Implementacija modela dubokog učenja za prepoznavanje objekata na ugradbenim računalnim platformama

Pepić, Robert

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:012729

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-08-04**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni studij

IMPLEMENTACIJA MODELA DUBOKOG UČENJA ZA PREPOZNAVANJE OBJEKATA NA UGRADBENIM RAČUNALNIM PLATFORMAMA

Diplomski rad

Robert Pepić

Osijek, 2023.

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

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 14.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Robert Pepić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1236R, 08.10.2021.
OIB studenta:	36236554306
Mentor:	izv. prof. dr. sc. Emmanuel Karlo Nyarko
Sumentor:	,
Sumentor iz tvrtke:	Filip Novoselnik
Predsjednik Povjerenstva:	prof. dr. sc. Robert Cupec
Član Povjerenstva 1:	izv. prof. dr. sc. Emmanuel-Karlo Nyarko
Član Povjerenstva 2:	izv. prof. dr. sc. Damir Filko
Naslov diplomskog rada:	Implementacija modela dubokog učenja za prepoznavanje objekata na ugradbenim računalnim platformama
Znanstvena grana diplomskog rada:	**Umjetna inteligencija (zn. polje računarstvo)**
Zadatak diplomskog rada:	U radu treba odabrati dva različita ugradbena uređaja pogodna za implementaciju modela dubokog učenja (npr. Google Coral, Nvidia Jetson Nano) te usporediti njihove karakteristike. Također, treba izgraditi i testirati nekoliko različith modela dubokog učenja za detekciju i prepoznavanje objekata pomoću standardnih biblioteka (npr. Tensorflow, PyTorch). Treba prilagoditi dobivene modele dubokog učenja za rad na odabranim ugradbenim platformama te razviti programsku podršku za rad s kamerom. Konačno, treba opisati i implementirati postupak kvantizacije modela te testirati i usporediti performanse modela na različitim platformama. (Tema rezervirana za: Robert Pepić) (Sumentor iz tvrtke: Filip Novoselnik, Protostar Labs d.o.o., Željeznička 10/A, 31551 Belišće, Hrvatska)
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	14.09.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 30.09.2023.

Ime i prezime studenta:	Robert Pepić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1236R, 08.10.2021.
Turnitin podudaranje [%]:	3

Ovom izjavom izjavljujem da je rad pod nazivom**: Implementacija modela dubokog učenja za prepoznavanje objekata na ugradbenim računalnim platformama**

izrađen pod vodstvom mentora izv. prof. dr. sc. Emmanuel Karlo Nyarko

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 studenta:

SADRŽAJ

1. UVOD 1

1.1. Zadatak diplomskog rada 1

2. UGRADBENE RAČUNALNE PLATFORME I RAČUNALNI VID 2

3. TRENIRANJE MODELA ZA DETEKCIJU OBJEKATA 5

3.1. Zahtjevi za implementaciju 5

3.2. EfficientDet model za detekciju objekata 5

3.3. Priprema skupa podataka 6

3.4. Uspostavljanje virtualnog okruženja za treniranje 8

3.5. Treniranje modela 9

3.6. Evaluacija modela 14

3.7. Kvantizacija modela i konverzija u TF Lite format 15

4. IMPLEMENTACIJA MODELA NA CORAL DEV BOARD 17

4.1. Coral Dev Board 17

4.2. Uspostavljanje Coral-a za rad 17

4.3. Zahtjevi za izvođenje modela na Edge TPU koprocesoru 18

4.4. Kompiliranje modela za izvođenje na Edge TPU 19

4.5. Testiranje performansi modela na Coral-u 20

4.6. Testiranje rada modela na Coral-u korištenjem USB kamere 21

5. IMPLEMENTACIJA MODELA NA JETSON NANO 26

5.1. Jetson Nano Developer Kit 26

5.2. Uspostavljanje Jetson-a za rad 26

5.3. Testiranje performansi TF Lite modela na Jetson-u 26

5.4. Ponovno treniranje modela 29

5.5. Konverzija modela u TensorRT format 30

5.6. Testiranje performansi modela na Jetson-u 31

5.7. Testiranje rada modela na Jetson-u korištenjem USB kamere 31

6. ZAKLJUČAK **40**

LITERATURA **41**

1. UVOD

Umjetne inteligencija (engl. *Artificial Intelligence, AI*), u posljednjem desetljeću, nedvojbeno je doživjela velik napredak i ubrzan razvoj, te je javnosti dostupnija nego ikad do sada. S napretkom u procesorskoj snazi računala, pohrani podataka i algoritmima strojnog učenja, umjetna inteligencija uspjela je dosegnuti nove visine u smislu svojih mogućnosti i svoje primjene. Uspon AI tehnologije revolucionirao je mnoge industrije, uključujući zdravstvo, financije, transport i proizvodnju [1]. Omogućio je strojevima obavljanje zadataka, za koje se prije mislilo da ih može obavljati samo čovjek, kao što su prepoznavanje slika, obrada prirodnog jezika ili donošenje odluka. Jedno od područja koje se zahvaljujući ovom usponu također počelo razvijati jest ugrađena umjetna inteligencija (engl. *Embedded artificial intelligence*), koje se bavi implementacijom umjetne inteligencije na ugradbene računalne platforme, male računalne uređaje koji su potom integrirani u veće sustave. Ugrađena umjetna inteligencija tim sustavima tada omogućuje izvođenje složenih zadataka i to bez potrebe za povezivanjem s vanjskim računalnim sustavom. Podaci se obrađuju kod samog izvora, što donosi određene prednosti kao što su primjerice smanjeno kašnjenje i veća sigurnost podataka. Ovakvi sustavi imaju sve veću primjenu u područjima kao što su robotika, sigurnosni sustavi, autonomna vozila i industrijska automatizacija [2]. Ovaj rad fokusirati će se na ugradbene računalne platforme, odnosno na implementaciju modela dubokog učenja za prepoznavanje objekata na iste.

1.1. Zadatak diplomskog rada

U okviru rada odabrane su dvije ugradbene računalne platforme koje je potrebno osposobiti za rad: Google Coral Dev Board [3] i NVIDIA Jetson Nano [4]. Proučiti će se njihove karakteristike, mogućnosti, ograničenja te načini na koji se one koriste. Zatim će biti potrebno istrenirati nekoliko modela dubokog učenja za detekciju objekata. Razmotriti će se modeli koji će biti korišteni u okviru rada. Ukratko će se objasniti princip njihovog rada te metrike kojima ih se vrednuje. Samo treniranje provoditi će se korištenjem programskog jezika Python, zajedno s odgovarajućim bibliotekama odnosno alatima za strojno učenje. Objasnit će se što je to kvantizacija modela te će se dobivene modele prilagoditi za rad na odabranim ugradbenim platformama. Prilagođeni modeli zatim će biti implementirani na odabrane platforme, te će njihov rad biti testiran. Testiranje modela biti će obavljeno pomoću USB kamere te će biti potrebno razviti programsku podršku za rad s istom. Na kraju će se usporediti i prokomentirati rezultati u pogledu performansi modela na odabranim ugradbenima platformama.

2. UGRADBENE RAČUNALNE PLATFORME I RAČUNALNI VID

Područje računalnog vida vrlo je široko i obuhvaća razne zadatke, kao što su primjerice klasifikacija slike, detekcija i prepoznavanje objekata na slici te segmentacija objekata. Ovisno o zadatku kojeg se želi riješiti i zadanim zahtjevima, postoje različiti pristupi kojima se može doći do zadovoljavajućih rješenja. Određeni zadaci, kao primjerice detekcija defektnih proizvoda na proizvodnoj traci, ovisno o proizvodu, mogu zahtijevati visoku točnost i preciznost. U slučajevima gdje su zahtjevi strogi, gdje se ne toleriraju pogreške, potrebni su veliki modeli koji zahtijevaju značajnu računalnu snagu i memoriju. Takvi modeli imaju jako puno parametara i trenirani su na velikim skupovima podataka.

Primjer jednog takvog modela je SSD (engl. *Single Shot Multi-Box Detector*) [5] model za detekciju objekata. SSD je baziran na konvolucijskoj neuronskoj mreži koja generira skup graničnih okvira (engl. *Bounding Box*) i vjerojatnosti instanca klasi unutar njih. Iz tog skupa se zatim, korištenjem tehnike koja se zove *Non-maximum suppression* (NMS), eliminiraju sve duplicirane detekcije istog objekta. Kao rezultat tada ostane samo onaj okvir koji najbolje opisuje položaj objekta na slici. Još jedan primjer modela sličnih sposobnosti bio bi Darknet-ov YOLO (engl. *You Only Look Once*) [6] model za detekciju objekata, treniran na COCO (engl. *Common Objects in Context*) [7] skupu podataka koji sadrži preko 330 tisuća označenih slika podijeljenih u 80 različitih klasa. Oba modela sposobna su za rad u stvarnom vremenu, pritom dajući zadovoljavajuće rezultate u raznim primjenama. U istraživanju [8] objavljenom od strane *BMC Medical Informatics and Decision Making*, uspoređeni su prethodno navedeni modeli, zajedno s još jednim popularnim modelom za detekciju objekata – RetinaNet [9]. Modeli su uspoređeni na zadatku detekcije i klasifikacije tableta u stvarnom vremenu. Na temelju dobivenih rezultata ispostavilo se da je YOLO daleko najbrži s brzinom obrade od 51 FPS (engl. *Frames Per Second*), no ujedno i najmanjom srednjom prosječnom preciznošću od 80.7% mAP (engl. Mean Average Precision). S druge strane, RetinaNet imao je najveći mAP od 82.9%, ali i najmanju brzinu obrade od 17 FPS. SSD imao je gotovo isti mAP kao i RetinaNet u iznosu od 82.7%, no naspram njega imao je skoro dvostruku brzinu obrade od 32 FPS. Dobiveni rezultati navedeni su kako bi se pokazalo da određeni modeli, koji obavljaju isti zadatak, nisu jednako prikladni u svim situacijama. Primjerice, YOLO bi bio dobar izbor ukoliko se zahtjeva rad u stvarnom vremenu, a zahtjevi točnosti i preciznosti nisu previše strogi. Nasuprot tome, RetinaNet bi bio pogodniji kod zadataka u kojima vremenski zahtjevi nisu strogi, no koji zahtijevaju visoku razinu točnosti i preciznosti, dok bi SSD mogao biti korišten u situacijama gdje je i jedno i drugo potrebno. Naravno, brzina obrade i performanse nisu jedini faktori koji utječu na odabir modela.

Faktori kao što su veličina samog modela i računalna snaga potrebna za njegovo izvođenje također su vrlo bitni, posebice u kontekstu ovoga rada. Kao što je već spomenuto, ugradbene računalne platforme mali su uređaji ograničenih resursa. Prethodno navedeni modeli relativno su veliki te bi njihova brzina obrade bila ograničena ukoliko bi ih se implementiralo na platforme koje nemaju dovoljno računalnih resursa za njihovo pravilno izvođenje. Ta ograničenja moguće je vidjeti u rezultatima istraživanja [10] u kojem su se testirale performanse YOLOv3 modela na uređajima kao što su NVIDIA Jetson Nano, NVIDIA Jetson Xavier NX [11] i Raspberry Pi 4B [12] s priključenim Intel Neural Compute Stick 2 [13]. Navedeni uređaji testirani su na snimkama veličina 768x436 i 1920x1080. Jetson Nano na tim je snimkama ostvario brzine od 1.7 i 1.6 FPS, Raspberry Pi postigao je 2.5 FPS na obje snimke, dok je Jetson Xavier dosegnuo brzine od 6.1 i 5.9 FPS. Može se primijetiti kako su te brzine znatno manje od one postignute u prethodno spomenutom istraživanju (51 FPS), provedenom na stolnom računalu. Međutim, u istom istraživanju testirane su i performanse YOLOv3-Tiny modela, varijante standardnog YOLOv3. YOLOv3-Tiny dizajniran je da bude manji i brži u usporedbi s originalnim modelom, što ga čini pogodnijim za ugradbene uređaje. Implementacijom navedenog modela te ponovnim testiranjem na istim snimkama Jetson Nano ostvario je brzinu obrade od 6.8 i 6.6 FPS, Raspberry Pi 18.8 i 19 FPS, a Jetson Xavier čak 41.1 i 35.6 FPS. Međutim, manja složenost modela negativno je utjecala na njegovu preciznost. U nekim slučajevima rezultati su bili gotovo 40% lošiji.

Navedenim rezultatima htjelo se ukazati na jedan važan korak kod implementacije modela na ugradbene uređaje – optimizaciju modela. Postoje razne tehnike optimizacije modela za rad na ugradbenim platformama, koje smanjuju njegovu složenost i memorijske zahtjeve. Od jednostavnijih, kao što su smanjivanje veličine ulaza modela ili korištenje manjeg broja slojeva u modelu, do složenijih, kao što je kvantizacija modela, o kojoj će više bit rečeno u narednim poglavljima. Tim tehnikama model se prilagođava platformi na koju ga se namjerava implementirati. Pri tome je izazov model učiniti efikasnijim u pogledu računalnih resursa koje on zahtjeva, no time što manje negativno utjecati na kvalitetu njegovog rada. Postoji nekolicina modela za detekciju objekata, kreiranih specifično za primjenu na ugradbenim platformama, koji su otvorenog koda i besplatno dostupni za korištenje. Neki primjeri takvih modela, koji su trenutno popularni, bili bi EfficientDet [14], SqueezeDet [15] i prethodno navedeni YOLOv3-Tiny. Takvi modeli često dolaze unaprijed istrenirani na velikim skupovima podataka i imaju sposobnost detekcije objekata iz većeg broja različitih klasa. Treniranje takvih modela od nule na velikom skupu podataka može biti skupo i dugotrajno, zahtijevajući snažan hardver. Stoga se unaprijed trenirani modeli često koriste kao početna točka za kreiranje vlastitih. Te modele moguće je

prilagoditi manjem skupu podataka kako bi obavljali drugi, srodni zadatak. Navedeno se postiže korištenjem tehnike koja se zove prijenosno učenje (engl. *Transfer learning*) [16]. Ideja iza prijenosnog učenja je da se znanje stečeno rješavanjem jednog problema ponovno iskoristi za rješavanje drugog, sličnog problema. Podrazumijeva ponovno treniranje modela korištenjem novog skupa podataka. Glavne prednosti ove tehnike su te što ona skraćuje vrijeme potrebno za treniranje modela te što zahtjeva manje podataka za isto. Iz tog razloga ova tehnika koristiti će se i tokom izrade ovog rada za treniranje vlastitih modela, a način njenog rada i implementacije bit će detaljnije objašnjen u narednom poglavlju.

