SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 314

OPTIMIRANJE DUBOKIH MODELA SA SLOJEVIMA PAŽNJE

Dario Oreč

Zagreb, veljača 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 314

OPTIMIRANJE DUBOKIH MODELA SA SLOJEVIMA PAŽNJE

Dario Oreč

Zagreb, veljača 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 2. listopada 2023.

DIPLOMSKI ZADATAK br. 314

Pristupnik:	**Dario Oreč (0036514122)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	prof. dr. sc. Siniša Šegvić

Zadatak: **Optimiranje dubokih modela sa slojevima pažnje**

Opis zadatka:

Duboki konvolucijski modeli danas su metoda izbora za mnoge zadatke računalnog vida. Nažalost, velika računska složenost tih modela isključuje mnoge zanimljive primjene. Taj problem možemo ublažiti agresivnom optimizacijom za ciljani računski uređaj. U okviru rada, potrebno je odabrati okvir za automatsku diferencijaciju te upoznati biblioteke za rukovanje matricama i slikama. Proučiti i ukratko opisati postojeće segmentacijske arhitekture utemeljene na konvolucijama i pažnji. Istražiti mogućnost optimizacije dubokih modela sa slojevima pažnje primjenom alata TensorRT. Vrednovati optimirane modele te prikazati i ocijeniti postignutu točnost. Radu priložiti izvorni i izvršni kod razvijenih postupaka, ispitne slijedove i rezultate, kao i potrebna objašnjenja te dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 9. veljače 2024.

Zahvaljujem mentoru prof. dr. sc. Siniši Šegviću i dr. sc. Josipu Šariću na ukazanom povjerenju i stručnim savjetima. Zahvaljujem svojoj obitelji i prijateljima na bezuvjetnoj podršci.

SADRŽAJ

1. Uvod

Posljednjih godina, modeli temeljeni na mehanizmu pažnje (engl. *attention-based models*) sve više pronalaze primjenu u području računalnog vida. Ovi modeli su pokazali dobre rezultate na zadacima poput klasifikacije, detekcije objekata i semantičke segmentacije [1] [3] [4]. Najbolji primjer ovih modela su Transformeri [2]. Iako su se Transformeri prvobitno koristili za obradu prirodnog jezika, danas pronalaze upotrebu i u računalnom vidu i u prepoznavanju govora [19] [7] [4].

Dobar uvodni primjer primjene modela Transformer u računalnom vidu je Vision Transformer (ViT) [1]. Za razliku od tradicionalnih modela računalnog vida, ViT se ne oslanja na konvoluciju već na mehanizam pažnje. Tretira ulaznu sliku kao niz jednakih dijelova (engl. *patches*) i koristeći mehanizam pažnje pronalazi globalne ovisnosti objekata i uzoraka u ulaznoj slici. Vision Transformer je otvorio nove mogućnosti rješavanja problema računalnog vida iskorištavanjem mehanizma pažnje.

Nadovezujući se na uspjeh ViT-a, a i drugih modela temeljenih na pažnji, model MaskFormer [3] koristi mehanizam pažnje za generiranje maski. Koristeći mehanizam pažnje, MaskFormer efektivno pronalazi uzorke i ovisnosti među objektima i dijelovima slika, te time generira precizne i točne segmentacijske rezultate. MaskFormer je pokazao odlične rezultate na više problema računalnog vida kao što su segmentacija instanci, semantička segmentacija i panoptička segmentacija. Primjer je modela temeljenog na pažnji koji daje kompetitivne rezultate u području računalnog vida.

Još jedan model koji koristi mehanizam pažnje je Mask2Former [4]. Mask2Former je nadogradnja na model MaskFormer koji uvodi novu vrstu pažnje u Transformer dijelu modela, takozvanu maskiranu pažnju. Daje osjetno bolje rezultate u odnosu na MaskFormer, ali zbog veće kompleksnosti modela proces zaključivanja traje duže.

Kroz ovaj rad, uz upotrebu pažnje u računalnom vidu, promatrati će se i ubrzavanje postupka zaključivanja korištenjem tehnologije NVIDIA TensorRT [16]. NVIDIA TensorRT je radni okvir koji služi za optimizaciju i ubrzavanje procesa zaključivanja treniranih dubokih modela za NVIDIA grafičke kartice. Koristeći optimizacije za NVIDIA grafičke kartice i tehnike optimizacije poput fuzije slojeva i kalibracije preciz-

nosti, TensorRT daje značajno ubrzanje procesa zaključivanja. Uz značajno ubrzanje, TensorRT optimizira modele bez gubitka točnosti.

Koristeći radni okvir PyTorch i programski jezik Python, u ovom radu su optimizirani modeli MaskFormer i Mask2Former koristeći radni okvir TensorRT.

2. Gusta predikcija scene

Gusta predikcija scene je jedan od temeljnih problema računalnog vida čiji je cilj naučiti mapiranje ulazne slike u kompleksnu izlaznu strukturu. Izlazna struktura može sadržavati rezultate semantičke segmentacije, procjene dubine i udaljenosti predmeta od kamere, segmentaciju objekata, panoptičku segmentaciju ili brojne druge. Za razliku od detekcije objekta koja pokušava detektirati pojedinačne objekte, gusta predikcija scene nastoji pružiti kompletno razumijevanje svakog dijela slike. U sklopu ovog rada obraditi će se i prikazati rezultati postojećih modela na problemima semantičke segmentacije.

Semantička segmentacija je proces dodjele oznake razreda svakom pikselu ulazne slike. Možemo reći da je semantička segmentacija klasifikacija na razini piksela, jer se svaki piksel klasificira u jednu od N klasa. Rezultat semantičke segmentacije je detaljna i lako razumljiva segmentirana scena. Semantička segmentacija nalazi upotrebu u rješavanju raznih problema računalnog vida, a ponajprije u autonomnoj vožnji, gdje su položaj ceste i objekata na cesti (znakovi, pješaci, druga vozila) ključni za sigurnu vožnju. Većina modela semantičke segmentacije se temelji na konvolucijskoj arhitekturi, te se isti treniraju na velikim označenim skupovima podataka.

2.1. Postojeće arhitekture

Duboke konvolucijske neuronske mreže (engl. *Deep Convolutional Neural Networks DCNNs*) su dugo vremena bile prvi izbor pri rješavanju problema guste predikcije scene. Duboke konvolucijske neuronske mreže su klasa neuronskih mreža čija se glavna primjena pronalazi u rješavanju zadataka prepoznavanja i obrade slike. Glavna komponenta DCNN-ova je konvolucijski sloj koji primjenjuje niz učenih filtara na ulazne podatke. Ovi filtri pronalaze karakteristične značajke ulaza poput rubova, oblika ili tekstura, omogućavajući mreži da pronađe bitne međuovisnosti u podacima. Slaganjem više konvolucijskih slojeva, što je karakteristično za DCNN-ove, dobijemo duboke konvolucijske modele. Kako bi se smanjio veliki memorijski otisak dubokih

modela zbog velikog broja izračuna, između konvolucijski slojeva se obično nalaze slojevi sažimanja (engl. *pooling layers*) koji smanjuju računalni i memorijski otisak, te povećavaju receptivno polje. Na kraju konvolucijskih modela se nalaze potpuno povezani slojevi koji interpretiraju značajke koje su konvolucije izvukle u ovisnosti o problemu kojeg rješavamo, klasifikacija, segmentacija ili nešto drugo.

Kako bi gusta predikcija scene radila dobro, obično su ulazi slike visoke rezolucije. Zbog visoke rezolucije slike i dubine konvolucijskih modela, znatno raste broj potrebnih izračuna i memorijski otisak za dobivanje rezultata predikcije scene. Stoga su se razvile nove porodice arhitektura koje rješavaju problem velikog broja izračuna i memorijskog otiska. Ove arhitekture, nekim novim pristupima, pokušavaju smanjiti memorijski otisak i broj izračuna, a pri tome donose bolje rezultate u odnosu na duboke konvolucijske modele.

Potpuno konvolucijske mreže (engl. *Fully Convolutional Networks FCN*) predstavljaju jedan od pristupa rješavanja problema u području dubokog učenja, posebno za zadatke guste predikcije scene. Cilj potpuno konvolucijskih mreža je zamijeniti potpuno povezane slojeve, koji su se kod tradicionalnih konvolucijskih modela nalazili korak pred generiranje predikcija, konvolucijskim slojevima. Arhitektura FCN-ova se obično sastoji od dva dijela: dio sažimanja i dio naduzorkovanja. Dio sažimanja se, kao i kod tradicionalnih konvolucijskih modela, sastoji od niza konvolucijskih slojeva i služi za izvlačenje značajki iz ulazne slike. Često se ovaj dio preuzme iz već unaprijed treniranih mreža poput ResNet-a [9]. Dio naduzorkovanja služi za postepeno uvećanje značajki do rezolucije ulazne slike. Naduzorkovanje značajki se obično postiže dekonvolucijama (transponiranim konvolucijama) [14] ili odsažimanjem (engl. *unpooling*). Glavna prednost u odnosu na duboke konvolucijske modele je što FCN-ovi donose sposobnost obrade slika različitih dimenzija. Dodatno FCN-ovi su pokazali bolje rezultate na brojnim skupovima podataka [12]. FCN-ovi su korak naprijed u zamijeni dosadašnjih dubokih konvolucijskih modela u rješavanju problema predikcije guste scene.

DeepLab [5] je model koji koristi nekoliko ključnih noviteta za poboljšanje rezultata guste predikcije scene. Iako su CNN-ovi ključna komponenta DeepLaba-a, stvarni napredak u rezultatima donosi korištenje konvolucija s uvećanim filtrima (atrous konvolucije), potpuno povezanih uvjetnih slučajnih polja (engl. *Conditional Random Fields CRFs*) [10] i atrous prostorno piramidalno sažimanje (engl. *Atrous Spatial Pyramid Pooling ASPP*) [5]. Atrous konvolucije uvode novi parametar, stopa dilatacije, koje predstavlja udaljenost između vrijednosti unutar konvolucijskog filtra. Ako je stopa dilatacije 1 onda atrous konvolucija postaje standardna konvolucija. Međutim,

ako je filtar dimenzije 3x3 a stopa dilatacije je 2, tada filtar proširuje prostor nad kojim radi na 5x5, čime efektivno i povećava svoje receptivno polje. Atrous konvolucije, za razliku od operacija sažimanja koje smanjuju dimenzionalnost ulaza, održavaju dimenzije ulaza i time čuvaju detaljnu prostornu informaciju. Atrous prostorno piramidalno sažimanje predstavlja komponentu modela DeepLab koje efektivno hvata višestruke kontekstualne informacije. ASPP se temelji na ideji prostornog piramidalnog sažimanja [8], koja sažima značajke na različitim skalama, te ih poslije spaja. Kod ASPP-a ovo sažimanje se izvodi koristeći atrous konvolucije s različitim stopama dilatacije. Ovaj koncept omogućava modelu da obradi sliku na više skala istovremeno. Uvjetna slučajna polja služe kao korak post-procesiranja za poboljšanje izlaza jer rješavaju specifične probleme povezane s kvalitetom segmentacije. Glavna prednost CRF-ova je njihova mogućnost modeliranja ovisnosti među pikselima slike, i time postižu preciznije i bolje rezultate segmentacije. DeepLab koristi CRF-ove za razmatranje interakcija svakog para piksela na slici, te se time dobiju zaglađeniji rezultati segmentacije, a pri tome se očuvaju važni detalji rubova i smanjuje fragmentacija. CRF-ovi su ključan korak post-procesiranja za DeepLab, jer ne samo da osiguravaju točnost konačne segmentacije, već je segmentacija vizualno uvjerljiva kada se gleda kao cjelina.

3. Modeli temeljeni na pažnji

3.1. Mehanizam pažnje

Mehanizam pažnje (engl. *attention mechanism*) u strojnom učenju je postao jedan od glavnih dijelova modeliranja nizova, omogućujući modeliranje ovisnosti između ulaza i izlaza modela, bez obzira na njihovu udaljenost u nizu. Dodavanjem težina različitim segmentima ulaza, mehanizam pažnje omogućava pronalazak bitnijih značajki unutar sekvence, te umanjuje važnost drugih manje bitnih značajki što omogućuje lakšu analizu konteksta sekvenci. Ovaj mehanizam također rješava problem dugoročnih ovisnosti, što je izazov za tradicionalne arhitekture poput povratnih neuronskih mreža (engl. *Recurrent Neural Networks RNN*) [20], izravno povezujući udaljene dijelove ulaza, čime se očuva kontekst i poboljšava razumijevanje. Zbog toga, prvobitno pronalazi upotrebu u modeliranju sekvenci promjenjivih duljina, odnosno u području prirodne obrade jezika (engl. *Natural Language Processing NLP*) gdje je razumijevanje konteksta ključno. Zbog velikog uspjeha u području obrade prirodnog jezika, mehanizam pažnje se počeo koristiti i u drugim područjima poput prepoznavanja slika (engl. *image recognition*). Modeli zasnovani na pažnji, čak i oni u području prepoznavanja slika, nadmašuju ranije arhitekture koje nisu imale sposobnost iskorištavanja globalnog konteksta informacija. Fleksibilnost i učinkovitost modela pažnje su otvorili nove mogućnosti u napretku strojnog i dubokog učenja.

Pažnja uvodi tri nove komponente: upit (engl. *query (Q)*), ključ (engl. *key (K)*) i vrijednost (engl. *value (V)*). Upit predstavlja trenutni element koji se obrađuje, primjerice riječ u rečenici. Ključ odgovara elementima u ulaznoj sekvenci s kojima će model usporediti upit. Ključ je povezan s određenim dijelom ulaza i služi za izračunavanje relevantnosti s upitom. Vrijednost je povezana sa svakim ključem i sadrži informacije na koje bi model trebao obratiti pažnju ako su ključ i upit slični. Vrijednosti su rezultati pažnje, odnosno njihova agregacija proizvodi izlaz modela. Pažnju možemo prikazati formulom 3.1. Ovdje Q, K i V predstavljaju upit, ključ i vrijednost, redom. Faktor skaliranja $\frac{1}{\sqrt{d_k}}$ služi kako bi se izbjegli preveliki skalarni produkti. Na kraju se

izlaz pažnje provlači kroz funkciju *softmax*, te se dobiju vjerojatnosne interpretacije rezultata.

$$\mathrm{A}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{3.1}$$

Koristeći ove komponente pažnja se može opisati kao mapiranje upita i skupa parova ključ-vrijednost u izlaz, gdje se sve komponente i izlaz mogu predstaviti pomoću vektora. Izlaz se računa kao otežana suma (engl. *weighted sum*) vrijednosti, gdje su težine, pridodane svakoj vrijednosti izračunate kao rezultat sličnosti između upita i odgovarajućeg ključa koristeći neku funkciju sličnosti. Veća sličnost predstavlja veću ovisnost komponenti. Rezultat sličnosti se najčešće računa kao skalirani vektorski umnožak.

