SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2756

GUSTA PREDIKCIJA PRIMJENOM SLOJEVA PAŽNJE

Leo Pleše

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2756

GUSTA PREDIKCIJA PRIMJENOM SLOJEVA PAŽNJE

Leo Pleše

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 2756

Pristupnik:	**Leo Pleše (0036505670)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	prof. dr. sc. Siniša Šegvić

Zadatak: **Gusta predikcija primjenom slojeva pažnje**

Opis zadatka:

Razumijevanje scena važan je zadatak računalnog vida s mnogim zanimljivim primjenama. U posljednje vrijeme vrlo zanimljive rezultate postižu modeli utemeljeni na slojevima pažnje. Ipak, jedna od glavnih prepreka prema boljoj generalizaciji i dalje je memorijska složenost algoritma backprop. Zadaća ovog rada je provesti objektivnu usporedbu s konvolucijskim modelima s obzirom na generalizacijsku moć te učinkovitost učenja i zaključivanja. U okviru rada, potrebno je odabrati okvir za automatsku diferencijaciju te upoznati biblioteke za rukovanje matricama i slikama. Proučiti i ukratko opisati postojeće algoritme za gustu predikciju. Odabrati slobodno dostupni skup slika te oblikovati podskupove za učenje, validaciju i testiranje. Uhodati učenje i zaključivanje. Vrednovati naučene modele te prikazati postignutu točnost i učinkovitost. Predložiti prikladne smjerove budućeg rada. Radu priložiti izvorni i izvršni kod razvijenih postupaka, ispitne slijedove i rezultate, uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 27. lipnja 2022.

Zahvaljujem se Profesoru Dr. Sc. Siniši Šegviću na velikoj pomoći pri izradi ovog rada u obliku niza savjeta, prijedloga, ideja i smjernica koje mi je dao kako bi ovaj rad bio što bolji. Hvala i g. Mateju Grciću, Ivanu Grubišiću i Josipu Šariću na vrlo korisnoj korespondenciji tijekom izrade rada.

Sadržaj

Uvod 1

1. Elementi dubokih modela za gustu predikciju 3

1.1. Gusta predikcija 3

1.2. Konvolucijski modeli 3

1.3. Transformerski modeli 8

2. Metode za semantičku segmentaciju 17

2.1. Transformerski modeli za semantičku segmentaciju 17

2.1.1. Segmenter 17

2.1.2. Transformer SWIN 19

2.2. Konvolucijski modeli za semantičku segmentaciju 23

2.2.1. SwiftNet 23

3. Programska izvedba i vanjske biblioteke 28

3.1. Vanjske biblioteke 28

3.2. Programska izvedba 28

3.2.1. Projekt: SwiftNet 29

3.2.2. Projekt: Segmenter 38

3.2.3. Projekt: SWIN 39

3.2.4. Projekt: Ansambli i arhitektura SwiftNet-Segmenter 40

4. Podatkovni skupovi 45

4.1. PASCAL Context 45

4.2. ADE20K 45

4.3. Cityscapes 46

5. Eksperimenti 47

5.1. Uvodni eksperimenti 47

5.1.1. Eksperimenti na skupu PASCAL Context 47

5.1.2. Eksperimenti na skupu Cityscapes 49
5.2. Usporedno vrednovanje na skupu Cityscapes 49
5.3. Usporedno vrednovanje na skupu ADE20K 59
5.4. Ansambli SwiftNeta i Segmentera 61
5.5. Združeno učenje Swiftneta i Segmentera 62
5.6. Združeno učenje piramide Segmentera 66
5.7. Interpretiranje odluka Segmentera 69
5.8. Kvalitativni eksperimenti 77
Zaključak 84
Literatura 86

Uvod

Duboko učenje dominantno je polje unutar strojnog učenja koje je glavni pristup razvoju umjetne inteligencije. Ključne primjene dubokog učenja su u računalnom vidu i obradi prirodnog jezika jer su razumijevanje slika i ljudskog jezika zadaci koji su čovjeku lagani, ali umjetno inteligentnom agentu predstavljaju jedne od najvećih izazova.

Unutar računalnog vida ima više razina razumijevanja slika tj. osnovnih zadataka: klasifikacija, detekcija i semantička segmentacija objekata na slikama. Cilj klasifikacije je pridijeliti jednu klasu svakoj slici, dok je cilj detekcije i semantičke segmentacije provesti lokalizaciju svih objekata na slici i svakom objektu pridijeliti određenu klasu. U detekciji se objekti pritom označavaju oznakom klase i opisanim pravokutnikom koji pristaje uz objekt, dok semantička segmentacija ide korak dalje u lokalizaciji označavanjem svakog piksela, zbog čega se naziva i zadatkom guste predikcije. Gusta predikcija je jedan od najkompleksnijih zadataka računalnog vida i ovaj rad se njime bavi. Cilj ovog rada je pokazati potencijal transformera – modela koji već duže dominiraju poljem obrade prirodnog jezika, a prije nekoliko godina postali i jednim od dominantnih pristupa u računalnom vidu. Taj potencijal transformerskih modela iskazat ćemo u vidu generalizacijske moći i performansi učenja i zaključivanja modela, također dati prikladne vizualizacije, i uz to usporediti transformere s konvolucijskim modelima. Usporedbom s konvolucijskim modelima koji su već tradicionalna vrsta dubokih modela u računalnom vidu, pokazat ćemo mogu li transformerski modeli dostići ili i prestići konvolucijske. Dodatno, predstavit ćemo nekoliko ansambala konvolucijskog i transformerskog modela te hibridni model koji ih spaja. Razvoj hibridnih modela i njihovo združeno učenje je također velik izazov jer kombinira dva dominantna tipa dubokih modela i time ima možda i još veći potencijal od jednog transformerskog ili konvolucijskog modela pojedinačno.

Konkretno, transformerski modeli koje proučavamo su Segmenter [1] i SWIN [2]. Segmenter je transformer koji je dizajniran za semantičku segmentaciju, a SWIN je transformer koji ima visoke performanse kako u klasifikaciji i detekciji tako i u semantičkoj segmentaciji. Od konvolucijskih modela odabrali smo piramidalni SwiftNet [3] koji je jedan od vodećih modela za gustu predikciju.

Spomenimo još modele na kojima počivaju modeli kojima se bavimo.

Od transformerskih modela, ideju mehanizma pažnje donosi rad [4] kojom je tzv. pokušao riješiti problem kodirana cijele rečenice u vektor fiksne duljine tako što je uveo pažnju. Pažnja je omogućila da model kodira samo one riječi iz rečenice koje su relevantne za predikciju ciljne riječi što je znatno učinkovitiji pristup uz usporedive performanse strojnog prevođenja. Začetak modela transformera donosi rad [5] – transformer dizajnira kao model tipa koder-dekoder koji se zasniva na mehanizmu pažnje. Provodi eksperimente strojnog prevođenja na kojima postiže odlične rezultate i od tad su transformeri postali dominantnim smjerom razvoja dubokih modela u obradi prirodnog jezika, uz povratne modele (RNN) koji su i se i dalje razvijali (npr. arhitektura LSTM [6] koja je kasnije razvijena). Nekoliko godina poslije, u računalnom vidu javio se ViT [7] koji je popularizirao modele zasnovane na pažnji u računalnom vidu. ViT je pokazao da se transformerskim modelom bez konvolucijskih slojeva mogu dobiti kompetitivni rezultati. Jedna od glavnih ideja ViT-a koje su omogućile primjenu transformera u klasifikaciji slika je podjela slike u regije koje su umjesto piksela osnovna jedinica za izračun pažnje, čime su dobiveni transformeri prihvatljive učinkovitosti. Nakon ViT-a javio se niz radova kao njegova poboljšanja u različitim aspektima te jedan od glavnih je bio DeiT [8] koji je primijenio niz augmentacija podataka i tehniku destilacije znanja gdje model učenik (transformer) uči od modela učitelja (konvolucijski model). Javili su se i radovi koji smanjuju računsku složenost pažnje, koja je kvadratno ovisna o veličini slike, na linearnu. Osim SWIN-a, to su Linformer [9], koji matrice ključa i vrijednosti projecira u nižu dimenziju, Nyströmformer [10] koji aproksimira samopozornost metodom Nyström i dr.

Od mnogih konvolucijskih modela za gustu predikciju spomenimo samo nekoliko vezanih uz ideju SwiftNeta. Prvi je DeepLab [11]. On koristi konvoluciju s naduzorkovanim jezgrama („atrous konvolucija“) čime učinkovito povećava receptivno polje, metodu ASPP koja koristeći jezgre s različitih razina uzorkovanja i s različitim receptivnim poljima prikuplja omogućuje predikciju na više mjerila, te kombinacijom dubokih konvolucijskih modela i probabilističkih grafičkih modela poboljšava lokalizacijsku točnost. Sljedeći je FCN [12] koji je tradicionalne modele (AlexNet [13], VGG net [14]) pretvorio u „potpuno konvolucijske“ – zamijenivši potpuno povezane slojeve konvolucijskim omogućio je gustu predikciju nad ulazom proizvoljne veličine i zadržao prostornu dimenziju izlaza. Tu je još i Ladder DenseNet [15] koji kombinira koder DenseNeta [16] (modul SPP [17]) s modulom za ljestvičasto naduzorkovanje. SwiftNet polazi od njega i donosi poboljšanje modula SPP u konceptu piramidalne fuzije značajki.

1. Elementi dubokih modela za gustu predikciju

Navedimo i ukratko opišimo neke od glavnih ideja zastupljenih u modelima za gustu predikciju. Istaknut ćemo elemente koji su korišteni u odabranim transformerskim i konvolucijskim modelima i još nekoliko elemenata koji su široko korišteni.