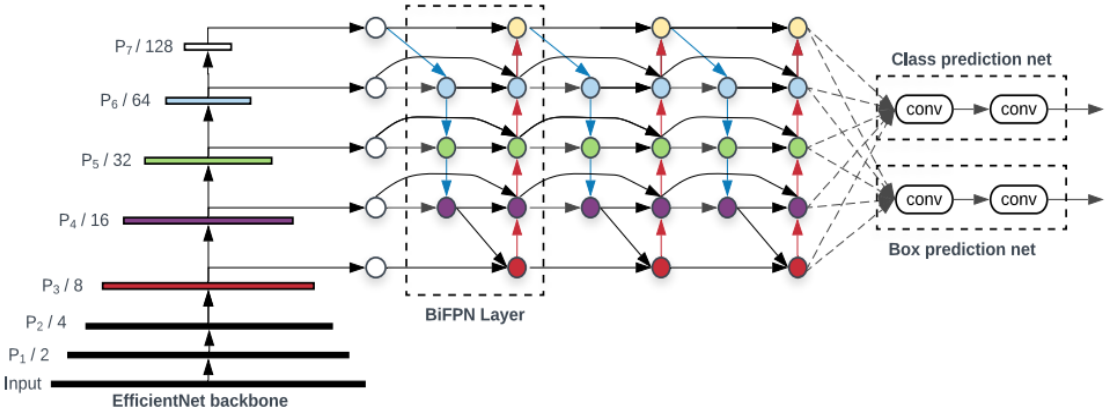
3. TRENIRANJE MODELA ZA DETEKCIJU OBJEKATA

3.1. Zahtjevi za implementaciju

Odabrane platforme imaju određene zahtjeve, koje modeli moraju ispuniti, kako bi se u potpunosti mogle iskoristiti njihove mogućnosti. U slučaju Coral Dev Bord-a model mora biti treniran korištenjem TensorFlow-a [17], zatim kvantiziran i konvertiran u TensorFlow Lite [18] format te kompiliran za izvođenje na uređaju. Jetson Nano s druge strane podržava različite formate modela, no zahtjeva da se model konvertira u TensorRT format. Oba postupka biti će detaljno objašnjena u narednim poglavljima. Za samo treniranje modela koristiti će se TensorFlow Lite Model Maker biblioteka jer sadrži niz korisnih značajki i pojednostavljuje proces treniranja modela. Jedno od njenih važnih svojstava je to što treniranje obavlja korištenjem tehnike prijenosa učenja te što za isto na raspolaganje nudi niz različitih, unaprijed istreniranih modela koji se mogu koristiti kao početna točka. Podržava treniranje modela za razne zadatke strojnog učenja, kao što su klasifikacija slike, klasifikacija teksta, klasifikacija zvuka i detekcija objekata. Za zadatak detekcije objekata na raspolaganje nudi EfficientDet i EfficientDet-Lite modele, koji će biti korišteni u okviru rada. Uz to, omogućuje pohranu modela u standardnom i u TF Lite formatu.

3.2. EfficientDet model za detekciju objekata

Modeli za detekciju objekata vremenom su postali sve složeniji, točniji i precizniji, no time ujedno veći i zahtjevniji u pogledu računalne snage potrebne za njihovo izvođenje. Prethodno spomenuti EfficientDet model kreiran je s namjerom postizanja balansiranog odnosa točnosti i efikasnosti pri detekciji objekata. Temelji se na EfficientNet [19] konvolucijskoj neuronskoj mreži za klasifikaciju slika, čiju arhitekturu proširuje dodatnim komponentama koje mu omogućuju sposobnost detekcije objekata.



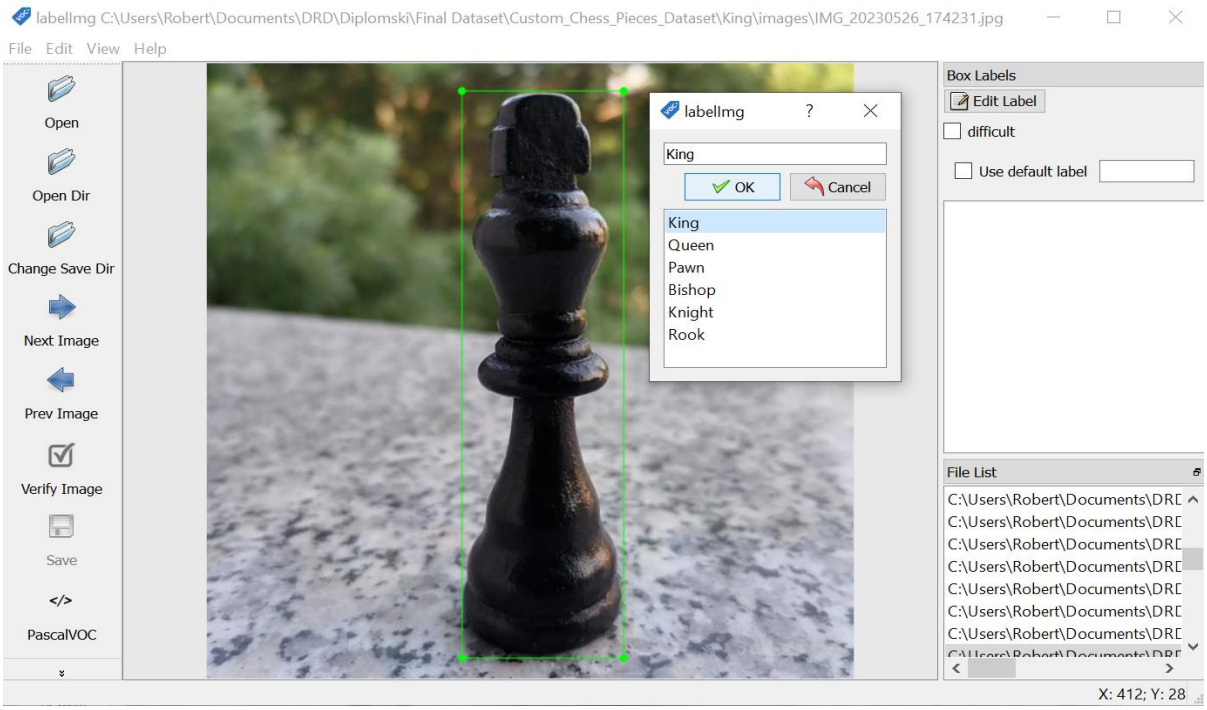
Slika 3.1 Arhitektura EfficientDet modela za detekciju objekata ([14])

Arhitekturu EfficientDet modela, koji se sastoji od tri ključna dijela, moguće je vidjeti na slici 3.1. Početni dio EfficientDet-a, takozvanu okosnicu (engl. Backbone), čini sami EficientNet model. On je odgovoran za izdvajanje značajki iz ulazne slike, što radi pomoću niza konvolucijskih slojeva. Konvolucijski slojevi dizajnirani su za izdvajanje uzoraka ili značajki iz ulaznih podataka. Nizanjem konvolucijskih slojeva, mreža može naučiti sve složenije i apstraktnije prikaze ulaza. Izlazi pojedinog sloja nazivaju se mape značajki (engl. *Feature maps*). Svaki sljedeći sloj obrađuje mape značajki koje proizvodi prethodni sloj, omogućujući mreži prepoznavanje sve složenijih značajki. Mape značajki posljednjih pet slojeva, koji su na prethodnoj slici označeni s oznakama od P_3 do P_7 , tada su predane drugom dijelu EfficientDet modela, takozvanoj mreži značajki (engl. Feature network). Ona se sastoji od niza slojeva dvosmjernih piramida značajki (engl. *Bi-directional Feature Pyramid Network - BiFPN*) koje provode spajanje značajki (engl. Feature fusion) različitih razina apstrakcije. Ovaj proces spajanja modelu omogućuje učinkovito prikupljanje i obradu informacija različitih razmjera, odnosno sposobnost otkrivanja objekata različitih veličina. Spojene značajke zatim se šalju posljednjem dijelu mreže, zaduženom za provođenje klasifikacije objekta i predviđanje graničnih okvira.

3.3. Priprema skupa podataka

Prije nego li se može započeti s treniranjem modela, za isto je potreban skup podataka. Kod modela za detekciju objekata taj skup podataka podrazumijeva skup slika na kojima se nalaze objekti od interesa i skup tekstualnih datoteka u kojima su pohranjene informacije o tim objektima. Svaka slika ima odgovarajuću datoteku u kojoj su pohranjene informacije o objektima unutar nje, kao što su položaj i klasa pojedinog objekta. Postoje razni formati u kojima se te informacije mogu pohraniti. Neki od trenutačno popularnih formata su YOLO, COCO i Pascal VOC formati [20]. Oni se međusobno razlikuju po načinu na koji reprezentiraju položaje objekata na slici, ali i po formatu datoteke u koju pohranjuju te informacije. U YOLO formatu položaj objekata definiran je pomoću točke koja predstavlja središte graničnog okvira koji opisuje promatrani objekta te njegovom visinom i širinom, a informacije o položaju i klasi pojedinih objekata pohranjuju se kao redovi u standardnoj tekstualnoj datoteci. U COCO formatu položaj objekata definira se pomoću točke koja predstavlja gornji lijevi vrh graničnog okvira promatranog objekta te njegovom visinom i širinom, a informacije se pohranjuju u JSON formatu. Pascal VOC format položaj objekata definira pomoću dvije točke koje predstavljaju gornji lijevi i donji desni vrh graničnog okvira, a informacije se pohranjuju u XML formatu. Postupak definiranja graničnih okvira oko objekata od interesa, odnosno definiranje njihovog položaja na slici, te dodjeljivanja klasa istima naziva se označavanje ili anotacija slika. Za potrebe ovog rada kreiran je vlastiti skup podataka. Kao objekti

od interesa odabrane su šahovske figure te je prikupljeno tisuću slika figura koje pripadaju istom šahovskom setu. Slike su zatim anotirane korištenjem grafičkog alata za anotaciju slika LabelImg [21] koji omogućuje jednostavno označavanje slika i automatsko generiranje datoteka s anotacijama u YOLO ili Pascal VOC formatu. Na slikama u nastavku moguće je vidjeti postupak označavanja objekata na slici unutar LabelImg aplikacije, te primjer sadržaja generirane datoteke s anotacijama u Pascal VOC formatu.



Slika 3.2 Označavanje slika unutar aplikacije LabelImg

```
<annotation>
  <folder>images</folder>
  <filename>IMG_20230526_174231.jpg</filename>
  <path>C:\Users\Robert\Documents\DRD\Diplomski\Chess_Pieces_Dataset\King\images\IMG_20230526_174231.jpg</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>640</width>
    <height>640</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>King</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>252</xmin>
      <ymin>28</ymin>
      <xmax>412</xmax>
      <ymax>618</ymax>
    </bndbox>
  </object>
</annotation>
```

Slika 3.3 Sadržaj generirane datoteke s anotacijama u Pascal VOC formatu

3.4. Uspostavljanje virtualnog okruženja za treniranje

Treniranje model provoditi će se korištenjem već spomenute TensorFlow Lite Model Maker biblioteke i programskog jezika Python. Za potrebe treniranja kreirati će se virtualno Python okruženje. Virtualno okruženja predstavlja izolirano okruženje s vlastitom verzijom Python-a i vlastitim paketima. Paketi instalirani unutar virtualnog okruženja neće utjecati na pakete sistemskog Pythona niti na pakete unutar drugih virtualnih okruženja. Time se sprječavaju potencijalni konflikti između različitih verzija paketa. Za kreiranje virtualnog okruženja koristiti će se venv modul [22]. Python verzija 3.9, koja će biti korištena u okviru ovog rada, sadrži venv modul kao dio standardne biblioteke te ga nije potrebno posebno instalirati. U nastavku je prikazano pozivanje naredbe za kreiranje virtualnog okruženja.

```
robert@robert-Legion-5-Pro-16ACH6:~$ python3 -m venv diplomski_env
```

Listing 3.1 Stvaranje virtualnog okruženja

Izvršenjem prethodne naredbe kreirano je virtualno okruženje naziva *diplomsi_env,* no kako bi se to novo kreirano okruženje pokrenulo potrebno je pozvati sljedeću naredbu.

```
robert@robert-Legion-5-Pro-16ACH6:~$ source diplomski_env/bin/activate
(diplomski_env) robert@robert-Legion-5-Pro-16ACH6:~$
```

Listing 3.2 Pokretanje virtualnog okruženja

Nakon izvršenja naredbe virtualno okruženje je aktivirano, što se može vidjeti pojavom naziva okruženja (diplomski_env) unutar terminala. Svi Python paketi koji budu instalirani nakon aktivacije virtualnog okruženja pripadati će tom okruženju. Prvi paket koji će biti instaliran unutar okruženja je TensorFlow Lite Model Maker. Instalacija će se pokrenuti pozivom sljedeće naredbe.

```
(diplomski_env) robert@robert-Legion-5-Pro-16ACH6:~$ pip install tflite-model-maker
```

Listing 3.3 Instalacija TensorFlow Lite Model Maker paketa

Model Maker će automatski instalirati i sve pakete koji su mu potrebni za ispravan rad, kao što su primjerice numpy, TensorFlow i TF Lite. Nakon toga instalirati će se JupyterLab razvojno okruženje [23] , unutar kojega će se pisati kod i provoditi treniranje modela. Pozivom naredbe u nastavku pokrenuti će se instalacija JupyterLab-a.

```
(diplomski_env) robert@robert-Legion-5-Pro-16ACH6:~$ pip install jupyterlab
```

Listing 3.4 Instalacija JupyterLab razvojnog okruženja

Završetkom instalacije uspješno je uspostavljeno okruženje za treniranje. Za pokretanje JupyterLab razvojnog okruženja potrebno je pozvati naredbu sa slike 3.8, nakon čega se može započeti pisanje koda odnosno s treniranje modela.

```
(diplomski_env) robert@robert-Legion-5-Pro-16ACH6:~$ jupyter-lab
```

Listing 3.5 Pokretanje JupyterLab razvojnog okruženja

3.5. Treniranje modela

Za početak je potrebno importirati nekoliko modula iz TF Lite Model Maker biblioteke. Prvi od njih je *model_spec* modul. On sadrži specifikacije već istreniranih modela za razne zadatke strojnog učenja. Na temelju tih specifikacija generiraju se modeli koji se koriste kao početna točka za treniranije prijenosnim učenjem. Zatim je potrebno importirat *object_detector* modul, koji je vrlo važan jer sadrži *DataLoader* i *ObjectDetector* klase. *DataLoader* je zadužen za učitavanje odnosno iteraciju kroz podatke, dok *ObjectDetector*, na temelju predanih specifikacija modela i *DataDoader*-a, omogućuje treniranje modela za detekciju objekata i njegovu evaluaciju. Posljednja stvar koju je potrebno importirati je *ExportFormat* klasa koja će se koristiti prilikom pohrane modela i njegove konverziju u TF Lite format. Izvršavanjem sljedećeg bloka koda importirati će se navedeni paketi potrebni za treniranje modela.

Linija	Kod
1:	**from** tflite_model_maker **import** model_spec
2:	**from** tflite_model_maker **import** object_detector
3:	**from** tflite_model_maker.config **import** ExportFormat

Listing 3.6 Importiranje paketa

Nakon toga moguće je započeti s pisanjem koda za treniranje modela. Prvo je potrebno učitati skup podataka. Za potrebe treniranja podaci su nasumično podijeljeni na skupove za treniranje, testiranje i validaciju. Skupa za treniranje sadrži osamsto slika u direktoriju *images* te njima odgovarajuće datoteke s anotacijama u *annotations* direktoriju. Skupovi za testiranje i validaciju strukturirani su isti način i svaki sadrži po sto slika i sto odgovarajućih datoteka s anotacijama. Za sva tri skupa kreirati će se posebni *DataLoader* objekt. Njih će se kreirati pomoću *from_pascal_voc()* metode *DataLoader*-a. Navedena metoda kao obavezne argumente zahtjeva putanje do direktorija sa slikama i anotacijama, te *label_map* argument koji sadrži nazive klasa

objekata u skupu podataka. Nazivi klasa moraju odgovarati onima korištenim tokom anotacije slika. Argument *label_map* može biti u obliku liste ili u obliku rječnika (engl. *Dictionary*) kojem su ključevi cijeli brojevi od jedan nadalje (nula je rezervirana za pozadinske slike odnosno slike bez objekata), a vrijednosti nazivi klasa objekata. U okviru rada to su nazivi vrsta šahovskih figura na engleskom jeziku (King, Queen, Pawn, Rook, Knight i Bishop). Stvaranje *DataLoader* objekata za treniranje, testiranje i validaciju postiže se izvršavanjem sljedećeg bloka koda.

