l)}} I^{(l)}(m) \cdot W^{(l)}(m,n) \tag{3-4}$$

U navedenoj jednadžbi (3-4), $A^{(l)}(n)$ predstavlja aktivaciju *n*-tog neurona, $B^{(l)}(n)$ je *bias*, a $W^{(l)}(m,n)$ su težine, dok $I^{(l)}{}_m(m)$ su ulazni podaci ili aktivacije s prethodnog sloja za *m*-ti neuron. Uz navedene ključne slojeve, konvolucijska neuronska mreža se sastoji i od aktivacijskih funkcija poput *ReLU, tanh i sigmoid* [12], koje dodaju nelinearnost u mrežu, omogućavajući joj učenje kompleksnih uzoraka.

Osim navedenih ključnih slojeva, u CNN se susreću slojevi navedeni u nastavku.

- **Normalizacija grupa (*batch norm*)** - predstavlja sloj koji ubrzava treniranje i poboljšava stabilnost mreže normaliziranjem ulaznih jedinica. Matematički je ovaj postupak definiran jednadžbama (3-5) i (3-6) [1].

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mu}{\delta} \tag{3-5}$$

$$\hat{y}^{(k)} = \gamma \hat{x}^{(k)} + \beta \tag{3-6}$$

U jednadžbama (3-5) i (3-6), $\hat{x}^{(k)}$ označava normaliziranu aktivaciju, $\hat{y}^{(k)}$označava rezultirajuću transformiranu aktivaciju, a γ i β su parametri koji se uče i označavaju razmjerno skaliranje i translaciju.

- **Dropout sloj** - *dropout* je tehnika regularizacije koja se koristi u konvolucijskim neuronskim mrežama (i drugim vrstama dubokih neuronskih mreža) za smanjenje prenaučenosti [13]. *Dropout* funkcionira tako što nasumično "isključuje" (postavlja na nulu) dio aktivacija u sloju tijekom faze učenja, što sprječava mrežu da se previše osloni na bilo koji pojedinačni neuron i potiče mrežu da nauči robusnije značajke. Operacija *dropout* za aktivaciju *x* u sloju može se matematički izraziti korištenjem jednadžbe (3-7) [5].

$$x' = x \odot r \tag{3-7}$$

U jednadžbi (3-7), simbol $\odot$ predstavlja Hadamardovo množenje, odnosno operaciju koja svaki element vektora *x* množi s odgovarajućim elementom vektora *r*. Rezultat ove operacije je vektor aktivacija nakon primjene *dropout* tehnike odnosno *x'*. Vektor *r* je generiran iz Bernoullijeve distribucije te ima istu dimenziju kao i *x*, gdje svaki element ima vjerojatnost *p* da bude 1, što znači da se odgovarajući neuron zadržava u modelu, dok ima vjerojatnost $(1-p1-p)$ da bude 0, što znači da se odgovarajući neuron privremeno isključuje iz računanja u toj iteraciji treninga.

3.1.1. Klasifikacija korištenjem neuronskih mreža

U središtu primjene konvolucijskih neuronskih mreža za klasifikaciju slika je sposobnost mreža da uče prepoznavati i kategorizirati slike u predefinirane klase. Jedna od prvih arhitektura u ovom području je LeNet-5, razvijena od strane Yann LeCun-a [14]. LeNet-5 služi kao temelj za mnoge napredne arhitekture poput AlexNet [15], VGG [16], ResNet [17] i Inception [18] mreže, koje su unaprijedile polje dubokog učenja kroz inovativne dizajnerske pristupe, poboljšavajući performanse i efikasnost obrade podataka.

LeNet-5 arhitektura

LeNet-5, koja se sastoji od sedam slojeva, optimizirana je za efikasnu klasifikaciju rukom pisanih znamenki, koristeći konvolucijske slojeve, slojeve sažimanja i potpuno povezane slojeve. Ova struktura mreže, s alterniranjem slojeva za ekstrakciju značajki i redukciju dimenzionalnosti, postavila je osnovu za razvoj složenijih CNN arhitektura.
LeNet-5 je koncipirana kako bi se obradile ulazne slike dimenzija 32x32 piksela, kroz seriju slojeva koji transformiraju podatke na način koji olakšava klasifikaciju. Arhitektura obuhvaća slojeve opisane u nastavku [14].

- **C1 - konvolucijski sloj** - koristi 6 filtera veličine 5x5 za proizvodnju mapa značajki dimenzija 28x28, svaki filter detektira specifične značajke na ulaznoj slici,
- **S2 - sloj sažimanja** - primjenjuje maksimalno sažimanje s filtrom 2x2 i korakom 2, reducirajući dimenzije mapa značajki na 14x14,
- **C3 - drugi konvolucijski sloj** - koristi 16 filtera veličine 5x5 za daljnju proizvodnju mapa značajki dimenzija 10x10 izvlačeći složenije značajke,
- **S4 - drugi sloj sažimanja** - smanjuje dimenzije mapa značajki na 5x5,
- **C5 - treći konvolucijski sloj** - funkcionira više kao potpuno povezani sloj zbog veličine filtra, sa 120 izlaza koji odgovaraju specifičnim kombinacijama značajki,
- **F6 - potpuno povezani sloj** - s 84 neurona, ovaj sloj dodatno obrađuje podatke za klasifikaciju,
- **Izlazni sloj** - koristi *softmax* funkciju za klasifikaciju slika u jednu od mogućih kategorija.

Navedena arhitektura je također vidljiva na slici 3.4.

Slika 3.4. LeNet-5 arhitektura.

LeNet-5 se pokazala jednako preciznom kao slične implementacije dostupne u literaturi za zadatke klasifikacije. Njena sposobnost postupnog izvlačenja složenijih značajki iz jednostavnih oblika omogućuje mreži efikasnu diferencijaciju među različitim kategorijama na temelju hijerarhije naučenih značajki. Brza obrada i donošenje odluka čine LeNet-5 pogodnom za implementaciju na uređajima s ograničenim resursima, uključujući FPGA.

3.1.2. Detekcija objekata putem neuronske mreže

Detekcija objekata predstavlja jedan od ključnih zadataka u području računalnog vida i dubokog učenja, gdje je cilj ne samo klasificirati objekte unutar slike, već i precizno odrediti njihovu lokaciju. U ovom kontekstu, Yolov3 Tiny se ističe kao jedna od najefikasnijih arhitektura, posebno prilagođena za brzu i resursno učinkovitu detekciju objekata. Ova arhitektura omogućava obradu u realnom vremenu s visokom točnošću, što je čini idealnom za implementaciju na FPGA platformama. FPGA platforme dodatno pojačavaju učinkovitost ove arhitekture zahvaljujući sposobnosti paralelne obrade i visoke energetske učinkovitosti.

Yolov3 Tiny arhitektura

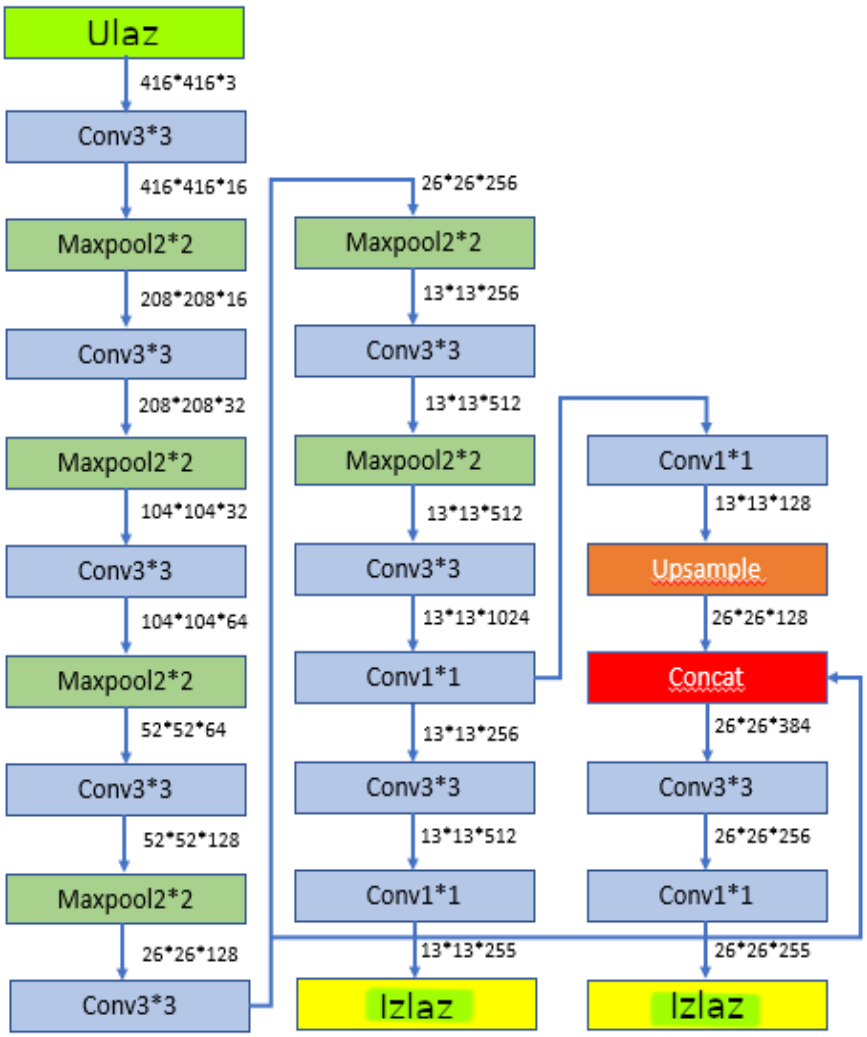
Yolov3 Tiny, kao lagana verzija kompletnog Yolov3 modela [19], optimizirana je za brze detekcije uz minimalnu potrošnju resursa. Ova pojednostavljena verzija zadržava osnovnu filozofiju Yolo pristupa; obradu cijele slike odjednom za predviđanje *bounding box-ova* i klasifikacije objekata. Time omogućava istovremeno prepoznavanje i lokalizaciju višestrukih objekata unutar jednog prolaza kroz mrežu. Zahvaljujući modularnoj strukturi, na ovakvoj arhitekturi se omogućuje modifikacija i konfiguracija slojeva i filtera kako bi se prilagodila specifičnim zahtjevima različitih aplikacija. Stoga se izdvajaju karakteristike opisane u nastavku [19].

- **Brzina i učinkovitost** - Yolov3 Tiny omogućava detekciju objekata u realnom vremenu, što je ključno za aplikacije poput video nadzora, autonomne navigacije i interaktivnih sustava.
- **Optimizacija za resurse** - smanjeni broj slojeva i filtera u odnosu na punu verziju Yolov3 omogućuje implementaciju na uređajima s ograničenim računalnim kapacitetom, uključujući FPGA platforme.
- **Fleksibilnost i prilagodljivost** - Yolov3 Tiny može se prilagoditi specifičnim zahtjevima aplikacije, optimizirajući balans između brzine, točnosti i potrošnje resursa.

Arhitektura je također sastavljena od više slojeva koji su isti kao i kod LeNet-5 mreže, gdje su ključni slojevi konvolucije, maksimalnog sažimanja te dodatni slojevi invertirane konvolucije (engl. *upsample*) i spajanja (engl. *concat*). Navedeni dodatni slojevi su opisani u nastavku [19].

- **Invertirana konvolucija** - poznata i kao *upsample* sloj, koristi se u arhitekturi Yolov3 Tiny za povećanje dimenzija značajki iz dubljih slojeva mreže. Ovaj sloj omogućuje da se podatci koji su prethodno sažeti pomoću maksimalnog sažimanja mogu vratiti na veće dimenzije, čime se olakšava detekcija objekata na različitim razinama preciznosti. Sloj *upsample* koristi tehniku interpolacije za povećanje prostornih dimenzija aktivacijskih mapa, što je ključno za precizno lokaliziranje objekata na slikama.
- **Sloj spajanja** - *concat* sloj, koristi se za kombiniranje značajki iz različitih slojeva mreže. Ovaj pristup omogućuje mreži da koristi informacije kako iz dubljih, tako i iz površinskih slojeva, pružajući bogatiji skup značajki za konačnu detekciju objekata. Spajanjem značajki iz više slojeva, Yolov3 Tiny može efikasnije identificirati i klasificirati objekte unutar slike, koristeći kombinaciju širokih i detaljnih perspektiva.
  Izgled Yolov3 Tiny arhitekture je prikazan na slici 3.5.

Na slici 3.5. vidljivo je da Yolov3 Tiny koristi dva izlazna sloja. Ovo omogućuje da se efikasno detektiraju objekti različitih veličina. Dakle, svaki izlazni sloj je odgovoran za predviđanje *bounding box*-ova i klasifikacije objekata, ali na različitim rezolucijama.



Slika 3.5. Izgled arhitekture Yolov3 Tiny modela [20].

Prvi izlazni sloj fokusira se na detekciju manjih objekata, koristeći visokorezolucijske značajke koje su sačuvane iz ranih slojeva konvolucije. Drugi izlazni sloj koristi značajke iz dubljih slojeva mreže koje su usmjerene na detekciju većih objekata [19]. Osim Yolov3 Tiny, razvijene su i druge napredne arhitekture za detekciju objekata koje poboljšavaju efikasnost obrade i preciznost identifikacije, a neke od njih su opisane u nastavku.

- **SSD (engl. *Single Shot MultiBox Detector*) -** ova arhitektura koristi jedinstveni prolaz kroz mrežu za predviđanje *bounding box*-ova i vjerojatnosti klasa, pružajući dobar balans između brzine i točnosti [21].
- **Faster R-CNN -** unapređenje na originalnom R-CNN, *Faster* R-CNN uvodi regijske prijedloge mreže (engl. *region proposal networks* - RPN) za brže i točnije predviđanje lokacija objekata [22].
- **Mask R-CNN -** proširenje na *Faster* R-CNN koje dodaje granu za predviđanje segmenata objekata, omogućavajući preciznu segmentaciju objekata uz detekciju [23].

Implementacija ovih arhitektura na FPGA platformama može dodatno poboljšati njihovu efikasnost glede obrade podataka, uz minimalnu potrošnju resursa spomenute platforme i znatno povećanje brzine obrade. Tim postupkom se iskorištava sposobnost FPGA da izvodi paralelne operacije i prilagođava se specifičnim zahtjevima detekcije objekata. FPGA platforme mogu biti programski konfigurirane kako bi optimalno podržale specifične arhitekture CNN-a. Također se i neuronske mreže mogu adekvatno optimizirati kako bi se postigla uzajamna korisnost s hardverske i softverske strane, omogućavajući brzu obradu i analizu slika u realnom vremenu, što je ključno za kritične aplikacije poput autonomnih vozila i sustava za nadzor. Kao i kod mreža namijenjenih klasifikaciji, optimizacija svih slojeva i težina ključna je i u konvolucijskim neuronskim mrežama za detekciju objekata. Ovaj proces osigurava efikasnost i preciznost u identifikaciji i lokalizaciji objekata unutar slika.

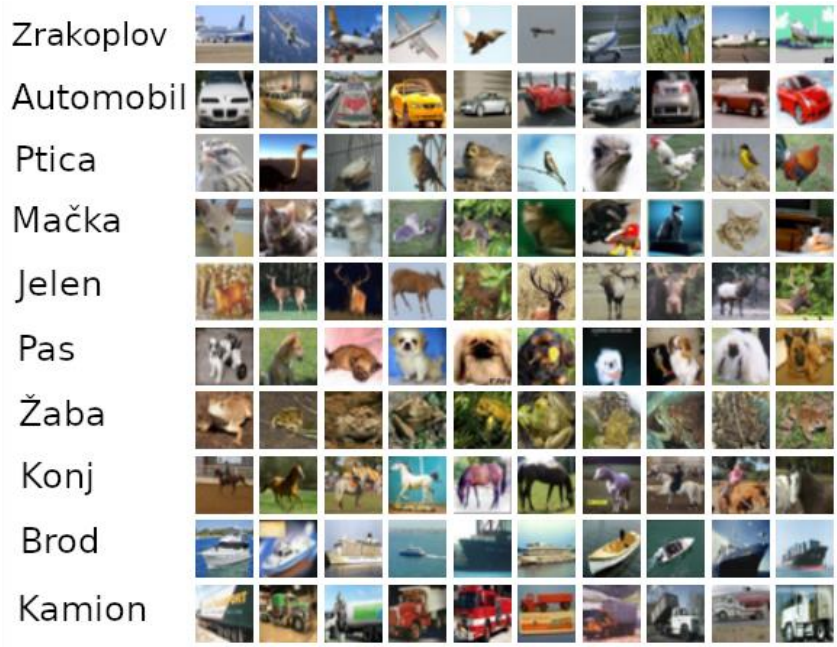
3.2. Pretvorba i prilagodba modela za FPGA

Prilagodba dubokih neuronskih mreža za FPGA ključna je za njihovu efikasnost i paralelnu obradu. Proces uključuje nekoliko faza, od inicijalne modifikacije arhitekture modela pomoću kvantizacije, preko odabira i pripreme skupa podataka, do treniranja mreže. Korištenjem alata poput Brevitas, omogućeno je fino podešavanje kvantizacijskih parametara modela, što je presudno za održavanje visoke točnosti uz smanjenu potrošnju resursa. Nakon treniranja, model se eksportira u ONNX (engl. *Open Neural Network Exchange*) format, omogućavajući korištenje FINN alata za efikasnu adaptaciju i kompilaciju modela za FPGA platforme. Ovaj pristup omogućava stvaranje optimiziranih DNN koje mogu izvoditi kompleksne zadatke obrade podataka u realnom vremenu na FPGA uređajima. Stoga prvi korak koji slijedi je obrada postojeće arhitekture u Brevitas-u.