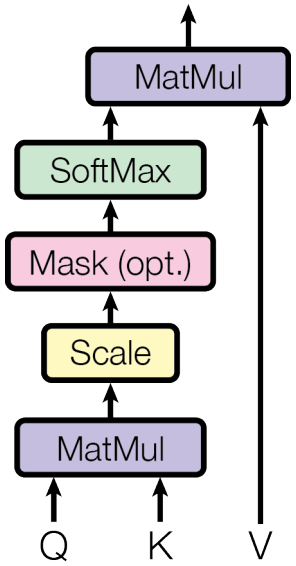
U nastavku ćemo se upoznati s nekim modelima temeljenim na pažnji.

3.2. Transformer

Transformer je model koji koristi mehanizam pažnje kako bi pronašao globalne ovisnosti među tokenima u sekvenci, obično riječi u rečenici. Prije Transformera, modeliranje sekvenci se radilo s RNN-ovima i LSTM ćelijama (engl. *Long Short-Term Memory*). Za razliku od Transformera, ove arhitekture nemaju mogućnost paralelne obrade sekvenci. Paralelizam kod Transformera omogućava puno brži trening, ali i bolju dugoročnu i globalnu ovisnost u ulaznim podatcima. Prvenstveno je bio namijenjen području obrade prirodnog jezika, gdje je omogućio razvoj moćnih modela poput BERT-a [7] (engl. *Bidirectional Encoder Representations from Transformers*) i GPT-a [19] (engl. *Generative Pre-trained Transformer*) koji danas pomjeraju granice prevođenja, razumijevanja i generiranja teksta. Iako dizajniran za NLP, Transformer je pronašao primjenu i u računalnom vidu. Primjerice, model Vision Transformer (ViT) je pokazao odlične rezultate kod klasifikacije slike i time postao konkurencija, dotad najsuvremenijim konvolucijskim modelima. Još jedna prednost Transformera je skalabilnost. Zbog paralelizma Transformer može raditi s velikim skupovima podataka i ulazima, te je i to jedan od razloga primjene Transformera na računalni vid.

3.2.1. Arhitektura

Transformer se sastoji od enkoder-dekoder strukture gdje oba dijela imaju posebnu ulogu u obradi ulaza i generiranju izlaza.



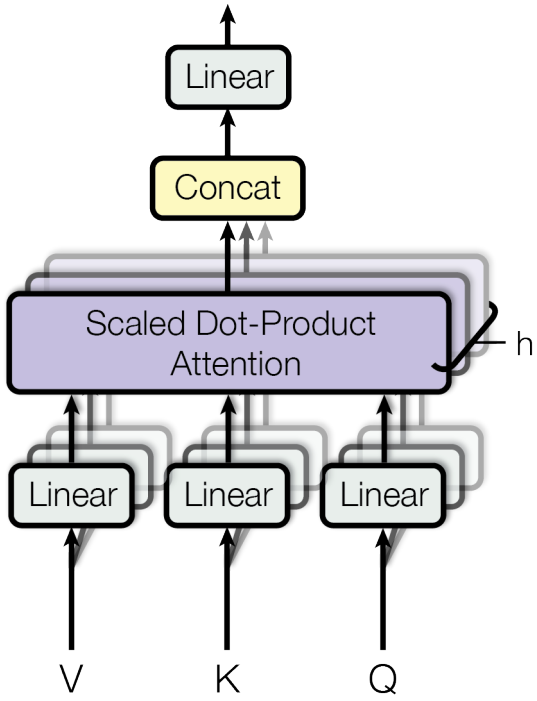
Slika 3.1: Na slici je prikazan mehanizam samopažnje. *Q*, *K* i *V* su upiti, ključevi i vrijednosti, redom. Izlaz samopažnje je otežana suma vrijednosti, gdje su težine izračunate kao matrični umnožak upita i ključeva. Slika preuzeta iz [2].

Enkoder se sastoji od proizvoljnog broja identičnih slojeva. Svaki sloj je građen od dva dijela: sloj višeglave samopažnje i jednostavne unaprijedne potpuno povezane neuronske mreže. Uloga enkodera je obraditi ulaznu sekvencu, pronaći ovisnosti i kontekst u toj sekvenci i proizvesti kontekstualiziranu reprezentaciju sekvence. Izlaz enkodera se dalje prosljeđuje dekoderu.

Dekoder, kao i enkoder, se sastoji od proizvoljnog broja identičnih slojeva, ali s dodatnim slojem koji izvodi samopažnju nad izlazom enkodera. Dekoder generira izlaz sekvencijalno, element po element. Ulaz dekodera se sastoji od dvije glavne komponente: izlazne sekvence i izlaz enkodera. Izlazna sekvenca je sekvenca koju je dekoder do sada generirao. U prevođenju teksta s jednog jezika u drugi, izlazna sekvenca predstavlja dio prevedene rečenice koju je dekoder preveo do tog trenutka. Izlaz enkodera je kontekstualizirana reprezentacija sekvence koja pomaže dekoderu fokusiranje na bitnije dijelove ulazne sekvence u svakom koraku generiranja izlaza. Dakle, enkoder daje kontekst ulaznoj sekvenci, pa dekoder, koristeći tu informaciju, generira smislen izlaz element po element (slika 3.3).

Svaki sloj enkoder-dekoder strukture sastoji se od mehanizma samopažnje, odnosno mehanizma višeglave pažnje i unaprijedne potpuno povezane neuronske mreže (engl. *Fully Connected Neural Network FCNNs*). Nakon svakog dijela, samopažnje i FCNN-a, dolaze slojevi *Add* i *Norm*. Ideja sloja *Add* dolazi od modela ResNet i omogućava lakši protok gradijenata tijekom procesa učenja, a samim time se omogućava

i treniranje dubljih modela. Sloj *Norm* je normalizirajući sloj (engl. *Layer Normalization*) koji osigurava da su srednja vrijednost izlaza blizu 0, odnosno standardna devijacija blizu 1. Normalizacija stabilizira proces učenja i pomaže u bržoj konvergenciji modela.



Slika 3.2: Na slici je prikazana višeglava pažnja koju Transformer koristi. *Q*, *K* i *V* su upiti, ključevi i vrijednosti, redom. Svaka glava odvojeno kroz linearne slojeve generira različite upite, ključeve i vrijednosti. Upiti, ključevi i vrijednosti zatim paralelno prolaze kroz sloj samopažnje. Izlazi samopažnje se spajaju i prosljeđuju kroz završni linearni sloj. Slika preuzeta iz [2].

Samopažnja je glavna komponenta Transformera koja djeluje unutar jedne sekvence (slika 3.1), za razliku od modela sekvence u sekvencu (engl. *sequence-to-sequence models*). Pomoću samopažnje svaki element je "informiran" od strane svih ostalih elemenata, čime se hvata globalna ovisnost elemenata sekvence, bez obzira na njihovu udaljenost u sekvenci. Ključ, vrijednost i upit su izvedeni iz istog ulaza. Za svaku poziciju sekvence se računa otežana suma i time svaki element postaje "informiran", te se na taj način dobije kontekstualna reprezentacija sekvence.

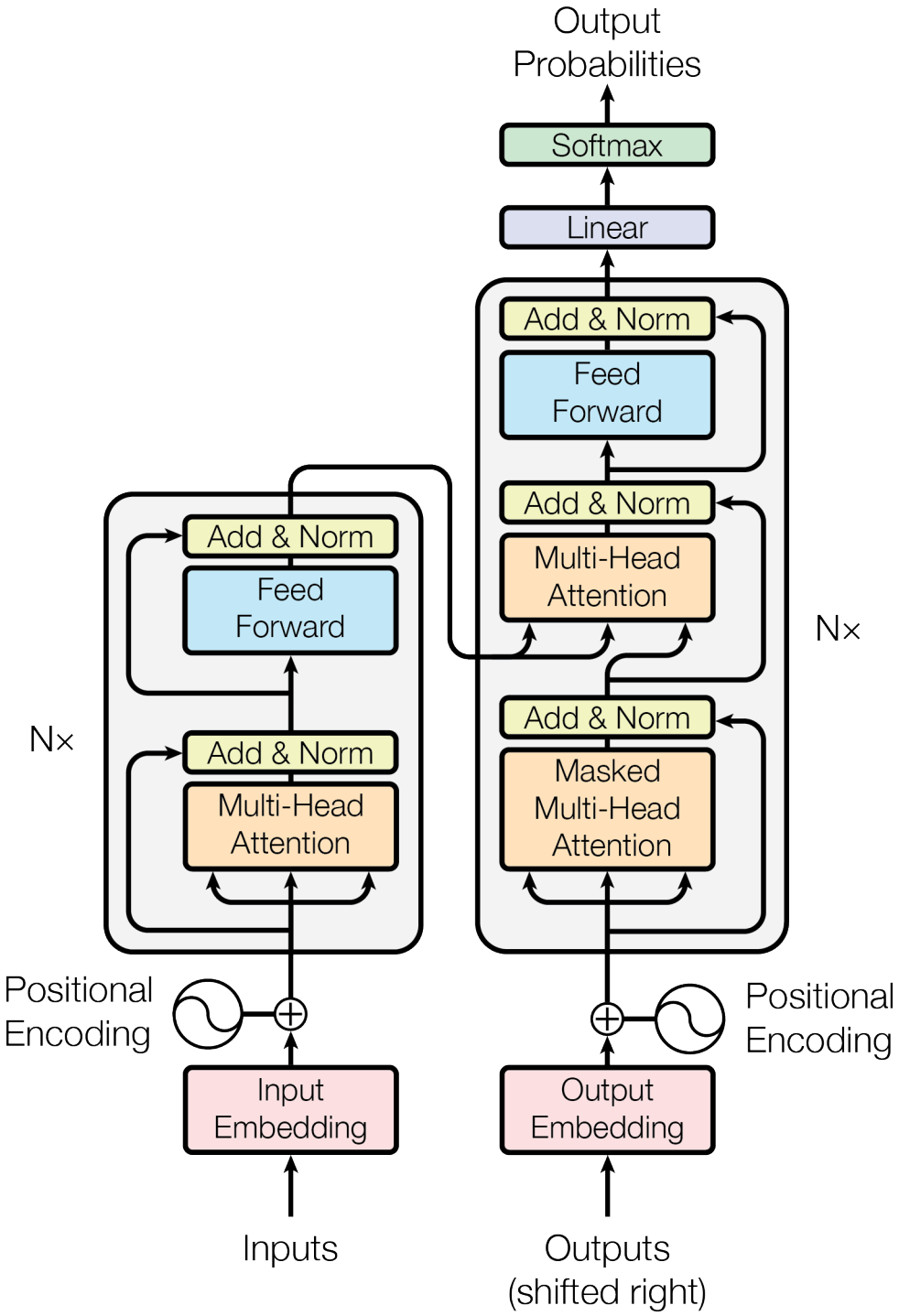
Višeglava samopažnja je nadogradnja mehanizma samopažnje (slika 3.2). Novi-

tet koji ovaj mehanizam uvodi je izvođenje nekoliko operacija samopažnje paralelno, svaka s vlastitim neovisnim skupom ključeva, upita i vrijednosti. Izlazi neovisnih paralelnih izračuna se spajaju u jedan izlaz, te se linearnom transformacijom prebacuju u željenu dimenzionalnost. Ideja višeglave pažnje omogućuje paralelnu obradu različitih neovisnih reprezentacijskih prostora. Primjerice, jedna glava može tražiti semantičko, a druga glava sintatičko značenje ulaza. Samim time, višeglava pažnja ima veću mogućnost pronalaska bitnih ovisnosti u ulazu.

Općenito, postoji niz prednosti korištenja samopažnje u odnosu na tradicionalnu pažnju (RNN, LSTM). Samopažnja omogućuje paralelizam obrade podataka i uzima u obzir veći raspon odnosa i ovisnosti unutar ulazne sekvence.

Unaprijedna mreža dolazi nakon samopažnje i unosi dodatnu nelinearnost u rezultat samopažnje. Sastoji se od dva linearna sloja, sa ReLU aktivacijskom funkcijom između.

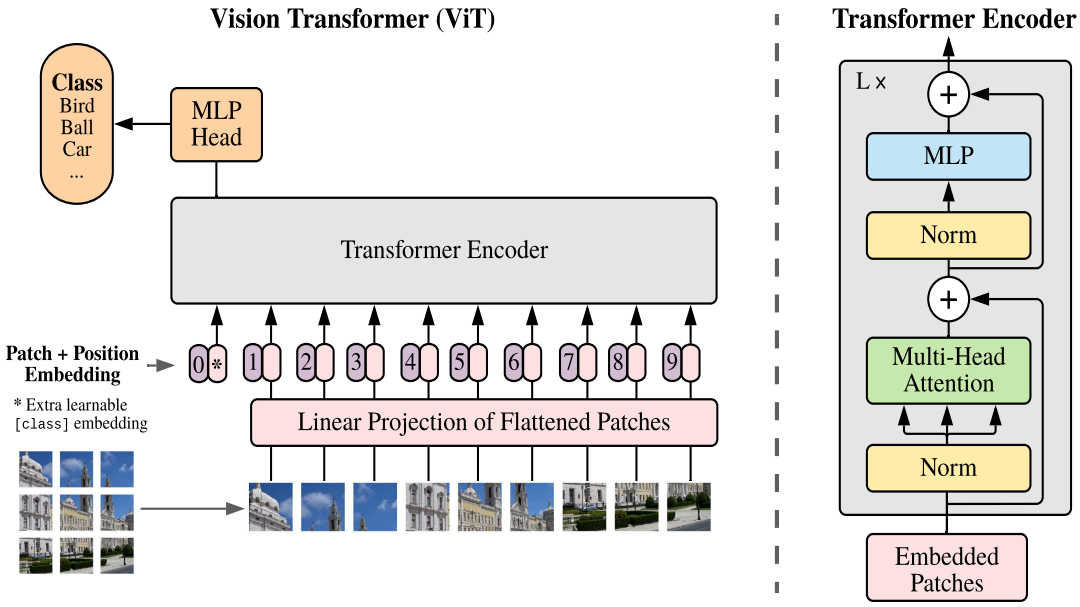
Kako Transformer radi nad nizovima podataka, prethodno opisana arhitektura nema informaciju o poretku elemenata u nizu. Rješenje nudi pozicijsko enkodiranje koje ubacuje informaciju o relativnoj ili apsolutnoj poziciji elemenata u nizu. Pozicijsko enkodiranje se dodaje na početak enkodera, odnosno dekodera. Često korištene funkcije za pozicijsko enkodiranje su sinusoidne funkcije.



Slika 3.3: Arhitektura modela Transformer. S lijeve strane vidimo enkoder sa proizvoljnim brojem (N) slojeva. Slojevi su građeni od samopažnje i unaprijedne mreže na čije se izlaze dodaju rezidualne veze, normiranje i zbrajanje radi učinkovitijeg postupka učenja modela. S desne strane se nalazi dekoder, također s proizvoljnim brojem slojeva (N). Na slici vidimo da dekoder dodaje dodatni sloj samopažnje za izlaze enkodera. Slika preuzeta iz [2].