Linija	**Kod**
1:	train_images_dir = './chess-pieces-dataset/train/images'
2:	train_annotations_dir = './chess-pieces-dataset/train/annotations'
3:	val_images_dir = './chess-pieces-dataset/validation/images'
4:	val_annotations_dir = './chess-pieces-dataset/validation/annotations'
5:	test_images_dir = './chess-pieces-dataset/test/images'
6:	test_annotations_dir = './chess-pieces-dataset/test/annotations'
7:	
8:	label_map = {1 : 'King', 2 : 'Queen', 3 : 'Pawn', 4 : 'Bishop', 5 : 'Knight', 6 : 'Rook'}
9:	
10:	train_data = object_detector.DataLoader.from_pascal_voc(train_images_dir,
11:	train_annotations_dir, label_map=label_map)
12:	validation_data = object_detector.DataLoader.from_pascal_voc(val_images_dir,
13:	val_annotations_dir, label_map=label_map)
14:	test_data = object_detector.DataLoader.from_pascal_voc(test_images_dir,
15:	test_annotations_dir, label_map=label_map)

Listing 3.7 Stvaranje DataLoader objekata za treniranje, testiranje i validaciju

Slijedeći korak je odabir specifikacije modela. Odabir specifikacije obavlja se pomoću *model_spec* modula koji, kao što je već rečeno, sadrži specifikacije već istreniranih modela za razne zadatke strojnog učenja, na temelju kojih se generiraju modeli koji služe kao početne točke za prijenosno učenje. Što se tiče detekcije objekata, *model_spec* na raspolaganje daje prethodno spomenuti EfficientDet-Lite model koji dolazi u pet različitih varijanti, numeriranih od nula do

četiri. One se međusobno razlikuju po složenosti arhitekture i broju parametara, a time automatski i veličinom samog modela (nula označava najmanji model). U okviru rada koristiti će se *efficientdet_lite0* [24] , *efficientdet_lite1* [25] i *efficientdet_lite2* [26] modeli, čije se karakteristike mogu vidjeti u nastavku.

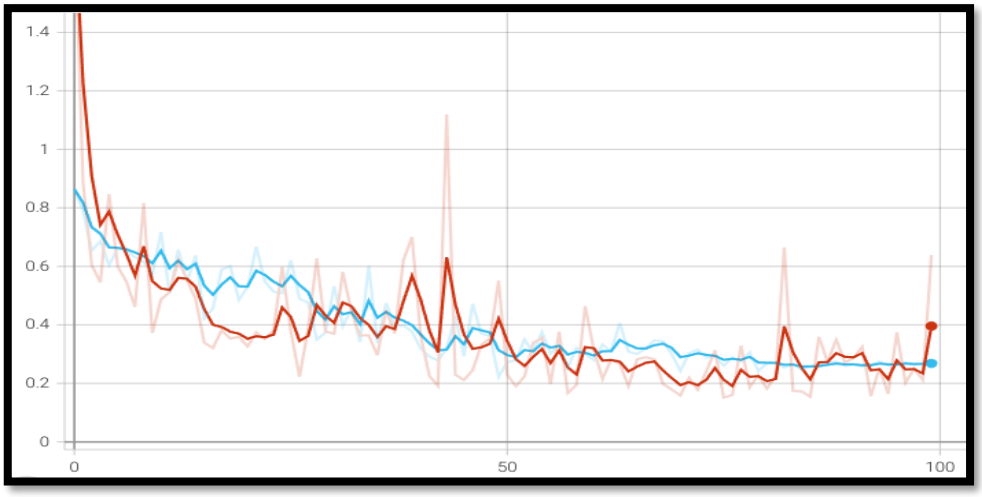
Model	**Dimenzije ulaza**	**Veličina(MB)**	**Latencija(ms) (Google Pixel 4 – CPU 4 threads)**	**mAP (COCO 2017)**
efficientdet_lite0	320x320	4.4	37	25.69%
efficientdet_lite1	384x384	5.8	49	30.55%
efficientdet_lite2	448x448	7.2	69	33.97%

Tablica 3.1 Karakteristike i performanse korištenih modela

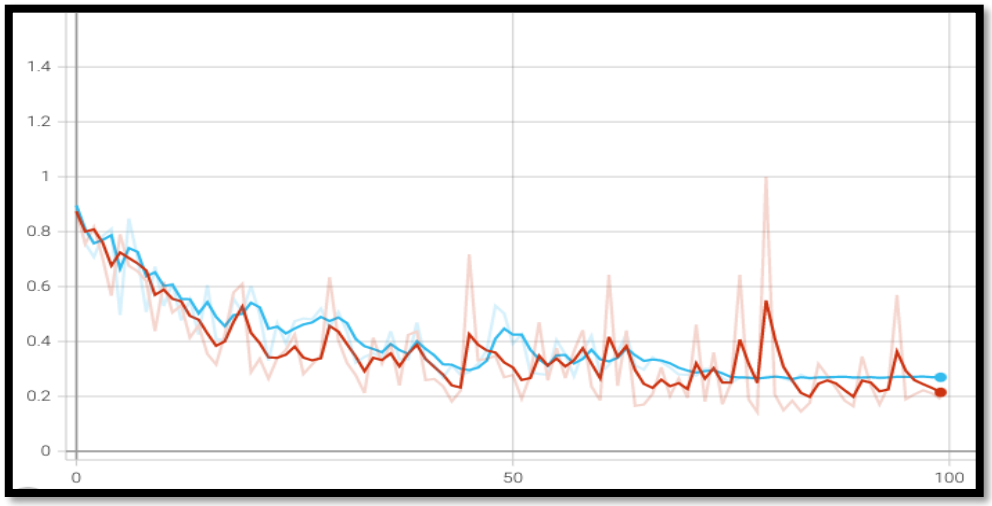
Sva tri modela biti će istrenirana na istom skupu podataka te kasnije implementirana na Coral Dev Board i Jetson Nano, nakon čega će se usporediti performanse modela na njima. Za treniranje je potrebna navedena *ObjectDetector* klasa, točnije njena *create()* metoda. Ona je ustvari ta koja će na temelju predane joj specifikacije generirati model koji predstavlja početnu točku za prijenosno učenje te započeti treniranje modela. Međutim, prije toga unutar same specifikacije modela potrebno je definirati *model_dir* parametar. On predstavlja putanju do direktorija unutar kojega će se pohranjivati kontrolne točke (engl. *Checkpoint*) koje će biti korištene za prikaz i praćenje napretka tokom treniranja. Uz njega će se, također unutar specifikacije, postaviti i parametar *save_freq* koji predstavlja frekvenciju kojom će se spremati kontrolne točke. Spremanje će se obavljati nakon svake epohe (engl. Epoch), odnosno svaki puta kada model prođe kroz sve podatke iz skupa za treniranje. Uz kreirane *DataLoader* objekte zadužene za iteraciju kroz podatke za treniranje i validaciju, te specifikaciju modela, *create()* metodi je kao argumente potrebno predati broj epoha i veličinu serije (engl. *Batch size*). Veličina serije predstavlja broj slika koji će proći kroz mrežu prije nego što se ažuriraju njene težine. Kako bi se ažurirale težine cijelog modela, a ne samo one koje se nalaze u posljednjem sloju, kako je postavljeno prema zadanim postavkama, funkciji će kao argument *train_whole_model* biti predana vrijednost *True*. Treniranjem cijelog modela rezultirati će boljim modelom, no vrijeme treniranja će zato biti duže. Samo treniranje provoditi će se na NVIDIA RTX 3050 Ti grafičkoj kartici za laptop uz instalirani CUDA Toolkit 11.2 [27]. U nastavku je prikazan kod kojim se započinje treniranje *efficientdet_lite0* modela od sto epoha i veličinom serije od dvije slike. Treniranje ostalih modela provodi se analogno.

Linija	Kod
1:	spec = model_spec.get('efficientdet_lite0')
2:	spec.config.model_dir ='./efficientDet-lite0-chess-piece-detection-model-checkpoints'
3:	spec.config.save_freq = 'epoch'
4:	
5:	model = object_detector.create(train_data = train_data,
6:	model_spec = spec,
7:	validation_data = validation_data,
8:	epochs = 100,
9:	batch_size = 2,
10:	train_whole_model = **True**)
Listing 3.8 Odabir specifikacije i treniranje modela	

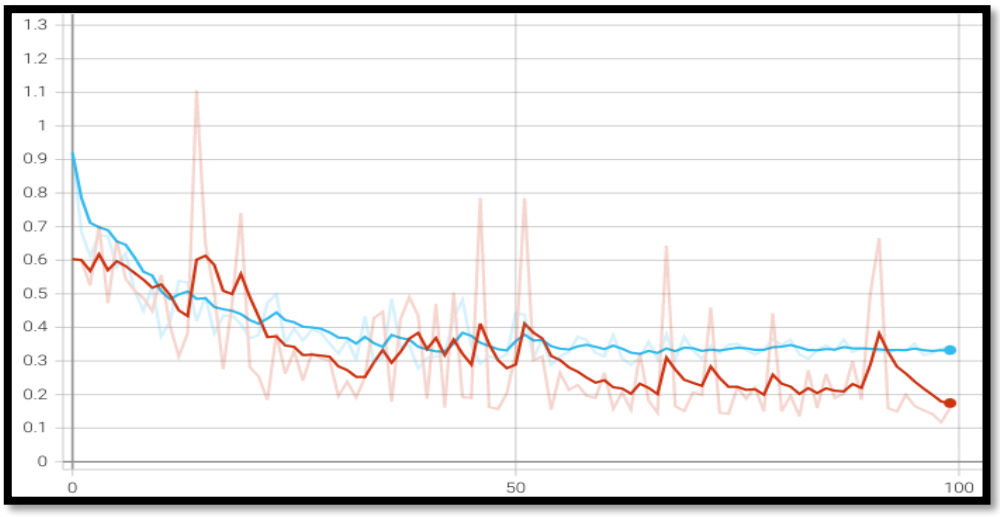
Za vizualizaciju procesa treniranja korišten je TensorFlow-ov alat TensorBoad [28]. Njemu se predaje putanja do direktorija unutar kojeg se nalaze kontrolne točke prikupljene tokom treniranja. Na temelju njih on zatim generira razne grafove vezene uz trenirani model. Grafove je moguće pratiti nakon, ali i tijekom treniranja. Od posebnog značaja je graf koji prikazuje pad vrijednosti kriterijske funkcije (engl. *Loss function*) po epohama treniranja. Kriterijska funkcija nam govori koliko model dobro predviđa, odnosno mjeri razliku između predviđenih rezultata i stvarnih vrijednosti. Tijekom treniranja, parametri modela se ažuriraju s ciljem smanjenja vrijednost kriterijske funkcije. U slučaju korištenih EfficientDet Lite modela, riječ je o focal loss [9] kriterijskoj funkciji. U nastavku su prikazani grafovi pada kriterijske funkcije preko sto epoha za sva tri istrenirana modela. Plavom bojom označena je kriterijska funkcija temeljena na validacijskom skupu podataka, a crvenom bojom kriterijska funkcija temeljena na skupu podataka za treniranje. Na temelju njihovog odnosa moguće je utvrditi koliko je dobro model istreniran. Općenito, postoje tri slučaja. U prvom slučaju, kada su njihove vrijednosti visoke, model nije dovoljno treniran (engl. *Underfitting*). U drugom slučaju, kada su njihove vrijednosti niske i konvergiraju, model je optimalno istreniran. U trećem slučaju, kada vrijednost validacijske kriterijske funkcije počne rast, a trening kriterijske funkcije nastavi padati, model se previše prilagodio skupu podataka za treniranje (engl. *Overfitting*).



Slika 3.4 Trening (crvena) i validacijska (plava) kriterijska funkcija efficientDet-Lite0 modela



Slika 3.5 Trening (crvena) i validacijska (plava) kriterijska funkcija efficientDet-Lite1 modela



Slika 3.6 Trening (crvena) i validacijska (plava) kriterijska funkcija efficientDet-Lite2 modela

3.6. Evaluacija modela

Istrenirane modele sada je potrebno evaluirati. Evaluacija se vrši na skupu podataka za testiranje. Za nju će se koristiti *evaluate()* funkcija samog modela te prethodno kreirani *test_data DataLoader* za iteraciju kroz skup podataka za testiranje. Funkcija kao rezultat vraća razne metrike, od kojih su posebno značajne prosječne preciznosti (AP) po klasi objekata unutar testnog skupa podataka te srednja prosječna preciznost (mAP). U nastavku je prikazana linija koda kojom se započinje evaluacija modela te rezultati evaluacije sva tri istrenirana modela. Rezultati evaluacije pokazali su kako je srednja prosječna preciznost sva tri modela veća od osamdeset i četiri posto, što je dovoljno za potrebe ovoga rada.

Može se primijetiti kako je rastom složenosti modela rasla i njihova preciznost, što je i bilo za očekivat. Treba napomenuti kako se dobiveni rezultati evaluacije odnose na modele u standardnom TensorFlow formatu. U sljedećem koraku modeli će biti kvantizirani i konvertirani u TF Lite format kako bi bili prikladni za implementaciju na ugradbene platforme. Nakon toga, sva tri modela će se ponovno evaluirati te će se usporediti rezultati evaluacije prije odnosno poslije kvantizacije i konverzije modela u TF Lite format.

Linija	**Kod**
1:	model.evaluate(test_data, batch_size = 2)

Listing 3.9 Evaluacija istreniranih modela

	efficientDet-Lite0	efficientDet-Lite1	efficientDet-Lite2
AP_King	0.7959	0.8032	0.8615
AP_Queen	0.8247	0.8686	0.9121
AP_Pawn	0.8330	0.8166	0.9155
AP_Bishop	0.8234	0.8701	0.8835
AP_Knight	0.8992	0.9317	0.9145
AP_Rook	0.8872	0.9158	0.8737
mAP	0.8439	0.8677	0.8935

Tablica 3.2 Rezultati evaluacije istreniranih modela

3.7. Kvantizacija modela i konverzija u TF Lite format

Kvantizacija modela [29] je tehnika kojom se težine modela pretvaraju iz visoko preciznog prikaza s pomičnim zarezom (npr. float32) u nisko precizni prikaz s pomičnim zarezom (npr. float16) ili cijeli broj (npr. int8). Konvertiranjem težina modela na reprezentacije niže preciznosti, veličina modela i brzina inferencije mogu se znatno poboljšati, bez prevelikog žrtvovanja točnosti i preciznosti modela. Coral Dev Board, osim TF Lite formata, za implementaciju zahtjeva da model bude u potpunosti cjelobrojno kvantiziran (engl. *Full integer quantization*). Kvantizaciju modela moguće je provoditi na dva načina, odnosno korištenjem jedne od dviju metoda. Prvi metoda je kvantizirati model tijekom treniranja (engl. *Quantization aware training*), a druga je kvantizirati model nakon treniranja (engl. *Post-training quantization*). Prethodno istrenirani modeli kvantizirati će se korištenjem druge metode. Postoji nekoliko razloga zašto je korištena upravo ta metoda. Prvi razlog je taj što su modeli već istrenirani. Drugi razloga je taj što prva metoda trenutno nije podržana za modele za detekciju objekata trenirane korištenjem TensorFlow verzije 2.x, kojom su modeli trenirani. Posljednji razlog je taj što TF Lite Model Maker, prilikom konvertiranja modela u TensorFlow Lite format, automatski obavlja *post-training* potpunu cjelobrojnu kvantizaciju. U nastavku je prikazan kod kojim se započinje kvantizacija i konverzija efficientDet-Lite0 modela u TF Lite format. Postupak je analogan za ostale modele.