3.2.1. Brevitas: kvantizacija modela LeNet-5 neuronske mreže

Kroz biblioteke Brevitas alata, poddijela Pytorch-a, omogućen je inovativni pristup u modifikaciji arhitekture neuronskih mreža kroz napredne mogućnosti kvantizacije. Ovo se razlikuje od tradicionalnih metoda optimizacije, omogućavajući razvoj resursno efikasnijih neuronskih mreža za specifične primjene. Kvantizacija u Brevitas-u, prilagođena za svaki sloj zasebno, ključna je za optimizaciju mreža na FPGA. Proces kreiranja i treniranja neuronske mreže u *Brevitas*-u ne samo da smanjuje potrošnju resursa u hardveru, već također omogućava enkapsulaciju modela unutar FPGA memorije. Osim toga, fokus je i na očuvanju visoke točnosti predikcije modela neuronske mreže, unatoč redukciji podatkovne preciznosti. Specifično, adaptacija i kvantizacija LeNet-5 modela predstavlja primjer kako je moguće ostvariti visokoučinkovitu obradu na ograničenim hardverskim resursima bez kompromitiranja performansi.

Neuronska mreža LeNet-5 arhitekture se može trenirati koristeći podatkovni skup CIFAR-10, koji obuhvaća deset različitih klasa objekata [24]. Primjer slika iz navedenog skupa podataka je prikazan na slici 3.6.



Slika 3.6. CIFAR-10 skup podataka [24].

Iako jednostavan, ovaj skup podataka pruža podlogu za razvoj klasifikacijske neuronske mreže. Raznolikost ovog skupa podataka omogućava testiranje i optimizaciju kvantiziranog modela, pružajući robustnu platformu za evaluaciju performansi i točnosti.

Proces kvantizacije

Kvantizacija je ključni proces u digitalnoj obradi signala i strojnom učenju, omogućavajući efikasnu digitalnu obradu i pohranu kontinuiranih ili visoko preciznih signala i podataka. U kontekstu dubokog učenja, kvantizacija se koristi za smanjenje veličine neuronske mreže i računske složenosti, pretvarajući reprezentacije težina u režim pokretnog zareza, aktivacija i gradijenata u reprezentaciju fiksnog zareza. Osnovna formula za kvantizaciju, kako je opisano u radu [25], može se izraziti jednadžbom (3-8):

$$Q(x) = \Delta \cdot \left[\frac{x}{\Delta} + \frac{1}{2}\right] \tag{3-8}$$

gdje $Q(x)$ predstavlja kvantiziranu vrijednost ulaznog signala x, Δ je širina kvantizacijskog intervala (korak kvantizacije), a $\lfloor \cdot \rfloor$ označava funkciju zaokruživanja na najbliži cijeli broj. Ova jednadžba primjenjuje princip zaokruživanja na najbližu kvantizacijsku razinu, efikasno smanjujući preciznost ulaznih vrijednosti na određeni skup diskretnih vrijednosti. Kvantizacija pridonosi stvaranju kvantizacijske greške, razlike između originalne i kvantizirane vrijednosti, što može utjecati na performanse modela, ali uz pažljivu primjenu i optimizaciju, ovaj utjecaj može biti minimiziran.

Ovakav proces unutar Brevitas-a odvija se na više razina modela, uključujući konvolucijske slojeve, aktivacijske funkcije i potpuno povezane slojeve. Koristeći Brevitas implementirane su 1, 4 i 8-bitne kvantizacije za težine i aktivacije. Ova strategija omogućila je znatno smanjenje veličine modela i poboljšanje efikasnosti uz minimalan utjecaj na točnost [26]. Svaki sloj modela pažljivo je prilagođen za optimalnu kvantizaciju gdje se broj bitova za težine i aktivacije određuje tako da se postigne balans između efikasnosti resursa i očuvanja performansi modela. U nastavku su prikazane specifične prilagodbe kvantizacije za različite vrste slojeva.

- **Konvolucijski slojevi** - adaptirani u *QuantConv2d*, ovi slojevi koriste 1-bitne kvantizirane težine, omogućujući smanjenje potrebnog prostora za pohranu i ubrzavanje izračuna.
- **Aktivacijski slojevi** - *QuantReLU* implementira kvantizaciju aktivacija, dodatno smanjujući potrebu za računskim resursima bez gubitka informacija o aktivaciji.
- **Potpuno povezani slojevi** - *QuantLinear* slojevi koriste kvantizirane težine za efikasno izvođenje linearnih transformacija uključenih u klasifikaciju.

Kroz proces kvantizacije, arhitektura neuronske mreže može doživjeti promjene, posebno u načinu kako se memorija koristi za pohranu težinskih vrijednosti s nižom preciznošću. No, sloj za normalizaciju, poznat kao *BatchNormalization2d*, generalno ostaje nepromijenjen tijekom

kvantizacije. To je zato što ovaj sloj ne uključuje aktivno treniranje težina kroz iteracije kao što to rade konvolucijski ili potpuno povezani slojevi. Umjesto toga, *BatchNormalization2d* sloj normalizira izlaze prethodnih slojeva koristeći fiksne parametre za skaliranje i pomak koji se izračunavaju tijekom faze treniranja mreže. Stoga iako kvantizacija utječe na to kako su težine pohranjene i obrađene, slojevi poput *BatchNormalization2d* ne moraju se mijenjati jer njihova funkcionalnost nije izravno ovisna o preciznosti težina. Izgled arhitekture napisan u Brevitas-u je prikazan na slici 3.7.

```python
input_channels = 3  # RGB images
input_size = 32     # 32x32 images
kernel_size = 5     # kernel size for LeNet
weight_bit_width = 2
act_bit_width = 2
hidden1 = 64
hidden2 = 64
hidden3 = 64
num_classes = 10

class QuantLeNet(nn.Module):
    def __init__(self):
        super(QuantLeNet, self).__init__()
        # Convolutional layers
        self.conv1 = QuantConv2d(input_channels, hidden1, kernel_size, bias=True, weight_bit_width=weight_bit_width)
        self.relu1 = QuantReLU(bit_width=act_bit_width)
        self.conv2 = QuantConv2d(hidden1, hidden2, kernel_size, bias=True, weight_bit_width=weight_bit_width)
        self.relu2 = QuantReLU(bit_width=act_bit_width)

        # Fully connected layers
        self.fc1 = QuantLinear(hidden2 * 5 * 5, hidden3, bias=True, weight_bit_width=weight_bit_width)
        self.relu3 = QuantReLU(bit_width=act_bit_width)
        self.fc2 = QuantLinear(hidden3, num_classes, bias=True, weight_bit_width=weight_bit_width)

        # Batch normalization and dropout
        self.bn1 = nn.BatchNorm2d(hidden1)  # Change to BatchNorm2d
        self.bn2 = nn.BatchNorm2d(hidden2)  # Change to BatchNorm2d
        self.bn3 = nn.BatchNorm1d(hidden3)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.relu1(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = self.bn1(x)
        x = self.dropout(x)

        x = self.relu2(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = self.bn2(x)
        x = self.dropout(x)

        x = x.view(-1, hidden2 * 5 * 5)
        x = self.relu3(self.fc1(x))
        x = self.bn3(x)
        x = self.dropout(x)

        x = self.fc2(x)
        return x
```

Slika 3.7. Arhitektura *QuantLenet-5* neuronske mreže u Brevitas-u.

Nakon kvantizacije arhitekture neuronske mreže i određivanja skupa podataka, sljedeći korak je treniranje prilagođene neuronske mreže *QuantLenet-5* putem Brevitas-a. Proces treniranja sastoji se od optimizacije težina mreže na temelju ulaznih podataka i njihovih oznaka. Za ovu svrhu, odabran je *Adam* optimizator zbog njegove efikasnosti i sposobnosti automatskog prilagođavanja brzine učenja. Kod za pokretanje treninga može se vidjeti na slici 3.8.

```python
def train(model, train_loader, optimizer, criterion):
    losses = []
    # ensure model is in training mode
    model.train()

    for i, data in enumerate(train_loader, 0):
        inputs, target = data
        inputs, target = inputs.to(device), target.to(device)

        # convert target to one-hot encoding
        target = torch.nn.functional.one_hot(target, num_classes=10)
        target = target.float()  # convert targets to float tensor

        optimizer.zero_grad()

        # forward pass
        output = model(inputs.float())
        loss = criterion(output, target)  # remove target.unsqueeze(1)

        # backward pass + run optimizer to update weights
        loss.backward()
        optimizer.step()

        # keep track of loss value
        losses.append(loss.data.cpu().numpy())

    return losses
```

Slika 3.8. Funkcija za treniranje *QuantLeNet-5* neuronske mreže.

Proces treniranja prolazi kroz faze pripreme podataka, računanja gubitka i ažuriranja težina modela. Analogno uz funkciju treniranja implementirane su funkcije testiranja i evaluacije neuronske mreže koje su u fokusu u narednim poglavljima. U posljednjem koraku istreniranu neuronsku mrežu potrebno je izvesti u ONNX format.

Izvoz u ONNX

Kroz fazu izvoza neuronske mreže u ONNX format, mreža se priprema za korištenje u alatu FINN. Ovaj proces omogućava smanjivanje razlike između visoke razine razvoja modela u Brevitas okruženju i specifičnih hardverskih zahtjeva FPGA uređaja. Naime izvoz modela u ONNX ključan je zbog nekoliko razloga. Ovakav format omogućuje standardiziranu upotrebu modela u različitim alatima, uključujući FINN. Drugo izvoz u ONNX format omogućava detaljno definiranje i čuvanje kvantizacijskih parametara modela, što je neophodno za očuvanje točnosti modela tijekom njegove implementacije na FPGA.

Uz očuvanje svih ključnih informacija o arhitekturi neuronske mreže i kvantizacijskim parametrima korak pretvaranja Pytorch (Brevitas) u ONNX [27] uključuje korake opisane u nastavku.

- **Definiranje ulaznog formata** - određivanje oblika ulaznih podataka modela, koji treba biti usklađen s formatom podataka koji će se koristiti prilikom implementacije na FPGA. U ovom slučaju, ulazni oblik je definiran kao (1, 3, 32, 32), što odgovara ulaznim slikama CIFAR-10 skupa podataka s tri kanala boja i dimenzijama 32x32 piksela.
- **Priprema ulaznih podataka** - generiranje reprezentativnog skupa ulaznih podataka je korak u kojem se stvara set ulaznih podataka koji odražava stvarne uvjete u kojima će se model koristiti. To je bitno jer modeli dubokog učenja mogu različito reagirati na promjene u ulaznim podacima zbog svoje visoke osjetljivosti na distribuciju ulaza.
- **Pokretanje eksporta** - kroz funkciju *export_finn_onnx*, model se eksportira u *QuantLeNet-ready.onnx* format, spremajući model za daljnje korake optimizacije i sinteze putem FINN-a.

Navedeni koraci su također vidljivi na slici 3.9. koja prikazuje programski kod za pretvorbu mreže u ONNX format.

```
import brevitas.onnx as bo
from brevitas.quant_tensor import QuantTensor


ready_model_filename = "QuantLeNet-ready.onnx"

input_shape = (1, 3, 32, 32)

input_a = np.random.randint(0,2, size = input_shape).astype(np.float32)
input_a = 2 * input_a - 1
scale = 1.0
input_t = torch.from_numpy(input_a * scale)
input_qt = QuantTensor(
    input_t, scale=torch.tensor(scale), bit_width=torch.tensor(1.0), signed=True
)

model.cpu()

bo.export_finn_onnx(
    model, export_path=ready_model_filename, input_t=input_qt
)
print("Model saved to %s" % ready_model_filename)
```

Slika 3.9. Kod za pretvorbu *QuantLeNet-5* mreže u ONNX format.

Ovaj proces ne samo da olakšava integraciju modela na FPGA, već i osigurava transparentnost i ponovljivost u procesu razvoja, omogućavajući ponovnu preciznu prilagodbu i optimizaciju parametara neuronskih mreža.

Kvantizacija i prilagodba Yolov3 Tiny modela neuronske mreže

Pristup kvantizaciji i prilagodbi neuronske mreže Yolov3 Tiny za implementaciju na FPGA platformama analogan je onome koji je korišten za LeNet-5 arhitekturu ilustrirajući fleksibilnost procesa optimizacije dubokih neuronskih mreža pomoću alata Brevitas i ONNX formata. Kao što je opisano za LeNet-5 i Yolov3 Tiny model je podvrgnut sličnom procesu kvantizacije [28]. Ključni parametri, kao što su širina bita za težine i aktivacije, prilagođeni su kroz iterativna testiranja kako bi se postigao optimalan balans između točnosti i resursnih zahtjeva FPGA platforme. Za potrebe kvantizacije, uređivana je YAML datoteka unutar koje je sadržan opis arhitekture neuronske mreže. Izgled takve arhitekture je prikazan na slici 3.10.

```yaml
# Parameters
nc: 1  # number of classes
use_hardtanh: False
depth_multiple: 1.0  # model depth multiple
width_multiple: 0.2  # layer channel multiple
anchors:
  - [94, 113, 49, 46, 169, 186]  # P4/16  # P4/16


bit_width:
 # - [8, 4, 4, 4, 4]  # in_weight_bit_width, in_act_bit_width, weight_bit_width, act_bit_width, out_weight_bit_width
# - [8, 4, 2, 8]  # in_weight_bit_width,  weight_bit_width, act_bit_width, out_weight_bit_width
- [4, 2, 4, 8]  #

# YOLOv3-tiny backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, QuantConv, [16, 3, 1, None, 1, True, True, in_weight_bit_width, act_bit_width]],  # 0
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],  # 1-P1/2
   [-1, 1, QuantConv, [32, 3, 1, None, 1, True, True, weight_bit_width, act_bit_width]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],  # 3-P2/4
   [-1, 1, QuantConv, [64, 3, 1, None, 1, True, True, weight_bit_width, act_bit_width]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],  # 5-P3/8
   [-1, 1, QuantConv, [128, 3, 1, None, 1, True, True, weight_bit_width, act_bit_width]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],  # 7-P4/16
   [-1, 1, QuantConv, [256, 3, 1, None, 1, True, True, weight_bit_width, act_bit_width]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],  # 9-P5/32
   [-1, 1, QuantConv, [512, 3, 1, None, 1, True, True, weight_bit_width, act_bit_width]],  # 10
  ]

# YOLOv3-tiny head
head:
  [[-1, 1, QuantConv, [1024, 3, 1, None, 1, True, True, weight_bit_width, act_bit_width]],
   [-1, 1, QuantConv, [256, 1, 1, None, 1, True, True, weight_bit_width, act_bit_width]],
   [-1, 1, QuantConv, [512, 3, 1, None, 1, True, True, weight_bit_width, act_bit_width]],  # 13 (P5/32-large)
   [-1, 1, QuantSimpleConv, [(nc+5)*3, 3, 1, None, 1, out_weight_bit_width, act_bit_width, use_hardtanh]],

   [[14], 1, Detect, [nc, anchors, False, use_hardtanh]],  # Detect(P4, P5) num_classes, anchors, in_conv, use_hardtanh
  ]
```

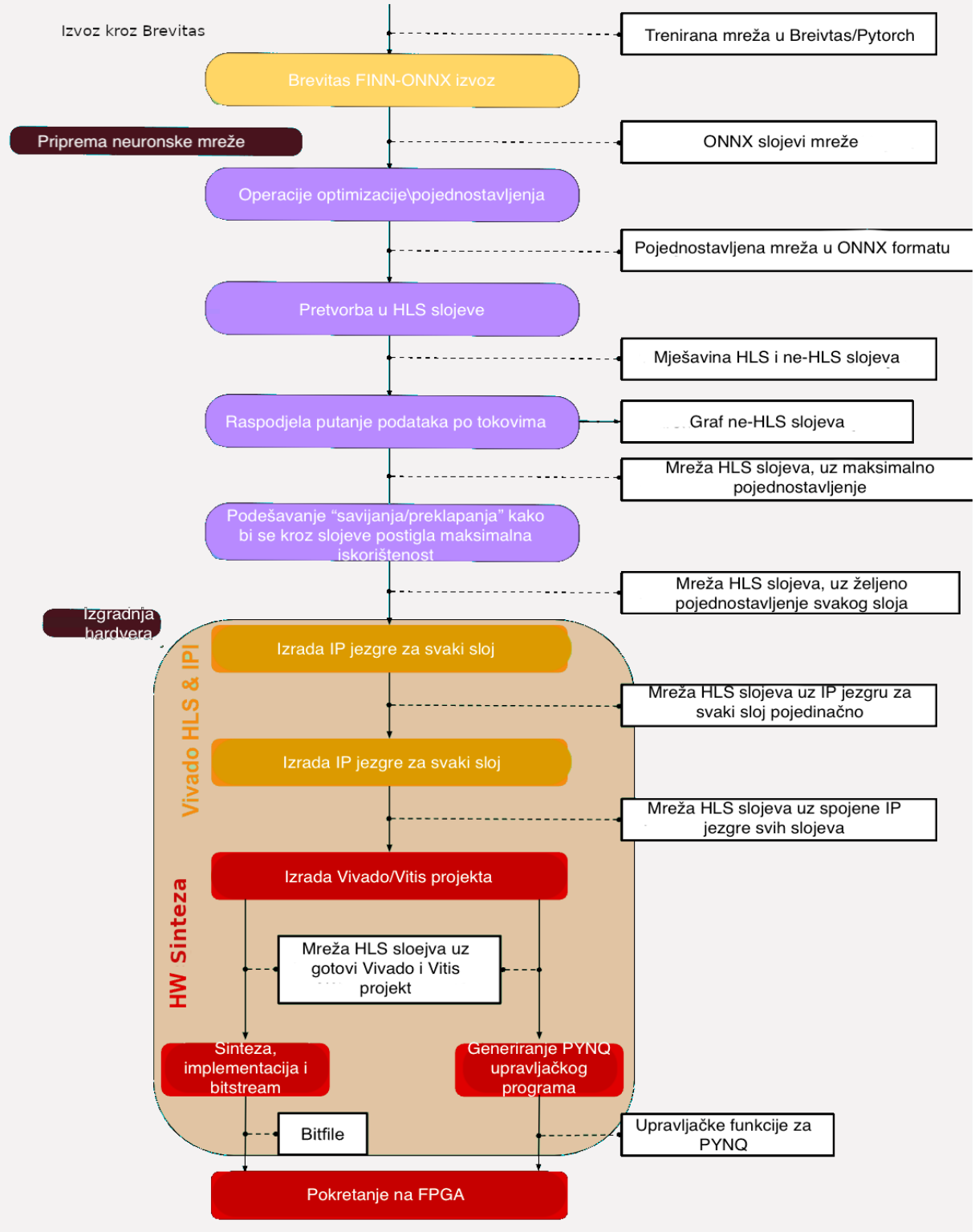
Slika 3.10. Arhitektura Yolov3 Tiny modela neuronske mreže.