3.3. Vision Transformer (ViT)

Kako je Transformer postao standardni izbor pri rješavanju problema obrade prirodnog jezika, potaklo je se pitanje njegove primjene, a i samim time zamijene konvolucijskih modela, u računalnom vidu. Vision Transformer (ViT) [1] je primjer primjene čiste arhitekture Transformera na probleme računalnog vida. Vision Transformer predstavlja revolucionaran pristup u računalnom vidu proširujući arhitekturu Transformera na zadatke obrade slike. Vision Transformer koristi arhitekturu transformera nad slikama, koje su standardni ulazi u model kod problema računalnog vida. Radi na način da dijeli sliku na jednake nepreklapajuće isječke (engl. *patches*) i te isječke šalje Transformerovom enkoderu. Ovo omogućava modelu da uhvati globalne ovisnosti na cijeloj slici, što uklanja potrebu korištenja konvolucija. Svaki dio se sastoji od jednakog broja elemenata (piksela). Isječci slike se tretiraju kao i riječi u nizu kod obrade prirodnog jezika. Kao i Transformer, ViT dodaje informaciju o pozicijama isječaka u originalnoj



Slika 3.4: Arhitektura modela ViT (lijevo). Slika se dijeli na jednake isječke koji se linearno projektiraju, te im se dodaje informacija o poziciji u originalnoj slici. Dodatno se dodaje "klasifikacijski token" koji sadrži globalnu informaciju ulazne slike i tako pomaže pri klasifikaciji. Linearne projekcije se predaju enkoderu transformera (desno). Kako bi se provela klasifikacije, rezultat enkodera se šalje višeslojnom perceptronu. Kao i kod Transformera, enkoder se sastoji od L identičnih slojeva, gdje svaki sloj sadrži višeglavu pažnju, unaprijednu potpuno povezanu mrežu i slojeve zbrajanja i normalizacije (*Add* i *Norm*) Slika preuzeta iz [1].

slici. Isječci slike se linearno projektiraju u takozvane *patch embeddings*. Linearne projekcije isječaka slike, zajedno s pozicijskim enkodiranjem, tvore ulaz u Transfor-

mer (slika 3.4).

Vision Transformer je pokazao iznimne rezultate na problemu klasifikacije slika, čime je potvrdio da se Transformer može prilagoditi sa sekvencijalnih zadataka obrade prirodnog jezika prema vizualnim zadacima računalnog vida. Uspjeh ViT-a u računalnom vidu je potaknuo daljnje istraživanje modela temeljenih na arhitekturi Transformera u računalnom vidu.

3.3.1. Arhitektura

Centar arhitekture Vision Transformera je integracija Transformer enkodera u područje računalnog vida. Koristeći mehanizam samopažnje ViT se odmiče od tradicionalne obrade slike koristeći konvolucijske neuronske mreže (engl. *Convolutional Neural Networks CNN*). Uz samopažnju, glavna inovacija je rastavljanje ulazne slike u isječke fiksne veličine. Linearnim ugrađivanjem se isječci šalju na ulaz enkodera koji hvata globalne ovisnosti među njima. Linearno ugrađivanje transformira piksel orijentiranu informaciju u format koji je pogodan za rad samopažnje. Razbijanjem slike na isječke gubi se informacija o poretku isječaka u originalnoj slici. Kako bi se otklonio ovaj problem, ViT koristi pozicijsko ugrađivanje (engl. *position embedding*), metoda koja modelu daje informaciju o stvarnom poretku isječaka u slici. Uz pretprocesiranje slike i enkoder Transformera, ViT sadrži i klasifikacijsku glavu koja je višeslojni perceptron (engl. *Multi-Layer Perceptron MLP*) i koja unosi dodatnu nelinearnost u model. Svrha klasifikacijske glave je procesirati i klasificirati reprezentaciju punu konteksta dobivenu kao izlaz enkodera. Klasifikacijska glava se sastoji od dva potpuno povezana linearna sloja sa aktivacijskom funkcijom ReLU (ili GeLU) između slojeva. Korištenjem klasifikacijske glave, ViT omogućava dobivanje rezultata iz kontekstom obogaćene reprezentacije koje je enkoder generirao.

3.4. MaskFormer

Semantička segmentacija se obično definira i rješava kao problem klasifikacije svakog piksela ulazne slike. MaskFormer je model temeljen na pažnji koji uvodi novi način rješavanja problema semantičke segmentacije pomoću klasifikacije maski. Predviđa skup binarnih maski gdje svaka maska predstavlja predikciju za jedan razred. Kako je klasifikacija maskom pogodna za rješavanje problema segmentacije instanci, koristeći ju za i rješavanje semantičke segmentacije nudi se univerzalan alat za rješavanje dva

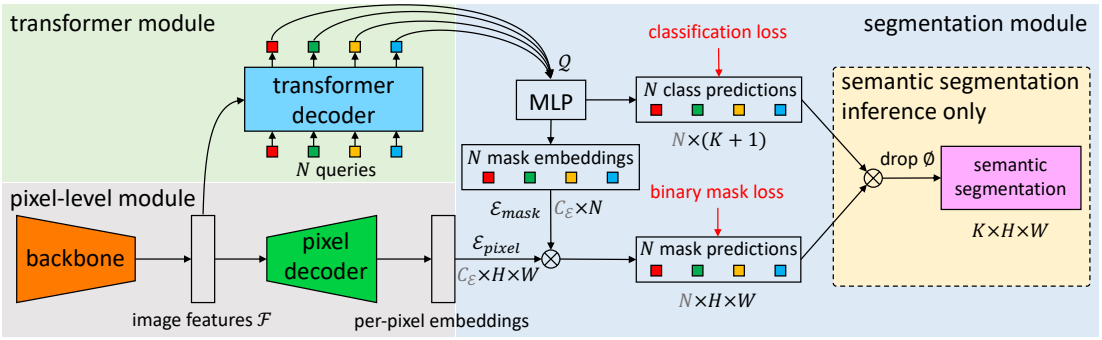
različita problema računalnog vida. Prema tome, MaskFormer zajedno rješava probleme semantičke segmentacije i segmentacije instanci koristeći isti model i funkciju gubitka.

Klasifikacija po pikselu je zadatak dodijele najvjerojatnije klase svakom pikselu u ulaznoj slici. Ovakav zadatak je pogodan za semantičku segmentaciju kojoj je i cilj svakom pikselu ulazne slike dodati oznaku nekog razreda. Klasifikacija po pikselu uvijek generira statičan broj predikcija, jednu predikciju po pikselu. S druge strane, klasifikacija maskom generira maske oko nekog objekta ili regije. Maska se obično definira kao binarna mreža gdje su vrijednosti maske jednake jedan kada pripadaju regiji ili objektu. Svaki piksel maske odgovora pikselu slike. Cilj je generirati masku po pikselima kojom se detektira cijeli objekt na slici. Klasifikacija maskom može generirati masku za svaku instancu objekta posebno umjesto jedne maske za cijeli razred. Broj izlaza je dinamičan i predstavlja skup maski gdje svaka maska predstavlja predikciju nekog objekta ili regije na slici. Zbog ovoga je klasifikacija maskom pogodna za segmentaciju instanci.

Kako bi primijenili klasifikaciju maskom nad semantičkom segmentacijom, potrebno je formulirati semantičku segmentaciju kao problem klasifikacije maskom. Klasifikacija maskom dijeli posao segmentacije na dva dijela. Prvi dio, slika se grupira u N regija, gdje N ne mora biti jednak broju razreda K. Svaka regija je prikazana binarnom maskom m_i. Drugi dio, svaku regiju treba asocirati s nekim od K razreda, odnosno distribucijom nad K razreda. Kako bi se provela klasifikacija maskom potrebno je definirati izlaz z kao vjerojatnost-maska par (p_i, m_i). Semantičko zaključivanje se može dobiti jednostavnim matričnim množenjem $argmax_{c\in\{1\ldots K\}} \sum_{i=1}^{N} p_i(c) \cdot m_i[h, w]$, gdje je $[h, w]$ lokacija piksela. Za semantičku segmentaciju, segmenti s istom kategorijom se spajaju, dok za segmentaciju instanci indeks *i* pomaže pri razlikovanju različitih instanci istog razreda [3].

3.4.1. Arhitektura

Model MaskFormer se sastoji od tri dijela: modul na razini piksela koji izvlači ugradnje po pikselima (engl. *per-pixel embeddings*) za generiranje binarnih maski, modul transformera koji koristi dekoder za generiranje N ugradnji po segmentu (engl. *per-segment embeddings*) i segmentacijski modul koji generira rezultat z na temelju prethodnih ugradnji.



Slika 3.5: Arhitektura modela MaskFormer. Okosnica izvlači značajke slike koje prosljeđuje transofrmer dekoderu i piksel dekoderu. Piksel dekoder postepeno povećava značajke slike te izvlači ugradnje po pikselu $\mathcal{E}_{pixel}$. Transformer dekoder stvara N ugradnji po segmentu $\mathcal{Q}$. Ugradnje po segmentu se dalje šalju višeslojnom perceptronu koji odvojeno generira N predikcija razreda s N odgovarajućih ugradnji maski $\mathcal{E}_{mask}$. Matričnim množenjem $\mathcal{E}_{pixel}$ i $\mathcal{E}_{mask}$ dobijemo N predikcija binarnih maski. Posebno, rezultate semantičke segmentacije možemo dobiti matričnim množenjem N predikcija razreda i binarnih maski. Slika preuzeta iz [3].

Modul na razini piksela radi na razini piksela i služi za generiranje značajki slike i ugradnji po pikselu koji se poslije koriste za stvaranje N binarnih predikcija maske. Ugradnja po pikselu predstavlja kontekstom bogatu reprezentaciju piksela slike. Ovaj modul prima na ulaz sliku dimenzije $H \times W$, te se sastoji od okosnice (engl. *backbone*) i piksel dekodera (engl. *pixel decoder*). Okosnica izvlači mape značajki, obično niske rezolucije, iz ulazne slike i šalje ih piksel dekoderu i transformer dekoderu. Obično se za okosnicu koriste ResNet50 ili ResNet101 [9]. Piksel dekoder povećava (engl. *upsamples*) mapu značajki kako bi generirao ugradnje po pikselima $\mathcal{E}_{pixel}$ dimenzija $C \times H \times W$. Za piksel dekoder se koristi FPN [11] (engl. *Feature Pyramid Network*).

Modul transformera se sastoji od standardnog dekodera Transformera i na ulaz prima značajke slike izvučene iz okosnice. Na temelju značajki i N pozicijski ugradnji (engl. *positional embeddings*), koje se također uče, računa N ugradnji po segmentu $\mathcal{Q}$. Za dekoder Transformera koristi se isti dizajn kao u *DEtection TRansformeru DETR-u* [13].

Segmentacijski modul prima ugradnje po segmentu $\mathcal{Q}$ i ugradnje po pikselima $\mathcal{E}_{pixel}$. Na temelju $\mathcal{Q}$, pomoću linearnog klasifikatora i softmax aktivacijske funkcije, računa vjerojatnosne predikcije razreda za svaki od N segmenata. Da bi dobili predikcije maskom koristi se višeslojni perceptron koji ugradnje po segmentima prebaciva u N ugradnji po maskama $\mathcal{E}_{mask}$ dimenzije C. Binarne predikcije maske se dobiju matričnim množenjem i-tih $\mathcal{E}_{mask}$ i $\mathcal{E}_{pixel}$ nakon čega slijedi sigmoidalna aktivacijska

funkcija.

3.5. Mask2Former

Masked-attention Mask Transformer (Mask2Former) je novi univerzalni model za segmentaciju slike izgrađen na temelju modela MaskFormer. Kao i MaskFormer sastoji se od okosnice, piksel dekodera i transformer dekodera. Uvodi nove promjene koje daju bolje rezultate i ubrzavaju učenje. Glavni novitet je korištenje maskirane pažnje (engl. *masked attention*) u transformer dekoderu. Maskirana pažnja izvlači lokalizirane značajke tako što ograničava pažnju unutar predviđenih područja maske. Koriste se značajke visoke rezolucije za lakše segmentiranje malih objekata, te se uvode optimizacije poput uklanjanja dropout slojeva i zamjena redoslijeda izvođenja samopažnje i unakrsne pažnje i smanjuje memorijski otisak pri treningu do tri puta računajući maskirani gubitak na nekoliko slučajno odabrani točki.

Maskirana pažnja je varijanta unakrsne pažnje (engl. *cross-attention*) koja se samo usmjerava na područje prednjeg plana (engl. *foreground*) predviđene maske za svaki upit. Pretpostavka rada maskirane pažnje je da su lokalne značajke dovoljne za ažuriranje značajki upita i da se kontekst može prikupiti kroz samopažnju. Maskiranu pažnju, sa rezidualnim vezama, možemo opisati sljedećom formulom:

$$\mathbf{X}_l = \text{softmax}(\mathbf{M}_l + \mathbf{Q}_l\mathbf{K}_l^T)\mathbf{V}_l + \mathbf{X}_{l-1}. \tag{3.2}$$

Ovdje, l predstavlja indeks sloja, $\mathbf{X}_l \in \mathbb{R}^{N\times C}$ predstavlja N C-dimenzionalnih značajki upita u l-tom sloju i $\mathbf{Q}_l = f_Q(\mathbf{X}_{l-1}) \in \mathbb{R}^{N\times C}$. $\mathbf{X}_0$ je ulazna značajka upita u dekoder Transformera. $\mathbf{K}_l$, $\mathbf{V}_l \in \mathbb{R}^{H_lW_l\times C}$ su značajke slike pod transformacijama f_K i f_V, redom. H_l i W_l su prostorne dimenzije značajki slike. f_Q, f_K i f_V su linearne transformacije. Pažnjina maska $\mathbf{M}_{l-1}$ na mjestu značajke (x, y) je opisana sljedećom formulom:

$$\mathbf{M}_{l-1}(x,y) = \begin{cases} 0, & \text{ako je } \mathbf{M}_{l-1}(x,y) = 1 \\ -\infty & \text{inače} \end{cases}. \tag{3.3}$$

Ovdje, $\mathbf{M}_{l-1} \in \{0,1\}^{N\times H_l\times W_l}$ predstavlja binarni izlaz maske predikcije promijenjene veličine prethodnog sloja dekodera Transformera. Veličina maske predikcije je promijenjena na dimenzije $\mathbf{K}_l$. $\mathbf{M}_0$ je binarna maska predikcije dobivena iz $\mathbf{X}_0$. Izbacivanjem $\mathbf{M}_{l-1}$ iz 3.2, maskirana pažnja postaje unakrsna pažnja.

Značajke visoke rezolucije općenito poboljšavaju performanse modela, posebice za male objekte, ali zahtijevaju puno hardverski resursa i izračuna [13]. Kako bi se

smanjio broj izračuna, Mask2Former koristi višeskalnu strategiju koja koristi značajke visoke rezolucije. Ideja je da se koriste piramidalne značajke koja se sastoji od značajki niske i visoke rezolucije. Svaka rezolucija se, od niske ka visokoj, daje odgovarajućem sloju dekodera Transformera (slika 3.6). Konkretno se koriste tri različite rezolucije i L slojeva dekodera.