Linija	Kod
1:	TFLITE_FILENAME = './efficientdet-lite0-chess-piece-detection-model.tflite'
2:	LABELS_FILENAME = 'chess-pieces-labels.txt'
3:	SAVED_MODEL_FILENAME = './efficientdet-lite0-chess-piece-detection-model-tf'
4:	model.export = (export_dir = ' . ' ,
5:	tflite_filename = TFLITE_FILENAME,
6:	label_filename = LABELS_FILENAME,
7:	saved_model_filename = SAVED_MODEL_FILENAME,
8:	export_format = [ExportFormat.TFLITE,
9:	ExportFormat.LABEL,
10:	ExportFormat.SAVED_MODEL])
Listing 3.10 Kvantizacija i konverzija modela u TF Lite format	

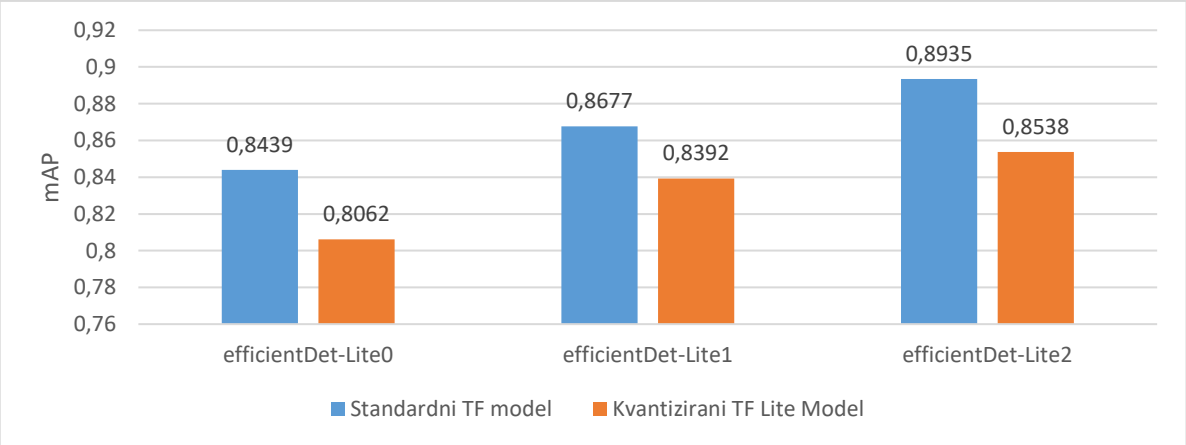
Nakon što su svi modeli kvantizirani i konvertirani u TF Lite format spremni su za implementaciju na ugradbene platforme, no prije toga će ih se evaluirati te dobivene rezultate usporediti s rezultatima evaluacije standardnih modela. U nastavku je prikazan kod kojim se evaluiraju modeli u TF Lite formatu, rezultati evaluacije sva tri modela te grafički prikaz usporedbe dobivenih rezultata prije i poslije konverzije. Kao što je i bilo za očekivati, preciznosti modela su opale nakon konverzije i kvantizacije, no ne značajno.

Linija	**Kod**
1:	model.evaluate_tflite(TFLITE_FILENAME, test_data)

Listing 3.19 Evaluacija TF Lite modela

	efficientDet-Lite0	efficientDet-Lite1	efficientDet-Lite2
AP_King	0.7244	0.7609	0.7859
AP_Queen	0.7627	0.8237	0.8675
AP_Pawn	0.7844	0.7872	0.8582
AP_Bishop	0.8034	0.8333	0.8498
AP_Knight	0.8916	0.9238	0.9147
AP_Rook	0.8708	0.9060	0.8467
mAP	0.8062	0.8392	0.8538

Tablica 3.3 Rezultati evaluacije TF Lite modela



Slika 3.7 Usporedba istreniranih modela

4. IMPLEMENTACIJA MODELA NA CORAL DEV BOARD

4.1. Coral Dev Board

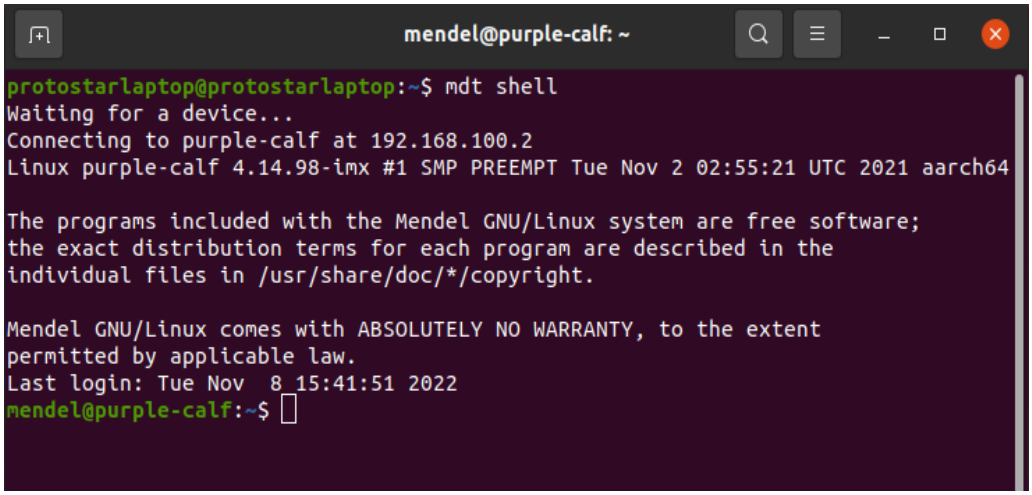
Coral Dev Board je jednopločno računalo razvijeno od strane Google-a. Ono što ga razlikuje od sličnih uređaja je to što sadrži takozvani Edge TPU (engl. *Tensor Processing Unit*). Edge TPU je koprocesor posebno dizajniran za brz i efikasan rad s tenzorima. On značajno ubrzava vrijeme inferencije kod raznih modela strojnog učenja, posebice kod dubokih neuronskih mreža. Sposoban je izvesti 4 trilijuna operacija (tera-operacija) u sekundi (TOPS), pritom koristeći 0,5 vata za svaki TOPS (2 TOPS po vatu) [30]. Ovo ga čini prikladnim za rad u okruženjima s ograničenim energetskim resursima, kao što su primjerice uređaji koji rade na bateriju. Sva zaključivanja izvode se lokalno na uređaju i za njih nije potreban pristup internetu, što osigurava privatnost podataka i omogućuje primjenu uređaja u okruženjima s ograničenim pristupom internetu.

4.2. Uspostavljanje Coral-a za rad

Prvi korak u uspostavljanju Coral Dev Borad-a jest na instalacija Mendel Linux-a [31] na isti. Mendel je distribucija Linuxa bazirana na Debian-u kreirana specifično za Coral i službeno je podržani operacijski sustav za isti. Mendel je moguće instalirati na dva načina, korištenjem microSD kartice ili serijskom konekcijom pomoću micro-B USB kabla. Oba načina su vrlo jednostavna i detaljno opisana na službenoj Coral stranici [32]. Mendel je potreban iz razloga što se sva komunikacija s pločom odvija preko njegovog shell-a kojem pristupamo s drugog računala. Nakon što je Mendel uspješno instaliran na Dev Board, sljedeća stvar koju je potrebno napraviti je instalirati *Mendel Development Tool* [33] (u nastavku MDT) na glavno računalo, odnosno na računalo kojim se spaja na Dev Board. Upravo MDT omogućuje spajanje na Mendelov shell pomoću kojeg se može upravljati pločom. Osim toga nudi i druge mogućnosti kao što su slanje podataka na ploču, preuzimanje podataka sa ploče ili instalacija paketa na ploču.

Ako je Mendel uspješno instaliran na ploču i MDT na glavno računalo, tada je moguće spojiti se na ploču. Prilikom prvog spajanja na ploču putem USB-a automatski se generiraju i pohranjuju privatni i javni SSH ključ što, ako je potrebno, omogućuje spajanje na ploču i preko interneta. Međutim, ukoliko se preko USB-a pokuša spojiti na ploču koja već ima generiran ključ, spajanje neće uspjeti. U tom slučaju potrebno je uspostaviti serijsku konekciju s pločom i manualno izbrisati ključ u *authorized_keys* datoteci. Nakon toga bit će moguće spojiti se na ploču. Samo spajanje s pločom inicira se pokretanjem “$mdt shell” naredbe u Linux terminalu glavnog

računala. Ukoliko je spajanje uspješno dobit će se poruka o tome da je uređaj pronađen i da je konekcija uspostavljena. Sada je moguće komunicirati sa shell-om ploče preko terminala na glavnom računalu. Primjer spajanja na shell ploče može se vidjeti na slici u nastavku.



Slika 4.1 Spajanje na shell ploče korištenjem MDT-a

Kako bi se olakšao rad s pločom od velike koristi je spojiti ploču na mrežu. Ovo uvelike pojednostavljuje stvari kao ažuriranje softwarea i instalaciju potrebnih paketa, jer bi ih se u suprotnom moralo prvo preuzeti na glavnom računalu te ih zatim manualno prebaciti na ploču i instalirati. To se može jednostavno postići spajanjem Ethernet kabla direktno na ploču ili odabiranjem Wi-Fi mreže pokretanjem naredbe "$nmtui" u shell-u uređaja. Sada kada je ploča spojena na mrežu moguće je prijeći na posljednji korak njenog uspostavljanja, kloniranje pyCoral API-ja [34] na ploču. PyCoral API je Python biblioteka koja omogućuje pristup Coral-ovim hardverskim i softverskim resursima. Omogućuje korištenje Pythona za interakciju i kontrolu različitih hardverskih komponenti ploče, od kojih je najvažniji Edge TPU zadužen za izvođenje i ubrzavanje modela strojnog učenja. Ovime je Coral uspješno uspostavljen i spreman za rad.

4.3. Zahtjevi za izvođenje modela na Edge TPU koprocesoru

Postoje određeni zahtjevi [35] koji moraju biti zadovoljeni kako bi se model mogao kompilirati za izvođenje na Edge TPU. Neki od njih, kao što su TF Lite format modela i njegova kvantizacija, objašnjeni su u prethodnim poglavljima, no oni nisu jedini. Uz njih postoji još određen broj zahtjeva koje treba ispuniti ukoliko se žele u potpunosti iskoristiti mogućnosti Edge TPU-a:

- Veličine tenzora su konstantne za vrijeme kompiliranja
- Parametri modela konstantni su za vrijeme kompiliranja

- Tenzori su 1-, 2- ili 3-dimenzionalni. Ako tenzor ima više od tri dimenzije, tada samo tri najdublje dimenzije mogu imati veličinu veću od 1
- Model koristi samo one operacije koje Edge TPU podržava

Neispunjavanje ovih zahtjeva moglo bi rezultirati time da se model uopće ne može kompilirati za Edge TPU ili da će se kompilirati samo njegov dio. Lista svih operacija koje Edge TPU podržava i sva poznata ograničenja dostupni su na službenoj Coral stranici.

4.4. Kompiliranje modela za izvođenje na Edge TPU

Posljednji korak prije implementacije modela na Coral je njegovo kompiliranje za izvođenje na Edge TPU. Za isto se koristi Edge TPU kompajler [36] kojem se jednostavno preda putanja do modela kojeg se želi kompilirati. Ako model ne ispunjava sve prethodno navedene zahtjeve i dalje ga je moguće kompilirati, no samo će se dio modela moći izvršavati na TPU-u. Na prvoj točki u grafu modela, gdje se dogodi nepodržana operacija, prevodilac dijeli graf na dva dijela. Prvi dio grafa, koji sadrži samo podržane operacije, kompilira se u prilagođenu operaciju koja se izvršava na TPU-u. Drugi dio grafa izvršava se na CPU-u. Važno je napomenuti da Edge TPU kompajler trenutačno ne može particionirati model više od jednom, tako da čim se dogodi nepodržana operacija, ta operacija i sve operacije nakon nje izvršavaju se na CPU-u, čak i ako među njima ima podržanih operacija. Kada završi s kompiliranjem, Edge TPU kompajler daje izvještaj o tome koliko se operacija može izvršiti na TPU-u, a koliko ih se (ako ih uopće ima) umjesto toga mora izvršiti na CPU-u. Kompajler kao rezultat vraća model spreman za izvođenje na TPU i jednu *log* datoteku. Kompilirani model ima isti naziv kao i originalni uz dodani nastavak "_edgetpu". Unutar *log* datoteke navedene su sve operacije korištene u modelu te koliki dio njih je uspješno mapiran na Edge TPU. Također, za neuspješno mapirane operacije dana su kratka obrazloženja o tome zašto iste nisu mapirane. Sadržaj *log* datoteke dobivene nakon kompiliranja efficientDet-Lite0 modela prikazan je na slici 4.2.

```
Edge TPU Compiler version 16.0.384591198
Input: efficientdet-lite0-chess-piece-detection-model.tflite
Output: efficientdet-lite0-chess-piece-detection-model_edgetpu.tflite
Operator                       Count      Status
QUANTIZE                       1          Mapped to Edge TPU
DEPTHWISE_CONV_2D              80         Mapped to Edge TPU
RESIZE_NEAREST_NEIGHBOR        12         Mapped to Edge TPU
RESHAPE                        10         Mapped to Edge TPU
CONCATENATION                  2          Mapped to Edge TPU
MAX_POOL_2D                    14         Mapped to Edge TPU
LOGISTIC                       1          Mapped to Edge TPU
ADD                            42         Mapped to Edge TPU
CUSTOM                         1          Operation is working on an unsupported data type
CONV_2D                        102        Mapped to Edge TPU
DEQUANTIZE                     2          Operation is working on an unsupported data type
```

Slika 4.2 Sadržaj log datoteke kompiliranog efficientDet-lite0 modela

4.5. Testiranje performansi modela na Coral-u

Sva tri modela uspješno su kompilirana te ih je sada moguće implementirati na Coral. Njihov rad prvo će se testirati na slikama, a zatim na snimci USB kamere. PyCoral API sadrži niz skripti koje se mogu koristiti za testiranje različitih vrsta modela strojnog učenja. Tako primjerice sadrži i *detect_image.py* skriptu, koja omogućuje testiranje modela za detekciju objekata na slici te mjerenje trajanja inferencije. Navedenom skriptom testirati će se rad i izmjeriti trajanje inferencije pojedinih modela, a dobiveni rezultati će kasnije biti korišteni za konačnu usporedbu ugradbenih platformi. Za testiranje rada modela na snimci USB kamere priključene na Coral napisati će se zasebna skripta koja će implementirati tu funkcionalnost.