1.1. Gusta predikcija

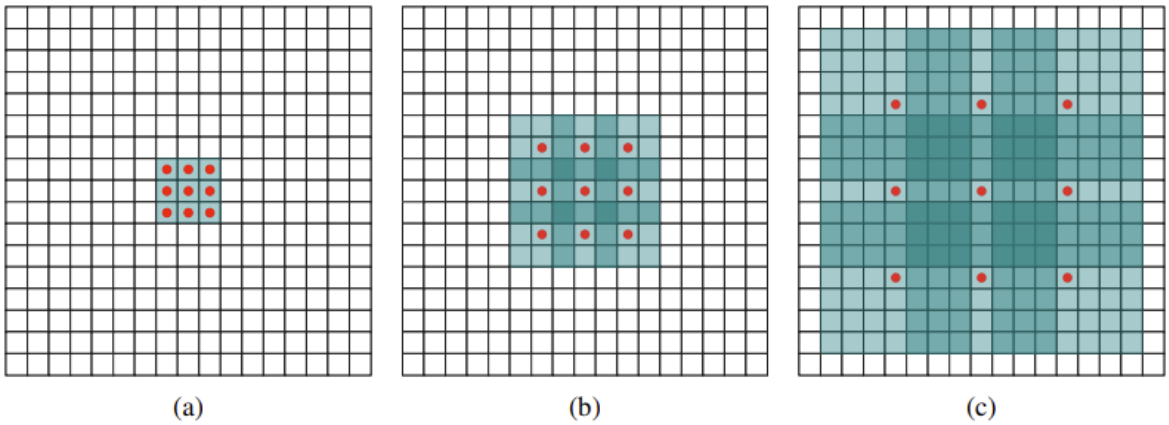
Za gustu predikciju možemo reći da predstavlja cilj kojem računalni vid teži – svaki piksel slike klasificirati u kategoriju objekta čiji je dio. Primjenu semantičke segmentacije nalazimo u mnogo područja – od analiza medicinskih slika do autonomne vožnje. Već dva navedena primjera jasno predstavljaju važnost da modeli korišteni za gustu predikciju moraju biti vrlo točni, a istovremeno i vremenski i prostorno vrlo učinkoviti kako bi u realnom vremenu na hardveru ograničene memorije i preciznosti mogli donositi čim bolje odluke koje mogu biti po život važne. Promotrimo stoga sad nekoliko elemenata koje *state-of-the-art* modele za semantičku segmentaciju čine uspješnima. Radi bolje preglednosti i sklada s tijekom kako su se ovi modeli razvijali, podijelimo elemente prema tipu modela u kojima se koriste – u elemente konvolucijskih i transformerskih modela.

1.2. Konvolucijski modeli

Današnji konvolucijski modeli koriste niz tehnika kojima proširuju receptivno polje konvolucije koja je po prirodi lokalnog karaktera i stoga dobra za prepoznavanje lokalnih struktura, no ne i globalnog konteksta slike. Neke od takvih poznatih tehnika su dilatirane konvolucije i SPP.

Dilatirane (ili atrous) konvolucije su vizualizirane (Slika 1). Na početku imamo jedan (Slika 1 – središnji) piksel koji je receptivnog polja 1x1 i njegova funkcija diskretna funkcija s realnom domenom F_0 dobivena je 0-dilatiranom konvolucijom. Pod a) je uobičajena konvolucija (funkcija F_1) koja je dobivena 1-dilatiranom konvolucijom nad F_0 i prema tome su težine konvolucijske jezgre udaljeni za 1 lokaciju; F_1 ima receptivno polje 3x3. Pod b) je F_2 koja je dobivena 2-dilatiranom konvolucijom nad F_1 i parametri su udaljeni za 2 piksela; F_2 ima receptivno polje 7x7. Slično, pod c) je F_3 je iz F_2 dobivena 4-

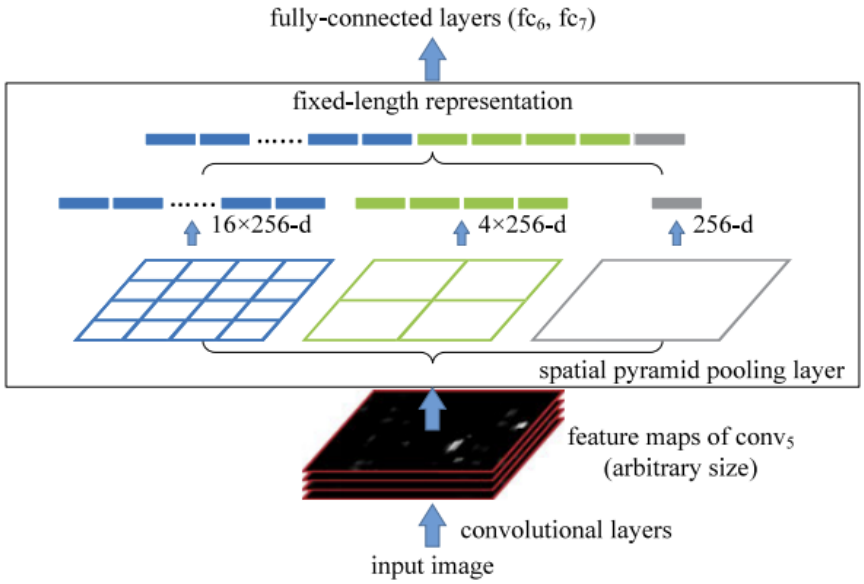
dilatiranom konvolucijom pa su težine udaljene 4 piksela i ona ima receptivno polje 15x15. Općenito je veličina receptivnog polja svakog piksela unutar F_{i+1} jednaka $(2^{i+2} - 1)$ x $(2^{i+2} - 1)$ gdje je F_{i+1} dobiven iz F_i 2^i-dilatiranom konvolucijom. To je eksponencijalno povećanje receptivnog polja. Prema tome, dilatirana konvolucija eksponencijalno povećava receptivno polje „preskakanjem“ pojedinih piksela, no bez gubitka rezolucije ili pokrivenosti ulazne slike tj. semantika ostaje ista. Npr. 2-dilatirana konvolucija ima samo 9 parametara kao i jezgra 3x3 obične konvolucije, a receptivno polje 7x7 je jednako onom obične konvolucije jezgre 5x5. Ovaj pristup među ostalim primjenjuje DeepLabv3+ [21] kako bi osim povećanja receptivnog polja izbjegao pretjerano poduzorkovanje, no na račun računske složenosti i memorijska otiska modela pri učenju [15].



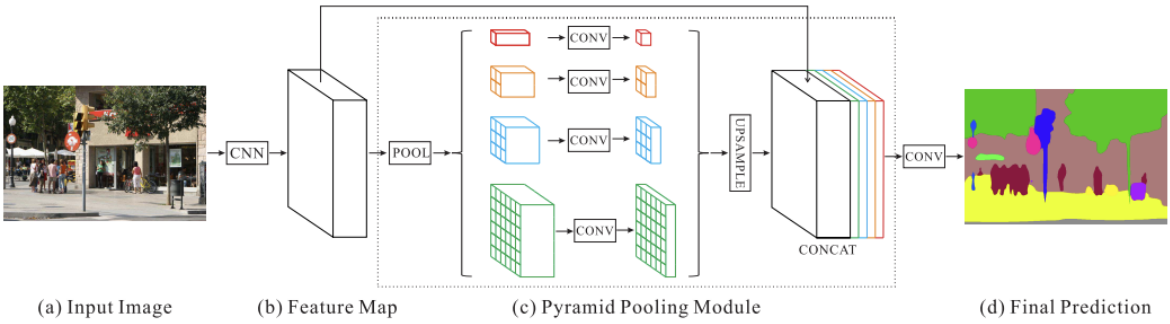
Slika 1 Ilustracija dilatirane konvolucije iz [20]. U a) je prikazano receptivno polje uobičajene konvolucije (1-dilatirana konvolucija), do b) se dolazi iz a) primjenom 2-dilatirane konvolucije i u c) se dolazi uz b) primjenom 4-dilatirane konvolucije. Korak dilatacije (ovdje 1/2/4) određuje koliko su piksela udaljene dvije susjedne težine konvolucijske jezgre.

Tehnika SPP (*spatial pyramid pooling*) sažima mape značajki i generira izlazne mape fiksne veličine iz ulaza proizvoljne veličine. Ova je tehnika temelj modela SPP-net [17] koji ju je primijenio u kontekstu konvolucijskog modela. Time se riješilo ograničenje potpunog povezanih slojeva koji primaju ulaz isključivo nepromjenjive veličine jer sad se SPP koristi za dovođenje izlaza konvolucijskog sloja (obično posljednjeg) na dimenziju koju prihvaća potpuno povezan sloj, što kod prijašnjih pristupa s klizećim prozorom nije bilo moguće. SPP koristi više rešetki/jezgara različitih veličina i omjera stranica (*aspect ratio*) jednakog omjeru širine i visine slike, a broj regija (*bins*) je fiksan za svaku rešetku neovisno o veličini slike. Slika 2 je vizualizacija. Umjesto zadnjeg sloja sažimanja je sloj SPP koji slijedi iza zadnjeg konvolucijskog sloja. U svakoj od M regija se sažimaju svih k mapa značajki (odzivi svake od k konvolucijskih jezgara sloja zadnjeg konvolucijskog

sloja $conv_5$). Ovdje su 3 jezgre sažimanja gdje je M redom jednak 16 (4x4; plavi), 4 (2x2; zeleni) i 1 (1x1; sivi), a ima k=256 konvolucijskih jezgara. Dobiveni rezultati ova 3 sažimanja se konkateniraju u reprezentaciju fiksne duljine koja se šalje u krajnje potpuno povezane slojeve (fc_6 i fc_7). Metoda SPP se ne koristi samo u klasifikaciji i detekciji kao kod SPP-neta, nego i u gustoj predikciji kao što to radi DeepLab [21], PSPNet [22] i također SwiftNet. PSPNet je modul SPP nadogradio u modul PPM (*pyramid pooling module*). PPM dodatno smanjuje gubitak informacije između podregija fuzijom značajki iz sve 4 razine u PPM-u koje se provuku kroz 1x1 konvoluciju, bilinearno naduzorkuju na veličinu ulazne mape značajki, konkateniraju po dubini zajedno s ulaznom mapom te se tako dobivena reprezentacija konačno provuče kroz konvoluciju čime se dobiva predikcija (Slika 3). SwiftNet pak uzima nešto prilagođeni model PPM-a iz [15] gdje je najveća razlika u tome što PPM samo povećava informaciju o kontekstu odnosno povećava receptivno polje, dok se klasifikacija događa tek nakon dekodera u mjesto nakon konvolucije [3].