Optimizacije, koje Mask2Former koristi, provode se na dekoderu Transformera. Provode se tri vrste optimizacija. Prvo, uvodi se maskirna pažnja, odnosno zamjenjuje se redoslijed izvođenja samopažnje i unakrsne pažnje. Pretpostavka ove optimizacije je što su značajke upita prvog sloja samopažnje ovisne o ulaznoj slici, pa zbog toga samopažnja manje vjerojatno može obogatiti informaciju i doprinijeti boljem krajnjem rezultatu. Drugo, značajke upita $\mathbf{X}_0$ se također uče u treningu i na kraju, u potpunosti se uklanja dropout sloj jer obično smanjuje performanse, a nema utjecaj na krajnji rezultat.

3.5.1. Arhitektura

Mask2Former se sastoji od okosnice, piksel dekodera i transformer dekodera. Okosnica je preuzeta iz modela MaskFormer.

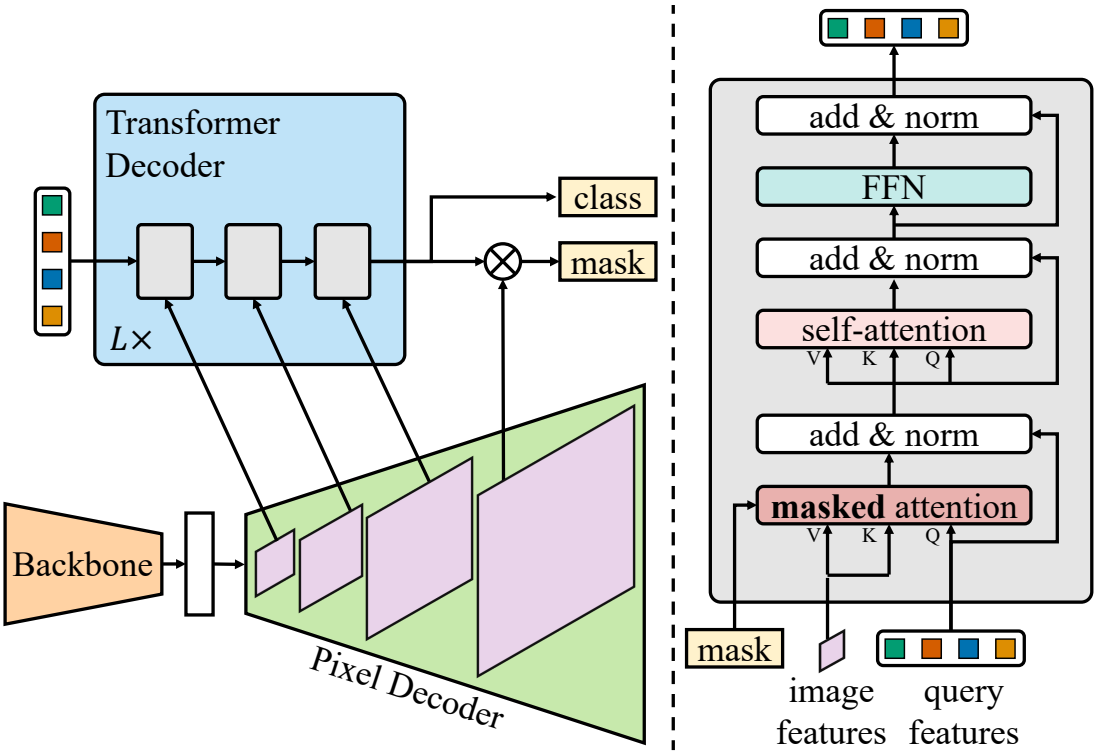
Piksel dekoder služi za postepeno povećanje značajki niske rezolucije dobivenih iz okosnice, te generiranje ugradnji po pikselu visoke rezolucije. Piksel dekoder koristi piramidu značajki koja se sastoji od značajki visoke i niske rezolucije. Izvučene značajke se jedna po jedna šalju slojevima transformer dekodera. Konkretno se koriste piramide značajki rezolucije 32, 16 i 8 puta manje od originalne slike. Ovakvim postupkom se osigurava da se lakše segmentiraju manji objekti na ulaznoj slici. Još jedna promjena u odnosu na MaskFormer je što Mask2Former za piksel dekoder koristi Multi-Scale-Deformable Attention *(MSDeformAttn)* Transformer [21].

Multi-Scale-Deformable Attention je nadogradnja piksel dekodera u odnosu na model MaskFormer. Pokazao se kao najbolji odabir za različite segmentacijske zadatke [4]. Inspiriran deformabilnim konvolucijama, *MSDeformAttn* se fokusira na rijedak skup ključnih točaka oko referentne točke. Na ovaj način se omogućava modelu da se fokusira na bitne značajke udaljenih, manjih i teže vidljivih objekata.

Problem korištenja pažnje nad slikama je što se mora koristiti nad svim prostornim lokacijama značajki slike, što je računski i vremenski vrlo zahtjevno. Zbog toga se koristi multi-scale-deformable pažnja. Ideja deformabilne pažnje je preuzeta od deformabilnih konvolucija. Deformabilna pažnja se fokusira na mali skup uzorkovanih točaka oko referentne točke, bez obzira na dimenzije mape značajki slike. Na ovaj

način se smanjiva broj izračuna, a model se fokusira na bitne značajke ulaza.

Dekoder transformera je najveća promjena u odnosu na model MaskFormer. Glavna promjena je korištenje maskirane pažnje. Maskirna pažnja izvlači lokalne značajke tako što se fokusira na dijelove predviđenih maski, a ne na cijelu mapu značajki. Transformer dekoder prima mape značajke, izvučene pomoću piramide značajke iz piksel dekodera, u uzastopne slojeve (round robin) (slika 3.6). Koristi se dekoder s L = 3, te zbog značajki visoke rezolucije dekoder sadrži sveukupno 9 slojeva. Dodatno, prethodne optimizacije se provode u ovom dijelu modela.



Slika 3.6: Arhitektura modela Mask2Former (lijevo). Arhitektura je građena prema modelu MaskFormer, pa se sastoji od okosnice, piksel dekodera i transformer dekodera (desno). Piksel dekoder uzastopno povećava značajke slike, te ih šalje transformer dekoderu. Transformer dekoder (desno) uz samopažnju koristi i maskiranu pažnju. Izlaz transformer dekodera i piksel dekodera matričnim množenjem daju predikcije maski (kao kod modela MaskFormer). Slika preuzeta iz [4].

4. NVIDIA TensorRT

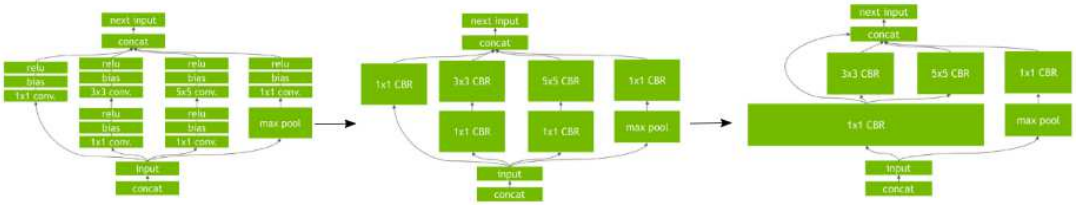
NVIDIA TensorRT [15] je SDK (Software Development Kit) za optimiranje istreniranih dubokih modela. Služi za ubrzavanje zaključivanja već treniranih dubokih modela, bez velikih gubitaka točnosti na NVIDIA hardveru. TensorRT sadrži optimizator zaključivanja dubokog učenja za istrenirane duboke modele i vrijeme izvođenja (engl. *runtime*). Nakon što smo istrenirali naš model u okviru koji koristimo i prebacimo model u format koji TensorRT može optimirati (npr. ONNX), dobijemo optimirano izvršavanje zaključivanja s većom propusnosti i manjom latencijom. Dizajniran je da radi s okvirima poput PyTorch-a i TensorFlow-a.

4.1. Optimizacije

NVIDIA TensorRT je dizajniran za optimiziranje i ubrzavanje procesa zaključivanja istreniranih dubokih modela na NVIDIA grafičkim karticama. Podržava više različitih tehnika optimizacije, a neke od njih su: kalibracija preciznosti (engl. *precision calibration*), fuzija slojeva (engl. *layer fusion*), automatsko podešavanje jezgre (engl. *kernel auto-tuning*), dinamičko upravljanje memorije tenzora (engl. *dynamic tensor memory management*) [15].

Kalibracija preciznosti omogućava dinamično ažuriranje preciznosti težina i aktivacija dubokog modela. TensorRT analizira težine i aktivacije tijekom zaključivanja modela, te ažurira njihove vrijednosti na optimalnu preciznost (FP32, FP16, INT8), a da se pri tome minimizira gubitak točnosti i poveća ubrzanje performansi.

Fuzija slojeva je optimizacijska tehnika koja kombinira više slojeva u jedan sloj. Fuzijom slojeva se smanjiva vrijeme izvođenja jer se spojeni slojevi ne izvšavaju odvojeno. TensorRT analizira graf izračuna modela i pronalazi slojeve koji su pogodni za fuziju. Postoje dva načina provođenja fuzije slojeva: vertikalna i horizontalna. Vertikalna fuzija spaja više slojeva, koji su naslagani jedan na drugi, u jedan sloj. Primjer vertikalne fuzije su uzastopne konvolucije nakon kojih slijede aktivacijske funkcije. Horizontalna fuzija spaja slojeve koji primaju isti ulaz i provode iste operacije. Fuzija



Slika 4.1: Primjer jednostavne konvolucijske neuronske mreže s aktivacijskim funkcijama (lijevo). Vertikalna fuzija (sredina) se primjenjuje na naslagane slojeve konvolucije i aktivacije (CBR). Horizontalna fuzija (desno) se primjenjuje na slojeve koji primaju isti tenzore i rade iste operacije nad njim [15].

slojeva smanjuje memorijski otisak, poboljšava učinkovitost grafičke kartice i pojednostavljuje graf izračuna.

Automatsko podešavanje jezgre se odnosi na iskorištavanje grafičke kartice na najoptimalniji način. TensorRT pronalazi najbolju kombinaciju parametara, primjerice broj dretvi i veličina bloka dretve, kako bi postigli optimalnu performansu na korištenoj grafičkoj kartici.

Dinamičko upravljane memorije tenzora predstavlja memorijske optimizacije koje TensorRT provodi nad tenzorima i hardverom kako bi se minimizirao memorijski otisak. TensorRT alocira memoriju na grafičkoj kartici samo onda kada je memorija potrebna tenzoru i ponovno iskorištava zauzetu memoriju kada je tenzor više ne koristi. Ova optimizacija smanjuje razmjenu podataka između procesora i grafičke kartice i time povećava performanse zaključivanja modela.

4.2. ONNX

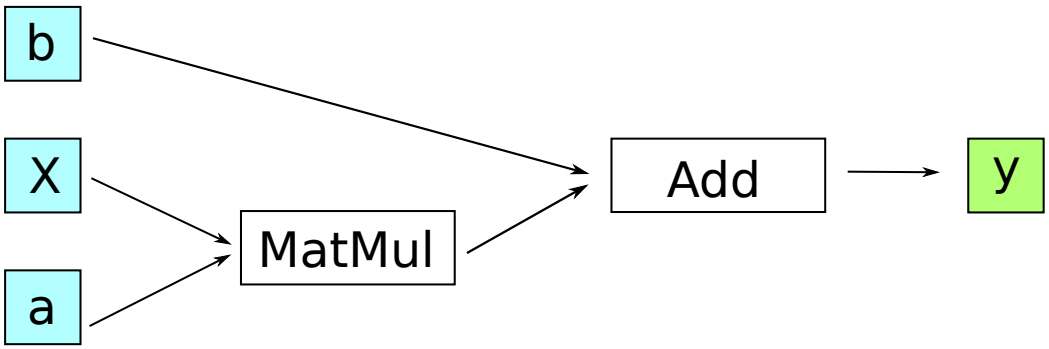
Kako bi TensorRT mogao provesti optimiranja nad našim modelom potrebno je model prikazati nekim formatom koji TensorRT može razumjeti. Jedan takav format je Open Neural Network Exchange, skraćeno ONNX [18].

ONNX je javno dostupni datotečni format napravljen za univerzalno predstavljanje modela strojnog i dubokog učenja bez obzira na okvir u kojem su modeli pisani i trenirani. ONNX definira skup operatora s kojima se mogu graditi duboki modeli. Međutim, ONNX ne podržava sve operatore koje možemo pronaći primjerice u PyTorch radnom okviru. Podržava uobičajeni skup operatora koji su, za veći dio modela i operacija, dovoljni, poput konvolucije, aktivacijskih funkcija, matričnog množenja i brojnih drugih operatora. Zbog toga se ponekad dogodi da naš model sadrži neke operacije

koje nisu podržane u skupu ONNX operatora. Rješenje je pokušati izmijeniti model na način da koristi operacije koje su podržane ili pokušati samostalno implementirati odgovarajući ONNX operator. Oba rješenja mogu biti izazovna i dugoročna.

Kada naš istrenirani model želimo prebaciti u ONNX format gradi se takozvani ONNX graf sa podržanim ONNX operatorima. Gradnja ONNX grafa se odnosi na implementaciju izračuna i funkcija našeg modela s ONNX operatorima. Glavni dijelovi ONNX grafa su ulazi (engl. *inputs*), izlazi (engl. *outputs*) i čvorovi (engl. *nodes*).

Uzmimo za primjer jednostavnu linearnu regresiju opisanu formulom $y = aX + b$, gdje je y rezultat vektor, a i b vektori koeficijenata i X matrica ulaza. Izgradnjom ONNX grafa nad linearnom regresijom dobiti ćemo tri ulaza a, b i X, jedan izlaz y i dva čvora. Čvorovi će biti prikazani ONNX operatorima, konkretno matričnim množenjem (operator *MatMul*) i zbrajanjem (operator *Add*).



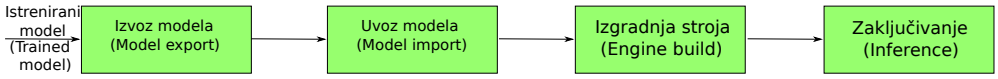
Slika 4.2: Primjer grafa kojeg bi ONNX generirao za problem linearne regresije. X, a i b su ulazni čvorovi, y je izlazni čvor, a *MatMul* i *Add* su čvorovi koji implementiraju odgovarajuće ONNX operacije.

Uz generiranje grafa, ONNX također optimizira sam graf. Kada se ulaz nikada ne mijenja, konkretno vektori koeficijenata a i b, ONNX ih spremi kao konstante unutar čvora koji ih prima na ulaz.

4.3. TensorRT tijek rada

Tijek rada (engl. *workflow*) TensorRT-a možemo podijeliti na uvoz (engl. *import*) modela, stvaranje stroja za zaključivanje (engl. inference engine) i zaključivanje [17]. Ako razvijamo model u jednom radnom okviru, ili jednostavno želimo spremiti naš naučeni model, a TensorRT zaključivanje provodimo u drugom radnom okviru, potre-

ban je dodatan korak izvoza modela u odgovarajući format, primjerice ONNX. Prema tome, tijek se rada TensorRT-a, u većini slučajeva, sastoji od četiri koraka.