Za početak će se modeli, zajedno sa slikama na kojima će se testirati njihov rad, premjestiti na Coral pomoću MDT-a. Za premještanje podataka s glavnog računala na Coral koristi se naredba „$mdt push <lokalna_putanja> <udaljena_putanja>“. Nakon što su modeli i slike premješteni na Coral, pomoću *detect_image.py* skripte testirati će rad sva tri modela. Skripti se kao argumenti predaju: putanja do modela, putanja do datoteke s oznakama, putanja do slike na kojoj se žele detektirati objekti te putanja na kojoj će se pohraniti slika na koju će se ucrtati detektirani objekti. Skripta će, po zadanom, na istoj slici inferenciju provesti deset puta, pri tome mjereći trajanje svake od njih. Skripta će kao rezultat vratiti sliku na kojoj su ucrtani detektirani objekti, te ispisati izmjerena vremena inferencije. Na temelju tih vremena izračunati će se prosječno vrijeme trajanja inferencije pojedinog modela. Pri tom se neće uzimati u obzir trajanje prve inferencije koja traje duže jer ona uključuje i učitavanje modela u memoriju. U nastavku je prikazana naredba kojom se testira rad efficientDet-Lite0 modela treniranog za detekciju šahovskih figura na proizvoljnoj slici te dobiveni rezultati. Postupak je analogan za ostale modele.

```
mendel@purple-calf:~/rp_dip/efficientDet-Lite-Edge-TPU/efficientDet-Lite0-Edge-TPU$
 python3 detect_image.py --model efficientdet-lite0-chess-piece-detection-model_edg
etpu.tflite --labels chess-pieces-labels.txt --output ./result.jpg --input test_img
.jpg --count 10
----INFERENCE TIME----
Note: The first inference is slow because it includes loading the model into Edge T
PU memory.
89.33 ms
71.29 ms
69.61 ms
65.97 ms
80.61 ms
71.62 ms
75.52 ms
63.90 ms
74.80 ms
65.92 ms
```

Slika 4.3 Testiranje rada efficientDet-Lite0 modela i mjerenje brzine inferencije na Coral-u

Slika 4.4 Rezultati inferencije efficientDet-Lite0 modela na proizvoljnoj slici

	efficientDet-Lite0	efficientDet-Lite1	efficientDet-Lite2
Trajanje inferencije (ms)	71.03	105.61	139.83
FPS	14.08	9.47	7.15

Tablica 4.1 Vrijeme trajanja inferencije na Coral-u izražena u milisekundama i FPS-u

4.6. Testiranje rada modela na Coral-u korištenjem USB kamere

Za kraj će se rad modela testirati još i na USB kameri. Ovo zahtijeva pisanje vlastite Python skripte koja će omogućiti tu funkcionalnost. Za rad s kamerom, odnosno za dohvaćanje i prikaz slika koristiti će se openCV [37] biblioteka. Za ucrtavanje objekata na sliku koristiti će se Python Imaging Library (PIL) biblioteka [38]. Za implementaciju modela i provođenje inferencije koristiti će se PyCoral API i TensorFlow Lite Interpreter. U TensorFlow Lite-u objekt *Interpreter* predstavlja ključnu komponentu koja izvršava modele strojnog učenja. Odgovoran je za učitavanje modela i izvođenje operacija zaključivanja na predanim mu ulaznim podacima. Ukratko, pruža sučelje koje omogućuje interakciju s modelom. U nastavku je prikazan kod kojim se model implementira na Coral te kojom se njegov rad testira korištenjem USB kamere.

Linija	Kod
1:	**import** numpy **as** np
2:	**import** cv2
3:	**from** PIL **import** Image
4:	**from** PIL **import** ImageFont
5:	**from** PIL **import** ImageDraw
6:	**from** pycoral.adapters **import** common
7:	**from** pycoral.adapters **import** detect
8:	**from** pycoral.utils.dataset **import** read_label_file
9:	**from** pycoral.utils.edgetpu **import** make_interpreter
10:	
11:	def draw_objects(draw, objs, scale_factor, labels):
12:	for obj in objs:
13:	bbox = obj.bbox
14:	draw.rectangle([(bbox.xmin * scale_factor, bbox.ymin * scale_factor),
15:	(bbox.xmax * scale_factor, bbox.ymax * scale_factor)],
16:	outline="red", width=2)
17:	font = ImageFont.truetype("LiberationSans-Bold.ttf", size=15)
18:	draw.text((bbox.xmin * scale_factor , bbox.ymin * scale_factor - 25),
19:	'%s %.2f' % (labels.get(obj.id, obj.id), obj.score),
20:	fill="white", font=font)
21:	
22:	labels = read_label_file('chess-pieces-labels.txt')
23:	TFLITE_FILENAME='efficientdet-lite0-chess-piece-detection-model_edgetpu.tflite'
24:	interpreter = make_interpreter(TFLITE_FILENAME)
25:	interpreter.allocate_tensors()

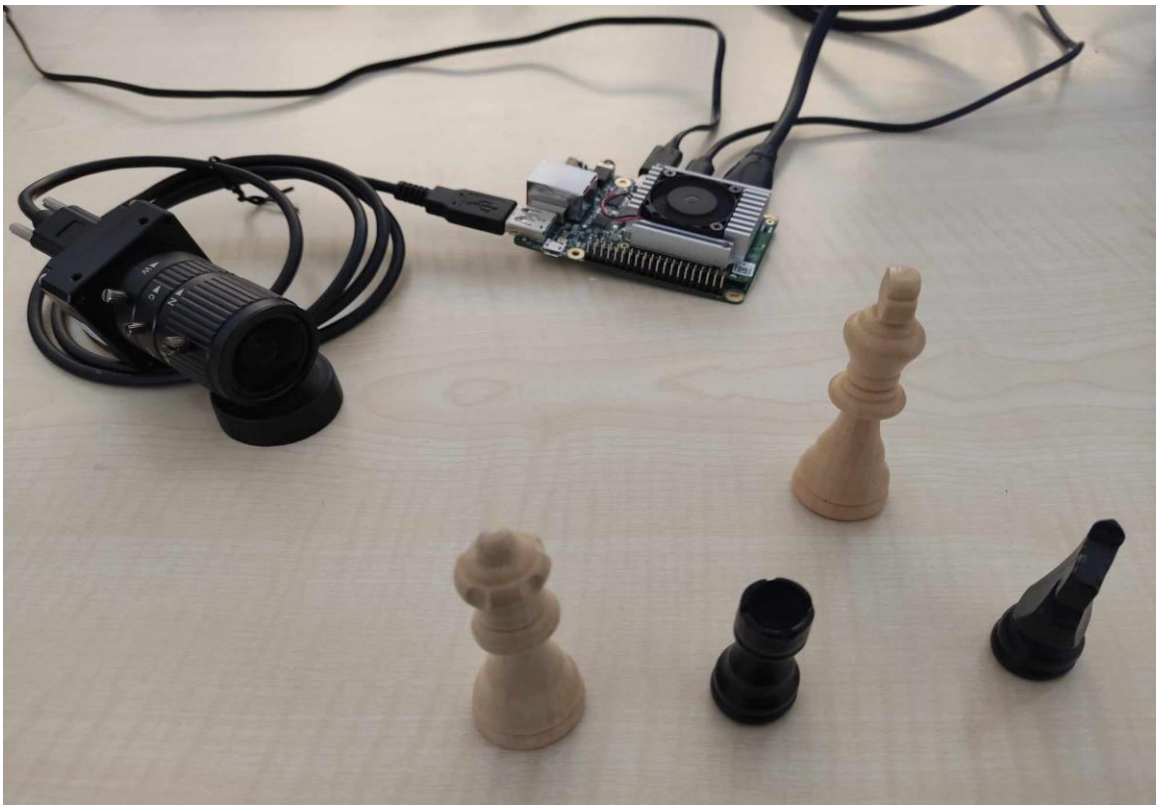
```
26:  usbCam = cv2.VideoCapture(0)
27:  while usbCam.isOpened():
28:      _, frame = usbCam.read()
29:      frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
30:      image = Image.fromarray(frame)
31:      _, scale = common.set_resized_input(interpreter, image.size,
32:                                          lambda size: image.resize(size, Image.Resampling.LANCZOS))
33:
34:      interpreter.invoke()
35:      objs = detect.get_objects(interpreter, score_threshold=0.5, image_scale=scale)
36:
37:      display_width = 1000
38:      scale_factor = display_width / image.width
39:      height_ratio = image.height / image.width
40:      image = image.resize((display_width, int(display_width * height_ratio)))
41
42:      draw_objects(ImageDraw.Draw(image), objs, scale_factor, labels)
43:      imgArr = np.array(image)
44:      imgArr= cv2.cvtColor(imgArr, cv2.COLOR_BGR2RGB)
45:      cv2Img=cv2.imshow("Chess piece object detection", imgArr)
46:      if cv2.waitKey(1) & 0xFF == ord('q'):
47:          Break
48:
49:  usbCam.release()
50:  cv2.destroyAllWindows()
```

Listing 4.1 Programska podrška za rad s kamerom

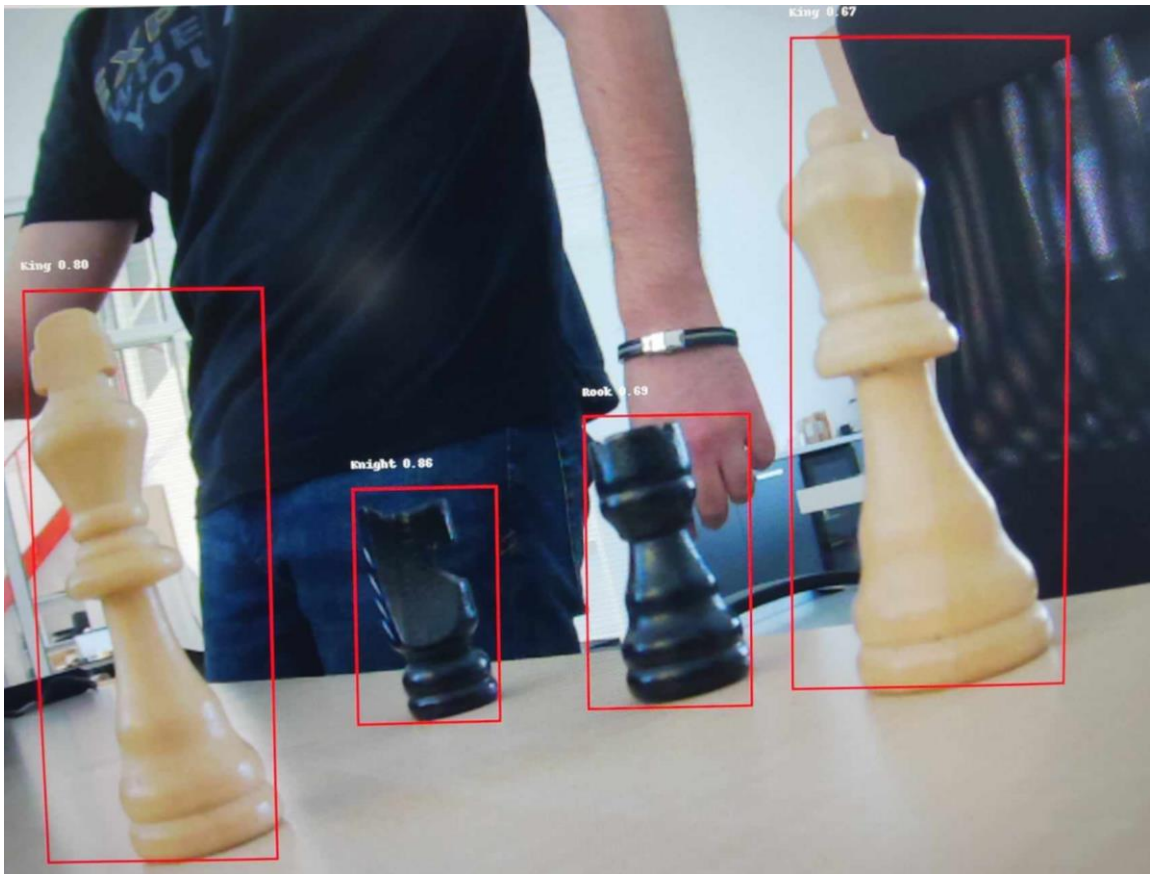
Za početak su importirani svi potrebni paketi. PyCoral API sadrži razne module za rad s različitim vrstama modela strojnog učenja. Iz njega su importirana dva modula: *detect* modul koji sadrži funkcije za rad s modelima za detekciju objekata te *common* modul koji sadrži funkcije primjenjive na raznim modelima. Uz njih su, također iz PyCoral-a, importirane i dvije funkcije: *read_label_file()* funkcija za učitavanje oznaka te *make_interpreter()* funkcija za kreiranje interpretera koji će model izvoditi na Edge TPU-u. Uz navedeno također su importirane numpy, PIL i cv2 biblioteke. Nakon importiranja paketa potrebno je učitati model, odnosno stvoriti *Interpreter* objekt koji će interpretirati model. Njega se jednostavno kreira pomoću prethodno spomenute *make_interpreter()* funkcije kojoj se preda putanja do modela. Na interpreteru je zatim potrebno pozvati metodu *allocate_tensors()* koja alocira memoriju za ulazne i izlazne tenzore. Ovime je model učitan u memoriju i spreman za korištenje.

Slike na kojima će se provoditi inferencija dohvaćati će se s USB kamere. Korištenjem *VideoCapture* klase kreirana je instanca *usbCam* koja predstavlja spojenu USB kameru. Metodom *read()* tada je moguće dohvatiti sliku s te instance, odnosno kamere. Dohvaćenu sliku zatim je potrebno konvertirati u RGB format, jer openCV slike učitava u BGR formatu. Nakon toga se pomoću metode *set_resized_input()* dohvaćenoj slici mijenja veličina kako bi ona odgovarala veličini koju modelu očekuje te ju se postavlja na ulaz u model. Metoda kao argumente prima interpreter modela, izvornu veličinu slike i funkciju za promjenu veličine slike. Funkciju za promjenu veličine slike potrebno je definirati samostalno. Ona kao argument mora primiti touple oblika (visina, širina) koji predstavlja dimenzije ulaza modela, a kao rezultat vratiti sliku odgovarajuće veličine. Metoda *set_resized_input()* kao rezultate vraća sliku promijenjene veličine i faktor skaliranja. Faktor skaliranja pohraniti će se u varijabli *scale*. On služi tome da se objekti detektirani na slici promijenjene veličine mogu pravilno prikazati na originalnoj slici.

Na slici koja je postavljena na ulaz modela sada se može izvršiti zaključivanje pozivom *invoke()* metode interpretera. Rezultati zaključivanja pohranjeni su u samom interpreteru, a dohvaća ih se pomoću *get_objects()* funkcije. Njoj se kao argumenti predaju interpreter modela, prag pouzdanosti i faktor skaliranja *scale*. Dohvaćeni rezultati skalirani su za prikaz na originalnoj slici. Za ucrtavanje položaja detektiranih objekata te ispisivanje njihovih klasa i pouzdanosti na sliku definirana je *draw_objects()* funkcija kojoj se predaju dohvaćeni rezultati. Sliku s ucrtanim objektima tada je moguće prikazati korištenjem *imshow()* funkcije. Cijela procedura dohvaćanja slike s kamere, provođenja inferencije te ucrtavanja i prikaza rezultata smještena je unutar *while* petlje. Petlja se ponavlja dajući dojam snimke uživo. Konačni rezultat prikazan je na slikama u nastavku.



Slika 4.7 Testiranje rada modela na Coral-u s USB kamerom



Slika 4.8 Rezultati testiranja modela na Coral-u s USB kamerom

5. IMPLEMENTACIJA MODELA NA JETSON NANO

5.1. Jetson Nano Developer Kit

Jetson Nano Developer Kit malo je računalo namijenjeno za razvijanje i testiranje različitih vrsta modela strojnog učenja. Razvila ga je tvrtka NVIDIA. Sadrži četvero-jezgreni ARM Cortex-A57 CPU, 4 GB LPDDR4 RAM-a te Maxwell GPU sa 128 jezgri, što mu omogućuje velike brzine obrade. Ključna značajka Jetson Nano Developer Kita je njegova sposobnost da koristi mogućnosti CUDA (engl. *Compute Unified Device Architecture*) i cuDNN (engl. *CUDA Deep Neural Network Library*) biblioteka. Ove softverske komponente iskorištavaju mogućnosti paralelne obrade GPU-a, ubrzavajući izračune neuronskih mreža. Uz to, kompatibilan je s brojnim popularnim okvirima za strojno učenje, kao što su TensorFlow, PyTorch, MXNet i Caffe.