Slika 2 Ilustracija modula SPP iz [17]. Na svaku mapu značajki iz conv5 se primjenjuju jezgre sažimanja veličine 4x4, 2x2 i 1x1, dobiveni sažetci se konkateniraju u vektor fiksne duljine i ta se reprezentacija dalje šalje u potpuno povezane slojeve.

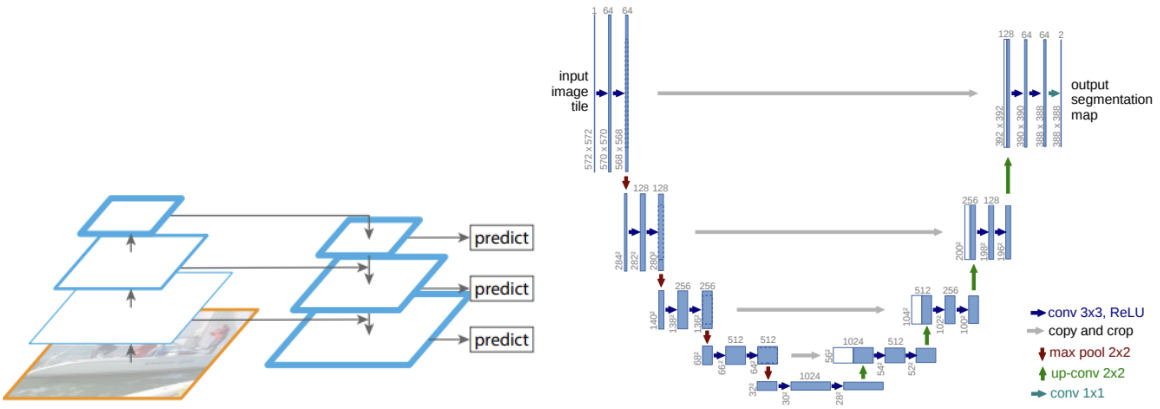


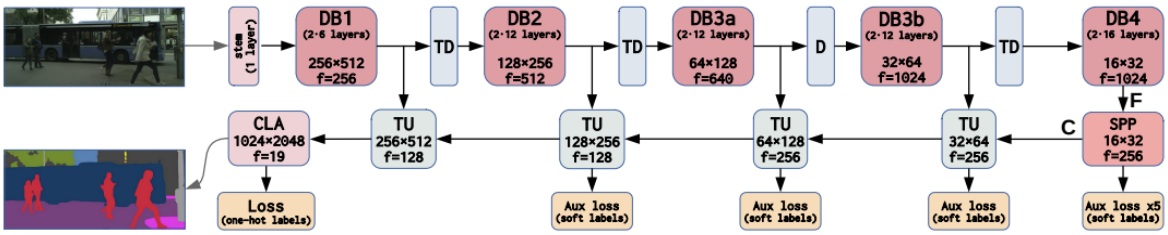
Slika 3 Ilustracija modula PPM iz PSPNeta iz [22]. Mapa značajki (pod b)) ulazi u modul PPM (pod c)) i sažima se jezgrama 1x1, 2x2, 3x3 i 6x6, sažetci prolaze 1x1 konvoluciju i naduzorkuju se na veličinu mape značajki, mapa značajki se sa sažetcima konkatenira na kraju PPM-a i dobiveno prolazi kroz konvoluciju čime se dobiva predikcija (pod d)).

Recimo još nekoliko riječi o generalnoj arhitekturi modernih modela za gustu predikciju. Oni su uglavnom su tipa koder-dekoder – koder smanjuje rezoluciju ulazne slike, a povećava dimenzionalnost značajki, dok dekoder prima izlaz kodera koji sadrži značajke male rezolucije koje onda dekoder naduzorkuje kako bi dobio konačnu segmentacijsku mapu na temelju koje se računa gubitak, tipično gubitak unakrsne entropije na razini piksela. Prema tome, smisao kodera je dobiti semantički vrijednu reprezentaciju na temelju koje se može raditi klasifikacija, a cilj je dekodera iz te reprezentacije doći do početnih lokaliziranih značajki slike. Rad FCN-a [12] je predstavio ideju takvih modela tzv. „potpuno konvolucijskih“ modela – oni zamjenjuju potpuno povezane slojeve na kraju modela konvolucijskim i zatim provode bilinearno naduzorkovanje (unatražna konvolucija / dekonvolucija) koje se uči. Kako bi poboljšali točnost jer se ovakvom izvedbom dobivaju nezadovoljavajuće grubi izlazi, kombiniraju izlaz zadnjeg sloja za predikciju s izlazima plićih konvolucijskih slojeva koji imaju finiji korak tj. kombiniraju predikcije dobivene iz značajki naduzorkovanih na grubim (dublji slojevi, značajke veće dimenzije) i finijim (plići slojevi, značajke manje dimenzije) rezolucijama.

Dodatno se taj tip arhitekture poboljšava time što se izlaz iz pojedine razine kodera, koji je više rezolucije no semantički slabiji, šalje uz izlaz prethodnog bloka dekodera, koji sadrži značajke niže rezolucije no semantički je bogatiji, kao dodatan ulaz odgovarajućem bloku dekodera. Pritom dani blok dekodera najprije mora naduzorkovati značajke niske rezolucije koje dolaze s prethodne razine dekodera na rezoluciju značajki koje dolaze lateralnom vezom iz odgovarajuće razine kodera, a značajke koje dolazi iz kodera treba provući kroz 1x1 konvoluciju kako bi se smanjila dimenzionalnost kanal. Zatim te dvije reprezentacije spaja tipično zbrajanjem po elementima i na kraju još provodi 3x3

konvoluciju radi smanjenja efekta naduzorkovanja. To čini ideju modela za ljestvičasto naduzorkovanje (npr. Ladder DenseNet [15]) odnosno modela piramidalne strukture za izlučivanje značajki (npr. FPN [23] i U-Net [24]). Oni rješavaju problem različitih mjerila tako što naduzorkovanjem i lateralnim vezama kroz više razina vraćaju izvornu rezoluciju. Zanimljivo je ideja kako se modeli poput FPN-a i U-Neta mogu koristiti kao generički moduli za izlučivanje značajki pa tako i u kontekstu transformera SWIN koji ima hijerarhijske mape značajki [2]. Navedena 3 modela prikazana su (Slika 4): FPN (gore lijevo) omogućuje neovisnu predikciju na svakoj mapi značajki u piramidi desno, U-Net (gore desno) koji u putu prema dolje koristi sažimanje maksimumom kojim se 2 puta povećava broj kanala te u putu prema gore zatim koristi „dekonvoluciju“ koja 2 puta smanjuje broj kanala, i Ladder DenseNet koji za osnovu koristi model DenseNeta [25] (blokovi DB – *dense blocks*) i uz njega modul SPP-a na kraju kodera te nakon dekodera koji se sastoji od TU blokova semantičke mape bilinearno naduzorkuje i segmentira. Treba reći da je DenseNet uveo novost da su svi slojevi povezani sa svim ostalim tj. konvolucijski model koristi konkatenaciju svih prethodnih značajki čime se dijele značajke i poboljšava tok gradijenta prema početku modela [15]. Tako modeli DenseNeta, uz tradicionalno modele ResNeta [26] koji su uveli rezidualni vezu za poboljšanje toka gradijenta, predstavljaju kralješnicu koja se koristi u suvremenim dubokih modelima pa tako i modelima za gustu predikciju. Konkretno, Ladder DenseNet, od kojeg SwiftNet preuzima dizajn dekodera i modul SPP (SPP samo u slučaju *singlescale* SwiftNeta), temelji se na DenseNetu, a koder SwiftNeta da ResNetu (alternativno na MobileNetu [27] koji koristi grupne konvolucije tipa *depthwise separable convolutions* koje smanjuju računsku složenost i broj parametara, no degradiraju točnost [3]).

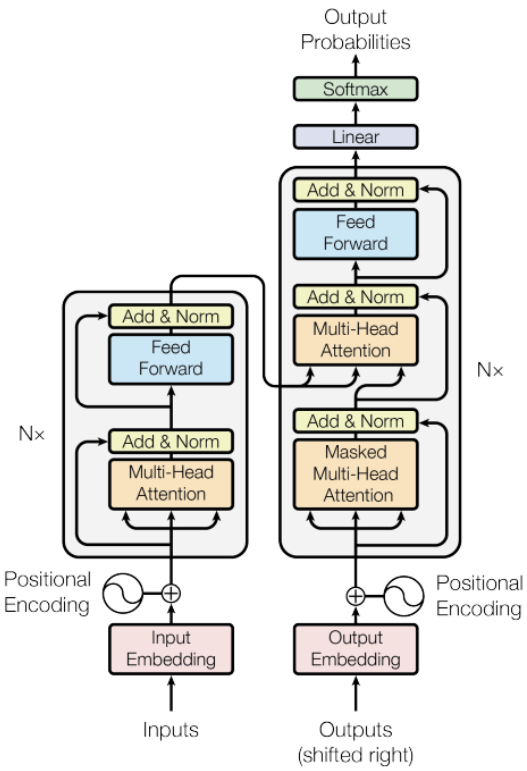




Slika 4 Ilustracija ideje FPN-a iz [23] (gore lijevo), U-Neta iz [24] (gore desno) i Ladder DenseNeta iz [15] (dolje). FPN omogućava predikciju na svakoj razini mapa značajki, U-Net u putu poduzorkovanja koristi sažimanje maksimumom 2x2, a u putu naduzorkovanja dekonvoluciju 2x2, Ladder DenseNet koristi blokove DenseNeta [25] u koderu (DB1-DB4), modul SPP za proširenje receptivnog polja (nakon kodera, a prije dekodera) i blokove naduzorkovanja (TU).

1.3. Transformerski modeli

Iako formalno ideju mehanizma pažnje donosi [4] u obliku pažnje koji danas nazivamo Bahdanau pažnja, rad *Attention Is All You Need* [5] predstavlja oblik pažnje koja se danas standardno koristi u svim transformerskim modelima u računalnom vidu. Također, [5] donosi dizajn transformera u kontekstu obrade prirodnog jezika, ali ta se arhitektura u istom načelnom obliku primjenjuje u računalnom vidu uz prilagodbe za domenu slika umjesto riječi, kao što ćemo vidjeti predstavivši [5] u predstavljanju ViT-a [7]. Krenimo s opisom univerzalnog modela transformera (Slika 5).