Slika 4.3: Tijek rada (engl. *workflow*) TensorRT-a. Istrenirani model u radnom okviru po želji, se izvozi u format kojeg TensorRT može prepoznati (ONNX). Model se učitava iz formata, parsira te se stvara TensorRT stroj za zaključivanje. U ovom koraku se provode TensorRT optimizacije nad modelom. Nakon izgrađenog stroja za zaključivanje, preostaje nam koristiti model za zaključivanje.

U ovome radu ćemo prikazati dva načina na koji se može izgraditi TensorRT stroj za zaključivanje koristeći programski jezik Python s dva različita paketa: *onnx_tensorrt* i *tensorrt*. Oba načina imaju svoje prednosti i mane. Paket *onnx_tensorrt* nudi vrlo jednostavan okvir za izgradnju TensorRT stroja za zaključivanje. No, ne nudi robustan način postavljanja parametara izgradnje stroja. S druge strane, paket *tensorrt* zahtijeva pažljivu pripremu podataka i objekata potrebnih za izgradnju stroja i provođenje zaključivanja, ali zbog toga i nudi veću slobodu pri postavljanju bitnih parametara za izgradnju stroja.

4.3.1. TensorRT tijek rada koristeći *onnx_tensorrt* paket

Prvi korak je izvoz modela u format ONNX kako bi TensorRT na temelju ONNX grafa mogao optimirati i izgraditi stroj za zaključivanje. Nakon izvoza modela slijedi uvoz modela u ONNX formatu. Slijedi gradnja optimiziranog TensorRT stroj za zaključivanje. Kako bi se mogao izgraditi stroj, potreban je jedan primjer ulaza modela, NVIDIA grafička kartica i drugi konfiguracijski parametri. U zadnjem koraku je potrebno predati stvorenom TensorRT stroju ulaz i provesti zaključivanje.

Programski kod 4.1 prikazuje tijek rada TensorRT-a u programskom jeziku Python. Kako bi se provelo zaključivanje na TensorRT stroju u Pythonu potrebni su moduli *torch*, *numpy*, *onnx*, *onnx_tensorrt*. Prije izvoza modela u ONNX format potrebni su istrenirani model i ulaz u model (linije 7-11 programskog koda 4.1). Model mora biti u evaluacijskom načinu rada kako bi se TensorRT stroj pravilno izgradio. Model i ulaz se moraju ručno prebaciti na grafičku karticu pozivom funkcije *to* koja kao argument prima znakovni niz *cuda*, što označava prebacivanje na grafičku karticu. Programski se izvoz odvija u dva koraka. Prvi korak je napraviti unaprijedni prolaz kroz

model i spremiti rezultat u *torch.out* varijablu (linija 14). Drugi korak je napraviti sam izvoz modela u ONNX format koristeći funkciju *torch.onnx.export* (linije 15-22). Funkcija *torch.onnx.export* kao argumente prima model, ulaz, putanju do spremanja modela u obliku znakovnog niza, verziju (parametar *opset_version*), naziv ulaznih, odnosno izlaznih čvorova u grafu (parametri *input_names* i *output_names*). Novije verzije sadrže veći broj podržanih operacija. Verzija (*opset_version*) u ovome radu je postavljena na najnoviju stabilnu verziju 16. Uvoz modela se jednostavno obavlja pozivajući funkciju *load* modula *onnx* prosljeđujući putanju do izvezenog modela (linija 25). Ponekad izvođenje naredbe za uvoz direktno nakon izvoza može rezultirati greškom. U tom slučaju se izvoz, odnosno uvoz odrađuje u odvojenim procesima. Nakon uvoza slijedi gradnja stroja za zaključivanje pozivom funkcije *prepare* modula *onnx_tensorrt.backend*. Funkcija *prepare* prima model i znakovni niz koji obilježava redni broj grafičke kartice (CUDA:0, CUDA:1, itd.), te kao rezultat vraća izgrađen stroj za zaključivanje (linija 28-31). Izgradnjom stroja za zaključivanje spremni smo raditi zaključivanje jednostavnim pozivom funkcije *run* nad našim strojem. Funkcija *run* prima ulaz u obliku NumPy polja i kao rezultat vraća predikciju modela (linija 34).

```python
import torch
import numpy as np
import onnx
import onnx_tensorrt.backend as backend

# Priprema ulaza i modela
model = TestModel()
model.to("cuda")
model.eval()
image = load_image("path/to/image.png")
x = torch.tensor(image).to("cuda")

# Izvoz modela
torch.out = model(x)
torch.onnx.export(
        model,
        x,
        "model_name.onnx",
        opset_version=16,
        input_names["input"],
        output_names=["output"]
)

# Uvoz modela
model = onnx.load("model_name.onnx")

# Stvaranja stroja za zaključivanje
engine = backend.prepare(
        model,
        device="CUDA:0"
)

# Zaključivanje
output = engine.run(np.array(image))
```

Programski kod 4.1: Tijek rada (engl. *workflow*) TensorRT-a opisan u četiri koraka koristeći paket *onnx_tensorrt*.

4.3.2. TensorRT tijek rada koristeći *tensorrt* paket

Kao i prethodni način, TensorRT tijek rada koristeći *tensorrt* paket se sastoji od izvoza i uvoza modela, gradnje stroja i zaključivanja. Potrebni moduli za ispravan rad TensorRT stroja su *onnx*, *numpy*, *tensorrt* i *pycuda*.

Programski kod 4.2 prikazuje tijek rada TensorRT-a. Izvoz modela se radi na identičan način kao i u prethodno opisanom načinu gradnje TensorRT stroja. Nakon izvoza modela dolazi uvoz i učitavanje modela. Uvoz modela se odvija u par koraka (linije 9-13 i 14-17). Prvo je potrebno inicijalizirati sve potrebne objekte (linije 9-13). Linija 9 inicijalizira TensorRT *Builder* i *Logger* objekte. Objekt *Builder* služi za izgradnju TensorRT stroja za zaključivanje iz definicije modela (mreže). Linija 10 stvara konfiguraciju *builder* objekta. Koristeći konfiguraciju možemo postavljati razne parametre s kojima kontroliramo način gradnje TensorRT stroja. Primjer je linija 11 s kojom postavljamo maksimalnu veličinu radnog prostora. Linija 12 stvara *network* objekt potreban za izgradnju TensorRT stroja. Nakon stvaranja svih potrebnih objekata, možemo parsirati model (linije 14-17). Linija 14 stvara parser kojim parsiramo model zapisan u formatu ONNX, sprema ga u objekt *network* i time završavamo postupak uvoza modela. Uvoz modela je dosta kompliciraniji u odnosu na uvoz modela koristeći *onnx_tensorrt* paket. Nakon što smo uvezli model, spremni smo za izgradnju stroja za zaključivanje. Izgradnja stroja za zaključivanje se obavlja pozivom funkcije *build_engine* objekta *builder* koja prima objekte *network* i *config* (linija 19). Kada imamo izgrađen stroj za zaključivanje možemo provesti zaključivanje. Međutim zaključivanje zahtijeva par koraka pripreme i nije jednostavno kao zaključivanje koristeći *onnx_tensorrt* paket. Kako bi proveli zaključivanje potrebno je prvo stvoriti kontekst izvršavanja (linija 22). Nakon stvaranja konteksta izvršavanja potrebno je pripremiti podatke i alocirati memoriju na grafičkoj kartici za ulazne i izlazne podatke (linije 23-27). Nakon alociranja memorije na grafičkoj kartici, potrebno je stvoriti objekt *cuda.Stream()* (linija 29). CUDA Stream objekt služi za asinkrono prebacivanje podataka na GPU i izvršavanje. Ovime smo završili pripremu za pokretanje zaključivanja modela. Zaključivanje modela se odvija pozivom funkcije *execute_async_v2* objekta *context* (linija 31). No prije nego provedemo zaključivanje potrebno je kopirati ulazne podatke na GPU (linija 30). Nakon zaključivanja potrebno je kopirati rezultat zaključivanja sa GPU-a na CPU kako bi mogli procesirati rezultat (linije 32 i 33). Kako bismo bili sigurni da je zaključivanje završeno i da je rezultat na CPU, koristimo sinkronizaciju pomoću objekta *stream* (linija 34). Nakon sinkronizacije možemo procesirati rezultat TensorRT stroja za zaključivanje.

```python
import tensorrt as trt
import numpy as np
import onnx
import pycuda.driver as cuda

# Priprema i izvoz modela identični kao linije 7-23 koda 4.1
...

# Inicijalizacija objekata i uvoz modela
builder = trt.Builder(trt.Logger(trt.Logger.WARNING))
config = builder.create_builder_config()
config.max_workspace_size = 1 << 32  # 4 GB
network = builder.create_network(1 <<
    int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))

parser = trt.OnnxParser(network, builder.logger)
with open(trt_model_path, 'rb') as model:
        parser.parse(model.read())

# Gradnja stroja za zaključivanje
engine = builder.build_engine(network, config)

#Priprema podataka za zaključivanje i zaključivanje
context = engine.create_execution_context()
input_data = np.array(image.cpu())
d_input = cuda.mem_alloc(1 * input_data.nbytes)
output_shape = (19, 128, 256)
output_size = int(np.prod(output_shape) *
    np.dtype(np.float32).itemsize)
d_output = cuda.mem_alloc(output_size)

stream = cuda.Stream()
cuda.memcpy_htod_async(d_input, input_data, stream)
context.execute_async_v2(bindings=[int(d_input),
    int(d_output)], stream_handle=stream.handle)
output_data = np.empty(output_shape, dtype=np.float32)
cuda.memcpy_dtoh_async(output_data, d_output, stream)
stream.synchronize()
```

Programski kod 4.2: Tijek rada (engl. *workflow*) TensorRT-a opisan u četiri koraka koristeći paket *tensorrt*. Izvoz modela se radi kao i u programskom kod 4.1.

5. Implementacija

U ovom poglavlju su opisane korištene tehnologije i implementacijski detalji eksperimenata provedenih u ovome radu.

5.1. Korištene tehnologije

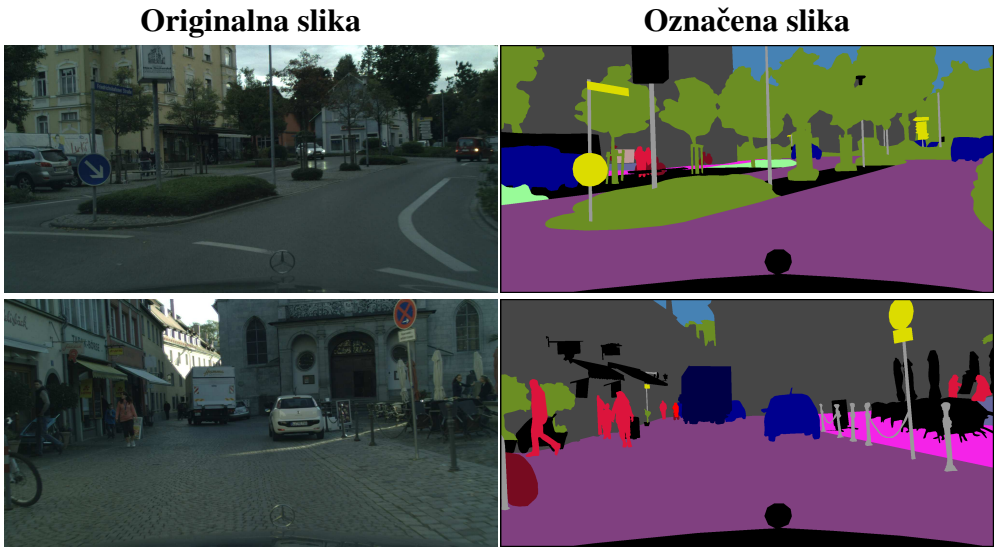
Svi eksperimenti su provedeni u programskom jeziku Python. Gradnja TensorRT stroja za zaključivanje i samo provođenje procesa zaključivanja su provedeni nad prethodno opisanim predtreniranim modelima MaskFormer i Mask2Former koji su implementirani koristeći programski okvir PyTorch i Detectron2, okvir za računalni vid. TensorRT stroj je izgrađen uz pomoć Python modula *onnx_tensorrt*. Modul *onnx_tensorrt* je modul koji koristi ONNX format modela za stvaranje TensorRT stroja za zaključivanje. Uspoređene su performanse modela s i bez stvaranja TensorRT stroja za zaključivanje. Model bez stvaranja TensorRT stroja za zaključivanje je napisan u Python-u i izvodi se koristeći PyTorch bez optimizacija. Performanse su uspoređene na problemu semantičke segmentacije i skupu podataka Cityscapes na kojem su modeli prethodno bili istrenirani.

5.1.1. Skup podataka Cityscapes

Skup podataka Cityscapes [6] je poznati skup podataka za razumijevanje gradske scene i semantičku segmentaciju. Sastoji se od slika uličnih scena snimljenih iz perspektive vozača automobila, visoke rezolucije i iz pedeset različitih gradova. Pruža različite scene i slike označene na razini piksela. Sadrži trideset razreda, no u praksi se često koristi smanjena verzija skupa podataka koji sadrži devetnaest razreda. Razredi su: cesta, pločnik, zgrada, zid, ograda, stup, bicikl, motor, vlak, bus, kamion, auto, semafor, prometni znak, vegetacija, teren, nebo, osoba, biciklist. Smanjeni broj razreda se koristi kako bi se modeli učili i fokusirali više na bitnije razrede poput ceste, pločnika, zgrada, vozila i pješaka. Uz slike visoke kvalitete, skup podataka Cityscapes nudi slike

dobivene kroz više mjeseci.

Cityscapes nudi 5000 slika označenih s visokom kvalitetom i 20000 slika označenih s nižom kvalitetom. Od 5000 slika označenih visokom kvalitetom, skup za treniranje sadrži 2975 slika, skup za validaciju 500 i skup za treniranje 1525 slika. Slike su rezolucije 1024x2048. Modeli su predtrenirani na ovom skupu podataka, a u sklopu ovog rada korišten je samo dio slika radi testiranja i usporedbe performansi i točnosti modela s i bez optimizacije zaključivanja okvirom TensorRT.



Slika 5.1: Primjer ulaznih podataka skupa podataka CityScapes (lijevo) i odgovarajuće označenih slika (desno).

5.2. Optimiranje modela MaskFormer koristeći radni okvir TensorRT

Programski kod 4.1 nam može poslužiti kao recept pri gradnji TensorRT stroja za zaključivanje za bilo koji model, pa tako i MaskFormer. Prije gradnje TensorRT stroja potrebno je učitati istrenirani model.

```
1 import pickle
2 import torch
3 import numpy
4 import onnx_tensorrt.backend as backend
5
6 model = Trainer.build_model(cfg)
7
8 state = pickle.load(open('model_final_38c00c.pkl', 'rb'))
9 for k, v in state["model"].items():
10         v = torch.tensor(v)
11         state["model"][k] = v
12 model.load_state_dict(state["model"])
13
14 model.to("cuda")
15 model.eval()
```

Programski kod 5.1: Priprema istreniranog modela za prebacivanje u ONNX format.