Ključni parametar je širina bita (engl. *bit width*) kojim se određuje stupanj kvantizacije. Model je treniran kroz standardni programski okvir *Darknet* [29] te je napravljen izvoz u ONNX format koristeći analogiju s klasifikacijskim primjerom.

3.2.2. FINN: Adaptacija neuronske mreže za FPGA

Adaptacija neuronske mreže za FPGA uz pomoć FINN kompajlera je proces koji uključuje više koraka, od inicijalne pripreme i optimizacije modela do konačne implementacije na željenom FPGA uređaju. Ovaj proces omogućava transformaciju modela dubokog učenja u format koji je optimalan za iskorištavanje paralelnih računskih sposobnosti i efikasnu upotrebu FPGA resursa, što rezultira brzim i energetski efikasnijim izvođenjem. Kroz FINN kompajler je napravljen sveobuhvatan postupak za pretvorbu istrenirane konvolucijske neuronske mreže na FPGA. Kompajler je razvijen unutar kontejnera *Docker* te su unutar kontejnera obuhvaćeni ključni alati za sintezu i implementaciju svih potrebnih elemenata za mrežu. Za automatsku kompilaciju koda kroz FINN koriste se sljedeći alati: Vivado, Vitis i Vitis HLS. U procesu se koriste različite hardverske komponente. Među njima su tablice za pretragu (engl. *Look-Up Table - LUT*) i memorija sa slučajnim pristupom bazirana na *LUT* (engl. *LUT-based Random Access Memory - LUTRAM*). Također se koriste bistabili (engl. *Flip-Flop - FF*), blokovi memorije sa slučajnim pristupom (engl. *Block Random Access Memor - BRAM*) i procesori digitalnih signala (engl. *Digital Signal Processor - DSP*). Dodatno, uključeni su ulazno/izlazne jedinice (engl. *Input/Output - IO*) te ulazni spremnici (engl. *Buffer Gate - BUFG*). Koraci koje je kroz cijeli postupak potrebno napraviti su prikazani na slici 3.11. te su opisani u ovom poglavlju.

Na slici 3.11. se može vidjeti da postoji niz koraka koji se mogu izvesti kako bi se postigla željena optimizacija. Drugim riječima postoje dva dijela, odnosno dio unutar kojeg se priprema neuronska mreža za obradu kroz Python i dio koji se odvija direktno kroz Vivado i Vitis HLS. Drugi dio je hardverski jer se preko naredbi FINN-a izvršava mapiranje i sinteza komponenata, tj. kreiranje IP (engl. *Intellectual Property*) jezgri. Stoga na osnovu navedenog postupka na slici 3.11. kreiran je tok rada gdje prvi korak uključuje pripremu modela i optimizaciju. Transformacije i sve funkcije od prvog softverskog (pojednostavljenje neuronskih mreža) i drugog hardverskog dijela (HLS) su implementirane u pozadini FINN kompajlera.



Slika 3.11. Shematski prikaz puta kojim model neuronske mreže prolazi kroz FINN.

Priprema modela i optimizacija

Model se inicijalno učitava u FINN kompajler, gdje se priprema za niz transformacija (Slika 3.12.) [30]. Ovaj korak osigurava da je model spreman za daljnju obradu i optimizaciju za FPGA uređaje. Transformacije koje se provode u inicijalnim koracima su navedene u nastavku.

- **Transformacija *InferShapes*** - ključna za osiguranje da sve dimenzije tenzora u modelu budu precizno definirane. To omogućava da se u kasnijim koracima obrade i optimizacije koriste točne informacije o strukturi podataka modela.
- **Operacija *FoldConstants*** - reducira broj operacija u modelu eliminacijom izračuna koji se mogu unaprijed odrediti. Ovo smanjenje broja operacija direktno pridonosi efikasnijem izvođenju modela na FPGA uređaju.
- **Imenovanje tenzora i čvorova** - pomoću *GiveReadableTensorNames* i *GiveUniqueNodeNames* transformacija se olakšava razumijevanje strukture modela. Smislena imena tenzora i čvorova ključna su za efikasno debugiranje i optimizaciju modela.
- **Transformacija *RemoveStaticGraphInputs*** - uklanja sve nepotrebne statičke ulaze u modelu, pojednostavljujući njegovu strukturu i smanjujući potrebne resurse za izvođenje.

Ove inicijalne transformacije postavljaju temelje za daljnje optimizacije modela za FPGA, uključujući prilagodbu slojeva i pakiranje modela za izvođenje.

```
from finn.util.basic import make_build_dir
from finn.util.visualization import showInNetron
import os

#build_dir = os.environ["FINN_BUILD_DIR"]


import onnx
from finn.util.test import get_test_model_trained
import brevitas.onnx as bo
from qonnx.core.modelwrapper import ModelWrapper
from qonnx.transformation.infer_shapes import InferShapes
from qonnx.transformation.fold_constants import FoldConstants
from qonnx.transformation.general import GiveReadableTensorNames, GiveUniqueNodeNames, RemoveStaticGraphInputs


model = ModelWrapper("QuantLeNet-ready.onnx")
model = model.transform(InferShapes())
model = model.transform(FoldConstants())
model = model.transform(GiveUniqueNodeNames())
model = model.transform(GiveReadableTensorNames())
model = model.transform(RemoveStaticGraphInputs())
model.save("end2end_CNV_w1a1_tidy_custom.onnx")
```

Slika 3.12. Primjer koda za učitavanje i pojednostavljenje modela.

Nakon optimizacije, model prolazi kroz niz transformacija za postizanje veće efikasnosti. Ove transformacije uključuju integraciju pretprocesiranja, kvantizaciju ulaznih podataka i postprocesiranje [30]. Ključni koraci ovog procesa se mogu razložiti na način prikazan slikom 3.13 te su opisani u nastavku.

- **Integracija preprocesiranja** - koristeći *ToTensor* transformaciju iz PyTorch biblioteke, model se priprema za obradu *UINT8* ulaznih podataka tako što ih normalizira dijeljenjem s 255. Ova transformacija omogućava modelu da efikasno procesira ulazne podatke u formatu koji je tipičan za slike.
- **Spoj preprocesirajućeg i glavnog modela** - korištenjem *MergeONNXModels* transformacije, preprocesirajući model se spaja s glavnim modelom. Ovo omogućava da cijeli proces, od pretprocesiranja do predikcije, bude integriran u jedinstveni model spreman za sintezu.
- **Kvantizacija ulaznih podataka** - postavljanjem tipa podataka ulaza na *UINT8* pomoću *set_tensor_datatype* metode, model se dodatno optimizira za rad s FPGA uređajima koji su efikasniji u obradi fiksnih tipova podataka.
- **Umetanje TopK čvora za postprocesiranje** - transformacija *InsertTopK* koristi se za dodavanje čvora na kraj modela koji omogućava efikasno određivanje *k* najvećih vrijednosti u izlazima modela, što je često korisno za klasifikacijske zadatke.

- **Dodatno pojednostavljenje modela** - identične transformacije onima iz inicijalnog koraka (Slika 3.12.) se pojavljuju i u ovom koraku, kao što su *InferShapes, FoldConstants, GiveUniqueNodeNames, GiveReadableTensorNames* i *RemoveStaticGraphInputs*, ponavljaju se i u ovom koraku, čime se model dodatno pojednostavljuje, optimizira i postaje čitljiviji.

```
from finn.util.pytorch import ToTensor
from qonnx.transformation.merge_onnx_models import MergeONNXModels
from qonnx.core.datatype import DataType

model = ModelWrapper("end2end_CNV_w1a1_tidy_custom.onnx")
global_inp_name = model.graph.input[0].name
ishape = model.get_tensor_shape(global_inp_name)
# preprocessing: torchvision's ToTensor divides uint8 inputs by 255
totensor_pyt = ToTensor()
chkpt_preproc_name = "end2end_CNV_w1a1_preproc_custom.onnx"
bo.export_finn_onnx(totensor_pyt, ishape, chkpt_preproc_name)

# join preprocessing and core model
pre_model = ModelWrapper(chkpt_preproc_name)
model = model.transform(MergeONNXModels(pre_model))
# add input quantization annotation: UINT8 for all BNN-PYNQ models
global_inp_name = model.graph.input[0].name
model.set_tensor_datatype(global_inp_name, DataType["UINT8"])
```

```
/home/davorin/PYNQ_WORK/finn_v2/finn/deps/qonnx/src/qonnx/transformation/infer_data_layouts.py:119: UserWarnin
  warnings.warn("Assuming 4D input is NCHW")
```

```
from qonnx.transformation.insert_topk import InsertTopK
from qonnx.transformation.infer_datatypes import InferDataTypes

# postprocessing: insert Top-1 node at the end
model = model.transform(InsertTopK(k=1))
chkpt_name = "end2end_CNV_w1a1_pre_post_custom.onnx"
# tidy-up again
model = model.transform(InferShapes())
model = model.transform(FoldConstants())
model = model.transform(GiveUniqueNodeNames())
model = model.transform(GiveReadableTensorNames())
model = model.transform(InferDataTypes())
model = model.transform(RemoveStaticGraphInputs())
model.save(chkpt_name)
```

Slika 3.13. Primjer integracije pretprocesiranja i glavnog modela.

Nakon pojednostavljivanja modela i stvaranja čitljivog prikaza njegove arhitekture, potrebno je primjeniti transformacije postojećih slojeva: konvolucije, maksimalnog sažimanja i potpuno povezanog sloja. To uključuje zamjenu postojećih slojeva s jednostavnijim matematičkim operacijama koje se mogu paralelno izvršavati. Korak u narednom dijelu koji obavlja ove transformacije jest korak racionalizacije (engl. *Streamlining*).

Racionalizacija i izmjena slojeva neuronske mreže

Prethodno objašnjene transformacije te postavljanje preciznosti težina neuronske mreže da budu *UINT8* su stvorile podlogu za izvođenje specifičnih i zahtjevnijih operacija čiji je utjecaj jasno vezan za slojeve konvolucije i maksimalnog sažimanja. Sljedeći korak uključuje dublje izmjene arhitekture, vraćajući se na osnovne operacije množenja i dijeljenja težina unutar slojeva. Transformacije koje se provode su *Streamlining*, *LowerConvsToMatMul*, *ConvertBipolarMatMulToXnorPopcount*, *AbsorbTransposeIntoMultiThreshold*, *AbsorbScalarMulAddIntoTopK* i*nferDataLayouts* i *RemoveUnusedTensors*, a objašnjene su u nastavku.

- **Streamlining** - transformacija *Streamline* automatizira proces pojednostavljenja modela tako što optimizira seriju operacija, uklanjajući nepotrebne slojeve i konvertujući operacije gdje je to moguće u efikasnije forme [31]. Matematički takvo pojednostavljenje može uključivati operacije kao što su eliminacija operacija koje ne doprinose izlazu (npr. množenje s jedinicom ili zbrajanje nule). Navedene operacije se mogu izostaviti, što uključuje i izostavljanje određenih čvorova i težina koje su "nepotrebne".
- **Pretvaranje konvolucija u množenja matrica** - korak *LowerConvsToMatMul* transformira konvolucijske slojeve u ekvivalentne operacije množenja matrica, što olakšava sintezu i optimizaciju za FPGA [31]. Odnosno, konvolucijski sloj koji transformira ulazni tensor *X* pomoću jezgre *K* može se izraziti kao množenje matrica.
- **Optimizacija za bipolarni i *XNOR* račun** - *ConvertBipolarMatMulToXnorPopcount* transformacija prilagođava množenje matrica za binarne i bipolarne ulaze, koristeći *XNOR* i brojanje jedinica za smanjenje kompleksnosti i povećanje brzine izračunavanja [31]. Transformacija se temelji na jednadžbi (3-9) [31], gdje su *X* i *K* kao binarni ulazni i kernel tenzori, a $\oplus$ predstavlja *XNOR* operaciju kroz koju se prilagođava način rada s matricom. Dodatno, funkcija *popcount* se koristi za brojanje jediničnih bitova u rezultatu XNOR operacije, što se koristi za izračun konačnog izlaza sloja.

$$Y = popcount(X \oplus \neg K) \tag{3-9}$$

- **Reorganizacija i apsorpcija operacija** - serija transformacija dostupnih u biblioteci *absorb* [31], kao što su *AbsorbTransposeIntoMultiThreshold* i *AbsorbScalarMulAddIntoTopK* integrira višestruke operacije u jednostavnije strukture [31], čime se smanjuje broj potrebnih računskih koraka, a matematički se mogu opisati kao

optimizacija koja uključuje permutacije dimenzija unutar složenijih operacija. Jedna od korištenih operacija je operacija transponiranja matrica.

- **Inferencija rasporeda podataka** - transformacija *InferDataLayouts* analizira i određuje optimalni raspored podataka unutar modela, osiguravajući da su podaci organizirani na način koji je najpogodniji za procesiranje na FPGA. Dimenzije tenzora se raspoređuju unutar modela neuronske mreže.
- **Uklanjanje nepotrebnih tenzora** - kroz *RemoveUnusedTensors*, model se čisti od svih tenzora koji nisu direktno uključeni u izračunavanje izlaza, čime se oslobađa memorija i resursi uređaja.

Navedene transformacije i pojednostavljenja su prikazani na slici 3.14.

```
from finn.transformation.streamline import Streamline
from qonnx.transformation.lower_convs_to_matmul import LowerConvsToMatMul
from qonnx.transformation.bipolar_to_xnor import ConvertBipolarMatMulToXnorPopcount
import finn.transformation.streamline.absorb as absorb
from finn.transformation.streamline.reorder import MakeMaxPoolNHWC, MoveScalarLinearPastInvariants
from qonnx.transformation.infer_data_layouts import InferDataLayouts
from qonnx.transformation.general import RemoveUnusedTensors

model = ModelWrapper("end2end_CNV_w1a1_pre_post_custom.onnx")
model = model.transform(MoveScalarLinearPastInvariants())
model = model.transform(Streamline())
model = model.transform(LowerConvsToMatMul())
model = model.transform(MakeMaxPoolNHWC())
model = model.transform(absorb.AbsorbTransposeIntoMultiThreshold())
model = model.transform(ConvertBipolarMatMulToXnorPopcount())
model = model.transform(Streamline())
# absorb final add-mul nodes into TopK
model = model.transform(absorb.AbsorbScalarMulAddIntoTopK())
model = model.transform(InferDataLayouts())
model = model.transform(RemoveUnusedTensors())
model.save("end2end_CNV_w1a1_streamlined_custom.onnx")
```

Slika 3.14. Primjer koda za specifične transformacije s ciljem pojednostavljenja modela mreže.

Svaka transformacija je implementirana s primjenom strogih matematičkih principa i algoritama kako bi se optimizirala i pojednostavila neuronska mreža. Ovim pristupom se postiže da završni proizvod, prilikom postavljanja na FPGA platformu, ne samo da ispunjava očekivane standarde brzine i uštede energije, nego i oponaša funkcije originalnog modela dubokog učenja.

Nakon detaljnjih transformacija, preostali segment je direktno transformiranje modela neuronske mreže u HLS kod, gdje je fokus na hardveru, odnosno definiranju i modifikaciji IP jezgri za svaki sloj neuronske mreže pojedinačno.

HLS i *Dataflow* particije

U pozadini FINN kompajlera IP jezgre su kreirane korištenjem HLS koda. To uključuje jezgre za konvoluciju, maksimalno sažimanje, potpuno povezane slojeve, normalizaciju podataka i druge funkcije, uz dodatak vlastitih IP jezgri za različite operacije. U koraku korištenja HLS i *Dataflow* particija, izrađuje se tok podataka unutar kojeg težine i *bias-i* putuju kroz neuronsku mrežu unutar FPGA. U tom koraku provedene su transformacije opisane u nastavku čiji je cilj optimizacija modela [30].