Slika 5 Osnovni dizajn arhitekture transformera iz [5]. Transformer se sastoji od kodera (lijevo) i dekodera (desno). Koder i dekoder se sastoje od N blokova. Svaki blok se sastoji od slojeva višeglavne pažnje MHA (narančasto) i unaprijednog modela FF (plavo), uz slojeve zbrajanja s ulazom koji rezidualnom vezom dolazi do izlaza sloja MHA/FF i slojeve normiranja koje se primijeni nakon zbrajanja. Ulazi u koder i dekoder se ugrađuju (*embedding*) i pribraja im se njihov pozicijski kod i taj zbroj ulazi u koder odnosno dekoder.

Transformer se sastoji od kodera (Slika 5 lijevo) ili/i dekodera (Slika 5 desno). Dvije glavne vrste slojeva od kojih se transformer sastoji su slojevi pažnje (konkretno pažnja s više glava, *multi-head attention*, MHA) i potpuno povezani slojevi (*feed forward*, FF). Koder i dekoder sačinjava određen broj blokova (Slika 5 – u primjeru je to isti broj slojeva N) gdje svaki blok ima prvo sloj MHA i zatim FF. Osim toga, postoji rezidualna veza koja ulaz u sloj zbraja s njegovim izlazom i zatim se normira. Napomena je da normiranje ne treba uvijek biti na kraju, ono može biti i npr. prije MHA odnosno prije FF kao što to radi SWIN. Na način kako je ovdje predloženo, izlaz sloja se računa kao u (1).

$$out = LayerNorm(x + layer(x)) \tag{1}$$

Dekoder osim ova 2 sloja u svakom bloku ima i sloj MHA koji provodi na izlazom kodera. Također, MHA koji je na početku bloka dekodera je modificiran maskiranjem tako da dio rečenice iza trenutne riječi ne bi ulazio u obzir pri predikciji modela – ovo je primjer

specifičnost u danoj primjeni u obradi prirodnog jezika. Na kraju dekodera se primjenjuje linearna transformacija koja se uči i funkcija softmax kako bi se izlaz pretvorio u vjerojatnost sljedećeg tokena.

Ulaz u koder i dekoder se zbraja sa svojim izlazom pozicijskih kodiranja/ugrađivanja (*positional embedding*) za taj element i tek tada ulazi u koder odnosno dekoder. Naučena pozicijska kodiranja pretvaraju ulazne tokene u vektore dimenzije modela. Pozicijsko kodiranje ima važnu ulogu jer transformer nema konvolucijskih slojeva, stoga je ono uvedeno kako bi se ugradila informacija o relativnoj ili apsolutnoj poziciji trenutnog ulaza – to je pozicija riječi u rečenici ili npr. piksela na slici ako prevedemo iz konteksta riječi u kontekst slika. Općenito je to pozicija tokena u nizu gdje token može značiti npr. riječ. Pozicijski kodovi iste su dimenzije kao tokeni (dimenzija određena za model) kako bi se mogli zbrojiti. Mogu biti fiksne funkcije ili se mogu učiti pa vrijedi isprobati više varijanti u oba smjera.

Obradimo sada slojeve transformera.

Prvo ukratko definirajmo drugu grupu slojeva što su potpuno povezani slojevi. Unaprijedni model se primjenjuje na svaku poziciju neovisno na isti način. Ovaj sloj čine dvije linearne transformacije (ekvivalentno dvjema konvolucijama s jezgrom 1x1) s nelinearnosti zglobnice (ReLU) u sredini između kao u (2). Treba reći da se parametri linearne projekcije razlikuju između slojeva iako se različite pozicije transformiraju jednako.

$$FF(x) = \max(0, xW_1 + b_1)\, W_2 + b_2 \tag{2}$$

Definirajmo sada sloj pažnje (Slika 6 lijevo). Pažnja je preslikavanje upita i skupa parova ključ-vrijednost u izlaz. Imamo tri vektora: upit (*query*, Q), ključ (*key*, K) i vrijednost (*value*, V). Izlaz je jednaku težinskoj sumi vrijednosti (V); težina pridijeljena vrijednosti se računa kao funkcija kompatibilnosti odnosno sličnosti upita (Q) i odgovarajućeg ključa (V). Taj tip pažnje se naziva *scaled dot-product attention* i računa se prema (3).

$$attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3}$$

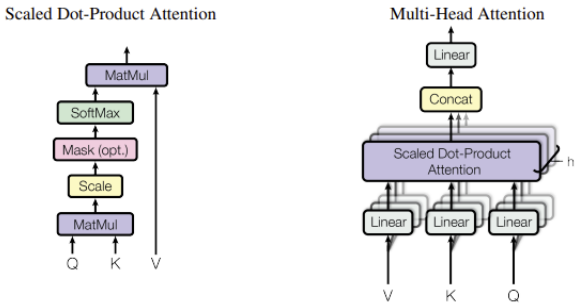
Q, K i V u praktičnoj implementaciji zapravo nisu vektori, već matrice u koje se ti vektori pakiraju kako bi se implementacija ubrzala. Vidimo se izračun uključuje skalarni umnožak matrica Q i K, nakon toga i skalarani produkt matrice težina vrijednosti (matrica koja je izlaz softmax-a). To je optimizirana operacija množenja matrica koja je vrlo učinkovito

implementirana u matematičkim bibliotekama, za razliku od npr. aditivne pažnje (Bahdanau pažnja) predložene u prije spomenutom radu [4], koja koristi unaprijedni model s jednim skrivenim slojem. U (3) se skalarani umnožak Q i K dijeli s korijenom dimenzije d_k što je dimenzija ključa i upita (dimenzija vrijednosti je d_v). Ta veličina kojom se normalizira je jednaka standardnoj devijaciji distribucije iz koje dolazi skalarni produkt Q i K (koji pak dolaze iz distribucije N(0, 1)) te je ta normalizacija operacija bez koje bi aditivna pažnja bolje računala sličnost Q i K jer bi kod skalarnog produkta gradijent softmax-a mogao „nestati“ za velike dimenzije d_k (navedeno u [5]).

Definiravši pažnju, napomenimo još da se transformeri mogu izvesti u 3 varijante. Prvo, ako imaju koder i dekoder, onda K i V dolaze iz kodera, a Q iz prethodnog sloja dekodera (gornja pažnja MHA u dekoderu - Slika 5), čime svaka pozicija u dekoderu obraća pažnju na sve pozicije ulaznog niza. Drugo, ako postoji samo koder, onda imamo sloj samopozornosti gdje Q, K i V dolaze iz istog ulaza što je izlaz prethodnog (od N) sloja kodera, čime svaka pozicija u pojedinom sloju kodera obraća pažnju na sve pozicije u prethodnom sloju. Treće, ako postoji samo dekoder, imamo sloj samopozornosti kojim svaka pozicija u dekoderu obraća pažnja na sve pozicije dekodera (bez pozicija u svim narednim slojevima ako se implementira maskiranje koje smo istaknuli kao specifičnost). Spomenimo da u računalnom vidu imamo najčešće transformere samo s koderom ili dekoderom pa se radi o samopozornosti. Složenost samopozornosti je kvadratna ovisno o veličini ulazne slike zbog skalarnih produkata u izračunu pažnje. To je nedostatak transformera što mnogi transformeri rješavaju na svoj način, jedan od njih je i SWIN.

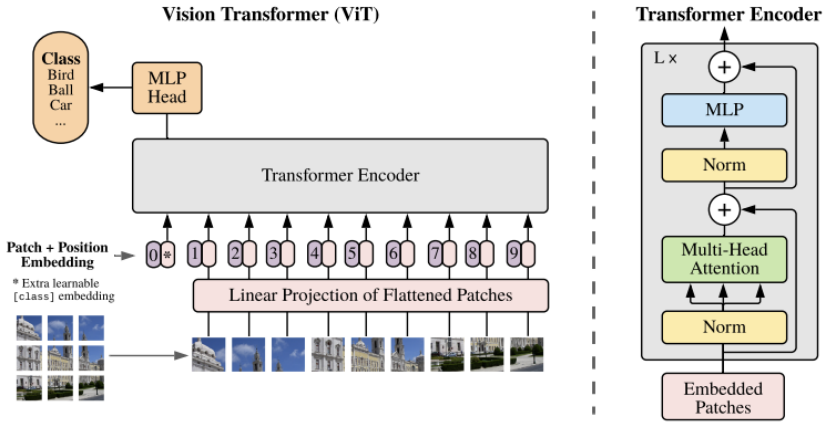
Definirajmo sada sloj višeglavne pažnje (MHA) u (4) (Slika 6 desno). Ovdje ćemo pod oznakama Q, K i V smatrati vektorima. Novost koju uvodi je više glava (*heads*) pažnje; broj glava = h. Svaka od tih glava ($head_i$) ima parametre koji se uče i kojima projecira matrice Q (parametrom W_i^Q) i K (parametrom W_i^K) u dimenziju d_k, a V (parametrom W_i^V) u d_v. Zatim se pažnja sa skalaranim produktom istovremeno računa na svih h glava i kako rezultat se dobiva h vrijednosti dimenzije d_v (od $head_1$ sve do $head_h$) pa se te vrijednosti konkateniraju (concat) u vektor dimenzije hd_v. Tako konkatenirani izlazni vektor se na kraju projecira parametrom W^O dimenziju modela d_{model}. U slučaju MHA tipično se dimenzija modela dijeli na h glava pa su dimenzija ključa i vrijednosti jednake d_{model}/h.

$$\begin{gathered} MHA(Q,K,V) = concat(head_1, \dots, head_h)W^O \\ head_i = attention(QW_i^Q, KW_i^K, VW_i^V) \end{gathered} \tag{4}$$



Slika 6. Ilustracija pažnje zasnovane na skalarnom umnošku (lijevo) i višeglavne pažnje (desno) iz [5]. Pažnja sa skalaranim umnošcima skalarno množi Q i K, umnožak skalira standardnom devijacijom distribucije iz koje taj umnožak dolazi, prolazi kroz softmax i izlaz se množi skalarno s V što daje pažnju. Višeglavna pažnja koristi h glava koje paralelno vlastitim parametrima projeciraju Q, K i V, zatim paralelno računaju pažnju nad tim projekcijama, i na kraju se rezultat svih glava konkatenira li linearno projecira što daje izlaz te pažnje.