Priprema modela se obavlja u par koraka (programski kod 5.1). Prvo je potrebno stvoriti instancu modela u koji ćemo učitati istrenirani model (linija 6). Nakon toga je potrebno učitati službeni istrenirani model. Učitavanje službenog modela, odnosno njegovog stanja (engl. *model state dict*), obavlja se pomoću modula *pickle* (linija 8). Nakon učitavanja stanja modela potrebno je sve težine modela pretvoriti u PyTorch tenzore (linije 9-11) kako bi stanje istreniranog modela mogli učitati u našu instancu modela. Kada smo sve težine istreniranog modela prebacili u tenzore, funkcijom *load_state_dict* učitavamo težine istreniranog modela u PyTorch model s kojim smo spremni graditi TensorRT stroj. Funkcija *load_state_dict* učitava težine modela iz spremljenog rječnika (engl *dictionary*) stanja u instancu tog modela. Rječnik stanja je Python rječnik koji mapira ime parametara sa odgovarajućim vektorom težina modela.

Prije no što krenemo s prebacivanjem modela u ONNX format potrebno ga je prebaciti na grafičku karticu i u evaluacijski način rada (linije 14 i 15).

5.2.1. Prebacivanje modela u ONNX format

```
1 input_shape = (3,512,1024)
2 x = torch.ones(input_shape).to("cuda")
3
4 torch.out = model(x)
5 torch.onnx.export(
6         model,
7         x,
8         "model_name.onnx"
9         export_params=True,
10        opset_version=16,
11        do_constant_folding=True,
12        input_names=["input"],
13        output_names=["output"]
14 )
```

Programski kod 5.2: Prebacivanje modela u ONNX format.

Prebacivanje modela u ONNX format je prikazano programskim kodom 5.2. Prije no što se model prebaci u ONNX format, potrebno je napraviti unaprijedni prolaz s tenzorom dimenzija naših podataka. Taj rezultat se sprema u varijablu *torch_out*. Ovaj korak je vrlo bitan, jer nakon prebacivanja u ONNX format naš model očekuje dimenzije ulaza koje smo mu proslijedili sa unaprijednim prolazom (linija 4). Ako se modelu preda ulaz dimenzije različit od očekivanog (linija 1) doći će do greške u izvođenju.

Samo prebacivanje modela u ONNX format se provodi s već spomenutom funkcijom *torch_onnx_export*. Uz prethodno pisane parametre ove funkcije 4.1, u programskom kodu 5.2 se koriste dva nova parametra *export_params* i *do_constant_folding*. Parametar *export_params* predstavlja zastavicu za izvoz parametara modela, te se postavlja kada izvozimo istrenirani model. Drugi parametar *do_constant_folding* se odnosi na optimizaciju koju ONNX provodi pri kreiranju grafa. *Constant folding* smanjuje broj izračuna tako što ulaze koji su konstante zamijeni čvorovima konstantama (engl. *constant nodes*).

Model MaskFormer je prebačen u ONNX format bez pogrešaka prilikom izvođenja (engl. *runntime errors*), što znači da su svi izračuni koje MaskFormer provodi podržani s odgovarajućim ONNX operatorima.

5.2.2. Učitavanje modela iz ONNX formata i gradnja stroja

Učitavanje modela i gradnja stroja se radi kao i u programskom kodu 4.1. U ovoj fazi može doći do grešaka tijekom gradnje stroja. Naime, ONNX i TensorRT ne podržavaju isti skup operatora. Zbog toga definirani ONNX graf, kojeg smo dobili prebacivanjem modela u ONNX format, može sadržavati operatore koje TensorRT ne podržava i tako dolazi do grešaka. U slučaju MaskFormer-a nije bilo poteškoća pri gradnji TensorRT stroja za zaključivanje.

5.2.3. Zaključivanje modela u TensorRT-u

Nakon što smo izgradili stroj možemo početi s zaključivanjem. Zaključivanje se provodi

```
1 image = load_and_prepare_image("path/to/image.png")
2
3 output = engine.run(np.array(image))
4
5 output = np.argmax(output[0], axis=0)
6 plt.imsave("inference_result.png", output)
```

Programski kod 5.3: Zaključivanje modela u TensorRT-u i primjer obrade rezultata zaključivanja.

pozivom funkcije *run* nad strojem za zaključivanje koja na ulaz prima NumPy polje. Programski kod 5.3 prikazuje jedan primjer obrade rezultata zaključivanja. Kako bi dobili rezultat semantičke segmentacije u obliku slike pozivamo funkciju *argmax* nad rezultatom i spremamo rezultat u obliku slike (linije 5 i 6).

5.3. Mask2Former TensorRT

Prebacivanje modela Mask2Former u TensorRT se također obavlja u četiri koraka: izvoz modela u ONNX format, uvoz modela iz ONNX formata, gradnja stroja i zaključivanje. Izvoz modela, uvoz modela i zaključivanje su identični kao i isti koraci kod MaskFormer modela. Gradnja stroja Mask2Former modela je imala greške pri izvođenju, pa je bilo potrebno pronaći i modificirati dijelove koda koji su stvarali greške.

5.3.1. Gradnja TensorRT stroja za zaključivanje

Za prebacivanje modela u ONNX format i za gradnju stroja za zaključivanje je bilo potrebno modificirati dijelove programskog koda. Konkretno, dva dijela izvornog programskog koda su stvarali probleme.

Prvi dio koda koji je stvarao probleme je takozvano indeksiranje tenzorom. Programski kod 5.4 prikazuje dio koda s kojim se nije mogao izgraditi stroj za

```
attn_mask[torch.where(attn_mask.sum(-1) ==
  ↪  attn_mask.shape[-1])] = False
```

Programski kod 5.4: Indeksiranja tenzora tenzorom, gdje se tenzor kojim se indeksira računa tijekom indeksiranja. Linija koda koja je stvarala grešku pri gradnji stroja za zaključivanje.

zaključivanje. Ovaj dio koda se nalazi u transformer dekoder dijelu modela. Razložimo i objasnimo programski kod 5.4 u detalje. Tenzor *attn_mask* je boolean tenzor koji predstavlja masku pažnje s kojom mehanizam pažnje radi. Operacija *attn_mask.sum(-1)* računa sumu duž zadnje dimenzije *attn_mask* tenzora. Rezultat sume je broj pozitivnih elemenata duž zadnje dimenzije. Operacija *attn_mask.shape[-1]* vraća veličinu posljednje dimenzije tenzora *attn_mask*. Funkcija *torch.where* je PyTorch funkcija koja prima uvjet a vraća tenzor elemenata gdje je uvjet zadovoljen.

```
row_sums = attn_mask.sum(-1)
row_mask = row_sums == attn_mask.shape[-1]
row_mask = row_mask
attn_mask = torch.where(row_mask.unsqueeze(-1),
  ↪  torch.tensor(False).to(device), attn_mask)
```

Programski kod 5.5: Programski kod kojim se zaobilazi indeksiranje tenzora tenzorom prikazan programskim kodom 5.4

. Konkretno, *torch.where(attn_mask.sum(-1) == attn_mask.shape[-1])* vraća indekse gdje je suma svakog elementa tenzora *attn_mask* jednaka veličini zadnje dimenzije. Na kraju se elementi na vraćenim indeksima postavljaju na False.
Ova linija koda ignorira određene elemente maske pažnje kako bi se mehanizam pažnje fokusirao na bitnije informacije u ulazu.

Programski kod 5.5 prikazuje kod kojim se zaobilazi indeksiranje tenzora tenzorom, koje stvara probleme pri gradnji TensorRT stroja za zaključivanje, bez utjecaja na rezultate i performanse modela. Drugi dio koda koji je stvarao probleme se nalazi u piksel dekoder dijelu modela. Piksel dekoder kojeg Mask2Former koristi je Multi-Scale Deformable Attention Transformer (*MSDeformAttn*). Naime, dio implementacije piksel dekodera je napisan koristeći programski kod CUDA. Problem nastaje pri prebacivanju modela u ONNX format.

Općenito, kada se CUDA kod prebaciva u ONNX format pomoću funkcije *torch.onnx.export*, ONNX graf ne generira čvorove koji sadrže ONNX operacije. Drugim riječima, generira se graf koji ima samo ulazne i izlazne čvorove. Primjerice, kada bismo linearnu regresiju napisali koristeći CUDA-u i pokušali ju prebaciti u ONNX format, generirani graf bi izgledao kao slika 4.2, bez unutrašnji čvorova *MatMul* i *Add*. Razlog ovakvog ponašanja je što ONNX "ne zna" kako CUDA kod povezati sa bilo kojim od svojih podržanih operatora. Iako su *MatMul* i *Add* podržane ONNX operacija, kada su napisane s programskim jezikom CUDA, ONNX ih ne može pravilno povezati sa svojim operacijama. Kako bi se CUDA operator ispravno pretvorio u ONNX bilo bi potrebno napraviti naš ONNX operator i povezati ga s CUDA kodom. No ovaj pristup zahtjeva odlično poznavanje CUDA-e, a i programskog jezika C++. Programskim jezikom C++ se gradi omotač oko CUDA koda i onda se C++ omotač povezuje s Python kodom. Kada bismo uspjeli napisati naš ONNX operator i povezati ga s našim CUDA kodom uspjeli bismo generirati ONNX graf s unutarnjim čvorovima koji bi pozivali naše CUDA implementacije operacija. No, ostaje nam i problem izgradnje TensorRT stroja za zaključivanje. Naime, ONNX i TensorRT ne podržavaju iste operacije. Kada bi naš ONNX graf sadržavao naše *custom* CUDA operacije i kada bi s tim ONNX grafom pokušali izgraditi TensorRT stroj za zaključivanje, naišli bismo na isti problem kao i kod prebacivanja CUDA koda u ONNX format, to jest TensorRT "ne zna" interpretirati naše operacije u ONNX grafu. I ovaj problem ima teorijsko rješenje pisanja naših TensorRT operacija i povezivanja istih s našim CUDA operacijama u ONNX grafu. No, kao i kod pisanja naših operacija za ONNX, i ovaj pristup zahtijeva iznimno znanje programskog jezika C++. Zbog ovoga, prebačeni Mask2Former s implementacijom multi-scale-deformable pažnje koristeći CUDA-u u

ONNX format neće zaključivati pravilno.

Dakle, jedan pristup je pisanje naših *custom* ONNX operacija, povezivanje istih s multi-scale-deformable pažnja CUDA operacijom i stvaranje ONNX grafa. Naravno, morali bismo iste operacije napisati i za TensorRT kako bi se iz ONNX grafa mogao izgraditi pravilan TensorRT stroj za zaključivanje. Pošto izravno prebacivanje CUDA koda u ONNX format (preko sučelja programskog jezika Python) nije podržano, preostaje nam alternativni pristup. Alternativni pristup je napisati implementaciju operacije multi-scale-deformable pažnje direktno koristeći programski jezik Python. Na ovaj način, pisanje *custom* ONNX i TensorRT operacija nije potrebno. U ovome radu je napisana operacija multi-scale-deformable pažnje koristeći programski jezik Python i uspješno smo izgradili TensorRT stroj za zaključivanje.

Programski kod 5.6 prikazujem implementaciju operacije multi-scale-deformable pažnje u programskom jeziku Python.

```python
def ms_deform_attn_core_pytorch(value, value_spatial_shapes, sampling_locations, attention_weights):
        N_, S_, M_, D_ = value.shape
        _, Lq_, M_, L_, P_, _ = sampling_locations.shape
        value_list = value.split([H_ * W_ for H_, W_ in value_spatial_shapes], dim=1)
        sampling_grids = 2 * sampling_locations - 1
        sampling_value_list = []
        for lid_, (H_, W_) in enumerate(value_spatial_shapes):
                # N_, H_*W_, M_, D_ -> N_, H_*W_, M_*D_ -> N_, M_*D_, H_*W_ -> N_*M_, D_, H_, W_
                value_l_ = value_list[lid_].flatten(2).transpose(1, 2).reshape(N_*M_, D_, H_, W_)
                # N_, Lq_, M_, P_, 2 -> N_, M_, Lq_, P_, 2 -> N_*M_, Lq_, P_, 2
                sampling_grid_l_ = sampling_grids[:, :, :, lid_].transpose(1, 2).flatten(0, 1)
                # N_*M_, D_, Lq_, P_
                sampling_value_l_ = F.grid_sample(value_l_, sampling_grid_l_,
                mode='bilinear', padding_mode='zeros', align_corners=False)
                sampling_value_list.append(sampling_value_l_)
        # (N_, Lq_, M_, L_, P_) -> (N_, M_, Lq_, L_, P_) -> (N_, M_, 1, Lq_, L_*P_)
        attention_weights = attention_weights.transpose(1, 2).reshape(N_*M_, 1, Lq_, L_*P_)
        output = (torch.stack(sampling_value_list, dim=-2).flatten(-2) *
        attention_weights).sum(-1).view(N_, M_*D_, Lq_)
        return output.transpose(1, 2).contiguous()
```

Programski kod 5.6: Implementacija multi-scale deformable attention funkcije korstići radni okvir PyTorch.

.

6. Rezultati

U ovom poglavlju su prikazani, opisani i uspoređeni rezultati dobiveni provođenjem eksperimenata s modelima MaskFormer i Mask2Former.

Rezultati su provedeni i dobiveni na NVIDIA GeForce RTX 3090 grafičkoj kartici i ugradbenom uređaju NVIDIA Jetson Xavier. Korišten je skup podataka Cityscapes. Rezultati su dobiveni na slikama rezolucije 128x256, 256x512 i 512x1024. Kao rezultati su mjereni ukupna brzina zaključivanja na 180 slika, vrijeme zaključivanja po slici, propusnost modela i usporedba rezultata semantičke segmentacije između TensorRT stroja i PyTorch modela. Za izgradnju TensorRT stroja korišten je paket *onnx_tensorrt*. Vizualni rezultati usporedbe su prikazani samo za rezultate dobivene na grafičkoj kartici, no isti rezultati vrijede i za ugradbeni uređaj Jetson.