- **Konverzija operacija u HLS slojeve** - transformacije poput *InferBinaryMatrixVectorActivation* i *InferQuantizedMatrixVectorActivation* automatski prepoznaju i prilagođavaju slojeve modela za implementaciju u HLS, koristeći se specifičnim FPGA resursima poput *DSP* blokova za efikasne operacije množenja i akumulacije. Ovaj korak omogućava direktno mapiranje operacija visoke razine iz neuronske mreže u optimizirane HLS module. Matematički, ove transformacije pretvaraju operacije množenja matrica i vektora ($Y=X*W+b$) u njihove binarne i kvantizirane ekvivalente, koji su prirodno efikasniji za implementaciju na FPGA platformama.
- **Prilagodba za specifične HLS slojeve** - transformacije *InferLabelSelectLayer* i *InferThresholdingLayer* su posebno dizajnirane za optimizaciju slojeva klasifikacije i kvantizacije, prilagođavajući ih za implementaciju u HLS-u koja poboljšava upotrebu memorije i računsku efikasnost.
- **Optimizacija U/I sučelja i strujanja podataka** - slojevi poput *InferConvInpGen* i *InferStreamingMaxPool* prilagođavaju ulazne i izlazne operacije modela za prijenos podataka, što je ključno za efikasnu paralelnu obradu na FPGA. Ove transformacije omogućavaju modelu da efikasno obrađuje podatke u realnom vremenu iskorištavajući sposobnost FPGA uređaja za visoku propusnost i nisku latenciju.
- ***Dataflow* particioniranje** - *CreateDataflowPartition* transformacija modularizira model u nezavisne, podatkovne dijelove koji mogu biti nezavisno sintetizirani i implementirani kao IP jezgre na FPGA. Ovaj korak omogućava fleksibilnu i modularnu implementaciju kompleksnih modela dubokog učenja, olakšavajući optimizaciju i skaliranje na hardverskom nivou.
- **Uklanjanje nepotrebnih operacija** - transformacije kao što su *RemoveCNVtoFCFlatten* i *AbsorbConsecutiveTransposes* služe za uklanjanje ili pojednostavljenje operacija koje ne

doprinose finalnom izračunavanju ili koje mogu biti efikasnije prikazane u kontekstu FPGA implementacije.

Navedene i opisane transformacije su vidljive u kodu na slici 3.15. te je svaka od njih usmjerena na iskorištavanje prednosti FPGA, od paralelizma i niske latencije do efikasne upotrebe hardverskih blokova poput *DSP* i *BRAM*, *LUT*-a i bistabila.

```
import finn.transformation.fpgadataflow.convert_to_hls_layers as to_hls
from finn.transformation.fpgadataflow.create_dataflow_partition import (
    CreateDataflowPartition,
)
from finn.transformation.move_reshape import RemoveCNVtoFCFlatten
from qonnx.custom_op.registry import getCustomOp
from qonnx.transformation.infer_data_layouts import InferDataLayouts

# choose the memory mode for the MVTU units, decoupled or const
mem_mode = "decoupled"

model = ModelWrapper("end2end_CNV_w1a1_streamlined_custom.onnx")
model = model.transform(to_hls.InferBinaryMatrixVectorActivation(mem_mode))
model = model.transform(to_hls.InferQuantizedMatrixVectorActivation(mem_mode))
# TopK to LabelSelect
model = model.transform(to_hls.InferLabelSelectLayer())
# input quantization (if any) to standalone thresholding
model = model.transform(to_hls.InferThresholdingLayer())
model = model.transform(to_hls.InferConvInpGen())
model = model.transform(to_hls.InferStreamingMaxPool())
# get rid of Reshape(-1, 1) operation between hlslib nodes
model = model.transform(RemoveCNVtoFCFlatten())
# get rid of Tranpose -> Tranpose identity seq
model = model.transform(absorb.AbsorbConsecutiveTransposes())
# infer tensor data layouts
model = model.transform(InferDataLayouts())
parent_model = model.transform(CreateDataflowPartition())
parent_model.save("end2end_CNV_w1a1_dataflow_parent_custom.onnx")
sdp_node = parent_model.get_nodes_by_op_type("StreamingDataflowPartition")[0]
sdp_node = getCustomOp(sdp_node)
dataflow_model_filename = sdp_node.get_nodeattr("model")
# save the dataflow partition with a different name for easier access
dataflow_model = ModelWrapper(dataflow_model_filename)
dataflow_model.save("end2end_CNV_w1a1_dataflow_model_custom.onnx")
```

Slika 3.15. Implementacija funkcija za rad s HLS kodom i IP jezgrama.

U okviru FINN-a, za svaku unaprijed definiranu IP jezgru, model se konfigurira korištenjem tehnike "presavijanja" (engl. *folding*). Pri tome se za svaki sloj određuju ključni parametri koji optimiziraju rad i efikasnost modela na FPGA platformi. Ovi parametri omogućuju preciznu kontrolu nad resursima i performansama. Navedeni parametri objašnjeni su u nastavku [30].

- **PE (engl. *processing elements*)** - broj jedinica za obradu unutar svakog sloja. Viši broj PE omogućava veću paralelizaciju i brže izvođenje operacija, no također povećava potrošnju hardverskih resursa.
- **SIMD (engl. *single instruction, multiple data*)** - broj operacija koje se mogu izvesti paralelno unutar jednog processing elementa. SIMD optimizacija omogućava istodobnu

obradu više podataka koristeći istu instrukciju, čime se povećava efikasnost izvođenja pri ograničenim resursima.

- **Dubina FIFO (engl. *first in, first out*)** - predstavlja veličinu međuspremnika za privremeno pohranjivanje podataka tijekom obrade. Memorije s većim kapacitetom FIFO-a omogućuju nesmetanu i kontinuiranu obradu podataka, no zauzimaju više hardverskog prostora.

Takva parametrizacija je definirana i prikazana u kodu na slici 3.16. gdje se vide uređeni parovi za svaki broj operacija i procesnih elemenata uz broj ulazno izlaznih tokova.

```
model = ModelWrapper("end2end_CNV_w1a1_dataflow_model_custom.onnx")
fc_layers = model.get_nodes_by_op_type("MatrixVectorActivation")
# each tuple is (PE, SIMD, in_fifo_depth) for a layer
folding = [
    (16, 3, [128]),
    (32, 32, [128]),
    (16, 32, [128]),
    (16, 32, [128]),
    (4, 32, [81]),
    (1, 32, [2]),
    (1, 4, [2]),
    (1, 8, [128]),
    (5, 1, [3]),
]
for fcl, (pe, simd, ififodepth) in zip(fc_layers, folding):
    fcl_inst = getCustomOp(fcl)
    fcl_inst.set_nodeattr("PE", pe)
    fcl_inst.set_nodeattr("SIMD", simd)
    fcl_inst.set_nodeattr("inFIFODepths", ififodepth)

# use same SIMD values for the sliding window operators
swg_layers = model.get_nodes_by_op_type("ConvolutionInputGenerator")
for i in range(len(swg_layers)):
    swg_inst = getCustomOp(swg_layers[i])
    simd = folding[i][1]
    swg_inst.set_nodeattr("SIMD", simd)

model = model.transform(GiveUniqueNodeNames())
model.save("end2end_CNV_w1a1_folded_custom.onnx")
```

Slika 3.16. Konfiguracija parametara i komponenata po sloju neuronske mreže.

Prethodna slika detaljno prikazuje optimizacije presavijanja i paralelizacije koje su primijenjene [32]. Kako bi se postigla takva optimizacija na FPGA, potrebno je prilagoditi određene parametre koji utječu na paralelizaciju i protok podataka kroz IP jezgre. Koraci koje je potrebno napraviti, a vidljivi su na slici 3.16 opisani su u nastavku.

- ***Folding* konfiguracija** - specificira kako će se gore navedene optimizacije primijeniti na svaki sloj modela. Ova konfiguracija je ključna za prilagodbu modela specifičnostima FPGA uređaja, balansirajući između brzine izvođenja i potrošnje resursa.

- **Konfiguracija slojeva i operacija** - svaki sloj modela, gdje je primaran fokus na slojeve za operacije množenja matrica i vektora (*MatrixVectorActivation*), konfigurira se s ciljem optimizacije. To uključuje postavljanje broja procesnih elemenata PE, SIMD širine i dubine FIFO memorije temeljeno na prethodno definiranim parametrima *folding-a*.
- **Prilagodba operacija generiranja konvolucija** - dodatno, optimizacija se primjenjuje i na slojeve odgovorne za generiranje ulaznih podataka za konvolucijske slojeve (*ConvolutionInputGenerator*), prilagođavajući SIMD širine za usklađenost s prethodno postavljenim parametrima optimizacije.

Definiranje i izgradnja projekta u FINN

Nakon detaljnih transformacija, konačni korak uključuje definiranje i konfiguraciju parametara za izgradnju projekta u Vivado alatu. U projektu su kroz ovaj korak generirane i ključne IP jezgre koje čine matematičku sastavnicu neuronske mreže. Na sljedećoj slici (Slika 3.17.) je prikazan kod unutar kojeg se specificiraju svi potrebni parametri za izgradnju mreže na FPGA.

```
test_pynq_board = "Pynq-Z2"
target_clk_ns = 10

from finn.transformation.fpgadataflow.make_zynq_proj import ZynqBuild
model = ModelWrapper("end2end_CNV_w1a1_folded_custom.onnx")
model = model.transform(ZynqBuild(platform = test_pynq_board, period_ns = target_clk_ns))
```

Slika 3.17. Kod za definiranje parametara pri izgradnji projekta za neuronsku mrežu.

U kodu prikazanom na prethodnoj slici (Slika 3.17.) su definirani parametri koji su opisani u nastavku.

- ***test_pynq_board*** - specificira željeni FPGA.
- ***target_clk_ns*** - određuje period takta u nanosekundama, što utječe na brzinu izvođenja modela.
- ***ZynqBuild* transformacije** - model se prilagođava za implementaciju na PYNQ-Z2 platformi, uzimajući u obzir njene hardverske kapacitete i ograničenja.

Finalni projekt je izgrađen kroz postupak prikazan na prethodnoj slici (Slika 3.17.) i kao takav je dostupan za pregled unutar Vivado alata. Budući da je sličnost između projekata neuronskih mreža LeNet-5 i Yolov3 Tiny iznimno velika u pogledu vrste slojeva, izgrađeni projekt prikazan je samo u primjeru s Yolov3 Tiny modelom koji se nalazi narednom potpoglavlju (Slika 3.19 i 3.20).

Adaptacija Yolov3 Tiny modela neuronske mreže u FINN alatu

Kao što je objašnjeno u poglavlju 3.2.1. o kvantizaciji neuronske mreže u Brevitas-u, ovdje je prikazana slična usporedba gdje su isti koraci transformacije i pojednostavljenja modela primijenjeni za FPGA. Međutim, raspored IP jezgri i konačni projekt se razlikuju zbog različitih arhitektura, što utječe na korake optimizacije resursa po svakom sloju mreže za komponente (*DSP*, *BRAM*, *LUT*, *PE*). Ove transformacije se provode kroz već spomenuti proces *folding-a* neuronske mreže iz poglavlja 3.2.2. U ovom dijelu konfiguracija se učitava kroz *pynq-z2.json* datoteku [10]. Detaljan postupak izgradnje projekta prikazan je na slici 3.18., uz definiranje odgovarajućih izlaza.

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
#import os
#import shutil

#build_dir = "/workspace/finn/notebooks/best_test/brevitas_model"

model_file = build_dir + "/yolo_pre_post_tidy.onnx"

final_output_dir = build_dir + "/output"

cfg = build.DataflowBuildConfig(
    output_dir          = final_output_dir,
    mvau_wwidth_max     = 80,
    folding_config_file = build_dir + "/pynq_z2.json",
    auto_fifo_depths    = False,
    #large_fifo_mem_style = 'auto',
    target_fps          = 100000,
    synth_clk_period_ns = 10.0,
    board               = "Pynq-Z2",
    #fpga_part           = "xc7z020clg400-1",
    shell_flow_type     = build_cfg.ShellFlowType.VIVADO_ZYNQ,
    generate_outputs=[
        build_cfg.DataflowOutputType.ESTIMATE_REPORTS,
        build_cfg.DataflowOutputType.BITFILE,
        build_cfg.DataflowOutputType.PYNQ_DRIVER,
        build_cfg.DataflowOutputType.DEPLOYMENT_PACKAGE,
    ]
)

build.build_dataflow_cfg(model_file, cfg)
```

Slika 3.18. Kod za generiranje projekta u Vivado uz definiranje izlaza.

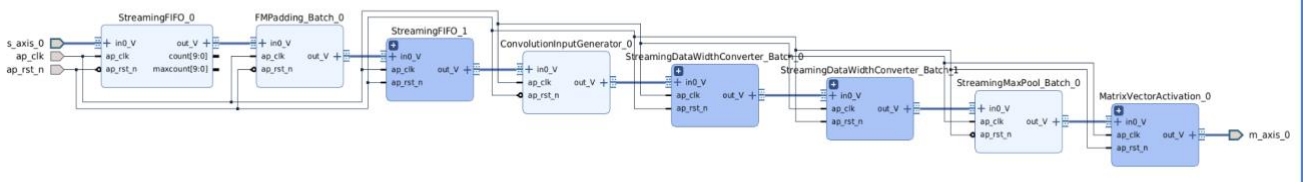
U programskom kodu prikazanom na slici 3.18. definiran je niz parametara, a neki od njih su objašnjeni u nastavku.

- **mvau_wwidth_max** - ovaj parametar određuje maksimalnu širinu bita za jedinice koje izvršavaju množenje i akumulaciju (*MVAU*) unutar FPGA. Veća širina bita može povećati preciznost izračuna, ali i potrošnju resursa. Prilagodbom ovog parametra balansira se između točnosti i efikasnosti resursa.

- **folding_config_file** - datoteka konfiguracije koja sadrži detaljne informacije o tome kako optimizirati slojeve modela (poput broja PE i SIMD jedinica po sloju).
- **target_fps** - ciljana brzina obrade u slikama po sekundi (engl. *frames per second*). Ovaj parametar je ključan za aplikacije koje zahtijevaju obradu u realnom vremenu, omogućavajući razvojnim inženjerima da postave specifične performanse koje model mora ispuniti.
- **synth_clk_period_ns** - period takta sinteze izražen u nanosekundama. Ovo je temeljni parametar koji određuje brzinu na kojoj FPGA izvršava operacije, utječući na ukupnu brzinu izvođenja modela.
- **Konfiguracija izgradnje i generiranje izlaza** - nakon postavljanja gore navedenih parametara, proces gradnje kreće u finalne korake koji uključuju sintezu modela u hardverski opis, generiranje bit datoteke potrebne za programiranje FPGA, kao i stvaranje PYNQ drivera i paketa za *deployment* koji omogućuju lako pokretanje i testiranje modela na ciljnoj FPGA pločici.

Raspored IP jezgri u Vivado-u

Nakon detaljnog upoznavanja s konfiguracijom parametara i izgradnje finalnog projekta, sljedeći korak je izgradnja pojedinačnih IP jezgri koje su međusobno povezane. Ovakva terminologija se još koristi pod imenom *stitched_ip* [30]. To je jedna od izlaznih jedinica *DataflowBuildConfig* konfiguracije. Dio izgradnje s pokaznim IP jezgrama je prikazan na slici 3.19.



Slika 3.19. Pregled izgrađenih i spojenih IP jezgri za pojedini sloj.

Na slici 3.19. se može vidjeti blok dizajn unutar Vivado alata, gdje su IP jezgre spojene u smisleni tok podataka koji odgovara operacijama unutar neuronske mreže. Ove jezgre implementiraju specifične funkcije kao što su *FIFO* redovi, obrada signala kroz konvolucijske slojeve, transformacije širine podatkovnih puteva, aktivacijske funkcije i operacije sažimanja.

Svaki blok u ovoj shemi predstavlja specifičnu IP jezgru koja se koristi u stvaranju konačnog FPGA dizajna, a detalji o pojedinoj jezgri navedeni su u nastavku.

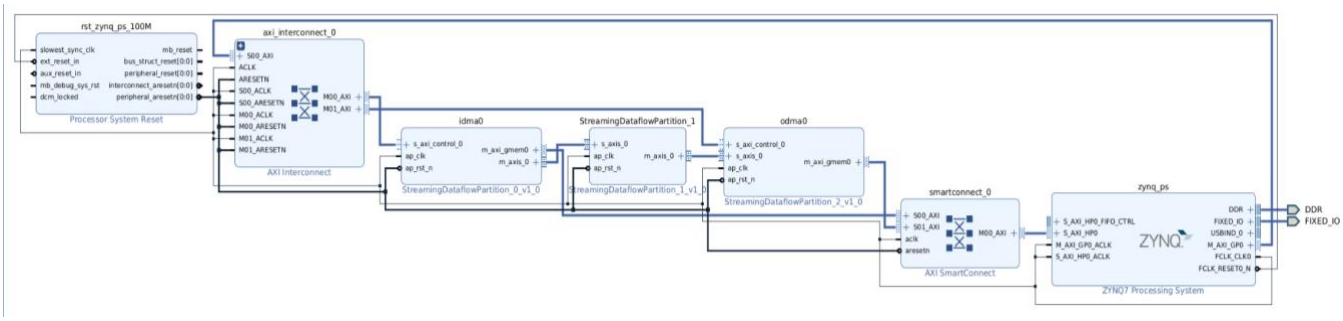
- ***StreamingFIFO_0*** - *FIFO* red koji se koristi za usklađivanje pristupa podacima, poboljšavajući efikasnost prijenosa podataka između različitih dijelova sustava.
- ***FMPadding_Batch_0*** - ovaj blok implementira dodavanje rubnih piksela (engl. *padding*) na ulazne podatke, što je često potrebno za konvolucijske slojeve kako bi se održala dimenzionalnost izlaza.
- ***ConvolutionInputGenerator_0*** - generira prozore podataka koji su potrebni za izračun konvolucije, pripremajući podatke za obradu kroz konvolucijske slojeve.
- ***StreamingDataWidthConverter_Batch_0*** - konverter širine podatkovnog puta omogućava prilagodbu širine bitova između različitih modula u lancu obrade, što je važno za kompatibilnost i optimizaciju performansi.
- ***MatrixVectorActivation_0*** - modul koji implementira operacije aktivacijske funkcije, često koristeći matricu za izračunavanje izlaznih aktivacija iz konvolucijskih slojeva.
- ***StreamingMaxPool_Batch_0*** - IP jezgra koja implementira operaciju sažimanja, često maksimalnog sažimanja, koja služi za redukciju dimenzionalnosti podataka i isticanje dominantnih značajki unutar segmenata ulazne slike.

Svaki od ovih modula dizajniran je da efikasno procesira podatke u realnom vremenu koristeći paralelnu arhitekturu FPGA. Proces spajanja ovih modula je ključan korak u dizajnu FPGA. Ciklus povezivanja IP jezgri se nastavlja ovisno o arhitekturi mreže, pa se slični blokovi mogu ponavljati za implementaciju višeslojnih konvolucijskih mreža. Time se održava strukturiran i efikasan tok podataka kroz dizajn. Ovi koraci vode do izrade finalnog projekta koji se sastoji od prethodno navedenih IP jezgri ključnih za rad CNN-a smještenih u *StreamingDataflowPartition* jezgru. Pored njih, u finalnom projektu se nalaze i gradivne IP jezgre koje su opisane u nastavku [33].

- ***ZYNQ7 processing system*** - predstavlja centralnu procesorsku jedinicu (engl. *Central Processing Unit* - CPU) koja upravlja radom cijelog sustava, uključujući konfiguraciju FPGA, komunikaciju s memorijom i I/O operacije. Povezan je s DDR memorijom, koja služi za pohranu podataka i izvršnih programa.
- ***AXI Interconnect*** - moduli omogućavaju komunikaciju između različitih dijelova sustava koristeći standardni AXI protokol. Ovi međusobni povezivači (engl. *interconnections*) služe za usmjeravanje podataka između CPU-a, IP jezgri i drugih periferija.
- ***idma0*** i ***odma0*** - DMA (engl. *Direct Memory Access*) kontroleri koji omogućavaju brzo prebacivanje podataka između memorije i IP jezgri bez potrebe za direktnim uključivanjem CPU-a, što značajno ubrzava obradu podataka.

- ***StreamingDataflowPartition*** moduli - IP jezgre koje predstavljaju pojedinačne dijelove neuronske mreže, specifično dizajnirane za paralelnu obradu podataka. Ove jezgre su rezultat optimizacije i kvantizacije modela u prethodnim koracima korištenjem alata FINN, a sada su integrirane u krajnji sustav kao dio FPGA dizajna.
- ***smartconnect_0*** - pametni međusobni povezivač koji optimizira korištenje širine pojedinih kanala i protok podataka unutar FPGA, prilagođavajući se potrebama specifičnog radnog opterećenja.
- **Processor system reset** blok - osigurava ispravno pokretanje i resetiranje cijelog sustava, sinkronizirajući različite komponente i osiguravajući da sve počinje raditi u usklađenom stanju.
- **Streaming *FIFO*** memorija - omogućava efikasno upravljanje protokom podataka između različitih slojeva obrade. Ovaj mehanizam je ključan za održavanje kontinuirane obrade podataka kroz cijeli sustav, osiguravajući da podaci koji prvi pristignu budu prvi i obrađeni. To eliminira zagušenja i optimizira *throughput* (prolaznost, protok).

Potpuno spojeni i izgrađeni projekt sa svim prethodno navedenim modulima je prikazan na slici 3.20.



Slika 3.20. Prikaz konačnog projekta u Vivado alatu za implementaciju neuronske mreže na FPGA.

Sve ove komponente zajedno čine konačni dizajn koji je spreman za sintezu i implementaciju na FPGA, pretvarajući pripremljeni model dubokog učenja u hardverski akcelerator. Iz konačnog dizajna se generiraju *bitstream* (.bit) i *hardware* (.hwh) datoteke koje se direktno učitavaju na PYNQ Z2 pločicu. Konačni dizajn i izgled komponenata koje se pojavljuju u projektu unutar Vivado alata je jednak kao i kod prethodnog primjera LeNet-5 neuronske mreže s razlikom u rasporedu IP jezgri zbog različitih slojeva u arhitekturi. Zbog toga je prikazan samo primjer izgrađenog projekta u Vivado alatu za Yolov3 Tiny model, jer su izgled projekta i sve IP jezgre pokriveni s jednim primjerom navedene neuronske mreže.

Generiranje upravljačkih funkcija za PYNQ FPGA platformu

Na osnovu izgrađenog projekta u Vivado alatu i potrebnih datoteka koje se implementiraju na *SD* karticu, finalni dio toka rada kroz FINN je generiranje potrebnih upravljačkih funkcija za inferenciju neuronske mreže na FPGA. Takav postupak uključuje slijedne korake opisane u nastavku.

1. Generiranje PYNQ upravljačkog programa:

- *MakePYNQDriver* (*"zynq-iodma"*) transformacija [30] u FINN-u stvara Python upravljačku funkciju specifičnu za model i ciljnu FPGA platformu. Ova upravljačka funkcija služi kao sučelje između modela implementiranog na FPGA i aplikacija koje ga koriste, omogućavajući jednostavno upravljanje i pokretanje modela. Transformacija iz koje se dobiva Python upravljačka funkcija je prikazana u kodu na slici 3.21.