Sada krenimo na reprezentativnog predstavnika transformera u računalnom vidu koji je kreirao arhitekturu bez konvolucijskih slojeva koristeći prethodno opisan model transformera – Vision Transformer (ViT [7]). Modifikacija koju je ViT uveo kako bi se transformer mogao primijeniti na slikama je podjela slike u regije (*patches*). Time se smanjuje broj tokena koji ulaze u transformer jer regija ima puno manje od piksela pa je i računska zahtjevnost djelomično smanjena. Te se regije linearno ugrađuju i zatim se daju kao ulaz transformeru. Slika 7 prikazuje arhitekturu.



Slika 7 Ilustracija arhitekture ViT-a iz [7]. ViT (lijevo) se sastoji od kodera u koji ulaze linearne projekcije izravnatih kvadratnih regija slike i njihovi pozicijski kodovi. Dodatan ulaz u koder je klasifikacijski token (znak *) koji se stavlja na 1. mjesto u nizu *embeddinga* regija i na temelju njega se klasificira slika. Desno je dana arhitektura kodera ViT-a – sastoji se od L slojeva čija je struktura jednaka strukturi kodera u standardnom modelu transformera, samo što se normiranje radi prije ulaza u sloj, umjesto nakon sloja MHA/MLP.

Transformer inače prima 1D niz ugrađenih tokena, no transformer ViT prima 2D sliku. Slika dimenzije H x W x C (C je broj kanala) se preoblikuje u regije veličine N x (P^2C) gdje je N broj regija, (P, P) dimenzija regije. N je jednak HW / P^2 i predstavlja duljinu ulaznih niza u transformer tj. broj tokena. Regije se u svim izravnaju i linearno projeciraju u dimenziju D što je dimenzija svakog latentnog vektora. Rezultat te projekcije se naziva kodovima regijama (*patch embeddings*) - zaobljeni pravokutnici roze boje koji ulaze u koder ViT-a (Slika 7). U nizu kodova regija, prije svih kodova se tijekom učenja dodaje *class* token što je posebni token tj. *embedding* koji se uči te se stanje tog tokena koristi na izlazu tokena kao reprezentacija ulazne slike pa se na temelju njega provodi i klasifikacija slike. Uz kodove regija, ulaz u transformer su pozicijski kodovi (*position embeddings*) koji smo objasnili u osnovnoj inačici transformera. ViT koristi 1D pozicijske kodove koji se uče (2D pozicijski kodovi nisu donijeli poboljšanje u odnosu na 1D [7]). Transformer ViT se sastoji od kodera koji je standardno oblikovan, no ovdje s prilagodbom za drugačiji ulaz. Konfiguracija kodera ViT-a dana s (5). Razlike u odnosu na standardni transformer su: i) sloj normiranja *LayerNorm* se primjenjuje prije sloja MSA (istovjetan MHA) i MLP (istovjetan FF), ii) na ulazu je N regija i klasifikacijski token (x_p^1,…,x_p^N + x_{class}) koji se projeciraju istim parametrom E, konkateniraju se i dodaju im se pozicijski kodovi E_{pos} i iii) u zadnjem sloju (l = L) se normira klasifikacijski token čime se dobiva izlazna reprezentacija y.

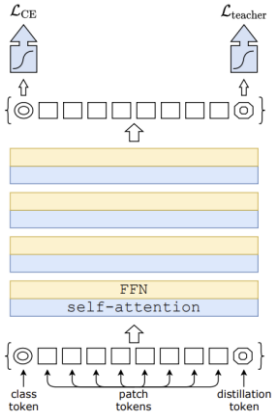
$$
\begin{aligned}
\mathbf{z}_0 &= [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\, \cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, && \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \\
\mathbf{z}'_\ell &= \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, && \ell = 1 \ldots L \\
\mathbf{z}_\ell &= \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, && \ell = 1 \ldots L \\
\mathbf{y} &= \text{LN}(\mathbf{z}_L^0)
\end{aligned}
\tag{5}
$$

Navedimo još standardne veće konfiguracije ViT-a (Slika 8). Konfiguracije su određene: brojem slojeva (L), dimenzijom modela (Slika 8 - skrivena veličina), veličinom skrivenog sloja u potpuno povezanim slojevima, brojem glava i konačno veličinom kvadratne regije. Tako npr. ViT-L/16 je transformer ViT-Large veličine regije 16 x 16. Treba još jednom napomenuti da je broj tokena N tj. duljina niza jednaka veličini slike HW podijeljene s P^2, a ne samo broju piksela HW, što implicira da manje regije daju računski zahtjevniji model. S druge strane, može se reći da za dani P ako se poveća veličina slike, također dobivani računski zahtjevniji model. To pak implicira da se ViT teže nosi s velikim slikama jer problem kvadratne složenosti pažnje, točnije samopozornost, počinje biti sve izraženiji.

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Slika 8. Osnovni podaci o pojedinim standardnim konfiguracijama modela ViT iz [7].

Predstavimo sada poznati model Data-efficient image transformer (DeiT [8]) koji je došao kao poboljšanje ViT-a. Uz različite augmentacije podataka kojima uvodi dodatnu regularizaciju podataka i koje su ključne za uspjeh DeiT-a, druga važna novost je da destilaciju znanja primjenjuje u kontekstu transformera. Slika 9 daje shemu. Uvodi se, uz klasifikacijski, još jedan dodatni token za destilaciju koji se primjenjuje slično kao klasifikacijski token samo što je cilj klasifikacijskog tokena reproducirati točnu oznaku, a cilj destilacijskog tokena je reproducirati oznaku koju daje učitelj. Pritom se koriste različiti tipovi destilacije – meka (*soft*) i s tvrdim oznakama (*hard-label*) (izrazi (6) i (7) redom). Obje računaju gubitak unakrsne entropije na temelju logita koje daje student, a glavna razlika je u drugoj komponenti gubitka – meka destilacija pribraja gubitak Kullback-Leiblerove divergencije između distribucije softmaxa koju daju logiti učitelja i učenika (cilj je minimizirati ju), a destilacija s tvrdim oznakama računa gubitak unakrsne entropije gledajući softmax logita učenika i oznaku koju daje učitelj. Tvrde oznake se mogu pretvoriti u meke oznake ako se koristi zaglađivanje oznaka (*label smoothing*) gdje se vjerojatnost točnoj klasi daje velika vjerojatnost (npr. 0.9), a suprotna vjerojatnost (mala) se dijeli na netočne klase. Važno je reći da je modelu učenik transformer DeiT koji se uči na upravo opisan način, a model učitelj može biti transformer ili konvolucijski model, ali je DeiT utvrdio da se bolja točnost dobiva s konvolucijskim učiteljem. Drugim riječima, model učenik više nauči od konvolucijskog nego transformerskog učitelja tj. više naslijedi induktivnu pristranost konvolucijskog učitelja.

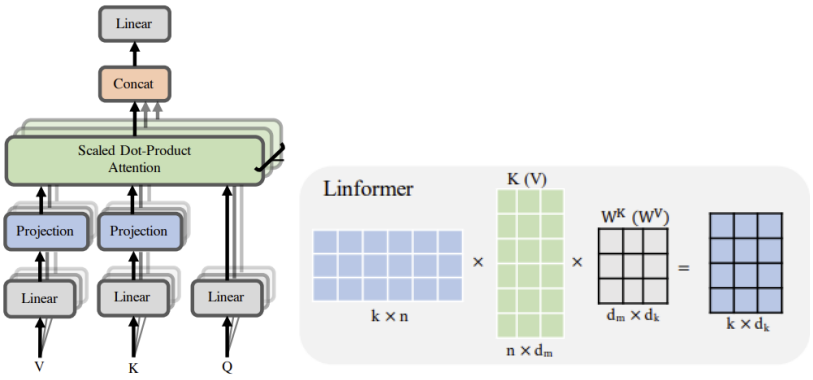


Slika 9. Arhitektura modela DeiT iz [8]. Na ulazu su osim *embeddinga* regija i klasifikacijski i destilacijski token. Ulaz prolazi kroz standardne blokove koji se sastoje od parova sloja samopozornosti i unaprijednog modela. Na kraju se na osnovi klasifikacijskog tokena radi klasifikacija u odnosu na oznaku, a na osnovi destilacijskog tokena klasifikacijska u odnosu predikciju modela učitelja. S tim u skladu ima i 2 gubitka – unakrsna entropija za klasifikacijski token i gubitak učitelja (također se uglavnom koristi unakrsna entropija).

$$\mathcal{L}_{\text{global}} = (1-\lambda)\mathcal{L}_{\text{CE}}(\psi(Z_{\text{s}}), y) + \lambda\tau^2 \text{KL}(\psi(Z_{\text{s}}/\tau), \psi(Z_{\text{t}}/\tau)) \tag{6}$$

$$\mathcal{L}_{\text{global}}^{\text{hardDistill}} = \frac{1}{2}\mathcal{L}_{\text{CE}}(\psi(Z_s), y) + \frac{1}{2}\mathcal{L}_{\text{CE}}(\psi(Z_s), y_{\text{t}}) \tag{7}$$

Na kraju, predstavimo još samo jedan od mnoštva transformera koji smanjuju računsku složenost pažnje na linearnu (osim SWIN-a). To je Linformer [9]. Linformer je rad koji je u kontekstu obrade prirodnog jezika predlaže novu definiciju višeglavne pažnje gdje uvodi projekciju K i V u nižu dimenziju (dimenzija k) prije primjene skalarnih umnožaka. Slika 10 lijevo plavim pravokutnicima označava te projekcije, i desno je prikazana projekcija K (ili ekvivalentno V) projekcijskim parametrom dimenzije k x n čime je rezultat dimenzije k x d_k umjesto n x d_k gdje je k << n kako bi se smanjili vremensko-prostorni zahtjevi što se očituje u bržem zaključivanju i linearnoj računskoj složenosti. Pokazano je da se ovakva aproksimacija pažnje opravdano može provesti jer je matrica samopozornosti niskog ranga: singularnom dekompozicijom matrice samopozornosti se dobilo da se većina (oko 90%) njenih svojstvenih vrijednosti akumilirana je u najvećih 128 svojstvenih vrijednosti koje su stoga dovoljne za opis većinskog dijela informacije samopozornosti. Dodatno je utvrđeno da je rang te matrice još manji u dubljim slojevima gdje se onda može dodatno smanjiti k. Linformer je predstavljen u okviru primjene razumijevanja prirodnog jezika, no također se može prevesti u domenu slika zamjenom pažnje s pažnjom koju on predlaže – npr. implementirati pažnju Linformera u ViT ili DeiT.