Programski kod 6.1 prikazuje kod za dobivanje rezultata brzine zaključivanja i propusnosti modela. Funkcija na ulaz prima model, Numpy polje ulaznih podataka, te broj iteracija zagrijavanja modela. Zagrijavanje modela se radi zbog prilagodbe izračuna modela za ulazne podatke i za optimiranje rada grafičke kartice (linije 4 do 7). Nakon zagrijavanja modela dolazi iterativno predavanje slika modelu i računanje vremena zaključivanja (linije 10 do 19). Kada se sve ulazne slike predaju modelu računa se ukupno vrijeme zaključivanja i propusnost modela (linije 20 do 22).

```
1  def benchmark(model, images, nwarmup=50):
2          print("Warm up ...")
3
4          with torch.no_grad():
5                  for i in range(nwarmup):
6                          features = model.run(images[i])
7          torch.cuda.synchronize()
8
9          timings = []
10         with torch.no_grad():
11                 for i in range(0, len(images)):
12                         start_time = time.time()
13                         pred_loc = model.run(images[i])
14                         torch.cuda.synchronize()
15                         end_time = time.time()
16                         timings.append(end_time - start_time)
17                         if i % 10 == 0:
18                                 print('Iteration %d/%d, avg
                                   ↪  batch time %.2f ms' % (i,
                                   ↪  len(images),
                                   ↪  np.mean(timings) * 1000))
19
20         print("Total time: %s seconds" % (np.sum(timings)))
21         print("Input shape:", images[0].shape)
22         print('Average throughput: %.2f images/second' %
           ↪  (len(images) / (np.mean(timings)*len(images))))
```

Programski kod 6.1: Programski kod za dobivanje rezultata brzine zaključivanja i propusnosti modela. Prikazani programski kod radi s TensorRT strojem za zaključivanje. Varijabla *images* je Numpy polje. Kako bi prikazani kod radio s Pytorch modelom potrebno je kao varijablu *images* predati PyTorch tenzor, te na linijama 6 i 13 *model.run(images[i])* zamijeniti s linijom *model(images[i])*.

6.1. Rezultati modela MaskFormer

Ovaj odjeljak opisuje i uspoređuje rezultate MaskFormer PyTorch modela i TensorRT stroja za zaključivanje dobivenih na grafičkoj kartici NVIDIA GeForce RTX 3090, te opisuje i uspoređuje rezultate istih modela dobivenih na ugradbenom uređaju Jetson Xavier.

6.1.1. Rezultati na NVIDIA GeForce RTX 3090 GPU

U ovom dijelu su prikazani i uspoređeni rezultati MaskFormer PyTorch modela i TensorRT stroja za zaključivanje na grafičkoj kartici NVIDIA GeForce RTX 3090, te su isti prikazani u tablici 6.1.

Rezultati su dobiveni na tri različite rezolucije slika, te se povećanjem rezolucije povećava vrijeme zaključivanja.

MaskFormer TensorRT stroj za zaključivanje znatno ubrzava vrijeme zaključivanja modela za rezolucije 128x256 i 256x512 u odnosu na PyTorch model. Međutim, za rezoluciju 512x1024 nema ubrzanja. Moguće objašnjenje je da zadana vrijednost maksimalne veličine radnog prostora nije dovoljna za optimalnu izgradnju TensorRT stroja za zaključivanje. Kako programski nismo uspjeli promijeniti maksimalnu veličinu radnog prostora, pokušali smo koristiti paket *tensorrt* za izgradnju TensorRT stroja za zaključivanje na rezoluciji 512x1024 i dobili smo osjetno ubrzanje (* redak u tablici 6.1). Važno je napomenuti da smo pokrenuli zaključivanje i na drugim rezolucijama koristeći paket *tensorrt*, ali zbog minimalnih oscilacija u ubrzanju u odnosu na paket *onnx_tensorrt*, ti rezultati nisu prikazani u ovome radu.

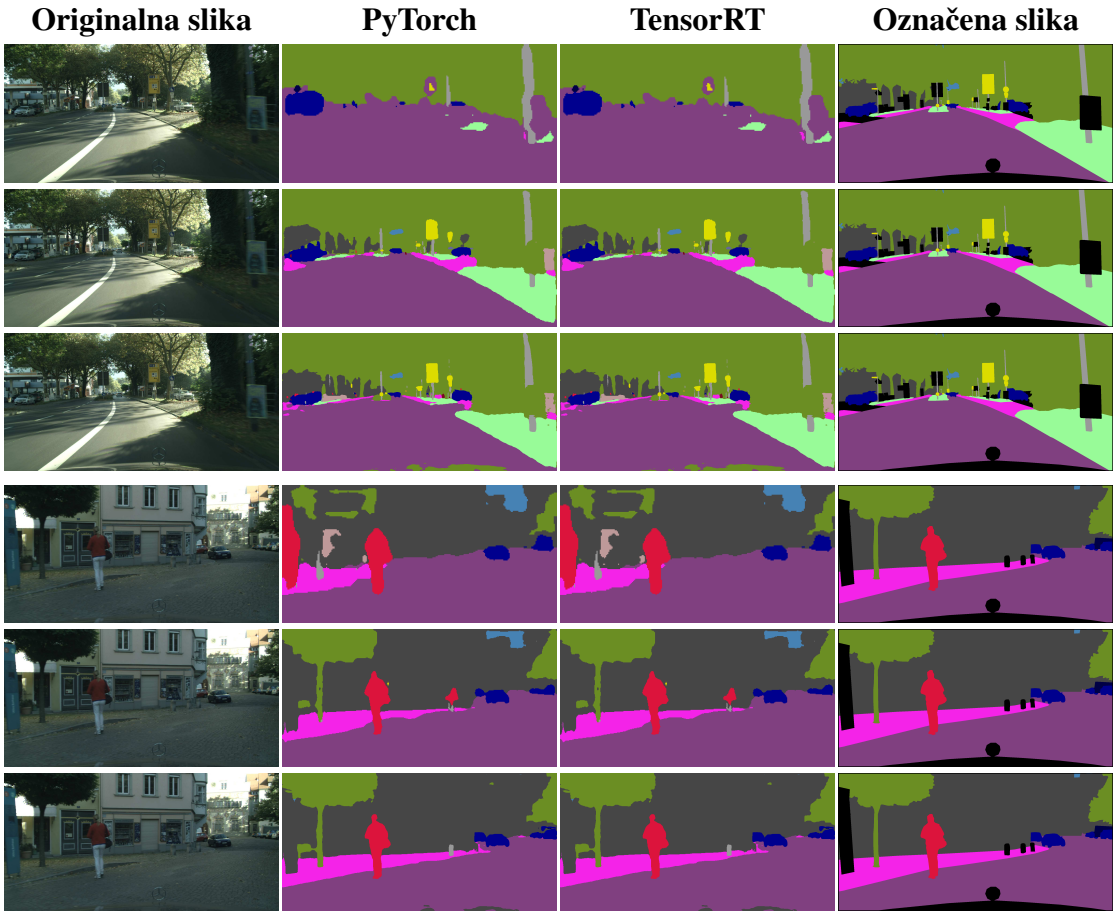
Dimenzije	Vrijeme zaključivanja	Propusnost	Vrijeme zaključivanja po slici	TensorRT
128x256	3.34 sekundi	53.93 slika/sekundi	18.55 ms	Ne
256x512	3.45 sekundi	52.21 slika/sekundi	19.15 ms	Ne
512x1024	4.33 sekundi	41.56 slika/sekundi	24.06 ms	Ne
128x256	0.868 sekundi	207.44 slika/sekundi	4.82 ms	Da
256x512	1.611 sekundi	111.72 slika/sekundi	8.95 ms	Da
512x1024	4.40 sekundi	40.92 slika/sekundi	24.44 ms	Da
512x1024*	3.51 sekundi	51.28 slika/sekundi	19.50 ms	Da

Tablica 6.1: Rezultati i usporedba TensorRT stroja za zaključivanje modela MaskFormer i modela MaskFormer. Po redcima su prikazani rezultati pojedinih rezolucija slika Po stupcima su prikazani rezultati vremena zaključivanja, propusnosti, vremena zaključivanja po slici i (ne)korištenje TensorRT stroja za zaključivanje. Zadnji redak (*) predstavlja rezultat zaključivanja izgradnjom TensorRT stroja pomoću paketa *tensorrt*.

Na slici 6.1 su vidljivi vizualni rezultati MaskFormer Pytorch modela i TensorRT stroja za zaključivanje. Lijevo se nalazi originalna slika koja je ujedno i ulaz modela. U sredini (dva stupca) se nalaze rezultati PyTorch modela i TensorRT stroja za zaključivanje, dok su desno označene slike. Iz rezultata je vidljivo da TensorRT stroj daje identične rezultate kao i PyTorch model na svim rezolucijama slike. Također, povećanjem rezolucije slike rezultati su bolji za oba modela.

	128x256	256x512	512x1024
Preciznost po pikselu	1.0	1.0	1.0

Tablica 6.2: Prikaz odnosa među rezultatima MaskFormer PyTorch modela i TensorRT stroja za zaključivanje (slika 6.1). Za računanje koeficijenta jednakosti korištena je preciznost po pikselu (engl. *pixel-wise accuracy*).



Slika 6.1: Vizualna reprezentacija rezultata semantičke segmentacije MaskFormer PyTorch modela i TensorRT stroja za zaključivanje. Lijevo je originalna slika, u sredini (dva stupca) se nalaze rezultati PyTorch modela i TensorRT stroja i desno su označene slike. Po redcima su prikazane rezolucije slika, 128x256, 256x512 i 512x1024, redom.

Kako bi se zaista uvjerili da TensorRT stroj daje identične rezultate kao i PyTorch model, izračunat je koeficijent sličnosti pomoću preciznosti po pikselu (engl. *pixel-wise accuracy*). Preciznost po pikselu je metrika koja računa preciznost predikcije modela na razini piksela, to jest uspoređuju se pikseli predikcije i stvarnih oznaka. Ovdje je preciznost po pikselu korištena radi usporedbe predikcije TensorRT stroja

i PyTorch modela. U tablici 6.2 su vidljive izračunate preciznosti po pikselu za sve rezolucije. Preciznost po pikselu za sve rezolucije iznosi 1 što nam govori da TensorRT stroj za zaključivanje daje identične rezultate kao i PyTorch model. Drugim riječima, TensorRT stroj ubrzava zaključivanje bez ikakvih gubitaka na točnost.

6.1.2. Rezultati na ugradbenom uređaju NVIDIA Jetson Xavier

U ovom dijelu su prikazani i uspoređeni rezultati MaskFormer PyTorch modela i TensorRT stroja za zaključivanje na ugradbenom uređaju NVIDIA Jetson Xavier, te su isti prikazani u tablici 6.3.

Kao i za RTX 3090 GPU, rezultati su dobiveni na tri različite rezolucije slika, te se povećanjem rezolucije povećava vrijeme zaključivanja.

Dimenzije	**Vrijeme zaključivanja**	**Propusnost**	**Vrijeme zaključivanja po slici**	**TensorRT**
128x256	9.22 sekundi	17.24 slika/sekundi	57.65 ms	Ne
256x512	20.91 sekundi	7.60 slika/sekundi	131.90 ms	Ne
512x1024	68.299 sekundi	2.33 slika/sekundi	429.40 ms	Ne
128x256	6.65 sekundi	23.90 slika/sekundi	41.81 ms	Da
256x512	17.25 sekundi	9.22 slika/sekundi	108.53 ms	Da
512x1024	60.288 sekundi	2.64 slika/sekundi	378.97 ms	Da

Tablica 6.3: Rezultati i usporedba TensorRT stroja za zaključivanje modela MaskFormer i modela MaskFormer na ugradbenom uređaju Jetson. Po redcima su prikazani rezultati pojedinih rezolucija slika. Po stupcima su prikazani rezultati vremena zaključivanja, propusnosti, vremena zaključivanja po slici i (ne)korištenje TensorRT stroja za zaključivanje.

Iz tablice 6.3 je vidljivo da TensorRT stroj za zaključivanje daje osjetno ubrzanje. Vizualna reprezentacija rezultata je identična kao i za rezultate na GeForce 3090 GPU, te su vidljivi na slici 6.1.

6.2. Rezultati modela Mask2Former

U ovom odjeljku su opisani i uspoređeni rezultati Mask2Former PyTorch modela i TensorRT stroja za zaključivanje dobivenih na grafičkoj kartici NVIDIA GeForce RTX 3090, te opisuje i uspoređuje rezultate istih modela dobivenih na ugradbenom uređaju Jetson Xavier.

6.2.1. Rezultati na NVIDIA GeForce RTX 3090 GPU

U ovom dijelu rada su prikazani i uspoređeni rezultati Mask2Former PyTorch modela i TensorRT stroja za zaključivanje. Dodatno, prikazani su i komentirani vizualni rezultati TensorRT stroja s implementacijom multi-scale-deformable pažnje u CUDA-i.

Dimenzije	**Vrijeme zaključivanja**	**Propusnost**	**Vrijeme zaključivanja po slici**	**TensorRT**
128x256	4.73 sekundi	38.00 slika/sekundi	26.32 ms	Ne
256x512	5.32 sekundi	33.82 slika/sekundi	29.57 ms	Ne
512x1024	9.30 sekundi	19.35 slika/sekundi	51.67 ms	Ne
128x256	0.927 sekundi	194.26 slika/sekundi	5.15 ms	Da
256x512	2.06 sekundi	87.18 slika/sekundi	11.47 ms	Da
512x1024	6.30 sekundi	28.55 slika/sekundi	35.01 ms	Da

Tablica 6.4: Rezultati i usporedba TensorRT stroja za zaključivanje modela Mask2Former i modela Mask2Former. Po redcima su prikazani rezultati pojedinih rezolucija slika. Po stupcima su prikazani rezultati vremena zaključivanja, propusnosti, vremena zaključivanja po slici i (ne)korištenje TensorRT stroja za zaključivanje.

Tablica 6.4 prikazuju rezultate vremena zaključivanja, propusnosti i vremena zaključivanja po jednoj slici za Mask2Former PyTorch model, odnosno Mask2Former TensorRT stroj za zaključivanje. Rezultati su dobiveni na 128x256, 256x512 i 512x1024 rezolucijama slika.