5.2. Uspostavljanje Jetson-a za rad

Jetson Nano Developer Kit koristi microSD karticu kao uređaj za pokretanje i glavnu pohranu. Na karticu je potrebno smjestiti JetPack SDK [39], službeni software za Jetson uređaje. JetPack pruža potpuno razvojno okruženje za razvoj i implementaciju hardverski ubrzanih AI aplikacija na NVIDIA Jetson modulima. JetPack uključuje Jetson Linux s bootloader-om, Linux kernel te Ubuntu desktop okruženje. Također, dolazi sa skupom unaprijed instaliranih biblioteka kao što su primjerice CUDA, cuDNN i TensorRT. U okviru rada korištena je JetPack verzija 4.6.1, preuzeta sa službene JetPack web stranice. Za smještanje slike JetPack SDK-a na microSD karticu korišten je alat *Balena Etcher* [40]. Karticu sa slikom JetPack-a zatim je potrebno umetnuti u microSD utor na donjoj strani Jetson Nano modula te isti uključiti u struju. Na Jetson se sada mogu spojiti monitor, tipkovnica i miš. Tijekom prvog pokretanja potrebno je obaviti neka početna podešavanja kao što su odabir jezika, vremenske zone, rasporeda tipkovnice te korisničkog imena i zaporke. Nakon toga moguće je prijaviti se korištenjem prethodno definiranog korisničkog imena i zaporke. Ukoliko je prijava uspješna, Jetson Nano je uspostavljen za rad i spreman za korištenje.

5.3. Testiranje performansi TF Lite modela na Jetson-u

Modelima prethodno implementiranim na Coral sada je moguće testirati performanse na Jetson-u. Jetson, za razliku od Coral-a, nema gotov alat odnosno skriptu za testiranje performansi TF Lite modela. Iz tog razloga je potrebno napisati takvu skriptu. Za tu svrhu ponovno će se koristiti *numpy*, *tensorflow* i *cv2* biblioteke, ali i Python-ova *time* biblioteka, koja će biti korištena za mjerenje trajanja inferencije. U nastavku se nalazi kod napisane skripte.

Linija	Kod
1:	**import** tensorflow **as** tf
2:	**import** numpy **as** np
3:	**import** time
4:	**import** cv2
5:	
6:	TFLITE_MODEL = 'efficientdet-lite1-chess-piece-detection-model.tflite'
7:	interpreter = tf.lite.Interpreter(model_path = TFLITE_MODEL)
8:	interpreter.allocate_tensors()
9:	input_details = interpreter.get_input_details()
10:	input_shape = input_details[0]['shape']
11:	height = input_shape[1]
12:	width = input_shape[2]
13:	input_tensor_index = input_details[0]['index']
14:	
15:	TEST_IMAGE = './test.jpg'
16:	image = cv2.imread(TEST_IMAGE)
17:	rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
18:	resized_image = cv2.resize(rgb_image, (width, height))
19:	input_image = np.expand_dims(resized_image, axis=0)
20:	interpreter.set_tensor(input_tensor_index, input_image)
21:	
22:	print("Testing mode {:s}".format(TFLITE_MODEL))
23:	print("-----INFERENCE TIME------")
24:	for i in range(10):
25:	start_time = time.perf_counter()

26:	interpreter.invoke()
27:	inference_time = time.perf_counter() - start_time
28:	print("Inference time {:d}: {:.2f} ms".format(i,inference_time*1000))

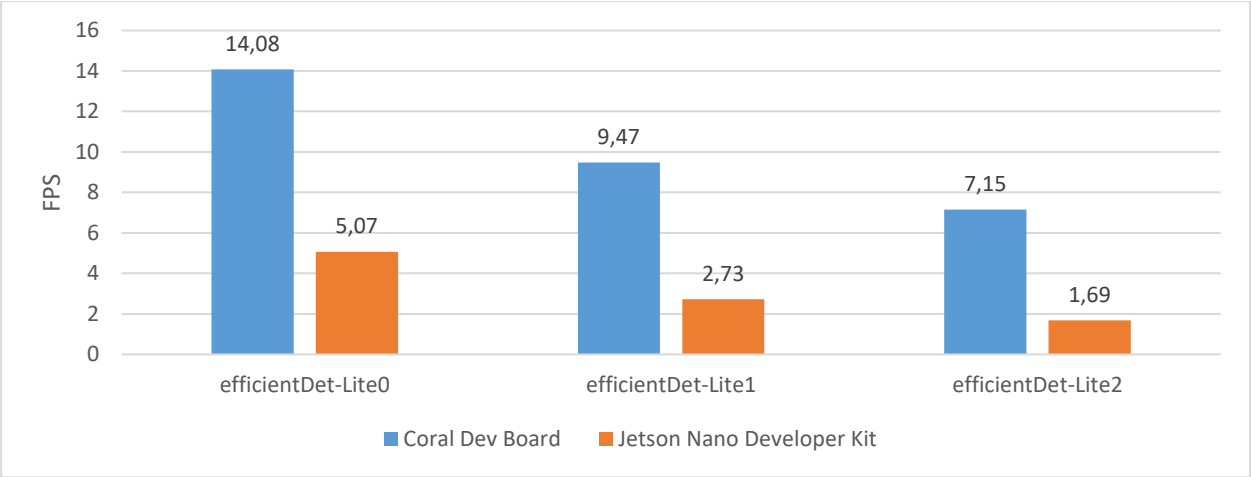
Listing 5.1 Programska podrška za rad s kamerom

Prvo je potrebno kreirati instancu *Interpreter* objekta koja će interpretirati modele. Instanca je kreirana tako što se njenom konstruktoru predala putanja do modela. Ista je zatim pohranjena u varijabli *interpreter*, nakon čega se na njoj pozvala metoda *allocate_tensors()* kojom je alocirana memorije za ulazne i izlazne tenzore. Na njoj je potom pozvana *get_input_details()* metoda kako bi se dobile dimenzije i indeks ulaza. Nakon toga se pomoću metode *imread()* učitala testna slika. Ona je zatim metodom *cvtColor()* konvertirana u RGB format te joj je metodom *resize()* veličina promijenjena kako bi odgovarala veličini ulaza modela. Istoj je također, metodom *expand_dims()*, dodana još jedna dimenzija jer model očekuje *batch* podataka. Metodom *set_tensor()* se tu sliku tada, pomoću indeksa ulaznog tenzora, postavlja na ulaz modela. Sada je pozivom *invoke()* metode na interpreteru moguće provesti inferenciju. Za mjerenje trajanja inferencije korištena je *perf_counter()* metoda. Samo mjerenje provesti će se deset puta po modelu. Na temelju rezultata izračunati će se srednje vrijeme trajanja inferencije, ne uzimajući u obzir prvo mjerenje koje traje duže zbog učitavanja modela. U nastavku su prikazani rezultati mjerenja za sva tri modela.

	efficientDet-Lite0	efficientDet-Lite1	efficientDet-Lite2
Trajanje inferencije (ms)	197.12	366.73	592.52
FPS	5.07	2.73	1.69

Tablica 5.1 Vrijeme trajanja inferencije TF Lite modela na Jetsonu-u izražena u milisekundama i FPS-u

Dobiveni rezultati znatno su lošiji u usporedbi s rezultatima dobivenim na Coral-u, što se može vidjeti iz tablice 5.1. Razlog tome je taj što se ne iskorištavaju sve mogućnosti Jetson-a. Jetson ima sposobnost akceleracije inferencije modela korištenjem TensorRT alata [41]. Konverzija modela u TensorRT format i ponovno testiranje performansi biti će obrađeno u narednim poglavljima.



Slika 5.1 Usporedba performansi TF Lite modela na Coral-u i Jetson-u

5.4. Ponovno treniranje modela

Ispostavilo se da verzije EfficientDet-Lite modela korištenih u radu nisu kompatibilne s TensorRT-om, odnosno da ih se ne može konvertirati u TensorRT format. Kako bi konačna usporedba platformi bila pravilnija odlučeno je ponovno istrenirati modele, no ovoga puta s kompatibilnim verzijama EfficientDet-Lite modela. Za treniranje modela ovoga puta će se koristiti *Google Brain AutoML* preuzet s njihovog git repozitorija [42] koji, isto kao i *TF Lite Model Maker*, sadrži EfficientDet-Lite modele namijenjene za prijenosno učenje. Ti modeli su kompatibilni i moguće ih je konvertirati u TensorRT format. Sam postupak treniranja vrlo je jednostavan i svodi se na poziv *main.py* skripte *autoML* repozitorija. Skripti je prilikom poziva potrebno predati nekoliko parametara, kao što je prikazano u nastavku.

```
(autoML_env) robert@robert-Legion-5-Pro-16ACH6:~/Documents/efDet2trt/automl/efficientdet$ python3 main.py --
mode=train_and_eval --train_file_pattern='CPD-tfrecords/train_tfrecord/-00000-of-00001.tfrecord' --val_file_
pattern='CPD-tfrecords/validation_tfrecord/-00000-of-00001.tfrecord' --model_name=efficientdet-lite0 --model
_dir='../../efficientdet-lite0-saved/'  --train_batch_size=4 --eval_batch_size=1 --eval_samples=150 --num_ex
amples_per_epoch=700  --num_epochs=500
```

Listing 5.2 Naredba za treniranje modela

Prvo je način rada skripte postavljen na treniranje i evaluaciju, postavljanjem *mode* parametra na „train_and_eval“. Zatim su joj predane putanje do skupova podataka za treniranje i evaluaciju preko parametara *train_file_pattern* i *val_file_pattarn.* Parametrom *model_name* odabire se varijanta EfficientDet-Lite modela koja će biti trenirana. Parametrom *model_dir* odabire se lokacija gdje će se pohranjivati kontrolne točke tokom treniranja. Zatim su definirane veličine *batch*-eva za treniranje i evaluaciju parametrima *train_batch_size* i *eval_batch_size*. Nakon toga je parametrima *num_examples_per_epoch* i *eval_samples* definiran ukupan broj slika za treniranje odnosno evaluaciju po epohi. Na kraju se parametrom *num_epochs* odabire ukupan broj epoha.

Nakon što su sva tri modela ponovno istrenirana, potrebno ih je spremiti u SavedModel formatu. Za tu svrhu koristi se skripta *inspector.py*, kojoj je također potrebno predati određene parametre. Prvo je način rada skripte postavljen na eksportiranje, postavljanjem *mode* parametra na „export“. Parametrom *model_name* navodi se koja je varijanta EfficientDet-Lite modela korištena tokom treniranja, a parametrom *model_dir* lokaciju na kojoj su spremljene kontrolne točke. Na kraju se parametrom saved_model_dir odabire gdje će se pohraniti model u SavedModel formatu. Te modele sada je moguće prebaciti na Jetson i konvertirati u TensorRT format.

```
(autoML_env) robert@robert-Legion-5-Pro-16ACH6:~/Documents/efDet2trt/automl/efficientdet/tf2$ python3 inspector.py
 --mode=export --model_name=efficientdet-lite0 --model_dir=../../../efficientdet-lite0-saved/ --saved_model_dir=..
/../../efficientdet-lite0-saved.out
```

Listing 5.3 Naredba za pohranu modela u SavedModel formatu

5.5. Konverzija modela u TensorRT format

Modele istrenirane u prethodnom poglavlju sada je moguće konvertirati u TensorRT format. Model u TensorRT formatu moguće je dobiti na više načina. U okviru rada za tu svrhu koristiti će se alat *trtexec* koji dolazi kao dio TensorRT biblioteke unaprijed instalirane na JetPack SDK-u. Pomoću njega moguće je konvertirati i testirati modele. Međutim, modele je prvo potrebno konvertirati u format koji odgovara *trtexec* alatu jer on ne može direktno konvertirati TensorFlow modele u TensorRT format. Alat prihvaća nekoliko različitih formata od kojih je, u okviru rada, odabran ONNX format (engl. *Open Neural Network Exchange*). Za konverziju modela u ONNX format korištena je *create_onnx.py* skripta koja se također nalazi unutar TensorRT biblioteke. Njoj se kao parametri jednostavno predaju dimenzije ulaza modela, putanja do modela u SavedModel formatu te putanja na kojoj se želi pohraniti model u ONNX formatu. Format naredbe kojom se pokreće skripta je oblika „$python3 create_onnx.py --input_size<dimenzije_ulaza_modela> --savedModel=<putanja_do_tf_modela> --onnx=<putanja_za_pohranu_onnx_modela> “.

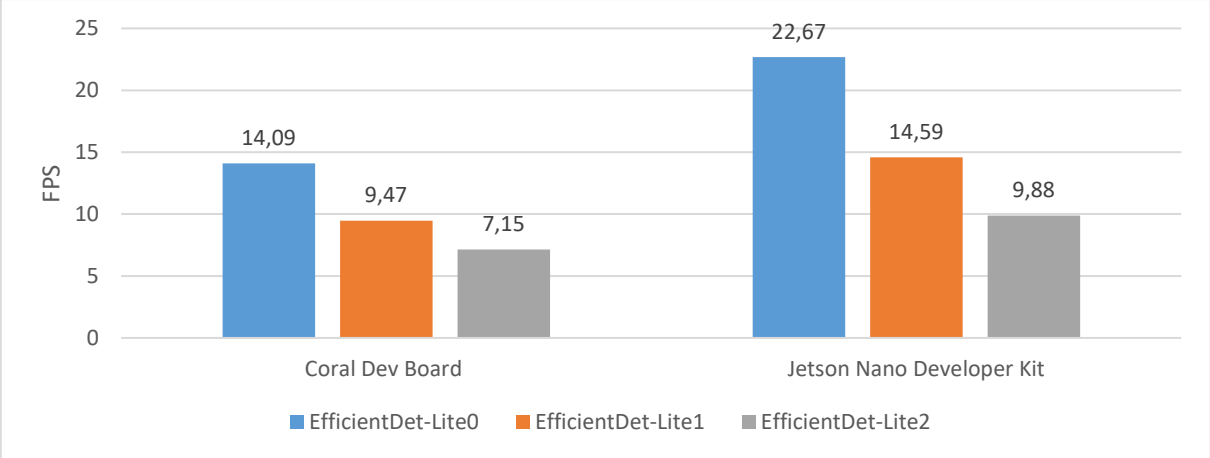
Nakon što su sva tri modela konvertirana u ONNX format, moguće ih je dalje konvertirati u TensorRT format korištenjem *trtexec* alata. Model u TensorRT formatu također se naziva i TensorRT *engine*. Alat *trtexec* koristi se iz terminala, a postupak konverzije vrlo je sličan prethodnom. Sam alat nalazi se na putanji „/usr/src/tensorrt/bin/trtexec“, a naredba kojom se započinje konverzija u TensorRT *engine* je oblika „$trtexec--onnx=<putanja_do_onnx_modela> --saveEngine= <putanja_za_pohranu_konvertiranog_modela> “. Sva tri modela ovom naredbom konvertirana su u TensorRT format te sada imaju .*engine* ekstenziju.