Slika 10. Ilustracija mehanizma pažnje koji predlaže Linformer iz [9]. Matrice K i V se prije ulaska u izračun pažnje sa skalarnim umnošcima (lijevo), projeciraju (plavo) u dimenziju k tako da je k<<n. Desno je dan primjer takve projekcije.

Neki od ostalih transformera koji smanjuju računsku složenost pažnje su Nyströmformer [10] koji koristi metodu Nyström kojom smanjuje složenost pažnju na O(n), Sparse Transformers [28] koji uvodi rijetku faktorizaciju matrice pažnje smanjujući složenost na $O(n\sqrt{n})$, Longformer [29] koji dovodi složenost na O(n) korištenjem pažnje koja se računa u lokalnim prozorima i „globalne pažnje“ koja se računa na nekoliko odabranih lokacija u ulazu itd.

Na kraju, možemo primijetiti da nijedan od predstavljenih transformera (ViT, DeiT, Linformer) nije namijenjen za semantičku segmentaciju – ViT i DeiT su za klasifikaciju slika, a Linformer donosi svoj oblik pažnje koji je univerzalno primjenjiv. Međutim, ViT odnosno DeiT je temelj mnogih transformera pa tako i Segmentera, stoga ih je bilo važno opisati, a opisom Linformera smo načeli opisivanje napora koji se ulažu u smanjivanje računske složenosti pažnje, što je problem koji rješava i SWIN. Upravo su Segmenter i SWIN jedni od prvih predstavnika transformera za gustu predikciju, s time da je SWIN i šire namjene.

Predstavimo onda u sljedećem poglavlju ta 2 transformera, uz SwiftNet kao odabranog glavnog predstavnika konvolucijskih modela.

2. Metode za semantičku segmentaciju

U prethodnom smo poglavlju predstavili sve važne komponente koje se koriste u *state-of-the-art* modelima za gustu predikciju, a posebno one elemente koji se koriste u trima modelima koje smo mi istražili i usporedili. Stoga ćemo u ovom poglavlju odmah izravno navesti novosti koje donose odabrani transformeri Segmenter i SWIN te konvolucijski model SwiftNet.

2.1. Transformerski modeli za semantičku segmentaciju

2.1.1. Segmenter

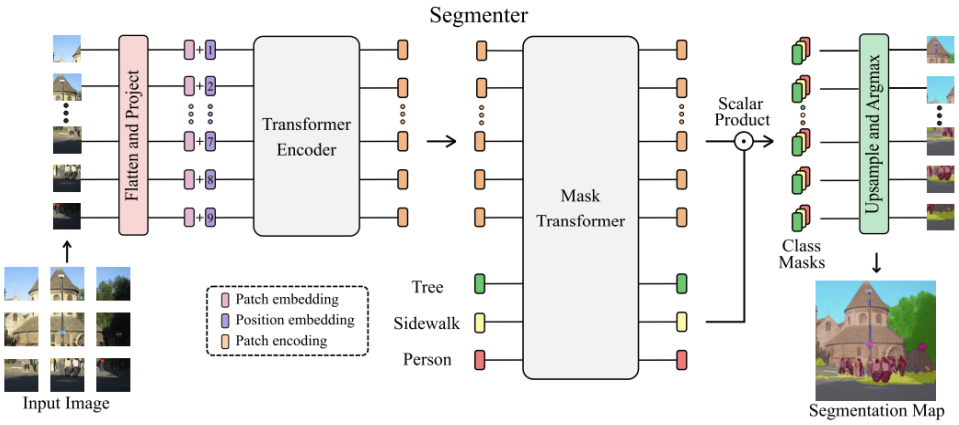
Segmenter je transformer dizajniran za gustu predikciju. Gradi se na ViT-u koji je namijenjen klasifikaciji i koristi ga kao koder. Izlazne *embeddinge* iz kodera ViT-a šalje u dizajnirani linearni dekoder ili dekoder koji se izvodi kao mask-transformer. Ta nadogradnja omogoućuje semantičku segmentaciju. Budući da koristi ViT, koristi modele ViT-a prednaučene za klasifikaciju i zatim ih prilagođava (*fine-tune*) na odabranom skupu učenju cijelog koder-dekoder modela Segmentera. Segmenter se uči *end-to-end* gubitkom unakrsne entropije na razini piksela.

Pogledajmo prvo osnovne konfiguracije Segmentera (Slika 11). Segmenter dolazi u najmanjoj (Ti – *tiny*), maloj (S – *small*), velikoj (B – *base*) i najvećoj (L – *large*) varijanti koje tim redom imaju sve više parametara, također se povećava dimenzija modela (veličina tokena), broj glava i broj slojeva. Osim ViT-a kao kralješnicu alternativno koristi DeiT (to je npr. Seg-B s oznakom † iza naziva).

Model	Backbone	Layers	Token size	Heads	Params
Seg-Ti	ViT-Ti	12	192	3	6M
Seg-S	ViT-S	12	384	6	22M
Seg-B	ViT-B	12	768	12	86M
Seg-B†	DeiT-B	12	768	12	86M
Seg-L	ViT-L	24	1024	16	307M

Slika 11 Osnovni podaci o pojedinim standardnim konfiguracijama modela Segmentera iz [1]. Model označen oznakom † (Seg-B†) ima, za razliku od ostalih navedenih, za kralješnicu transformer DeiT umjesto ViT.

Inovativna je arhitektura koju Segmenter donosi (Slika 12). Na prvi pogled se dizajn čini prilično jednostavnim (još jednostavniji bi izgledao da je ucrtan linearni dekoder), što je i bit koja omogućava učinkovit i brz rad te visoku točnost Segmentera.



Slika 12. Arhitektura Segmentera iz [1]. Segmenter se sastoji od kodera i dekodera (sivi pravokutnici). U koder ulaze standardno projecirane poravnate regije ulazne slike (*patch embeddings*) i njihovi pozicijski *embeddingi*, a izlaze kodovi regija (*patch encodings*). Kodovi regija ulaze u dekoder koji može biti tipa linearnog dekodera ili mask-transformera. Ovdje je skiciran mask-transformer koji kao ulaz uzima i *embeddinge* klasa. Oba ulaza mask-transformer obrađuje združeno, iz njega izlaze *embeddingi* klasa i *embeddingi* regija koji se skalarno množe i dobivene maske se naduzorkuju bilinearno na veličinu slike, provuku kroz softmax, pronađe se klasa najveće vjerojatnosti i dobiva se segmentacijska mapa.

Na slici (Slika 12) lijevo je koder, a desno dekoder Segmentera. Regije (*patches*) se standardno izravnaju u vektor i projeciraju i tako dobivenom *patch embeddingu* se pribraja njegovo *position embedding* čiji zbroj ulazi u koder. Izlaz kodera i kodovi klasa (*class embeddings*) ulaze u dekoder (ovdje transformer) koji na izlazu nakon nekoliko koraka daje oznake klasa za svaki piksel u obliku segmentacijske mape. Definirajmo arhitekturu formalnije.

Koder je standardnog tipa, ulaz se normira prije ulaza u modul MSA i MLP, koristi se normalna pažnja sa skalarnim umnošcima, samopozornost gdje su Q, K i V iste dimenzije d. Ulaz se sastoji od N regija veličine (P,P) slike HxW s C kanala pa je $N = HW/P^2$. Poravnate ulazne regije se iz dimenzije regije P^2C projeciraju u dimenziju modela D čime se dobiva niz *patch embeddinga* x_0 kojem se doda *position embedding* pos koji se uči. Koder kodira ulazni niz kodova regija $z_0 = [z_{0,1},\ldots, z_{0,N}]$ ($z_0 = x_0 + pos$) i njihovih pozicijskih kodova u niz kodova regija $z_L = [z_{L,1},\ldots, z_{L,N}]$ (L je broj slojeva kodera). Definicija kodera je u (8) lijevo i pažnje u (8) desno.

$$\begin{aligned} \mathbf{a_{i-1}} &= \mathrm{MSA}(\mathrm{LN}(\mathbf{z_{i-1}})) + \mathbf{z_{i-1}}, \\ \mathbf{z_i} &= \mathrm{MLP}(\mathrm{LN}(\mathbf{a_{i-1}})) + \mathbf{a_{i-1}}, \end{aligned} \qquad \mathrm{MSA}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V} \tag{8}$$

U dekoder preslikava niz kodova regija (*patch encodings*) z_L u segmentacijsku mapu s dimenzije H x W x K (za skup s K klasa). Tako dobivena segmentacijska mapa je dobivena na razini regija pa se sad bilinearno naduzorkuje na veličinu slike kako bi se dobila konačna mapa na razini piksela. Dva su tipa dekodera: linearni i mask-transformer. Linearni dekoder primjenjuje linearni sloj na kodove regija kojim iz dimenzije D na ulazu dobiva K-dimenzionalni izlaz što su logiti klasa na razini regija koji se bilinearno naduzorkuju s dimenzije N x K = HW/P^2 x K = H/P x W/P x K faktorom P na H x W x K. Izlaz se provuče kroz softmax i dobije se konačna segmentacijskih mapa. U zaključivanju se na izlaz primjenjuje argmax kako bi se dobila jedna klasa za svaki piksel, a u učenju se iz izlaza i oznake računa unakrsna entropija.