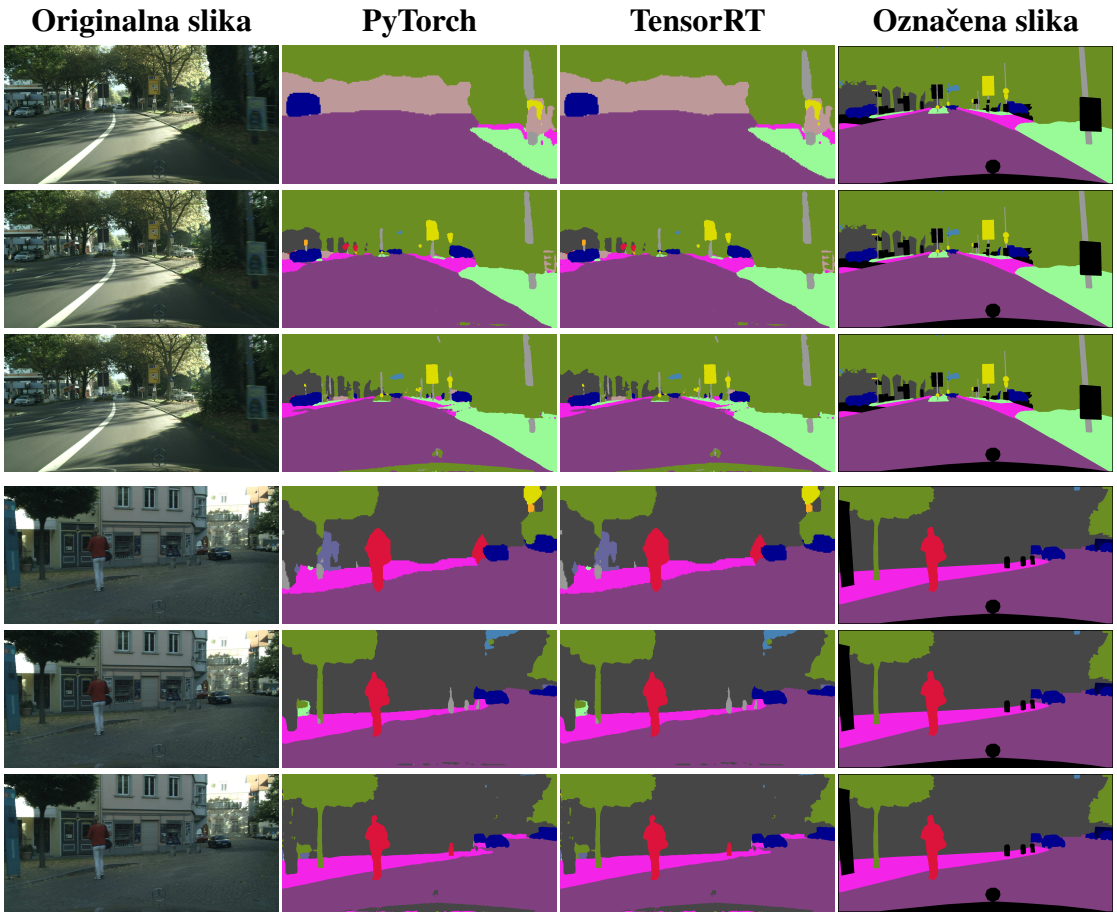
Na svim rezolucijama Mask2Former TensorRT stroj ubrzava zaključivanje. Na rezoluciji 128x256 i 256x512 ubrzanje je čak 5, odnosno skoro 3 puta veće. Za rezoluciju 512x1024 se i dalje vidno osjeti ubrzanje koje TensorRT nudi. Očekivano je da se povećanjem rezolucije ulaznog podatka smanji ubrzanje zaključivanja TensorRT stroja. Povećanjem rezolucije ulaza se povećava broj i kompleksnosti izračuna modela, te memorijski otisak modela, a samim time i efikasnost optimizacije TensorRT-a.

	128x256	256x512	512x1024
Preciznost po pikselu	1.0	1.0	1.0

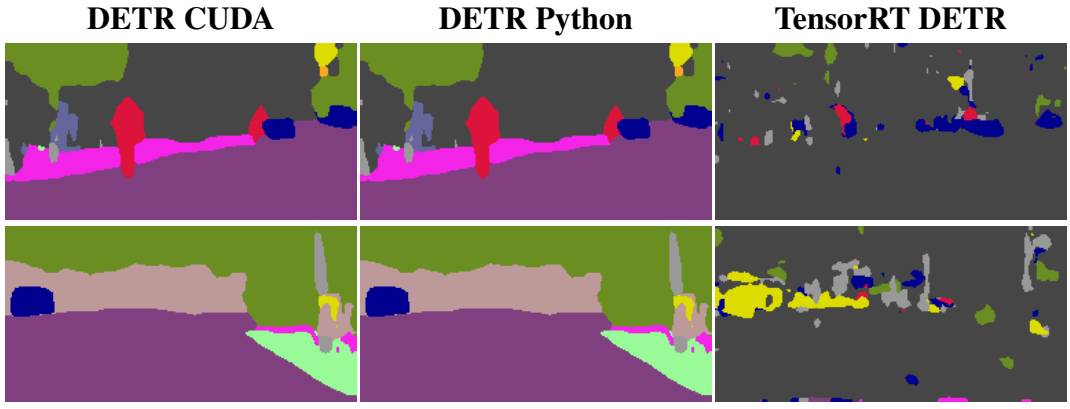
Tablica 6.5: Prikaz odnosa među rezultatima Mask2Former PyTorch modela i TensorRT stroja za zaključivanje (6.2). Za računanje koeficijenta jednakosti korištena je preciznost po pikselu (engl. *pixel-wise accuracy*).

Na slici 6.2 su prikazani vizualni rezultati semantičke segmentacije modela na različitim rezolucijama. Rezolucije slika su poredane po redcima. Lijevi stupac predstavlja ulaz u model, u sredini (dva stupca) nalaze se rezultati PyTorch modela i TensorRT stroja za zaključivanje, u desnom stupcu se nalaze označene slike. Kao i kod MaskFormer modela, TensorRT stroj daje identične rezultate, na svim rezolucijama, kao i PyTorch model. Da su rezultati TensorRT stroja identični rezultatima PyTorch modela potvrđuje tablica 6.5. Tablica 6.5 prikazuje razliku među rezultatima modela koristeći preciznost po pikselu. Preciznost po pikselu je za sve rezolucije 1, što potvrđuje da TensorRT stroj za zaključivanje ubrzava zaključivanje bez gubitaka točnosti.

Na slici 6.3 su prikazani rezultati semantičke segmentacije s implementacijom multi-scale-deformable pažnje u CUDA-i, odnosno u Pythonu. Lijevo je prikazan rezultat PyTorch modela s implementacijom u CUDA-i, a u sredini je rezultat PyTorch modela s implementacijom u Pythonu. Ako izračunamo preciznost po pikselima ova dva rezultata dobijemo iznos od 1, što znači da Python implementacija multi-scale-deformable pažnje ne utječe na točnost zaključivanja modela. Desno na slici je rezultat TensorRT stroja s implementacijom multi-scale-deformable pažnje u CUDA-i. Rezultat je ovakav iz razloga što se cijeli dio CUDA koda nije prebacio u ONNX format, te je model proveo zaključivanje bez tog dijela koda. Ovi rezultati su dobiveni na rezoluciji 128x256, no i na većim rezolucijama ishod je identičan.



Slika 6.2: Vizualna reprezentacija rezultata semantičke segmentacije Mask2Former PyTorch modela i TensorRT stroja za zaključivanje. Lijevo je originalna slika, u sredini (dva stupca) se nalaze rezultati PyTorch modela i TensorRT stroja i desno su označene slike. Po redcima su prikazane rezolucije slika, 128x256, 256x512 i 512x1024, redom.



Slika 6.3: Vizualna reprezentacija rezultata semantičke segmentacije PyTorch modela s implementacijom multi-scale-deformable pažnje u CUDA-i (lijevo), implementacijom multi-scale-deformable pažnje u Pythonu (sredina) i rezultati TensorRT stroja s implementacijom multi-scale-deformable pažnje u CUDA-i (desno).

6.2.2. Rezultati na ugradbenom uređaju Jetson Xavier

U ovom dijelu su prikazani i uspoređeni rezultati Mask2Former PyTorch modela i TensorRT stroja za zaključivanje na ugradbenom uređaju NVIDIA Jetson Xavier, te su isti prikazani u tablici 6.6.

Dimenzije	**Vrijeme zaključivanja**	**Propusnost**	**Vrijeme zaključivanja po slici**	**TensorRT**
128x256	18.01 sekundi	8.83 slika/sekundi	114.14 ms	Ne
256x512	40.13 sekundi	3.96 slika/sekundi	251.99 ms	Ne
512x1024	- sekundi	- slika/sekundi	- ms	Ne
128x256	10.63 sekundi	14.95 slika/sekundi	66.90 ms	Da
256x512	33.47 sekundi	4.75 slika/sekundi	210.58 ms	Da
512x1024	- sekundi	- slika/sekundi	- ms	Da

Tablica 6.6: Rezultati i usporedba TensorRT stroja za zaključivanje modela Mask2Former i modela Mask2Former na ugradbenom uređaju Jetson. Po redcima su prikazani rezultati pojedinih rezolucija slika. Po stupcima su prikazani rezultati vremena zaključivanja, propusnosti, vremena zaključivanja po slici i (ne)korištenje TensorRT stroja za zaključivanje. Zbog memorijskih ograničenja uređaja Jetson rezultati na rezoluciji 512x1024 nisu dobiveni.

I za model Mask2Former TensorRT daje osjetno ubrzanje zaključivanja. No, za ugradbeni uređaj Jetson nismo dobili rezultate na rezoluciji 512x1024 zbog memorijskih ograničenja uređaja. Vizualni rezultati su vidljivi na slici 6.2.

7. Zaključak

Modeli temeljeni na pažnji su postali glavni odabir pri rješavanju problema obrade prirodnog jezika, no danas pronalaze primjenu i u računalnom vidu. Kroz model Vision Transformer korištenje pažnje u računalnom vidu je postala stvarnost. Dalji napredak i primjena modela temeljenih na pažnji dovela je do razvoja kompetitivnih modela koji daju odlične rezultate. Primjer takvih modela su MaskFormer i Mask2Former. Oba modela koriste pažnju za rješavanje problema računalnog vida, poput semantičke i panoptičke segmentacije.

Kako modeli postaju kompleksniji i veći, povećava se i broj izračuna koje provode. Povećanjem izračuna se povećava i vrijeme zaključivanja modela. Jedan način ubrzanja proces zaključivanja je tehnologija NVIDIA TensorRT. Ova tehnologija omogućava ubrzavanje zaključivanja modela na NVIDIA grafičkim karticama, bez gubitka točnosti.

Kroz ovaj rad se pokazalo ubrzavanje zaključivanja modela MaskFormer i Mask2Former koristeći TensorRT. Pokazano je da TensorRT značajno ubrzava zaključivanje uz minimalne, ili nikakve, gubitke točnosti. Međutim i dalje postoje ograničenja. Naime, koristeći Python, TensorRT ne može efektivno izgraditi stroj za zaključivanje ako je dio modela napisan u CUDA-i.

Za budući rad bi se mogao istražiti ovaj problem i pokušati pronaći način kako izgraditi TensorRT stroj bez da se implementacije CUDA koda moraju implementirati direktno u Python kodu.

LITERATURA

[1] Dosovitskiy A., Beyer L., Kolesnikov A., Weissenborn D., Zhai X., Unterthiner T., Dehghani M., Minderer M., Heigold G., Gelly S., Uszkoreit J., i Houlsby N. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. URL https://arxiv.org/abs/2010.11929.

[2] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez N. A., Kaiser L., i Polosukhin I. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

[3] Cheng B., Schwing G. A., i Kirillov A. Per-pixel classification is not all you need for semantic segmentation, 2021. URL https://arxiv.org/abs/2107.06278.

[4] Cheng B., Misra I., Schwing G. A., Kirillov A., i Girdhar R. Masked-attention mask transformer for universal image segmentation, 2022. URL https://arxiv.org/abs/2112.01527.

[5] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, i Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs, 2017. URL https://arxiv.org/abs/1606.00915.

[6] Cityscapes. Skup podataka cityscapes. URL https://www.cityscapes-dataset.com/.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, i Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, i Jian Sun. *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition*, stranica

346–361. Springer International Publishing, 2014. ISBN 9783319105789. doi: 10.1007/978-3-319-10578-9_23. URL http://dx.doi.org/10.1007/978-3-319-10578-9_23.

[9] He K., Zhang X., Ren S., i Sun J. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

[10] Philipp Krähenbühl i Vladlen Koltun. Efficient inference in fully connected crfs with gaussian edge potentials, 2012. URL https://arxiv.org/abs/1210.5644.

[11] Tsung-Yi Lin, Dollár P., Girshick R., He K., Hariharan B., i Belongie S. Feature pyramid networks for object detection, 2017. URL https://arxiv.org/abs/1612.03144.

[12] Jonathan Long, Evan Shelhamer, i Trevor Darrell. Fully convolutional networks for semantic segmentation, 2015. URL https://arxiv.org/abs/1411.4038.

[13] Carion N., Massa F., Synnaeve G., Usunier N., Kirillov A., i Zagoruyko S. End-to-end object detection with transformers, 2020. URL https://arxiv.org/abs/2005.12872.

[14] Hyeonwoo Noh, Seunghoon Hong, i Bohyung Han. Learning deconvolution network for semantic segmentation, 2015. URL https://arxiv.org/abs/1505.04366.

[15] NVIDIA. Nvidia tensorrt, . URL https://developer.nvidia.com/tensorrt.

[16] NVIDIA. Nvidia tensorrt documentation, . URL https://docs.nvidia.com/deeplearning/tensorrt/quick-start-guide/index.html#ecosystem.

[17] NVIDIA. Nvidia tensorrt workflow and ecosystem, . URL https://docs.nvidia.com/deeplearning/tensorrt/quick-start-guide/index.html#ecosystem.

[18] ONNX. Open neural network exchange. URL https://onnx.ai/onnx/operators/index.html.

[19] Alec Radford, Karthik Narasimhan, Tim Salimans, i Ilya Sutskever. Improving language understanding by generative pre-training. 2018. URL https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.

[20] Ilya Sutskever, Oriol Vinyals, i Quoc V. Le. Sequence to sequence learning with neural networks, 2014. URL https://arxiv.org/abs/1409.3215.

[21] Zhu X., Su W., Lu L., Li B., Wang X., i Dai J. Deformable detr: Deformable transformers for end-to-end object detection, 2021. URL https://arxiv.org/abs/2010.04159.

Optimiranje dubokih modela sa slojevima pažnje

Sažetak

U ovom radu opisan je mehanizam pažnje kao moguća zamjena za konvolucijske modele te njegova uspješna primjena u području računalnog vida. Detaljno je opisan model Vision Transformer, koji je među prvima koristio mehanizam pažnje za rješavanje problema klasifikacije. Također su detaljnije objašnjeni modeli MaskFormer i Mask2Former. Oba modela na efikasan i jedinstven način rješavaju višestruke izazove računalnog vida koristeći mehanizam pažnje. U radu je opisan NVIDIA TensorRT, razvojni alat za optimizaciju istreniranih modela i ubrzanje postupka zaključivanja, te je predstavljen ONNX format za pohranu dubokih modela. Opisan je postupak stvaranja TensorRT stroja koristeći programski jezik Python i okvir PyTorch. Modeli MaskFormer i Mask2Former su optimizirani uz pomoć NVIDIA TensorRT-a, a rezultati su uspoređeni i prikazani između PyTorch modela i TensorRT strojeva.

Ključne riječi: Pažnja, MaskFormer, Mask2Former, TensorRT, optimizacija, ONNX

Optimizing deep models based on attention layers

Abstract

This paper describes the attention mechanism as a potential alternative to convolutional models and its successful application in computer vision problems. The Vision Transformer, one of the first models to employ the attention mechanism for classification tasks, is explained in detail. The MaskFormer and Mask2Former models are also closely described. Both models effectively and uniquely address multiple computer vision challenges by using the attention mechanism. NVIDIA TensorRT, an SDK for optimizing trained models and accelerating inference process, is explained along ONNX format for storing deep models. The process of creating TensorRT engine using Python and the PyTorch framework is also described. The MaskFormer and Mask2Former models are optimized using NVIDIA TensorRT, and a comparison of results between the PyTorch models and TensorRT engines is presented.

Keywords: Attention, MaskFormer, Mask2Former, TensorRT, optimization, ONNX