```
from finn.transformation.fpgadataflow.make_pynq_driver import MakePYNQDriver
model = model.transform(MakePYNQDriver("zynq-iodma"))
```

Slika 3.21. Generiranje upravljačke funkcije za kreirani Vivado projekt.

2. Priprema direktorija za implementaciju:

- Kreira se direktorij koji će sadržavati sve potrebne datoteke za implementaciju modela, uključujući (.bit) datoteku, (.hwh) datoteku, upravljačke funkcije i potrebne biblioteke (Slika 3.22.). Kao ključna datoteka za pokretanje je (*driver.py*). Navedene datoteke se kopiraju u kreirani direktorij te on služi kao paket koji se može distribuirati i lako instalirati na željenom FPGA uređaju.

```
from shutil import copy
from distutils.dir_util import copy_tree

# create directory for deployment files
deployment_dir = make_build_dir(prefix="pynq_deployment_")
model.set_metadata_prop("pynq_deployment_dir", deployment_dir)

# get and copy necessary files
# .bit and .hwh file
bitfile = model.get_metadata_prop("bitfile")
hwh_file = model.get_metadata_prop("hw_handoff")
deploy_files = [bitfile, hwh_file]

for dfile in deploy_files:
    if dfile is not None:
        copy(dfile, deployment_dir)

# driver.py and python libraries
pynq_driver_dir = model.get_metadata_prop("pynq_driver_dir")
copy_tree(pynq_driver_dir, deployment_dir)
```

```
['/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/qonnx/core/__init__.py',
 '/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/qonnx/core/datatype.py',
 '/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/qonnx/util/basic.py',
 '/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/qonnx/util/__init__.py',
 '/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/finn/util/data_packing.py',
 '/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/finn/util/__init__.py',
 '/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/driver_base.py',
 '/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/driver.py',
 '/tmp/finn_dev_davorin/pynq_deployment_ppage9xm/validate.py']
```

Slika 3.22. Kod za izradu foldera s svim potrebnim datotekama pri inferenciji modela na FPGA.

Svi navedeni direktoriji i datoteke se arhiviraju u (.zip) datoteku kako bi se cjelokupni tok rada kroz FINN zaključio. Postupak generiranja i kreiranja finalne arhivirane datoteke za inferenciju modela neuronske mreže je identičan i za model neuronske mreže za detekciju objekata. Na ovaj način se osigurava da su svi potrebni elementi za inferenciju modela na FPGA organizirani i spremni za upotrebu.

3.3. Implementacija i evaluacija na FPGA

U ovom dijelu je opisana implementacija i evaluacija (LeNet-5 i Yolov3) neuronskih mreža koje su prethodno optimizirane za rad na FPGA platformama. Proces uključuje detaljnu prilagodbu i testiranje modela na PYNQ-Z2, čime se ocjenjuje njihova praktična primjenjivost i performanse u stvarnim uvjetima. PYNQ-Z2 je razvojna pločica bazirana na Xilinx Zynq-7000 SoC, koja

omogućuje jednostavnu implementaciju i testiranje različitih aplikacija, uključujući neuronske mreže, zahvaljujući svojoj kombinaciji ARM procesora i PL logike (engl. *Programmable Logic*). PYNQ (engl. *Python Productivity for Zynq*) je okruženje koje koristi Python programski jezik za programiranje FPGA-a, čime se pojednostavljuje razvoj hardversko-softverskih rješenja [34]. Ova pločica podržava razvoj prilagođenih rješenja za paralelnu obradu i visoku energetsku učinkovitost, omogućujući optimalno iskorištavanje resursa FPGA arhitekture za ubrzanje složenih izračuna i obradu velikih količina podataka. Cilj je potvrditi efikasnost optimizacijskih postupaka, uključujući kvantizaciju i specifične prilagodbe slojeva, kako bi se osiguralo da modeli ne samo da održavaju visoku točnost, već i iskorištavaju prednosti koje FPGA platforme nude u pogledu brzine i energetske učinkovitosti. Evaluacija točnosti na FPGA uređajima ključan je korak koji demonstrira potencijal ovih tehnologija za buduće aplikacije dubokog učenja.

3.3.1. Metode LeNet-5 neuronske mreže za klasifikaciju

Za demonstraciju implementacije i evaluacije, koristi se model optimiziran za klasifikaciju s arhitekturom LeNet-5, pripremljen i konfiguriran za izvođenje na FPGA-u. Proces uključuje nekoliko ključnih koraka koji su opisani u nastavku.

- **Preuzimanje i instalacija datoteka na PYNQ-Z2 platformi** - prethodno generirana arhivirana datoteka se preuzima i prenosi na PYNQ-Z2 platformu. Unutar arhivirane datoteke nalaze ključne upravljačke funkcije (*driver.py* i *driver_base.py*) koje omogućavaju jednostavniji pristup pri pokretanju modela neuronske mreže.
- **Ekstrakcija i pokretanje modela neuronske mreže** - ekstrahira se sadržaj preuzete arhive na FPGA instaliraju se potrebne Python biblioteke i pokreće skriptu za izvršavanje inferencije, koristeći predefinirane *bitstream* i ulazne datoteke. Rezultat inferencije pohranjen je u obliku *output.npy* datoteke, koja se može analizirati i vizualizirati na PYNQ-u. Potrebne komande za pokretanje su prikazane na slici 3.23.

```
sudo python3 -m pip install bitstring
sudo python3 driver.py --exec_mode=execute --batchsize=1 --bitfile=resizer.bit --inputfile=input.npy
```

Slika 3.23. Potrebna biblioteka za instalaciju na PYNQ platformi te komanda za inferenciju modela neuronske mreže.

- **Validacija točnosti modela** - važan korak u procesu implementacije je validacija točnosti modela na PYNQ-Z2 platformi. Ovo uključuje instalaciju paketa za pristup skupu podataka, pokretanje validacijske skripte za izračun točnosti na CIFAR-10 skupu podataka i usporedbu rezultata s prethodno navedenim točnostima. Točnost izvršenog modela na

FPGA trebala bi biti usporediva s točnostima dobivenim tijekom procesa treniranja i optimizacije. Primjer pokretanja komande za validaciju i pristup podacima je prikazan na slici 3.24.

```
sudo pip3 install git+https://github.com/fbcotter/dataset_loading.git@0.0.4#egg=dataset_loading
sudo python3 validate.py --dataset cifar10 --batchsize 1000
```

Slika 3.24. Komande za pristup skupu podataka s PYNQ platforme i pokretanje validacijske skripte za CIFAR10 skup podataka.

Procesi opisani za rad s CIFAR-10 skupom podataka primjenjuju se na identičan način i za MNIST skup podataka. Dovoljno je promijeniti naziv skupa podataka u odgovarajućoj liniji koda (npr. u liniji prikazanoj na slici 3.24.) i učitati MNIST podatke u FPGA-ovom *Jupyter Notebook* okruženju.

3.3.2. Metode Yolov3 Tiny neuronske mreža za detekciju

Kao što je prethodno razmatrana implementacija i evaluacija modela za klasifikaciju na FPGA, sličan pristup se primjenjuje i za metode neuronskih mreža specijalizirane za detekciju objekata. U ovom slučaju, razvijen je *Jupyter Notebook* koji koristi optimizirani model za detekciju implementiran na PYNQ-Z2 platformi, s dodatnim funkcionalnostima za obradu i vizualizaciju rezultata detekcije.

Jupyter Notebook sadrži sve potrebne skripte za izvršavanje inferencije modela na FPGA platformi, uključujući pripremu ulaznih slika, pokretanje modela te obradu i vizualizaciju rezultata. U nastavku su navedeni ključni dijelovi *notebooka* i opis njihove funkcionalnosti.

- **Učitavanje, priprema modela i potrebnih biblioteka** - za pravilnu inferenciju neuronske mreže s ciljem detekcije objekata, potrebna je biblioteka *torch* koja se naknadno instalira na PYNQ. Model neuronske mreže za detekciju je već optimiziran i spreman za upotrebu na FPGA platformi te korištenjem klase *FINNExampleOverlay*, učitava s pripadajućim *bitstream*-om i težinama čime se priprema za inferenciju (Slika 3.25.).

```
import os
import glob
import cv2
import numpy as np
import time
import torch
import torch.nn as nn

from matplotlib import pyplot as plt

from driver.driver import io_shape_dict
from driver.driver_base import FINNExampleOverlay
from utils import (clip_coords, scale_coords, letterbox,
                   xywh2xyxy, non_max_suppression,
                   visualize_boxes)
from models import Detect
```

```
driver = FINNExampleOverlay(
    bitfile_name="./bitfile/finn-accel.bit",
    platform="zynq-iodma",
    io_shape_dict=io_shape_dict,
    batch_size=1,
    runtime_weight_dir="runtime_weights/",
)
```

Slika 3.25. Izgled *Jupyter Notebook* dijela za učitivanja modela i potrebnih biblioteka.

- **Učitavanje ulaznih slika** - ulazne slike za detekciju se učitavaju iz određenog direktorija, pripremajući ih za obradu modelom (Slika 3.26.). Korištenje funkcije *letterbox* osigurava da su sve slike pravilno dimenzioniranje i spremne za procesiranje.

```
test_img_folder = "../../inputs/images/"
test_img_paths = glob.glob(test_img_folder + "*.jpg")
output_path = "../../test_outputs/"
if not os.path.exists(output_path):
    os.makedirs(output_path)
```

Slika 3.26. Učitavanje slika i specificiranje potrebnih izlaznih putanja.

- **Specifikacija predefiniranih parametara** - za preciznu detekciju objekata na slikama, model koristi predefinirane parametre koji obuhvaćaju klase objekata *names*, broj klasa *nc* te *anchor boxes* koje su ključne za određivanje položaja i veličine detektiranih objekata. Parametar *scale* se koristi za skaliranje izlaznih vrijednosti modela, što je bitno za

interpretaciju rezultata na originalnoj veličini slika. Na temelju ovih parametara, kreiran je detekcijski sloj *Detect*, koji služi za obradu izlaznih podataka modela i izvlačenje relevantnih informacija o detekciji. Kod kojim se postavljaju navedeni parametri te poziva funkcija *Detect* prikazan je na slici 3.27.

```
names = ['dog, cat']
nc  = 2
anchors = np.array([[10,14,23,27,37,58]]) / np.array([32])
scale = np.load("./bitfile/scale.npy")
detect_head = Detect(nc, anchors)
```

Slika 3.27. Specificiranje parametra za preciznu izradu i klasificiranje koordinata okvira.

- **Priprema slika** - za svaku sliku iz definiranog skupa, originalna slika se učitava i kopira. Slika se zatim proporcionalno prilagođava (koristeći funkciju *letterbox*) na dimenzije koje odgovaraju ulaznom sloju neuronske mreže (u ovom primjeru 416x416 piksela), osiguravajući da originalni omjer stranica ostane nepromijenjen. Slika se dodatno obrađuje tako da se boje kanala preurede iz *BGR* (koji koristi *OpenCV*) u *RGB* format, koji se obično koristi u obradi slika dubokim učenjem.
- **Izvršavanje inferencije** - pripremljena slika se prosljeđuje FPGA upravljačkoj funkciji koja pokreće inferenciju dubokog učenja. Rezultat inferencije je skup predikcija koje uključuju koordinate *bounding box*-a identifikaciju klasa detektiranih objekata (npr. “pas", "mačka") i pripadajuće vjerojatnosti. Rezultati se skaliraju i preuređuju kako bi se omogućila daljnja obrada.
- **Obrada i vizualizacija rezultata** - rezultati inferencije se obrađuju korištenjem funkcije *non_max_suppression* za uklanjanje redundantnih *bounding box*-ova. Za svaki detektirani objekt, *bounding box* se prilagođava originalnoj veličini slike, a zatim se vizualizira na originalnoj slici. Ovaj korak uključuje iscrtavanje *bounding box*-a s imenima klasa i vjerojatnostima.
- **Spremanje i prikaz rezultata** - modificirane slike s vizualiziranim detekcijama se spremaju u definiranom izlaznom direktoriju i mogu se prikazati unutar *Jupyter Notebook-a*. Ovaj pristup omogućava jednostavnu analizu i evaluaciju učinka modela dubokog učenja na detekciji objekata u stvarnom vremenu. Svi prethodno navedeni koraci su prikazani na slici 3.28.