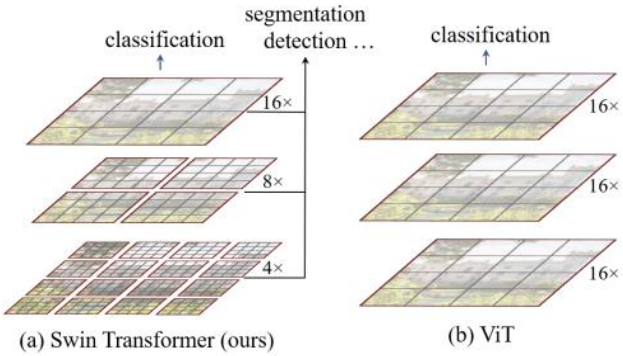
Drugi tip dekodera je *mask transformer*. U taj dekoder uz kodove regija z_L ulazi K *embeddinga* klasa cls = [cls_1,…,cls_K] dimenzije D i svaki *class embedding* je slučajno inicijaliziran jednom klasom (Slika 12). Mask-transformer kroz M slojeva obrađuje *patch* i *class embeddinge* zajedno. Na izlazu skalarno množi L2 normalizirane *embeddinge* regija z_M' (N x D) i *embeddinge* klasa c (K x D) čime se dobiva izlaz N x K koji se ekvivalentno kao kod linearnog dekodera prvo preoblikuje u H/P x W/P x K i zatim bilinearno naduzorkuje na H x W x K. Na kraju se primjeni softmax i zatim normiranje *layer norm* čime se dobiva segmentacijska mapa. Mask-transformer dekoder pokazao se uspješnijim od linearnog dekodera pa se on koristi u glavnim eksperimentima Segmentera.

2.1.2. Transformer SWIN

SWIN je široko primjenjiv transformer za računalni vid – od klasifikacije slika do guste predikcije kao što je detekcija i semantička segmentacija. To je pokazao na standardnim skupovima ImageNet (klasifikacija), COCO (detekcija) i ADE20K (segmentacija). Time ima za cilj postaviti novi tip modela koji bi se slično konvolucijskim modelima u računalnom vidu (odnosno transformerima u NLP-u) počeo primjenjivati širom područja CV-a. Kao glavne razloge zašto je prilagodba transformera za slike toliko izazovna navodi: i) veliku raznolikost u mjerilu koja ne postoji među riječima u rečenici pa je fiksna veličina tokena standard nastao u NLP-u koji pak nije prihvatljiv za primjene u CV-u gdje se problem vrlo različitih mjerila objekata na slikama javlja već u detekciji, te ii) veliku

rezoluciju slika – slike imaju puno više piksela nego što rečenica ima riječi – i to je onda problem posebno u semantičkoj segmentaciji gdje se klasifikacija događa na razini piksela. Ove probleme SWIN adresira hijerarhijskim dizajnom čije se reprezentacije dobivaju izračunima unutar pomaknutih lokalnih prozora (***S**hifted **win**dows*) prema kojima je Swin (*Shifted windows transformer*) dobio ime. Time se dobiva linearna računska složenost pažnje kao što ćemo pokazati te fleksibilan rad s različitim mjerilima.

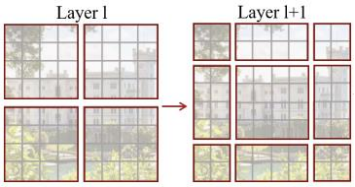
Slika 13 pod a) skicira hijerarhiju prozora koju SWIN predlaže. Sivo su označene regije (*patches*), a crveno prozori unutar kojih se lokalno računa pažnja. Prozori na najplićoj razini modela obuhvaćaju 4x4 regije (Slika 13 - primjeru), a idući prema dubini modela u svakoj sljedećoj razini se spajaju po 2x2 prozora tako da redom po razinama (koje ćemo detaljnije opisati) ima: 16 (4x4) prozora s 4x4 regije, 4 (2x2) prozora s 8x8 regija i 1 (1x1) prozor sa 16x16 regija. Upravo činjenica da je broj regija po prozoru je fiksan (ovdje 4x4) hiperparametar isti za svaku razinu dovodi do složenosti pažnje koja linearno ovisi o veličini slike. Važno je da se prozori unutar jednog sloja ne preklapaju što znači da je svaku regiju zahvaća samo jedan prozor u tom sloju. Vidimo da se ovakva hijerarhija mapa značajki može generički koristiti – bilo u klasifikaciji, detekciji ili segmantaciji. Slika 13 pod b) prikazuje još skicirane mape značajki ViT-a što su mape niske rezolucije koje je ista u svim slojevima i postoji samo 1 prozor unutar koji je veličine slike što dovodi do kvadratne složenosti pažnje koja se ovdje računa globalno na cijeloj slici.



Slika 13 Ilustracija ideje prozora SWIN-a iz [2]. Pod b) su prozori standardnih transformera za klasifikaciju kao što je ViT – sve mape su iste (niske) rezolucije i pažnja se računa za cijelu slike, a pod a) su prozori SWIN-a (crveni rub) koji sadrže fiksni broj regija (4x4) koji se prema dubljim slojevima spajaju iz manjih u sve veće prozore – ovdje su mape različitih rezolucija koje se mogu koristiti za više primjena npr. klasifikaciju, detekciju ili segmentaciju.

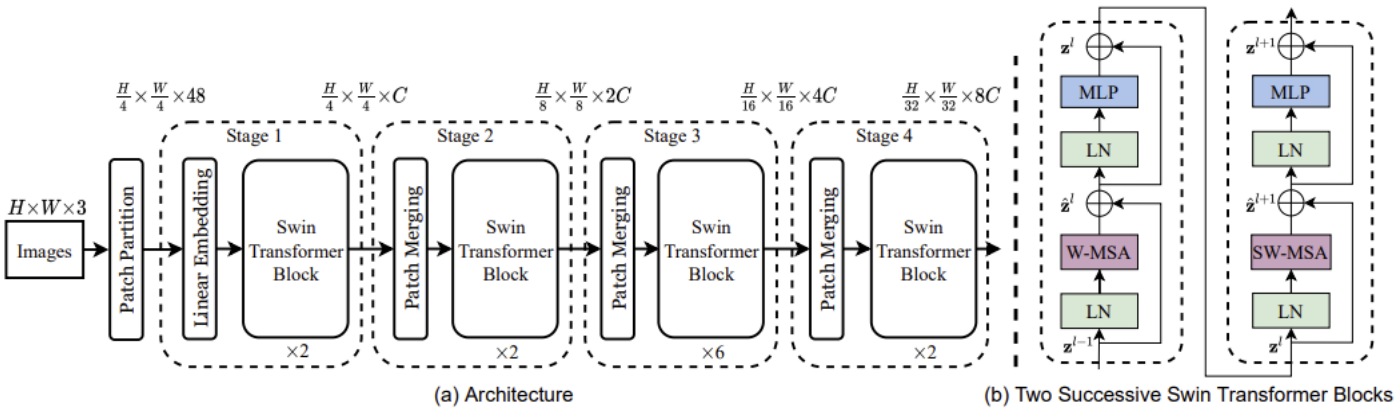
Iako se slika particionira u prozore koji se ne preklapaju u nijednoj regiji, prozori su postavljeni na različite pozicija unutar slike u različitim slojevima – odavde dolazi *shifted*

u imenu Swina. Pogledajmo taj koncept (Slika 14). Prikazana su dva uzastopna sloja unutar neke razine SWIN-a. Oba sloja imaju istu veličinu prozora od 4x4 regije, no u sloju l+1 nisu svi prozori te veličine i broj prozora je veći. To se dogodilo zbog pomaka 2x2 prozora iz sloja l u l+1 koji dovodi do preklapanja prozora, ali to je preklapanje između prozora u dva uzastopna sloja. To preklapanje omogućuje veze među prozorima koje poboljšavaju generalizaciju transformera. Uz to je smanjena latencija zbog izvedbe koja je učinkovita jer regije upita Q unutar nekog prozora dijele isti skup ključeva K.



Slika 14. Ilustracija pomičnih prozora SWIN-a iz [2]. Prikazana su 2 uzastopna sloja. U sloju l (lijevo) je inicijalna particija slike u prozora – 4 prozora od 4x4 regije, u svakom prozoru se zasebno računa pažnja. U sljedećem sloju l+1 (desno) su prozori pomaknute za 2 regije prema dolje desno čime je dobiveno 9 prozora različitih veličina. Taj pomak omogućuje prenošenje konteksta iz jednog u drugi prozor.

Predstavimo sad arhitekturu SWIN-a u konfiguraciji Swin-T (Slika 15).



Slika 15 Arhitektura SWIN-a iz [2]. Pod a) je arhitektura SWIN-a na primjeru SWIN-T Slika se na početku particionira u regije, zatim prolazi kroz 4 razine. U prvoj razini se regije linearnim slojem projeciraju u dimenziju C i ta dimenzija značajki se zadržava u toj razini. Slijede 2 bloka SWIN-a. Sljedeće 3 razine su drugačije od ove, a s međusobnom istim elementima: spajanje regija i blokovi SWIN-a. Spajanje regija spaja po 2x2 susjedna prozora u 4 puta veći prozor čime se dimenzija 4x povećava, no zatim se to linearno projecira u 2x manju dimenziju čime se efektivno dobiva 2x veća dimenzija na izlazu svake sljedeće razine. Pod b) je struktura 2 uzastopna bloka SWIN-a – ista je struktura, a pažnja je različita. Sloj l računa pažnju unutar inicijalnih prozora što je W-MSA, a sloj l+1 unutar prozora particije koja je pomaknuta u odnosu na inicijalnu što je SW-MSA.

Prvi korak je particionirati sliku kao što to radi ViT – u tokene što su nepreklapajuće regije 4x4 (*Patch Partition*) koji onda imaju dimenziju značajki od 4x4x3 = 48. Slijedi 1. razina SWIN-a (*Stage 1*) u kojoj prvo linearni sloj (*Linear Embedding*) projecira te značajke u dimenziju C i zatim slijedi nekoliko (u ovoj razini 2) blokova transformera SWIN (*Swin Transformer Blocks*) koji ne mijenjaju broj tokena (on ostaje HxW/16 odnosno H/4 x W/4). U sljedećoj razini (*Stage 2*) i svakoj sljedećoj razini (razine 3 i 4) je drugačiji dizajn – prvo ide sloj spajanja regija (*Patch Merging*) i zatim blokovi transformera SWIN (koji su isti kao u 1. razini). Sloj spajanja regija na početku 2. razine spaja grupe od 4 (2x2) susjedne regije (prisjetimo se - Slika 13) čime se broj tokena smanji 4 puta (sad je novi broj tokena H/8 x W/8). Tako se dobivene značajke konkateniraju u 4 puta veću dimenziju tj. 4C što se zatim linearnim slojem projecira u 2C. Naredni blokovi Swin transformera zadržava rezoluciju na H/8 x W/8 i dimenziju značajki od 2C pa je izlaz 2. razine H/8 x W/8 x 2C. Ekvivalentno se ponavlja u 3. i 4. razini (spajanje 2x2 regija iz prethodne razine čime se broj tokena smanjuje 4 puta i zatim se značajke linearno projeciraju u 2 puta manju dimenziju) gdje su izlazi onda redom H/16 x W/16 x 4C i H/32 x W/32 x 8C. Dobro je napomenuti da rezolucije mapa značajki na izlazu razina odgovaraju rezolucijama tradicionalnih konvolucijskih modela poput ResNeta što znači da se ova arhitektura može koristiti umjesto tih modela.