5.6. Testiranje performansi modela na Jetson-u

Performanse dobivenih TensorRT *engine*-a sada je moguće testirati s istim alatom kojim su generirani. Za tu svrhu koristiti će se naredba „$trtexec --loadEngine=<putanja_do_*engine*-a> --batch=1 --useCudaGraph“. Ona će učitati model u memoriju te na njegov ulaz smjestiti nasumične podatke, odnosno „sliku“. Zatim će nekoliko puta provesti inferenciju te kao rezultat vratiti prosječno vrijeme trajanja inferencije, odnosno prosječno vrijeme potrebno da model obradi jednu „sliku“. U nastavku su prikazani dobiveni rezultati za sva tri modela izražena u milisekundama i FPS-u. Dobiveni rezultati TensorRT modela puno su bolju u odnosu na rezultate TF Lite modela

	efficientDet-Lite0	efficientDet-Lite1	efficientDet-Lite2
Trajanje inferencije (ms)	44.11	68.56	101.22
FPS	22.67	14.59	9.88

Tablica 5.2 Vrijeme trajanja inferencije TensorRT modela na Jetson-u izraženo u milisekundama i FPS-u



Slika 5.2 Konačna usporedba performansi platformi

5.7. Testiranje rada modela na Jetson-u korištenjem USB kamere

Za implementaciju modela na Jetson Nano te za njihovo testiranje korištenjem USB kamere napisana je skripta koja obavlja tu funkcionalnost. Sam postupak implementacije nešto je složenije nego što je bio kod Coral Dev Bord-a. Iz tog razloga kod za implementaciju razlomljen je na manje dijelove koji će biti individualno objašnjeni i prokomentirani. Prvi i osnovni korak bio je

importirati sve potrebne paketa. Konkretno, riječ je o TensorRT, openCV, pycuda i numpy bibliotekama. Zatim je definirana pomoćna klasa *HostDeviceMem*. Ona će se kasnije koristiti prilikom alokacije memorije samog Jetsona (RAM) i memorije grafičke kartice (VRAM). Isječak koda kojim se postiže navedeno prikazan je u nastavku.

Linija	Kod
1:	**import** tensorrt **as** trt
2:	**import** numpy **as** np
3:	**import** cv2
4:	**import** pycuda.driver as **cuda**
5:	class HostDeviceMem(object):
6:	def __init__(self, host_mem, device_mem):
7:	self.host = host_mem
8:	self.device = device_mem
9:	def __str__(self):
10:	return "Host:\n" + str(self.host) + "\nDevice:\n" + str(self.device)
11:	def __repr__(self):
12:	return self.__str__()
Listing 5.4 Importiranje potrebnih biblioteka i definiranje HostDeviceMem klase	

Nakon toga definirati će se klasa *TrtModel* čija će instanca predstavljati TensorRT model spreman za korištenje. Unutar klase definirati će se nekoliko atributa i metoda potrebnih za instanciranje samog modela. Prva metoda koja će biti definirana je __init__ metoda. Ona predstavlja konstruktor same klase. U nastavku je prikazan kod koji ju sačinjava.

Linija	Kod
1:	class TrtModel:
2:	def __init__(self, engine_path, max_batch_size=1, dtype=np.float32):
3:	self.engine_path = engine_path

4:	self.dtype = dtype
5:	self.logger = trt.Logger(trt.Logger.WARNING)
6:	self.runtime = trt.Runtime(self.logger)
7:	self.engine = self.load_engine(self.runtime, self.engine_path)
8:	self.max_batch_size = max_batch_size
9:	self.context = self.engine.create_execution_context()
10:	self.inputs, self.outputs, self.bindings, self.stream = self.allocate_buffers()
Listing 5.5 Konstruktro TrtModel klase	

Unutar atributa *engine_path* pohraniti će se putanja do TensorRT *engine*-a kojeg se želi implementirati. Nakon tog, unutar atributa *logger*, pohraniti će se instanca *Logger* klase. *Logger* je potreban za stvaranje instance *Runtime* klase koja je pohranjena u *runtime* atribut i nužna za učitavanje modela u memoriju. Za samo učitavanje modela definirana je statička metoda *load_engine()* koja kao argumente prima prethodno definirani *runtime* i putanju do *engine*-a. Njen sadržaj moguće je vidjeti na slici 5.5. Učitani model je tipa *ICudaEngine* i pohranjen je u *engine* atribut. Zatim se na novo kreiranom *engine* atributu poziva metoda *create_execution_context()* kojom se stvara kontekst za provođenje inferencije te ga se pohranjuje u atribut *context*. Kontekst za provođenje inferencije je tipa *IExecutionContext*. Višestruki *IExecutionContext*-i mogu postojati za jednu instancu *ICudaEngine*-a, dopuštajući da se isti *ICudaEngine* koristi za izvođenje više *batch*-eva podatak istovremeno. Posljednji korak kojeg je potrebno napravit prije nego se model može koristiti je alokacija memorije za ulaze i izlaze iz modela. Za tu svrhu definirana je metoda *allocate_tensors()* čiji sadržaj je moguće vidjeti u nastavku.

Linija	**Kod**
1:	@staticmethod
2:	def load_engine(trt_runtime, engine_path):
3:	trt.init_libnvinfer_plugins(**None**, "")
4:	with open(engine_path, 'rb') as f:
5:	engine_data = f.read()

6:	engine = trt_runtime.deserialize_cuda_engine(engine_data)
8:	return engine
Listing 5.6 Kod load_engine metode	

Linija	**Kod**
1:	def allocate_buffers(self):
2:	inputs = []
3:	outputs = []
4:	bindings = []
5:	
6:	stream = cuda.Stream()
7:	
8:	for binding in self.engine:
9:	size=trt.volume(self.engine.get_binding_shape(binding))*self.max_batch_size
10:	host_mem = cuda.pagelocked_empty(size, self.dtype)
11:	device_mem = cuda.mem_alloc(host_mem.nbytes)
12:	bindings.append(int(device_mem))
13:	
14:	if self.engine.binding_is_input(binding):
15:	inputs.append(HostDeviceMem(host_mem, device_mem))
16:	print(HostDeviceMem(host_mem, device_mem))
17:	else:
18:	outputs.append(HostDeviceMem(host_mem, device_mem))
19:	
20:	return inputs, outputs, bindings, stream
Listing 5.7 Kod load_engine metode	

U kontekstu TensorRT-a, "vezivanje" odnosno *binding* odnosi se na vezu između ulaznih i izlaznih podataka i odgovarajućih memorijskih lokacija na uređaju (GPU) gdje se ti podaci nalaze tijekom zaključivanja. Vezivanje u biti mapira ulazne i izlazne podatke modela neuronske mreže na određene memorijskim adrese na uređaju. Metoda *allocate_buffers()* zadužena je za alokaciju memorije za ulaze i izlaze modela. Ona za svaki *binding* unutar modela alocira potrebnu količinu memorije na uređaju, ali isto tako i na *host*-u. *Host* se odnosi na CPU i RAM Jetson-a. To radi iz razloga što TensorRT koristi CPU i RAM za predobradu i prijenos podataka, a GPU za samu inferenciju. Ovo znatno ubrzava cijeli proces jer omogućava da CPU dohvaća nove podatke dok GPU obrađuje prethodne. Metoda je također zadužena i za stvaranje CUDA *stream*-a. Upravo je CUDA stream mehanizam koji omogućuje asinkrono izvršavanje i upravljanje GPU zadacima određenim redoslijedom te istodobnu obradu i prijenosa podataka.

Posljednje metoda koja će se definirati unutar *TrtModel* klase je __call__ metoda. U Python-u, __call__ metoda omogućuje korištenje objekata određene klase kao funkcije. Ona će omogućiti da se instanci modela preda slika te da ona kao rezultat vrati rezultate inferencije. U pozadini će ona prvo korištenjem funkcije *memcpy_htod_async()* (engl. host to device*)* kopirati sliku na ulaz u model. Zatim će se na prethodno kreiranom *context*-u pozvati metoda *execute_async()* koja će pokrenuti inferenciju. Nakon što je inferencija gotova, rezultati će se dohvatiti pomoću funkcije *memcpy_dtoh_async()* (engl. device to host). U nastavku je prikazana definicija __call__ metode.

Linija	Kod
1:	`def __call__(self, x:np.ndarray, batch_size=2):`
2:	`x = x.astype(self.dtype)`
3:	`np.copyto(self.inputs[0].host, x.ravel())`
4:	
5:	`for inp in self.inputs:`
6:	`cuda.memcpy_htod_async(inp.device, inp.host, self.stream)`
7:	`self.context.execute_async(batch_size=batch_size, bindings=self.bindings,`
8:	`stream_handle=self.stream.handle)`
9:	`for out in self.outputs:`
10:	`cuda.memcpy_dtoh_async(out.host, out.device, self.stream)`

11:	self.stream.synchronize()
12:	
13:	return [out.host.reshape(batch_size,-1) for out in self.outputs]

Listing 5.8 Kod __call__ medtode TrtModel klase

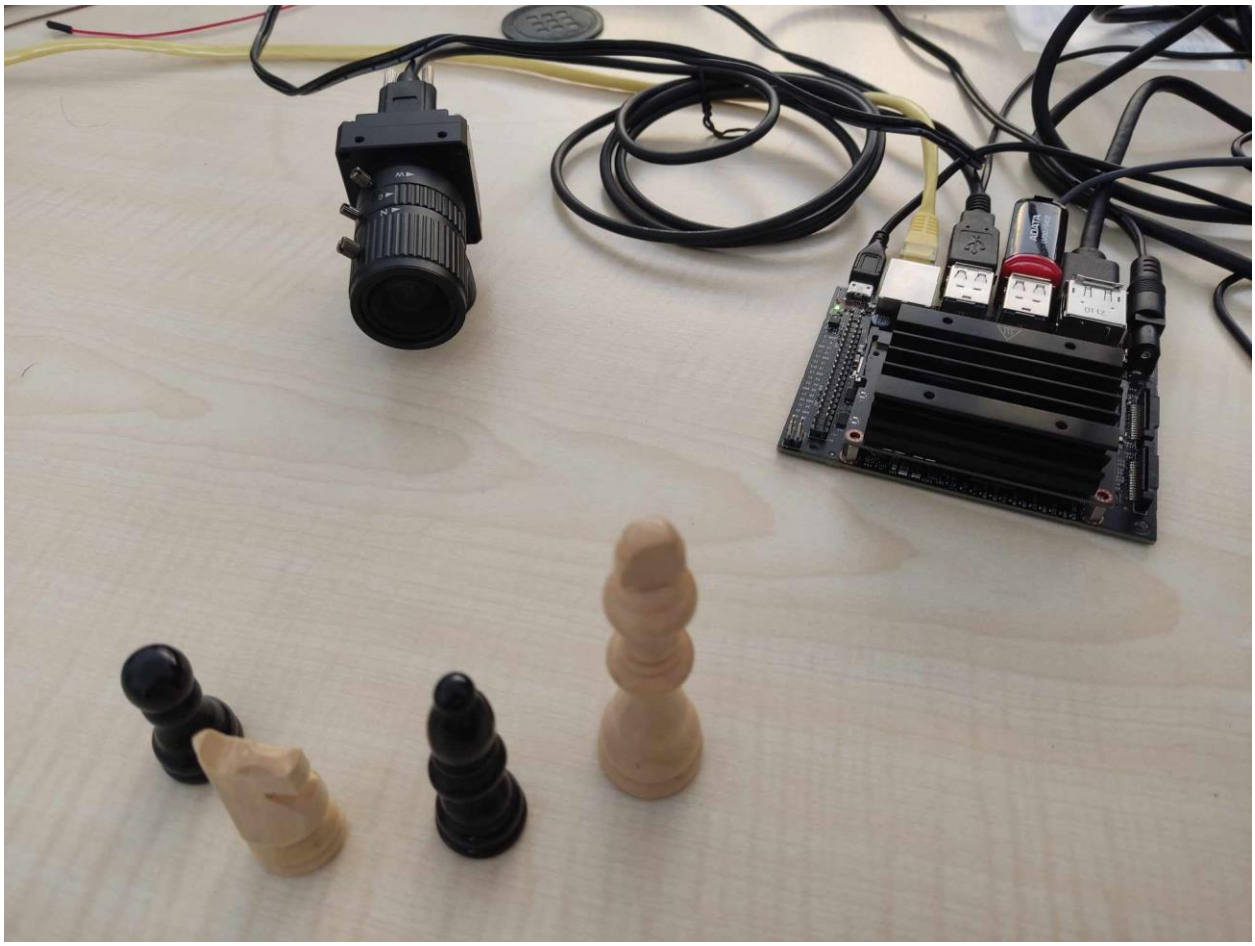
Ovime je klasa *TrtModel* u potpunosti definirana te ju je sada moguće koristiti za učitavanje modela i provođenje inferencije. Sada jedino preostaje napisati programsku podršku za rad s kamerom. Sam postupak gotovo je isti kakav je bio kod Coral-a i svodi se na sljedeće koraka: dohvaćanje slike s kamere, skaliranje slike kako bi ona odgovarala dimenzijama ulaza modela, postavljenje slike na ulaz modela i provođenje inferencije, dohvaćanje rezultata i njihovo skaliranja na dimenzije originalne slike te konačno ucrtavanje rezultata na originalnu sliku i njen prikaz. U nastavku je prikazana kod kojim se implementiraju navedeni koraci te konačni rezultati.

Linija	Kod
1:	if __name__ == "__main__":
2:	batch_size = 1
3:	trt_engine_path = "efficientdet-lite0.engine"
4:	model = TrtModel(trt_engine_path)
5:	shape = model.engine.get_binding_shape(0)
6:	
7:	input_height = shape[1]
8:	input_width = shape[2]
9:	
10:	usbCamera = cv2.VideoCapture(0)
11:	
12:	while usbCamera.isOpened():
13:	_, frame = usbCamera.read()
14:	image = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