```
for number, test_img_path in enumerate(test_img_paths):
    img_org = cv2.imread(test_img_path)
    img = img_org.copy()

    h, w, _ = img_org.shape
    img, ratio, (dw, dh) = letterbox(img, (416,416), auto=False)

    img = img[:, :, ::-1]
    img = img.astype(np.uint8)
    driver_in = np.expand_dims(img, 0)

    output = driver.execute(driver_in)
    output = scale*output
    output = output.transpose(0,3,1,2)

    output = torch.from_numpy(output)
    pred = detect_head([output])[0]

    pred = non_max_suppression(pred, conf_thres=0.20, iou_thres=0.10, classes=None, max_det=1000)

    boxes_detected, class_names_detected, probs_detected = [], [], []
    # Process predictions
    for i, det in enumerate(pred):  # per image
        if len(det):
            # Rescale boxes from img_size to im0 size
            det[:, :4] = scale_coords(img.shape, det[:, :4], img_org.shape).round()
            # Print results
            for c in np.unique(det[:, -1]):
                n = (det[:, -1] == c).sum()  # detections per class
                print(f"{n} {names[int(c)]}")  # add to string
            # Write results
            for *xyxy, conf, cls in reversed(det):
                c = int(cls)  # integer class
                label = f'{names[c]} {conf:.2f}'
                boxes_detected.append(xyxy)
                class_names_detected.append(names[c])
                probs_detected.append(conf)

    # Visualize.
    image_boxes = visualize_boxes(img_org, boxes_detected, class_names_detected, probs_detected)
    cv2.imwrite(output_path+f"{number}.jpg", image_boxes)