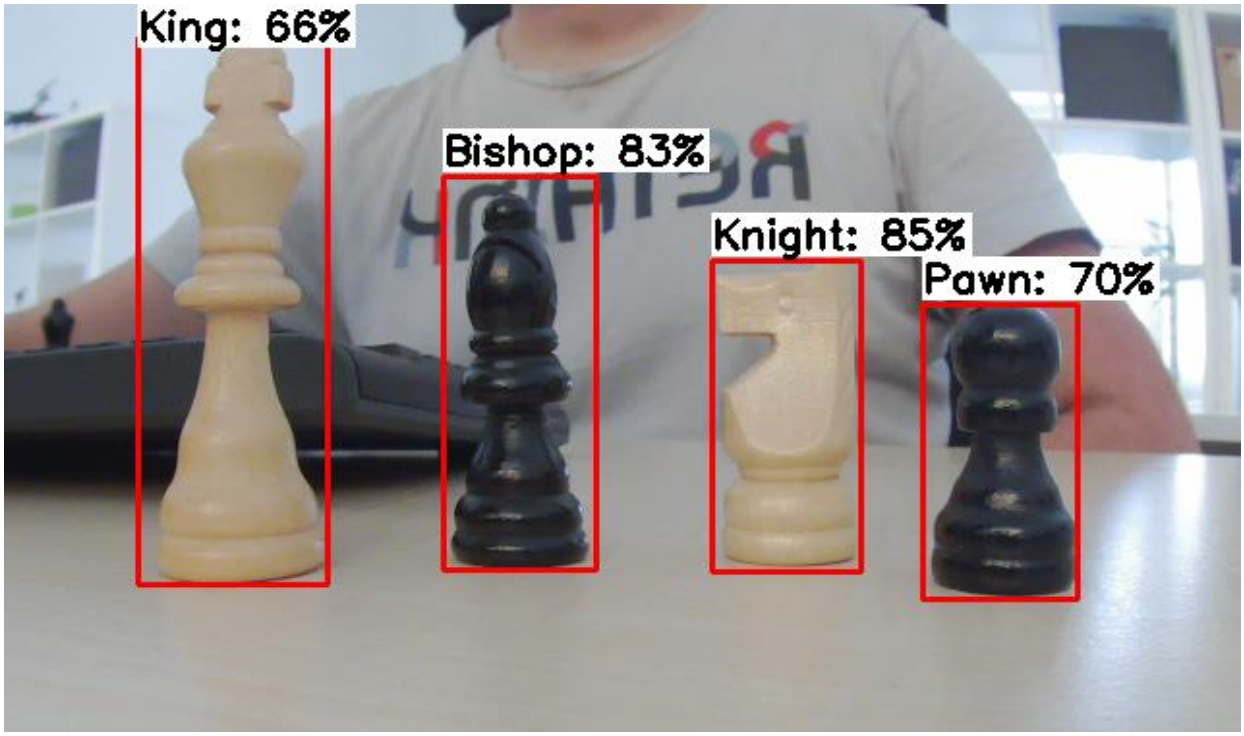
```
15:         imgH, imgW, imgC = image.shape
16:         scale_factor_w = imgW / input_width
17:         scale_factor_h = imgH / input_height
18:
19:         resized_image = cv2.resize(image, (input_width, input_height))
20:         input_image = np.expand_dims(resized_image, axis=0)
21:
22:         result = model(input_image, batch_size)
23:         num_detections_idx, boxes_idx, scores_idx, classes_idx  = 0, 1, 2, 3
24:         boxes = result[boxes_idx][0]
25:         classes = result[classes_idx][0]
26:         scores = result[scores_idx][0]
27:         boxes = np.reshape(boxes, (100, 4))
28:
29:         conf_threshold = 0.4
30:
31:         for i in range(len(scores)):
32:             if ((scores[i] > conf_threshold) and (scores [i] <=1.0)):
33:                 ymin = int(max(1, (boxes[i][0] * scale_factor_h )))
34:                 xmin = int(max(1, (boxes[i][1] * scale_factor_w)))
35:                 ymax = int(min(imgH, (boxes[i][2] * scale_factor_h)))
36:                 xmax = int(min(imgW, (boxes[i][3] * scale_factor_w)))
37:
38:                 cv2.rectangle(image, (xmin,ymin), (xmax,ymax), (0,0,250), 2)
39:                 if str(classes[i]) == "3e-45":
40:                     object_name = "Pawn"
```

```
41:                 if str(classes[i]) == "7e-45":
42:                     object_name = "Rook"
43:                 if str(classes[i]) == "6e-45":
44:                     object_name = "Knight"
45:                 if str(classes[i]) == "1e-45":
46:                     object_name = "Queen"
47:                 if str(classes[i]) == "0.0":
48:                     object_name = "King"
49:                 if str(classes[i]) == "4e-45":
50:                     object_name = "Bishop"
51:
52:                 label = '%s: %d%%' % (object_name, int(scores[i]*100))
53:                 labelSize, baseLine = cv2.getTextSize(label,
54:                                                       cv2.FONT_HERSHEY_SIMPLEX,  0.7, 2)
55:                 label_ymin = max(ymin, labelSize[1] + 10)
56:                 cv2.rectangle(image, (xmin, label_ymin - labelSize[1] - 10),
57:             (xmin + labelSize[0], label_ymin + baseLine-10), (255, 255, 255), cv2.FILLED)
58:                 cv2.putText(image, label, (xmin, label_ymin-7),
59:                                                   cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 0), 2)
60:
61:         cv2.imshow('Object detector', image)
62:         if cv2.waitKey(1) & 0xFF == ord('q'):
63:             break
64:
65: cv2.destroyAllWindows()
```

Listing 5.9 Kod __main__ funkcije



Slika 5.3 Testiranje rada modela na Jetson-u korištenjem USB kamere



Slika 5.4 Rezultati testiranja modela na Jetsonu-u korištenjem USB kamerom

6. ZAKLJUČAK

U okviru rada bile su odabrane dvije ugradbene računalne platforme: Google Coral Dev Board i NVIDIA Jetson Nano. Obje platforme uspostavljene su za rad te se na njih implementiralo nekoliko modela dubokog učenja za detekciju objekata. Sami modeli, bazirani na različitim varijantama EfficientDet-Lite modela, istrenirani su na vlastitom skupu podataka primjenom tehnike prijenosa učenja. Ti modeli zatim su prilagođeni za izvođenje na pojedinoj platformi. U slučaju Coral Dev Board-a to je bila potpuna 8-bitna kvantizacija te konverzija u TensorFlow Lite format. Jetson Nano, s druge strane, zahtijevao je konverziju modela u TensorRT format. Nakon uspješne prilagodbe, modeli su implementirani na ugradbene platforme i njihov rad testiran je korištenjem USB kamere. Uz to su testirane i performanse pojedine platforme mjerenjem prosječnog vremena inferencije za svaki od modela. U konačnici se, za zadatak detekcije objekata, Jetson Nano pokazao kao brža platforma.

LITERATURA

[1] D. M. West i J. R. Allen, »How artificial intelligence is transforming the world,« Brookings, 24 4 2018. [Mrežno]. Available: https://www.brookings.edu/research/how-artificial-intelligence-is-transforming-the-world/. [Pokušaj pristupa 2 4 2023].

[2] V. S. D. Prasad, »Applications and Benefits of Edge AI,« Embedded Computing Design, 8 3 2022. [Mrežno]. Available: https://embeddedcomputing.com/technology/ai-machine-learning/applications-and-benefits-of-edge-ai. [Pokušaj pristupa 3 4 2023].

[3] »Dev Board | Coral,« Coral, [Mrežno]. Available: https://coral.ai/products/dev-board. [Pokušaj pristupa 20 4 2023].

[4] »Jetson Nano Developer Kit,« NVIDIA, [Mrežno]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit. [Pokušaj pristupa 17 4 2023].

[5] W. Liu, D. Anguelov, . D. Erhan, C. Szegedy, S. Reed, F. Cheng-Yang i C. B. Alexander , »SSD: Single Shot MultiBox Detector,« arXiv, 29 12 2016. [Mrežno]. Available: https://arxiv.org/abs/1512.02325. [Pokušaj pristupa 14 4 2023].

[6] J. Redmond, »YOLO: Real-Time Object Detection,« Darknet, [Mrežno]. Available: https://pjreddie.com/darknet/yolo/. [Pokušaj pristupa 3 4 2023].

[7] »COCO - Common Objects in Context,« cocodataset, [Mrežno]. Available: https://cocodataset.org/#home. [Pokušaj pristupa 3 4 2023].

[8] L. Tan, T. Huangfu, L. Wu i W. Chen, »Comparison of RetinaNet, SSD, and YOLO v3 for real-time pill identification,« BMC Medical Informatics and Decision Making, 22 11 2021. [Mrežno]. Available: https://bmcmedinformdecismak.biomedcentral.com/articles/10.1186/s12911-021-01691-8. [Pokušaj pristupa 15 4 2023].

[9] T.-Y. Lin, P. Goyal, R. Girshick, K. He i P. Dollar, »Focal Loss for Dense Object Detection,« arXiv, 7 2 2018. [Mrežno]. Available: https://arxiv.org/abs/1708.02002v2. [Pokušaj pristupa 17 4 2023].

[10] H. Feng, G. Mu, S. Zhong, P. Zhanga i T. Yuan, »Benchmark Analysis of YOLO Performance on Edge,« Cryptography, 16 6 2022. [Mrežno]. Available: https://doi.org/10.3390/cryptography6020016. [Pokušaj pristupa 17 4 2023].

[11] »World's Smallest AI Supercomputer: Jetson Xavier NX,« NVIDIA, [Mrežno]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/. [Pokušaj pristupa 17 4 2023].

[12] »Raspberry Pi 4,« Raspberry pi, [Mrežno]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. [Pokušaj pristupa 4 17 2023].

[13] »Intel® Neural Compute Stick 2,« Intel, [Mrežno]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/neural-compute-stick.html. [Pokušaj pristupa 4 17 2023].

[14] M. Tan, R. Pang i Q. V. Le, »EfficientDet: Scalable and Efficient Object Detection,« arXiv, 27 7 2020. [Mrežno]. Available: https://doi.org/10.48550/arXiv.1911.09070. [Pokušaj pristupa 19 4 2023].

[15] B. Wu, A. Wan, F. Iandola, P. H. Jin i K. Keutzer, »SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous

Driving,« arXiv, 11 7 2019. [Mrežno]. Available: https://doi.org/10.48550/arXiv.1612.01051. [Pokušaj pristupa 19 4 2023].

[16] N. Donges, »What Is Transfer Learning? Exploring the Popular Deep Learning Approach.,« Built In, 25 8 2022. [Mrežno]. Available: https://builtin.com/data-science/transfer-learning. [Pokušaj pristupa 19 4 2023].

[17] »TensorFlow,« TensorFlow, [Mrežno]. Available: https://www.tensorflow.org. [Pokušaj pristupa 20 4 2023].

[18] »TensorFlow Lite | ML for Mobile and Edge Devices,« TensorFlow, [Mrežno]. Available: https://www.tensorflow.org/lite. [Pokušaj pristupa 4 20 2023].

[19] M. Tan i Q. V. Le, »EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,« arXiv, 11 9 2020. [Mrežno]. Available: https://arxiv.org/abs/1905.11946. [Pokušaj pristupa 8 6 2023].

[20] S. Pokhrel, »Image Data Labelling and Annotation — Everything you need to know,« towardsdatascience, 11 3 2020. [Mrežno]. Available: https://towardsdatascience.com/image-data-labelling-and-annotation-everything-you-need-to-know-86ede6c684b1. [Pokušaj pristupa 26 4 2023].

[21] »GitHub - heartexlabs/labelImg - LabelImg,« [Mrežno]. Available: https://github.com/heartexlabs/labelImg. [Pokušaj pristupa 29 5 2023].

[22] »venv - Creation of virtual environments,« python.org, [Mrežno]. Available: https://docs.python.org/3/library/venv.html. [Pokušaj pristupa 6 6 2023].

[23] »Project Jupyter | Installing Jupyter,« jupyter.org, [Mrežno]. Available: https://jupyter.org/install. [Pokušaj pristupa 6 6 2023].

[24] »EfficientDet-Lite0 Object detection model,« TensorFlow Hub, [Mrežno]. Available: https://tfhub.dev/tensorflow/lite-model/efficientdet/lite0/detection/default/1. [Pokušaj pristupa 6 6 2023].

[25] »EfficientDet-Lite1 Object detection model,« TensorFlow Hub, [Mrežno]. Available: https://tfhub.dev/tensorflow/lite-model/efficientdet/lite1/detection/default/1. [Pokušaj pristupa 2023 6 6].

[26] »EfficientDet-Lite2 Object detection model,« TensorFlow Hub, [Mrežno]. Available: https://tfhub.dev/tensorflow/lite-model/efficientdet/lite2/detection/default/1. [Pokušaj pristupa 6 6 2023].

[27] »CUDA Toolkit 11.2 Downloads | NVIDIA Developer,« NVIDIA, [Mrežno]. Available: https://developer.nvidia.com/cuda-11.2.0-download-archive. [Pokušaj pristupa 24 6 2023].

[28] »TensorBoard: TensorFlow's visualization toolkit,« TensorFlow, [Mrežno]. Available: https://www.tensorflow.org/tensorboard. [Pokušaj pristupa 24 6 2023].

[29] »The Ultimate Guide to Deep Learning Model Quantization and Quantization-Aware Training,« Deci, [Mrežno]. Available: https://deci.ai/quantization-and-quantization-aware-training/. [Pokušaj pristupa 25 6 2023].

[30] »Edge TPU performance benchmarks,« Coral, [Mrežno]. Available: https://coral.ai/docs/edgetpu/benchmarks/. [Pokušaj pristupa 25 6 2023].

[31] »What is Mendel Linux?,« Coral, [Mrežno]. Available: https://coral.googlesource.com/docs/+/refs/heads/master/ReadMe.md. [Pokušaj pristupa 26 6 2023].

[32] »Get started with the Dev Board,« Coral, [Mrežno]. Available: https://coral.ai/docs/dev-board/get-started/#flash-the-board. [Pokušaj pristupa 25 6 2023].

[33] »Mendel Development Tool (mdt),« Coral, [Mrežno]. Available: https://coral.ai/docs/dev-board/mdt/#install-mdt. [Pokušaj pristupa 26 6 2023¸].

[34] »PyCoral API overview,« Coral, [Mrežno]. Available: https://coral.ai/docs/reference/py. [Pokušaj pristupa 26 6 2023].

[35] »Model requirements,« Coral, [Mrežno]. Available: https://coral.ai/docs/edgetpu/models-intro/#model-requirements. [Pokušaj pristupa 26 6 2023].

[36] »Edge TPU Compiler,« Coral, [Mrežno]. Available: https://coral.ai/docs/edgetpu/compiler/. [Pokušaj pristupa 26 6 2023].

[37] »Releases - OpenCV,« OpenCV, [Mrežno]. Available: https://opencv.org/releases/. [Pokušaj pristupa 11 8 2023].

[38] »Pillow · PyPI,« PyPl, [Mrežno]. Available: https://pypi.org/project/Pillow/. [Pokušaj pristupa 12 8 2023].

[39] »JetPack SDK | NVIDIA Developer,« NVIDIA, [Mrežno]. Available: https://developer.nvidia.com/embedded/jetpack. [Pokušaj pristupa 14 8 2023].

[40] »balenaEtcher - Flash OS images to SD cards & USB drives,« Balena, [Mrežno]. Available: https://etcher.balena.io. [Pokušaj pristupa 14 8 2023].

[41] »TensorRT SDK | NVIDIA Developer,« NVIDIA, [Mrežno]. Available: https://developer.nvidia.com/tensorrt. [Pokušaj pristupa 25 8 2023].

[42] »GitHub - google/automl: Google Brain AutoML,« Google, [Mrežno]. Available: https://github.com/google/automl. [Pokušaj pristupa 25 8 2023].

SAŽETAK

U okviru rada odabrane dvije ugradbene računalne platforme: Google Coral Dev Board i NVIDIA Jetson Nano. Dan je uvid u značajne karakteristike odabranih platformi te su obje uspostavljene za rad. Na njih se zatim implementiralo nekoliko modela dubokog učenja za detekciju objekata. Sami modeli, bazirani na različitim varijantama EfficientDet-Lite modela, istrenirani su na vlastitom skupu podataka, nakon čega si i evaluirani. Treniranje modela provedeno je primjenom tehnike prijenosa učenja. Istrenirani modeli zatim su prilagođeni za izvođenje na pojedinoj platformi. U slučaju Coral Dev Board-a, prilagodba je podrazumijevala potpunu 8-bitnu kvantizaciju modela te njihovu konverzija u TensorFlow Lite format. S druge strane, prilagodba modela za Jetson Nano odnosila se na njihovu konverziju u TensorRT format. Nakon uspješne prilagodbe, modeli su implementirani na ugradbene platforme i njihov rad testiran je korištenjem USB kamere. Uz to su testirane i performanse pojedine platforme, mjerenjem prosječnog vremena inferencije svakog modela, te su dobiveni rezultati uspoređeni.

Ključne riječi: Detekcija objekata, Ugradbene računalne platforme, Računalni vid

IMPLEMENTATION OF DEEP LEARNING MODELS FOR OBJECT RECOGNITION ON EMBEDDED COMPUTER PLATFORMS

In the context of the thesis, two embedded computer platforms were selected: the Google Coral Dev Board and the NVIDIA Jetson Nano. An insight into the significant characteristics of the selected platforms is given, and both were set up for work. Several deep learning models for object detection were then implemented onto them. The models themselves, based on different variants of the EfficientDet-Lite model, were trained on a custom made data set, after which they were evaluated. Model training was carried out using the transfer learning technique. The trained models were then modified to run on the selected platforms. In the case of the Coral Dev Board, the modifications involved full 8-bit integer quantization of the models and their conversion to the TensorFlow Lite format. On the other hand, the modifications of the models for the Jetson Nano refered to their conversion to the TensorRT format. After this was done, the models were implemented onto the embedded platforms and were tested using a USB camera. In addition, the performance of each platform was tested by measuring the average inference time of each model, after which the obtained results were compared.

Keywords: Computer vision, Embedded computer platforms, Object detection