    plt.imshow(cv2.cvtColor(image_boxes, cv2.COLOR_BGR2RGB))
    plt.show()
```

Slika 3.28. Dio *Jupyter Notebook-a* čiji je zadatak detekcija objekata iscrtavanje i spremanje okvira detekcije.

Kroz niz postupaka i transformacija iz prethodnih poglavlja može se demonstrirati uspješnost implementacije i inferencije neuronske mreže na FPGA. Time se obuhvaćaju svi ključni koraci za postizanje primjera koji se može koristiti u praktičnim aplikacijama unutar FPGA tehnologije. Ovaj proces ne samo da ilustrira mogućnost FPGA uređaja da akcelerira napredne algoritme dubokog učenja, već i naglašava važnost detaljne pripreme i optimizacije modela za postizanje optimalnih performansi. Stoga je fokus sljedećeg odjeljka na ostvarivanju optimalnih rezultata

koristeći prethodnu metodologiju te procjeni učinkovitosti implementiranih neuronskih mreža u realnim uvjetima.

4. REZULTATI IMPLEMENTACIJE I INFERENCIJE NEURONSKIM MREŽA

U ovom poglavlju analiziraju se rezultati implementacije i inferencije optimiziranih modela neuronskih mreža na FPGA, s posebnim fokusom na modele za klasifikaciju (koristeći LeNet-5 na MNIST i CIFAR-10 skupovima podataka) i detekciju (koristeći Yolov3 Tiny za detekciju mačaka, pasa, osoba i automobila). U nastavku je prikazana detaljna analiza performansi modela, uključujući točnost klasifikacije i detekcije, brzinu inferencije, kao i potrošnju hardverskih resursa na FPGA platformi.

4.1. Metodologija mjerenja

Za cjelovitu evaluaciju modela implementiranih na FPGA uređajima, koriste se metrike navedene u nastavku.

- **Točnost modela -** mjeri se postotak pravilno klasificiranih primjera za klasifikacijske zadatke, a preciznost, odziv i *mAP* (engl. *mean average precision*) koriste se za zadatke detekcije. Ove metrike pružaju uvid u opću sposobnost modela da točno interpretira ulazne podatke.
- **Brzina inferencije -** vrijeme koje je modelu potrebno da obradi ulaz i generira izlaz mjeri se kao ključni pokazatelj za realno-vremenske aplikacije. Ova mjera se prikazuje kao broj slika po sekundi ili milisekundi po slici, ovisno o veličini ulaznog skupa.
- **Potrošnja resursa na FPGA -** analiza upotrebe FPGA resursa poput *LUT*-ova, *FF*-ova, *BRAM*-ova i *DSP*-ova provodi se nakon sinteze i implementacije. Ova analiza pomaže u razumijevanju koliko je model optimiziran za korišteni hardver. PYNQ Z2 raspolaže sa sljedećim resursima:
    - **LUT** : 53 200,
    - **FF** : 106 400,
    - **BRAM** : 315 36K-BRAM blokova,
    - **DSP** : 220.
- **Energetska efikasnost** - sekundarna metrika koja može biti od ključne važnosti za specifične aplikacije, uključujući mobilne i ugradbene sustave.

4.2. Rezultati klasifikacije

Performanse LeNet-5 mreže testirane su na MNIST skupu podataka s tri konfiguracije kvantizacije: W1, W4 i W8. Ove konfiguracije odnose se na računsku preciznost težina (W) u bitovima. Analizirani su točnost modela, brzina inferencije i potrošnja FPGA resursa za svaku konfiguraciju prema opisanom postupku u radu[8]. Stoga, analogno metodologiji mjerenja, rezultati su opisani u nastavku.

4.2.1. LeNet-5 na MNIST skupu podataka

- **Točnost modela -** u Tablici 4.1. prikazani su postotci točnosti za sve tri konfiguracije kvantizacije s vidljivim povećanjem točnosti s povećanjem bitne širine. Ovo sugerira da veća preciznost težina i aktivacija može pridonijeti boljem prepoznavanju obrazaca [1], što je ključno za klasifikacijske zadatke poput prepoznavanja rukom pisanih brojeva. Međutim, veća preciznost dolazi s povećanjem potrebe za računalnim resursima, što može utjecati na brzinu inferencije i energetsku efikasnost.

Tablica 4.1. Točnost modela LeNet-5 na MNIST skupu podataka za različite konfiguracije kvantizacije.

Konfiguracija	Točnost (%)
W1	71.12
W4	81.24
W8	89.96

- **Brzina inferencije -** brzina inferencije modela za svaku konfiguraciju kvantizacije detaljno je analizirana kako bi se utvrdilo prosječno vrijeme potrebno za obradu jedne slike. Dobiveni rezultati se mogu koristiti za usporedbu sa sličnim projektima, gdje je korišten sličan pristup, a isti skup podataka [1]. Na osnovu rezultata u Tablici 4.2. se može primijetiti da dok se povećava točnost s većom kvantizacijom, dolazi do blagog smanjenja u brzini inferencije. Ovo ukazuje na kompromis između točnosti i brzine, koji je ključan za aplikacije u stvarnom vremenu gdje je brzina inferencije jednako važna kao i točnost.

Tablica 4.2. Inferencija neuronskih mreža za različite kvantizacije na MNIST skupu podataka.

Konfiguracija	Vrijeme inferencije (ms/slika)	Protok (slika/s)
W1	0.07721	12950.176
W4	0.08413	11790.695
W8	0.10892	9180.921

- **Potrošnja FPGA resursa** - detaljnom analizom obuhvaćena je potrošnja ključnih FPGA resursa, uključujući *LUT*-ove, *FF*-ove, *BRAM*-ove i *DSP*-ove, za svaku od navedenih konfiguracija. U radu [33] također je na sličan način prikazana usporedba rezultata, gdje je stavljen fokus na stupanj kvantizacije i analognu potrošnju resursa. Analiza je pružila važne uvide u efikasnost implementacije modela na hardveru što je prikazano u Tablici 4.3.

Tablica 4.3. Potrošnja FPGA Resursa na MNIST skupu podataka.

Konfiguracija	LUT	FF	BRAM	DSP
W1	10.82%	5.02%	0.64%	4.55%
W4	12.60%	6.51%	2.22%	12.27%
W8	17.45%	8.48%	2.86%	15.45%

Za različite konfiguracije, upotreba *LUT*-ova, *FF*-ova, *BRAM*-ova i *DSP*-ova se smanjuje kako se preciznost smanjuje. Time se omogućava proširenje neuronske mreže u smislu arhitekture. Na primjer, LeNet-5 arhitektura može biti značajno povećana dodavanjem novih konvolucijskih slojeva ili slojeva sažimanja, što bi potencijalno moglo značajno povećati točnost modela neuronske mreže.

- **Energetska efikasnost** - energetska efikasnost modela mjeri se kroz potrošnju snage tijekom operacija inferencije izraženu u vatima (W). Cilj je minimizirati potrošnju energije bez značajnog utjecaja na performanse modela.

Tablica 4.4. Potrošnja snage za različite konfiguracije kvantizacije na MNIST skupu podataka.

Konfiguracija	Potrošnja snage (W)
W1	2.5
W4	2.9
W8	3.1

Merenjem potrošnje snage tokom inferencije za svaku kvantizaciju u Tablici 4.4. prikazana je energetska efikasnost. Pokazalo se da smanjenjem bitne širine težina i aktivacija dolazi do smanjenja potrošnje energije.

4.2.2. LeNet-5 na CIFAR-10 skupu podataka

Analiza performansi modela LeNet-5 na CIFAR-10 skupu podataka [8] provedena je analogno mjerenjima na MNIST skupu, no s određenim modifikacijama kako bi se prilagodila složenijoj prirodi CIFAR-10 slika. CIFAR-10 slike su u boji, što predstavlja dodatni izazov u odnosu na MNIST skup podataka koji se sastoji od sivih slika. Stoga su provedene evaluacije na istim konfiguracijama kao u prethodnom poglavlju.

- **Točnost modela -** s obzirom na dodatnu informaciju boje kod CIFAR-10 slika, prilagodbe modela u poglavljima 3.2.1 i 3.2.2 bile su ključne u očuvanju točnosti klasifikacije, što se može vidjeti i na rezultatima iz Tablice 4.5. gdje je veća bitna širina (manji stupanj kvantizacije) postigla veću točnost.

Tablica 4.5. Točnosti neuronskih mreža na CIFAR10 skupu podataka.

Konfiguracija	Točnost (%)
W1	63.12
W4	71.24
W8	77.96

- **Brzina inferencije -** u Tablici 4.6. se prikazuju rezultati inferencije i protoka slika po sekundi za različite konfiguracije na FPGA platformi PYNQ Z2. Može se vidjeti da manja bitna širina (W1) ima najkraće vrijeme inferencije od 0.33343 ms i najveći protok od

2990.176 slika/s što ga čini najefikasnijim u smislu brzine obrade. Kako se bitna širina povećava (W4 i W8), vrijeme inferencije raste na 0.41413 i 0.61892 ms, dok se protok smanjuje na 2431.311 slika/s i 1643.321 slika/s. To sugerira da veći nivoi kvantiziranosti zahtijevaju više vremena za obradu svake slike, smanjujući tako ukupan broj slika koje se mogu obraditi u sekundi [35].

Tablica 4.6. Inferencija neuronskim mreža za različite kvantizacije na CIFAR10 skupu podataka.

Konfiguracija	Vrijeme inferencije (ms/slika)	Protok (slika/s)
W1	0.33343	2990.176
W4	0.41413	2431.311
W8	0.61892	1643.321

- **Potrošnja FPGA resursa** - može se u Tablici 4.6. vidjeti da viša bitna širina, dok poboljšava točnost, značajno povećava potrošnju resursa. Bitna širina W1 koristi najmanje, dok W8 koristi najviše.

Tablica 4.7. Potrošnja FPGA Resursa na CIFAR10 skupu podataka.

Konfiguracija	LUT	FF	BRAM	DSPs
W1	11.69%	6.68%	2.22%	14.09%
W4	16.73%	8.48%	3.49%	18.64%
W8	21.05%	9.42%	5.40%	25.00%

- **Energetska efikasnost** - mjerenje potrošnje energije tijekom operacija inferencije je prikazano u Tablici 4.8. Rezultati pokazuju da konfiguracija s većom bitnom širinom (W8) troši najviše energije (4.0 W), dok konfiguracija s manjom bitnom širinom (W1) troši najmanje (2.1 W). To ukazuje na to da viša kvantiziranost, iako može poboljšati performanse modela, dolazi uz značajno povećanje energetske potrošnje, što naglašava važnost optimizacije za energetsku efikasnost.

Tablica 4.8. Potrošnja snage za različite konfiguracije kvantizacije na CIFAR10 skupu podataka.

Konfiguracija	Potrošnja snage (W)
W8	4.0
W4	2.7
W1	2.1

Kroz rezultate klasifikacije LeNet-5 modela na MNIST i CIFAR-10 skupovima podataka demonstrira se kako prilagodbe u kvantizaciji i konfiguraciji resursa utječu na performanse, brzinu i energetsku efikasnost na FPGA platformi. Analiza pokazuje da je moguće postići balans između točnosti i efikasnosti uzimajući u obzir ograničenja hardvera.

4.3. Rezultati detekcije

U ovom poglavlju analiziraju se performanse Yolov3 Tiny modela neuronske mreže prilagođenog za FPGA implementaciju putem FINN kompajlera, usmjerenog na detekciju mačaka i pasa, osoba te automobila.

4.3.1. Yolov3 Tiny za detekciju mačaka i pasa

U ovom dijelu analizira se prilagodba Yolov3 Tiny modela za FPGA implementaciju putem FINN kompajlera, usmjerena na detekciju mačaka i pasa.. Ovaj model neuronske mreže demonstrira kako se efikasnost i preciznost mogu balansirati s ograničenim resursima FPGA uređaja. Za različite razine kvantizacije i preciznosti težina, odabrana je optimalna konfiguracija za različite detekcij prikazana na slici 3.10. Takva konfiguracija se odnosi na ulazne i izlazne težine i aktivacijske jedinice neuronske mreže, definirana je kao: [4, 2, 2, 8]. U narednim poglavljima su prikazani rezultati rada neuronske mreže za detekciju s fokusom na: preciznost, odziv, brzinu inferencije i potrošnju FPGA resursa. Analogno za svaki skup podataka za detekciju je korištena ista metrika. Preciznost i odziv u slučaju detekcije mačaka i pasa su prikazani u Tablici 4.9.

Tablica 4.9. Preciznost i odziv za detekciju mačaka i pasa.

Objekt	Preciznost (%)	Odziv (%)
Mačke i psi	53.18	57.64

Preciznost od 53.18% i odziv od 57.64% za oba objekta ukazuju na umjereno uspješnu detekciju. Iako ove vrijednosti nisu iznimno visoke, one su znakovite za realno-vremenske aplikacije gdje je brzina često prioritizirana nad apsolutnom preciznošću. Ovako balansirane vrijednosti preciznosti i odziva pokazuju da model umjereno dobro identificira većinu relevantnih objekata bez velikog broja lažno pozitivnih detekcija. Ipak, ovi brojevi nisu optimalni. U obzir treba uzeti i razinu kvatizacije koja dodatno utječe na konačnu preciznost i odziv. Osim toga FPGA je specifičan uređaj gdje je fokus na paralelizaciji svih slojeva, što može dovesti da neuronska mreža za detekciju nema izvanredne rezultate detekcije kao na GPU ili CPU.

Budući da je riječ i o vremenski ovisnim aplikacijama, Tablica 4.10. sadrži ključne informacije o tome je li neuronska mreža kompetentna za rad u stvarnom vremenu.

Tablica 4.10. Brzina inferencije za detekciju mačaka i pasa.

Objekt	Brzina (slika/s)
Mačke i psi	20.17

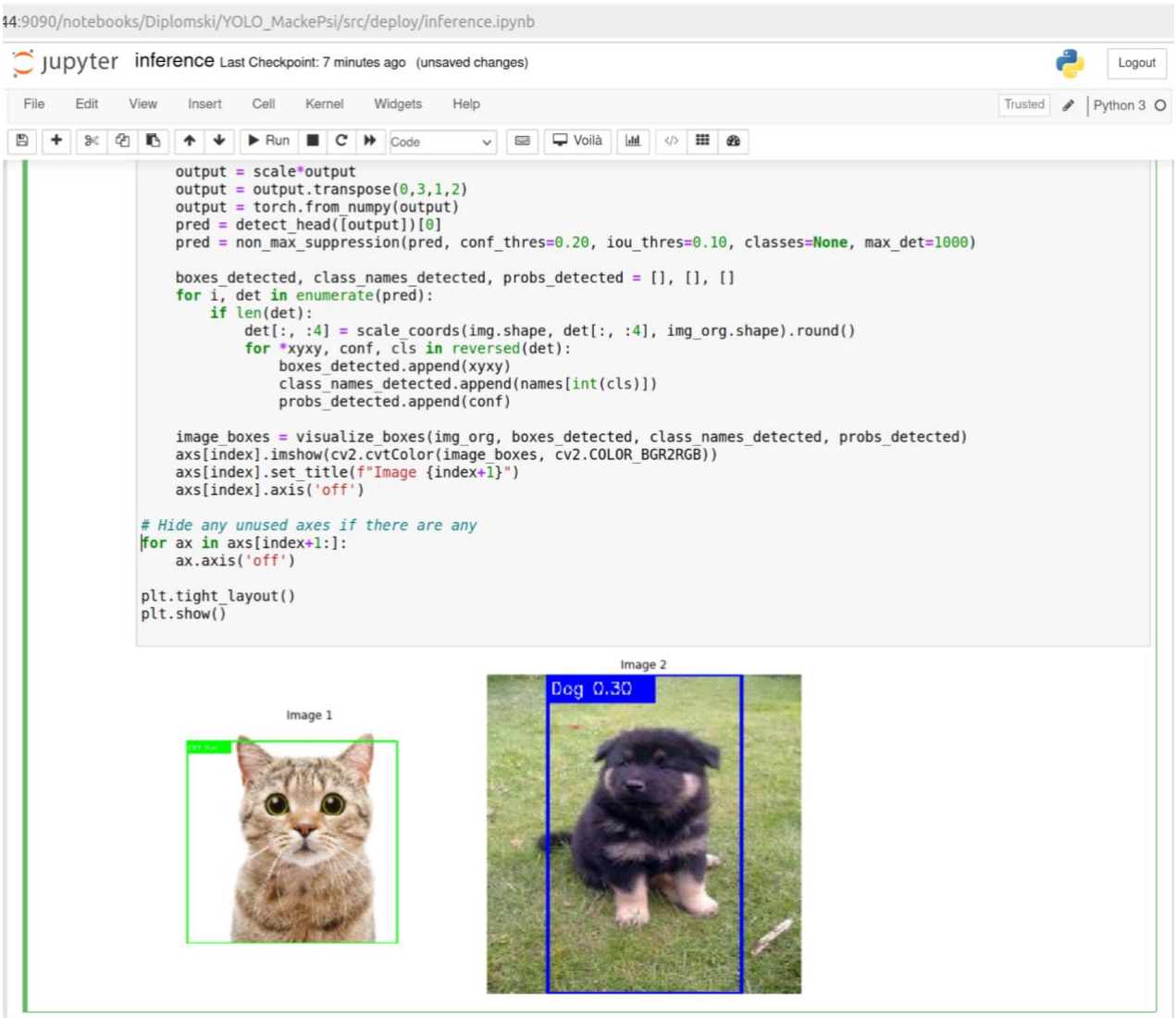
Održavanje visoke brzine uz zadržavanje točnosti je izazov, posebice na FPGA platformama koje su limitirane s obzirom na optimizaciju modela, paralelno procesiranje i dostupne resurse. Stoga inferencija od 20.17 slika u sekundi značajna je jer predstavlja dobar rezultat za složenu neuronsku mrežu i obradu slika više kvalitete. Ovo je postignuto uz nisku energetsku potrošnju, koja se kreće između 3W i 5W, što je usporedivo s energetskim zahtjevima kod klasifikacijskih problema.

U konačnici, kako bi neuronska mreža efikasno radila te postizala zadovoljavajuće rezultate i brzine inferencije, faktor koji joj to omogućava jesu gradivni blokovi od FPGA vidljivi u Tablici 4.11.

Tablica 4.11. Potrošnja FPGA resursa za detekciju mačaka i pasa.

Resurs	Upotreba (%)
LUT	58.06%
FF	22.69%
BRAM	24.44%
DSP	76.82%

Implementacija modela Yolov3 Tiny na FPGA platformi uključuje značajnu upotrebu resursa, pri čemu se koristi 58.06% dostupnih LUT-ova i 22.69% FF-ova, što odražava složenost modela i potrebu za većim brojem resursa u obradi podataka. Brojka od 76.82% DSP blokova naglašava upotrebu digitalnih operacija množenja, što je tipično za konvolucijske neuronske mreže koje zahtijevaju intenzivne matematičke operacije za obradu slika. BRAM jedinica koristi se 24.44%, što omogućava pohranu podataka ključnih za procesiranje, osiguravajući da su potrebni podaci brzo dostupni tijekom inferencije. Ovaj pristup omogućava sveobuhvatnu evaluaciju detekcijskih sposobnosti modela Yolov3 Tiny ističući kako se različiti aspekti modela (preciznost, brzina i potrošnja resursa) mogu optimizirati za specifične potrebe aplikacija. Rad inferencije neuronske mreže na FPGA je prikazan na slici 4.1. gdje se može vidjeti da su detektirani objekti poput mačke i psa te da je nacrtan adekvatan *bounding box*.



Slika 4.1. Primjer inferencije i detekcija objekata (mačke i psa) na PYNQ Z2 platformi.

4.3.2. Yolov3 Tiny za detekciju osoba

Nakon što su analizirane performanse Yolov3 Tiny modela za detekciju mačaka i pasa, fokus se prebacuje na primjenu istog modela za detekciju osoba. Kroz implementaciju na FPGA platformi, uz upotrebu FINN kompajlera, demonstrira se način na koji model može biti prilagođen za različite scenarije detekcije, u ovom slučaju za detekciju osoba. Identične metrike kao za prethodnu detekciju su provedene kroz Tablice 4.12., 4.13. i 4.14.

Tablica 4.12. Preciznost i odziv za detekciju osoba.

Objekt	Preciznost (%)	Odziv (%)
Osoba	83.43	75.50

U kontekstu FPGA implementacije pomoću FINN kompilatora, preciznost (83.43%) i odziv (75.50%) pokazuju bolju optimizaciju modela za detekciju osoba u odnosu na prethodni primjer. Ova razina preciznosti i odziva osigurava da se osobe mogu pouzdanije identificirati bez značajnog broja lažno pozitivnih rezultata.

Tablica 4.13. Brzina Inferencije za detekciju osoba.

Objekt	Brzina (slika/s)
Osoba	18.12

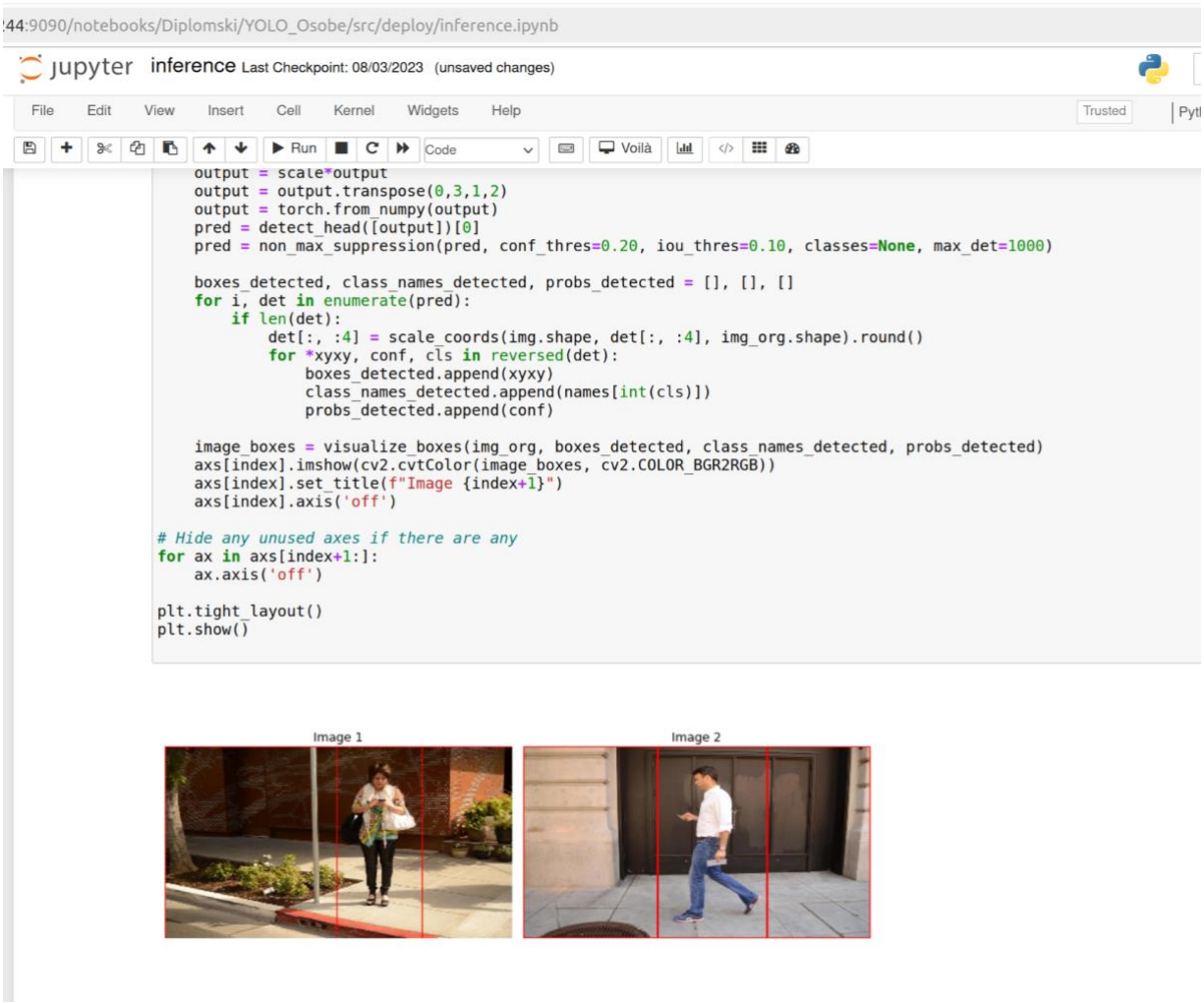
Brzina inferencije od 18.12 slika u sekundi (Tablica 4.13.) demonstrira sposobnost modela da efikasno obradi ulazne slike na FPGA. Takva brzina se dobiva iz iterativnih testiranja razina kvantizacije kroz transformacije koje su opisane u poglavlju 3.2. Povećanje brzine inferencije bi značajno utjecalo na preciznost i odziv, gdje bi se oni proporcionalno smanjivali zbog manje preciznosti težinskih vrijednosti.

Tablica 4.14. Potrošnja FPGA resursa za detekciju osoba.

Resurs	Upotreba (%)
LUT	58.42%
FF	22.72%
BRAM	24.44%
DSP	68.64%

Značajna upotreba *LUT*-ova, *FF*-ova i *DSP* blokova iz Tablice 4.14. ilustrira složenost modela koji je prilagođen za detekciju osoba. Optimalna iskorištenost ovih resursa ključna je za održavanje visokih performansi modela na FPGA. Može se zaključiti da postoji još prostora za poboljšanje modela neuronske mreže glede resursnog prostora i povećanja razina kvantizacije. Također, u

obzir dolazi i drugačiji raspored kod transformacije presavijanja (poglavlje 3.2.) kako bi se maksimizirala resursnu zauzetost FPGA. Vizualizacija detekcije osoba je prikazana na slici 4.2. gdje se identificiraju i precizno označavaju prisutne osobe, demonstrirajući sposobnost modela da se prilagodi različitim scenarijima primjene.



Slika 4.2. Detekcija osoba u različitim okruženjima na FPGA u *Jupyter Notebook*-u.

4.3.3. Yolov3 Tiny za detekciju automobila

Analogno prethodnim analizama pri detekciji drugih objekata, performanse optimizirane neuronske mreže Yolov3 Tiny su mjerene i testirane na skupu podataka za automobile. Koristeći identičan kompajler i postupak mreža je implementirana na FPGA s ciljem detekcije automobila. Stoga u Tablica 4.15. se prikazuju rezultati preciznosti i odziva implementiranih stavki u primjeru detekcije automobila.

Tablica 4.15. Preciznost i odziv za detekciju automobila.

Objekt	Preciznost (%)	Odziv (%)
Automobili	31.06	21.07

Za detekciju automobila, preciznost od 31.06% i odziv od 21.07% reflektiraju izazove s kojima se suočava model u stvarnim scenarijima. Niska preciznost i odziv rezultat su kvantizacije modela prije implementacije na FPGA. To znatno smanjuje sposobnost da točno identificira i klasificira objekte, posebno u kompleksnim ili pretrpanim scenama. Ovi rezultati ukazuju na potrebu za daljnjim finim podešavanjem i optimizacijom modela te odabirom i pripremom većeg skupa podataka kako bi se poboljšale performanse detekcije.

Tablica 4.16. Brzina inferencije za detekciju automobila.

Objekt	Brzina (slika/s)
Automobili	19.87

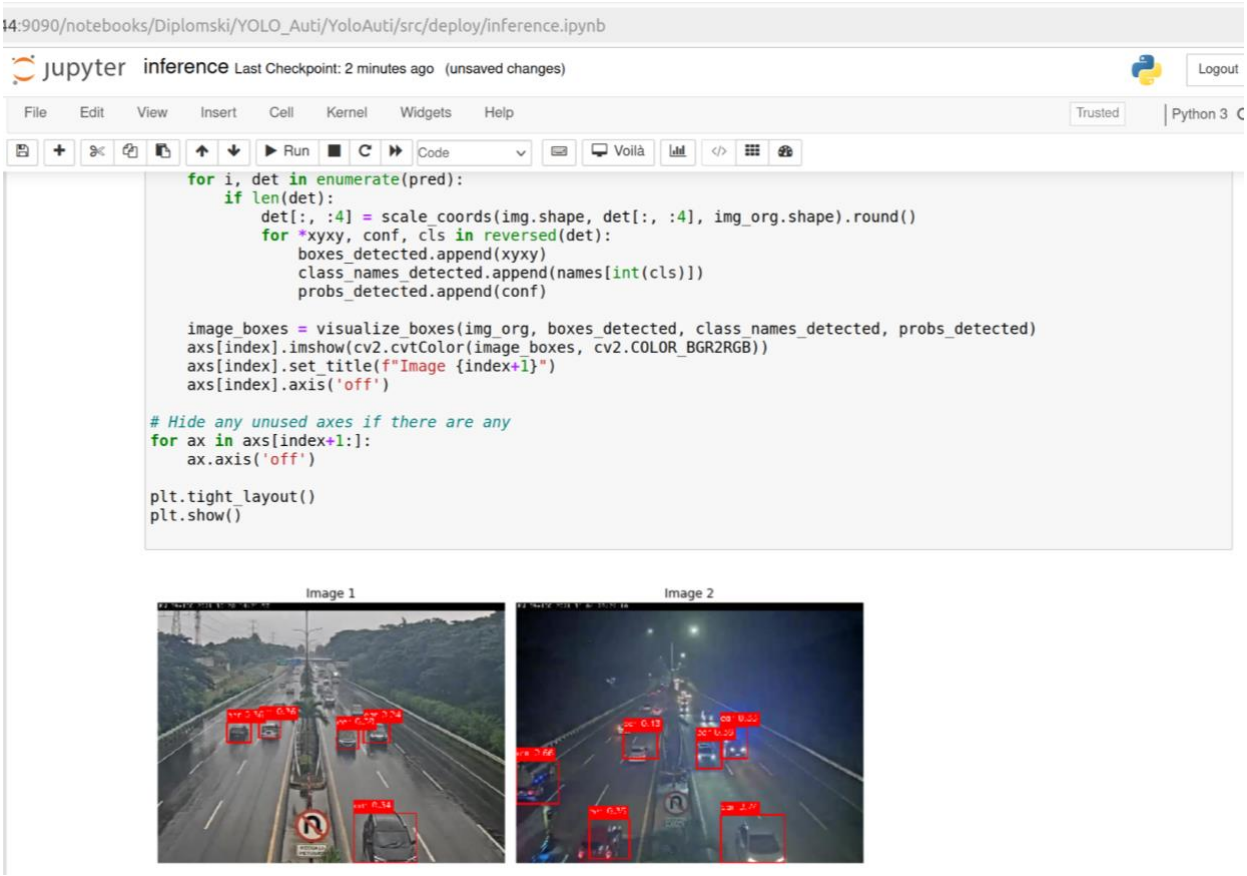
Iz Tablice 4.16. se može vidjeti da brzina inferencije od 19.87 slika po sekundi pokazuje da model neuronske mreže može efikasno procesirati ulazne slike unutar FPGA. Ovakva brzina je u čestim slučajevima dovoljna za mnoge aplikacije gdje je brz odziv važan kao što je praćenje prometa. Različita inferencijska vremena (primjer u odnosu na druge detekcije) mogu biti utjecaj i različitih veličina ulaznih slika i procesa obrade koje se mogu prethodno odvijati.

Tablica 17. Potrošnja FPGA resursa za detekciju automobila.

Resurs	Upotreba (%)
LUT	49.25%
FF	28.07%
BRAM	31.75%
DSP	79.09%

Model koristi gotovo polovicu dostupnih *LUT*-ova, što pokazuje složenost logike potrebne za obradu podataka. Manje od trećine FF-ova koristi se za upravljanje sekvencijalnim operacijama, dok je značajan dio BRAM jedinica alociran za pohranu podataka, omogućujući brz pristup i

učinkovitu inferenciju. Visoka upotreba DSP blokova naglašava potrebu za intenzivnim računskim operacijama. Detekcija većeg broja instanci automobila je napravljena na FPGA pri čemu su rezultati zajedno s *bounding box*-ovima u *Jupyter Notebook-a* prikazani na slici 4.3.



Slika 4.3. Primjer detekcije automobila unutar *Jupyter Notebook*-a na FPGA.

Zaključno, rezultati pokazuju da se ograničeni resursi FPGA mogu efikasno iskoristiti za postizanje zadovoljavajuće brzine i preciznosti. Analizom je utvrđena sposobnost FPGA platformi za izvršavanje složenih zadataka dubokog učenja, poput prethodno primijenjenih detekcija objekata. U dosadašnjem dijelu rada, kao implementacija dubokih neuronskih mreža na FPGA platformi, koristila se fiksna konfiguracija gdje je jedan model učitan i primjenjivan tijekom cijelog procesa. Iako takva metoda omogućuje postizanje zadovoljavajuće preciznosti i performansi, fleksibilnost sustava u prilagodbi promjenjivim uvjetima je ograničena. S primjenom dinamičke rekonfiguracije, otvorile bi se nove mogućnosti koje bi omogućile sustavu prilagodbu specifičnim zahtjevima u stvarnom vremenu. Na taj način, različiti modeli mogli bi se učitavati iz *DDR RAM*-a ovisno o vanjskim čimbenicima, bez potrebe za zaustavljanjem sustava. Na primjer, u aplikacijama koje koriste dronove za detekciju objekata, sustav bi mogao koristiti različite YOLO modele za detekciju ljudi, vozila ili površina, ovisno o visini na kojoj dron leti.

5. DINAMIČKO REKONFIGURIRANJE FPGA SUSTAVA ZA PRILAGOĐENU DETEKCIJU

Primjena dinamičkog rekonfiguriranja omogućuje FPGA uređajima da mijenjaju programirane *bitstream* datoteke prema zadanim poticajima kao što su promjene udaljenosti ili vremena. Pri nižim visinama sustav može detektirati ljude, dok na višim visinama može preći na detekciju automobila ili polja, pružajući veću operativnu fleksibilnost. Za sličnu problematiku postoje različiti referentni radovi. Primjerice, dinamička parcijalna rekonfiguracija omogućava FPGA uređajima brzo i efikasno mijenjanje dijelova svojih konfiguracija, što je ključno za adaptivne sustave kao što su softverski definirani radio uređaji ili prilagodljive letne (engl. *flight*) platforme [36, 37]. U ovim studijama se naglašava kako dinamička rekonfiguracija ne samo da poboljšava funkcionalnu adaptivnost, već i optimizira potrošnju energije i računalne resurse, što je presudno za misije s ograničenim resursima.

Pregled o dinamičkoj i parcijalnoj rekonfiguraciji FPGA pruža dublji uvid u arhitekture, metode i primjene, nudeći osnovu za implementaciju ove tehnologije u realnim operativnim scenarijima [38]. Ovi radovi stvaraju temelj za razvoj naprednih letnih sustava koji se mogu automatski prilagoditi specifičnim operativnim uvjetima u realnom vremenu. Ova istraživanja su također polazna točka istraživanja ovog diplomskog rada, gdje je cilj primjeniti rekonfiguraciju kreiranih programa za FPGA.

Implementacija dinamičke rekonfiguracije na PYNQ-Z2

Implementacija dinamičkog rekonfiguriranja omogućava FPGA platformama, poput PYNQ Z2, da tijekom rada automatski mijenjaju svoje izvršne *bitstream* datoteke na temelju specifičnih poticaja. Ova tehnologija pruža sustavima sposobnost prilagodbe funkcija u realnom vremenu, čime se znatno poboljšava operativna fleksibilnost i efikasnost. Jedan od ključnih sustava za primjenu dinamičke rekonfiguracije u ovom radu je *Systemd*. To je osnovni sustav inicijalizacije u većini Linux distribucija, koji koristi konfiguracijske datoteke za definiranje i upravljanje uslugama [39]. To pruža preciznu kontrolu nad procesima, omogućujući pokretanje određenih operacija pod specifičnim uvjetima. Na PYNQ Z2 platformi, *systemd* je ključan alat za upravljanje dinamičkim rekonfiguracijama FPGA, u realnom vremenu ovisno od promjena u okruženju.

Konfiguracija *systemd* usluge za dinamičko rekonfiguriranje

Za implementaciju usluge koja omogućuje dinamičko rekonfiguriranje FPGA modula na temelju promjena u visini leta, koristi se konfiguracijska datoteka *systemd*-a. Ova datoteka kontrolira kako i kada se pokreće određena skripta, omogućavajući prilagodbu detekcijskih modela ovisno o uvjetima okruženja. Na slici 5.1. je vidljiv primjer konfiguracije *systemd* usluge za testiranje u ovom radu.

```
[Unit]
Description=Dynamic FPGA Reconfiguration for Altitude-Based Detection
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/bin/switch_model.sh
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Slika 5.1. Konfiguracija *systemd* usluge za definiranje skripte za rekonfiguraciju.

Na slici 5.1. se može vidjeti da se preko *ExecStart-a* pokreće skripta *switch_model.sh* koja nadzire trenutne uvjete leta i mijenja FPGA konfiguraciju za detekciju. Skripta može sadržavati logiku za detekciju visine leta i automatsko učitavanje odgovarajućeg detekcijskog modela u FPGA. Primjer kako bi skripta za automatsko učitavanje i promjenu modela izgledala je prikazan na slici 5.2.

```
#!/bin/bash
# Detektiraj trenutnu visinu i učitaj odgovarajući model
if [ $(detect_altitude) -lt 50 ]; then
  # Niže visine - detekcija ljudi
  load_model /path/to/human_detection.bit
else
  # Više visine - detekcija automobila
  load_model /path/to/car_detection.bit
fi
```

Slika 5.2. Osnovni primjer implementacije skripte za dinamičku rekonfiguraciju.

U skripti na slici 5.2. se koristi uvjetna logika za detekciju visine leta i na temelju toga se određuje koji će model biti učitan u FPGA memoriju. Detaljan opis skripte se može predstaviti u stavkama opisanim u nastavku.

- **Detekcija visine leta** - skripta započinje provjerom visine leta pomoću hipotetičke funkcije *detect_altitude*. Ova funkcija vraća trenutnu visinu, koja se koristi za odlučivanje koju će se konfiguraciju učitati.
- **Uvjetna logika** - ako je visina manja od 50 metara (oznaka -lt na slici 5.2.), skripta pretpostavlja da su operativni uvjeti takvi da je potrebna detekcija ljudi. To je korisno u situacijama kao što su traganje i spašavanje na nižim visinama, gdje je važno brzo identificirati i locirati ljude. Ako je visina veća od 50 metara, skripta prelazi na konfiguraciju za detekciju automobila, što može biti korisno za misije koje uključuju nadzor cesta ili velikih otvorenih područja s veće visine.
- **Učitavanje modela** - skripta poziva funkciju *load_model* s odgovarajućim putem do *bitstream* datoteke (.bit) koja sadrži prethodno sintetizirani FPGA model za određenu detekciju. Ovo osigurava da se FPGA brzo i efikasno rekonfigurira za novi operativni način.

Uz navedene *bitstream* datoteke koje se učitavaju, mogu se također učitati određene Python skripte. Primjer jednog takvog učitavanja je prikazan na slici 5.3.

```
  GNU nano 4.8                                    /usr/local/bin/bootpy.sh
#!/bin/bash

. /etc/environment
for f in /etc/profile.d/*.sh; do source $f; done

BOOT_MNT=/boot
BOOT_DEV=/dev/mmcblk0p1
FIRST_PY=$BOOT_MNT/first.py
SECOND_PY=$BOOT_MNT/second.py

if !(mount | grep -q "$BOOT_MNT") ; then
   mount $BOOT_DEV $BOOT_MNT
fi

if test -f "$FIRST_PY"; then
   python3 $FIRST_PY
fi

if test -f "$SECOND_PY"; then
   python3 $SECOND_PY
fi
```

Slika 5.3. Implementacija skripti za rekonfiguraciju na FPGA gdje se pokreću dvije različite skripte ovisno o uvjetima izvršavanja.

Ovim istraživanjem i pristupom dinamičke rekonfiguracije bi se omogućilo da se programski modeli neuronskih mreža spreme u FPGA *DDR* memoriju te da se tijekom leta takvi modeli korištenjem navedenih skripti iz prethodnih slika učitaju ovisno o vanjskim poticajima (vremenskim ili senzorskim). Takav pristup bi iznimno bio koristan u primjeru rada s dronom, gdje bi se očitovala prednost jednostavne reprogramljivosti FPGA platforme bez direktnog spajanja na nju. Dakako, kroz *systemd* se pruža robustan i fleksibilan mehanizam za upravljanje ovim procesima, osiguravajući da je sustav spreman i prilagođen zahtjevima misije odmah pri pokretanju (spremanje modela na *DDR* memoriju). Također, korištenje *systemd*-a za upravljanje procesima prilikom pokretanja omogućava jednostavniju administraciju i bolju kontrolu nad sistemskim resursima, što dodatno doprinosi optimizaciji performansi cjelokupnog sustava.

6. ZAKLJUČAK

Razvoj, optimizacija i implementacija dubokih neuronskih mreža na FPGA platformama predstavlja ključni korak u omogućavanju brze i energetski efikasne obrade podataka za razne primjene, uključujući detekciju i klasifikaciju objekata. Ovaj rad bavi se detaljnom analizom i primjenom naprednih tehnika kvantizacije modela i prilagođavanja arhitekture DNN-a za efikasno pokretanje na FPGA platformama, koristeći alate kao što su Brevitas i FINN.

Kroz implementaciju na Xilinx PYNQ-Z2 platformi, kroz rad se demonstrira način na koji se konvolucijske neuronske mreže mogu uspješno optimizirati i implementirati na FPGA za zadatke detekcije i klasifikacije objekata. Analizom rezultata implementacije LeNet-5 modela na MNIST i CIFAR-10 skupovima podataka, kao i Yolov3 Tiny modela za detekciju mačaka, pasa, osoba i automobila, pokazano je da kvantizacija omogućava značajne uštede u potrošnji resursa i energije, uz zadržavanje visoke razine točnosti i brzine inferencije.

Primjena dinamičke rekonfiguracije na FPGA uređajima dodatno je omogućila prilagodbu sustava specifičnim operativnim uvjetima, što je ključno za scenarije kao što su detekcija objekata snimljenih dronom. Dinamičko rekonfiguriranje sustava omogućava fleksibilno prilagođavanje modela dubokog učenja, čime se postiže visoka operativna fleksibilnost i efikasnost, posebno u realno-vremenskim aplikacijama.

Zaključno, integracija dubokih neuronskih mreža s FPGA tehnologijom ne samo da predstavlja tehnički izazov već i priliku za stvaranje novih paradigmi u računarstvu. Ovaj rad pruža vrijedne uvide i praktične primjere kako se može ostvariti efikasna i efektivna implementacija DNN modela na FPGA, što će značajno doprinijeti budućem razvoju tehnologije i umjetne inteligencije.

LITERATURA

[1] Liang, S., Yin, S., Liu, L., Luk, W., & Wei, S. (2018). FP-BNN: Binarized Neural Network on FPGA. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, (pp. 1-10). ACM, February 2018.

[2] Sharma, H., Park, J., Suda, N., Lai, L., Chau, B., Chandra, V., & Esmaeilzadeh, H. (2016). From High-Level Deep Neural Models to FPGAs. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, (pp. 1-12). IEEE, October 2016.

[3] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2019). Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(11), (pp. 2072-2085), November 2019.

[4] Chen, M., Wei, S., & Zhang, L. (2022). An Intelligent Real-Time Object Detection System on Drones. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(11), 1234-1245, November 2022.

[5] Kriegeskorte N, Golan T. Neural network models and deep learning. Curr Biol. 2019 Apr 1;29(7):R231-R236. doi: 10.1016/j.cub.2019.02.034. PMID: 30939301.

[6] Potter, G. (2019). Neural Networks and Deep Learning. *Data-Driven Science and Engineering*. https://doi.org/10.1017/9781108380690.007.

[7] J. Stanisz, K. Lis, T. Kryjak, M. Gorgon, "Optimisation of the PointPillars network for 3D object detection in point clouds," in 2020 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA), 2020. Link

[8] "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks." Dostupno na: https://arxiv.org/abs/1809.04570

[9] "Hardware-software implementation of a DNN for 3D object detection using FINN - a demo." Dostupno na: https://ieeexplore.ieee.org/document/9556414

[10] Mittal, S. (2018). A Survey of FPGA-Based Accelerators for Convolutional Neural Networks. IEEE Transactions on Neural Networks and Learning Systems, 30(4), (pp. 1234-1249), April 2018.

[11] Shawahna, A., Sait, S. M., & El-Maleh, A. (2019). FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. IEEE Access, 7, (pp. 7823-7859), January 2019.

[12] A. Ghosh, A. Sufian, F. Sultana, A. Chakrabarti, D. De. "Fundamental Concepts of Convolutional Neural Network". Dostupno na: https://link.springer.com/chapter/10.1007/978-3-030-32644-9_36

[13] U. Michelucci. "Fundamentals of Convolutional Neural Networks". In Advanced Applied Deep Learning. Dostupno na: https://link.springer.com/chapter/10.1007/978-1-4842-4976-5_3

[14] Lecun, Yann & Bottou, Leon & Bengio, Y. & Haffner, Patrick. (1998). Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE. 86. 2278 - 2324. 10.1109/5.726791.

[15] Krizhevsky, A., Sutskever i., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. NIPS

[16] Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv.

[17] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. CVPR.

[18] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. CVPR.

[19] Adarsh, Pranav & Rathi, Pratibha & Kumar, Manoj. (2020). Yolo v3-Tiny: Object Detection and Recognition using one stage improved model. 687-694. 10.1109/ICACCS48705.2020.9074315.

[20] Eldor ibragimov & Lee, Jong-Jae & Gil, Heungbae & Lee, Jung. (2023). A Vision-Based System for Inspection of Expansion Joints in Concrete Pavement. 10.20944/preprints202305.0701.v1. https://www.researchgate.net/publication/370663153_A_Vision-Based_System_for_Inspection_of_Expansion_Joints_in_Concrete_Pavement

[21] Liu, W., et al. (2016). "SSD: Single Shot MultiBox Detector." ECCV.

[22] Ren, S., et al. (2015). "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." NIPS.

[23] He, K., Gkioxari, G., Dollár, P., and Girshick, R., “Mask R-CNN”, arXiv e-prints, 2017. doi:10.48550/arXiv.1703.06870.

[24] Krizhevsky, A. (n.d.). The CIFAR-10 and CIFAR-100 datasets. Pristupljeno [6.3. 2024] CIFAR-10 and CIFAR-100 datasets

[25] Gray, R., & Neuhoff, D. (2022). Quantization. *Encyclopedia of Mathematics*. https://www.csd.uoc.gr/~hy438/lectures/Quantization.pdf

[26] Xilinx. (n.d.). *Brevitas* [Software]. Pristupljeno [6.3. 2024] https://github.com/Xilinx/brevitas

[27] Xilinx. (n.d.). ONNX export tutorial. In *Brevitas documentation*. Pristupljeno [6.3. 2024], from https://xilinx.github.io/brevitas/tutorials/onnx_export.html

[28] "LPYolo: Low Precision Yolo for Face Detection on FPGA." Dostupno na: https://arxiv.org/abs/2207.10482

[29] Redmon, J. (n.d.). *Darknet: Yolo object detection* [Programska podrška]. Pristupljeno [7.3. 2024], from https://github.com/pjreddie/darknet/tree/master

[30] Xilinx. (n.d.). FINN. Preuzeto sa https://finn.readthedocs.io/en/latest/index.html

[31] Umuroglu, Y., & Jahre, M. (2017). Streamlined Deployment for Quantized Neural Networks. *arXiv preprint arXiv:1709.04060*. Preuzeto sa https://arxiv.org/abs/1709.04060

[32] Quentin Ducasse, Pascal Cotret, Loïc Lagadec, Rob Stewart. Benchmarking Quantized Neural Networks on FPGAs with FINN. DATE Friday Workshop on System-level Design Methods for DeepLearning on Heterogeneous Architectures, Feb 2021, Grenoble, France. hal-03085342

[33] Stewart, R., Nowlan, A., Bacchus, P., Ducasse, Q., & Komendantskaya, E. (2021). Optimising Hardware Accelerated Neural Networks with Quantisation and a Knowledge Distillation Evolutionary Algorithm. *Electronics*, 10(4), 396. https://doi.org/10.3390/electronics10040396

[34] TUL Corporation. (n.d.). *PYNQ-Z2 user manual (v1.0)*. Preuzeto s https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf [24. 6. 2024]

[35] Eldafrawy, M., Boutros, A., Yazdanshenas, S., & Betz, V. (2020). FPGA Logic Block Architectures for Efficient Deep Learning Inference. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13, 1 - 34. https://doi.org/10.1145/3393668.

[36] K. Vipin and S. A. Fahmy, "Mapping adaptive hardware systems with partial reconfiguration using CoPR for Zynq," *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Montreal, QC, Canada, 2015, pp. 1-8, doi: 10.1109/AHS.2015.7231169.

[37] Atef, Ahmed & Mohamed Mahmoud, Ahmed & Nagy, Ahmed & Gamal, Youssef & Shalash, Ahmed & Ismail, Yehea. (2017). Design guidelines for the high-speed dynamic partial reconfiguration based software defined radio implementations on Xilinx Zynq FPGA. 1-4. 10.1109/ISCAS.2017.8050456.

[38] Kizheppatt Vipin and Suhaib A. Fahmy. 2018. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. ACM Comput. Surv. 51, 4, Article 72 (July 2019), 39 pages. https://doi.org/10.1145/3193827

[39] SUSE. (2021). *Systemd Basics*. Preuzeto s https://documentation.suse.com/smart/systems-management/pdf/systemd-basics_en.pdf

SAŽETAK

Diplomski rad se bavi istraživanjem razvoja, optimizacije i implementacije dubokih neuronskih mreža na FPGA platformama. Glavni problem koji se istražuje je kako učinkovito kvantizirati i prilagoditi duboke neuronske mreže za obradu na ograničenim hardverskim resursima. Koristeći alate poput Brevitas, FINN, Vivado i Vitis HLS, provedena je kvantizacija i prilagodba konvolucijskih neuronskih mreža (LeNet-5 i Yolov3). Implementacija na Xilinx PYNQ-Z2 platformi pokazala je da ove metode omogućuju uspješnu optimizaciju za zadatke detekcije i klasifikacije objekata. Dinamička rekonfiguracija sustava omogućila je prilagodbu specifičnim operativnim uvjetima, značajno poboljšavajući energetsku efikasnost i fleksibilnost sustava.

Ključne riječi: Brevitas, FINN, konvolucijske neuronske mreže (CNN), FPGA, Vitis HLS

ABSTRACT

Development, optimization and implementation of deep neural networks on FPGA

This thesis investigates the development, optimization, and implementation of deep neural networks on FPGA platforms. The main problem addressed is how to effectively quantize and adapt deep neural networks for processing on limited hardware resources. Utilizing tools such as Brevitas, FINN, Vivado, and Vitis HLS, quantization and adaptation of convolutional neural networks (LeNet-5 and Yolov3) were performed. Implementation on the Xilinx PYNQ-Z2 platform demonstrated that these methods enable successful optimization for object detection and classification tasks. Dynamic system reconfiguration allowed adaptation to specific operational conditions, significantly improving the system's energy efficiency and flexibility.

Keywords: Brevitas, FINN, convolutional neural networks (CNN), FPGA, Vitis HLS

ŽIVOTOPIS

Davorin Miličević rođen je 2000. godine u Kiseljaku. Gimnaziju je završio u srednjoj školi Ivan Goran Kovačić u Kiseljaku s odličnim uspjehom. Nakon toga, završio je preddiplomski studij na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Tijekom studija, sudjelovao je na različitim projektima u firmi Protostar Labs, uključujući projekte s Europskom Svemirskom Agencijom (ESA) te projektima u auto industriji, gdje je stekao značajna iskustva u radu s ugradbenim uređajima (FPGA, STM32, Jetson) te programiranju i testiranju ECU-ova. Trenutno se bavi programiranjem i implementiranjem različitih algoritama i modela umjetne inteligencije na ugradbene sustave.