Navedimo osnovne konfiguracije SWIN-a: Swin-T, Swin-S, Swin-B, Swin-L. Konfiguracije se razlikuju po dimenziji značajki C u koju se linearno projeciraju (dimenzija kanala skrivenih slojeva u 1. razini) i broju slojeva u svakoj od 4 razine. Broj regija po širini/visini prozora je u svima M = 7. Konfiguracije su dizajnirane tako da Swin-B ima sličnu računsku složenost kao ViT-B odnosno DeiT-B, a u odnosu na Swin-B, Swin-T je 0.25x, Swin-S 0.5x i Swin-L 2x veći/računski zahtjevniji od Swin-B. Također je zanimljivo dizajnirano da je Swin-T slične računske složenosti ResNeta-50, a Swin-S ResNeta-101.

Slika 15 pod b) detaljnije prikazuje dva uzastopna bloka transformera Swin unutar neke razine. Razlikuju se u vrsti pažnje koju provode – W-MSA ili SW-MSA. Obje su višeglavne pažnje koje se računaju unutar svakog prozora zasebno, no SW-MSA ima pomak prozora, a W-MSA radi nad inicijalnim particioniranjem bez pomaka. U slučaju kad je više od 2 Swin bloka u razini, oni se naizmjenično ponavljaju. To je slučaj u 3. razini gdje je 6 Swin blokova što znači 3 takva para. Ponovimo, mapa značajki se particionira u fiksan broj prozora veličine M x M regija. Slika 14 daje primjer mape 8x8 i

M = 4 pa ima 2x2 prozor u sloju l. U sloju l+1 se prozori pomiču za floor(M / 2) po širini i visini što ovdje pomak od 2 regije od inicijalne particije u sloju l. Pogledajmo sad primjenu modula pažnje W-MSA i SW-MSA u (9). Prvo se normira izlaz prethodnog bloka (blok l-1), zatim se u trenutnom bloku (blok l) pojedine razine računa W-MSA nad tim ulazom i izlaz te pažnje se zbraja s izlazom prethodnog bloka čime se dobiju konačne izlazne značajke modula W-MSA. Te značajke se normiraju i provuku kroz unaprijedni model MLP pa se rezidualnom vezom pribroje izlazu MLP-a čime se dobiju konačne značajke modula MLP. Sad se tako dobiveni izlaz ovog MLP-a šalje u naredni transformerski blok gdje se sve isto ponavlja osim umjesto W-MSA je SW-MSA.

$$\begin{aligned}
\hat{\mathbf{z}}^l &= \text{W-MSA}\left(\text{LN}\left(\mathbf{z}^{l-1}\right)\right) + \mathbf{z}^{l-1}, \\
\mathbf{z}^l &= \text{MLP}\left(\text{LN}\left(\hat{\mathbf{z}}^l\right)\right) + \hat{\mathbf{z}}^l, \\
\hat{\mathbf{z}}^{l+1} &= \text{SW-MSA}\left(\text{LN}\left(\mathbf{z}^l\right)\right) + \mathbf{z}^l, \\
\mathbf{z}^{l+1} &= \text{MLP}\left(\text{LN}\left(\hat{\mathbf{z}}^{l+1}\right)\right) + \hat{\mathbf{z}}^{l+1},
\end{aligned} \tag{9}$$

Dodatan element koji SWIN dodaje u izračun samopozornosti je relativna pozicijska pristranost što je element B u (10). Matrice Q, K i V ovdje su dimenzije M^2 x d (ne više N x d jer je tu broj tokena jednak broju regija koje zahvaća prozora) pa je B dimenzije M^2 x M^2. Kako je relativna pozicija iz [-(M-1), M-1], ukupno je 2M-1 relativnih pozicija u odnosu na dani prozor pa se kreira manja parametrizirana matrica dimenzija (2M-1)x(2M-1) iz koje se uzimaju vrijednosti za B. Eksperimentalno je pokazano da je takva pristranost bolja od apsolutne pozicijske pristranosti ili nekorištenja ovog člana pozicijske pristranosti.

$$\text{Attention}(Q, K, V) = \text{SoftMax}(QK^T/\sqrt{d} + B)V, \tag{10}$$

2.2. Konvolucijski modeli za semantičku segmentaciju

2.2.1. SwiftNet

SwiftNet je konvolucijski model tipa koder-dekoder za gustu predikciju koji uvodi inovativnu ideju dijeljenih piramidalnih reprezentacija i fuzije heterogenih značajki unutar dekodera. Dijeljenje značajki ima jak regularizacijski učinak zbog čega je SwiftNet upravo posebno dobar za predikciju na razini piksela na slikama s vrlo raznolikim mjerilima.

Prikažimo prvo osnovni blok kodera na primjeru ResNet-18 što je jedna od standardnih kralješnica koje SwiftNet koristi u koderu. Slika 16 prikazuje zadnju rezidualnu jedinicu danog rezidualnog bloka kodera. Ulaz u rezidualni blok prolazi kroz 3x3 konvoluciju, sloj

normiranja po minigrupama *batchnorm*, aktivaciju zglobnice ReLU, zatim ponovo 3x3 konvoluciju i *batchnorm* te se taj izlaz sumira s ulazom koji dolazi rezidualnom vezom i na to se još primjeni ReLU. Vrijednost ReLU-a na kraju rezidualne jedinice šalje se u sljedeći blok kodera, no značajke koje odlaze lateralnom vezom prema odgovarajućem bloku dekodera nose vrijednost sume prije tog ReLu-a, a ne vrijednost ReLU-a jer je takva konfiguracija pokazala bolju točnost.

Slika 16. Arhitektura zadnje rezidualne jedinice pojedinog bloka kodera SwiftNeta iz [3]. Strukture je normalne rezidualne jedinice. Izlaz (ReLU desno) se šalje u idući blok kodera, a suma (zeleni krug) se latelatnom vezom šalje u odgovarajući blok naduzorkovanja u dekoderu SwiftNeta.

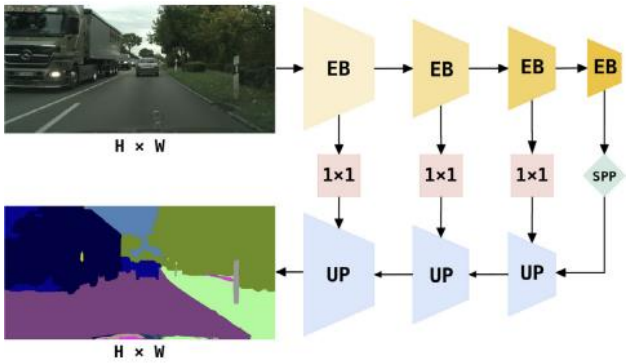
Sad opišimo osnovni dizajn SwiftNeta što je *singlescale* SwiftNet (Slika 17). On nema piramidu rezolucija, već radi na samo 1 slici. Sastoji se od kodera, SPP-a i dekodera. Koder se implementira uglavnom kao ResNet-18 [26] ili MobileNet V2 [27] i sastoji se od 4 bloka EB (žuti trapezi - Slika 17) koji redom daju značajke rezolucije redom 4, 8, 16 i 32 puta manje. Cilj kodera je dobiti značajke visoke dimenzije koje se mogu koristiti za prepoznavanje.

Za dekoder koristi lagani modul za ljestvičasto naduzorkovanje kao u [15] i sastoji se od 3 bloka naduzorkovanja UP (plavi trapezi - Slika 17). Cilj dekodera je kroz nekoliko blokova naduzorkovati značajke na rezoluciju ulazne slike. To postiže postupnim spajanjem reprezentacije više rezolucije iz odgovarajućeg bloka kodera sa semantički jačim reprezentacijama iz prethodnog sloja dekodera (ove su značajke niske rezolucije radi smanjenja vremensko-prostornih zahtjeva). Prije tog spajanja se značajke iz prethodnog sloja bilinearno naduzorkuju na rezoluciju značajki koje dolaze iz bloka kodera, a značajke iz bloka kodera se prvo provuku kroz 1x1 konvoluciju (crveni kvadrati - Slika 17) jer su većih dimenzija. Spajanje se sastoji od zbrajanja tih dvaju ulaza i 3x3 konvolucije.

Primijetimo sada da su koder i dekoder asimetrični što je suprotno tradicionalnih modela tipa koder-dekoder koji su simetrični. Asimetričnost se očituje u tome što blokovi kodera imaju niz konvolucija 3x3, dok blokovi dekodera samo jednu takvu konvoluciju i također koder povećava dimenziju značajki, a dekoder ostavlja dimenziju istom. SwiftNet odabire navedeni asimetričan dizajn jer smatra da dekoderu, koji doprinosi lociranju semantičkih granica, ne treba toliki kapacitet kao koderu, koji služi za prepoznavanje. Stoga gradi

lagani dekoder s minimalne složenosti, a da točnost i dalje ostane na razini dekodera u simetričnim arhitekturama.

Spomenimo još modul SPP (romb - Slika 17) koji koristi iz [15] (opisan u 1. poglavlju). SPP modul služi širenju receptivnog polja tako što generira mape značajki koristeći rešetke dimenzija 1x1, 2x2, 4x4 i 8x8 (napomena: pri zaključivanju se mijenja omjer širina/visina tako da npr. umjesto 1x1 se koristi 1x2 jer se koriste slike npr. čitave rezolucije 1024x2048 za Cityscapes umjesto kvadratnih isječaka 768x768 koji se koriste u učenju).

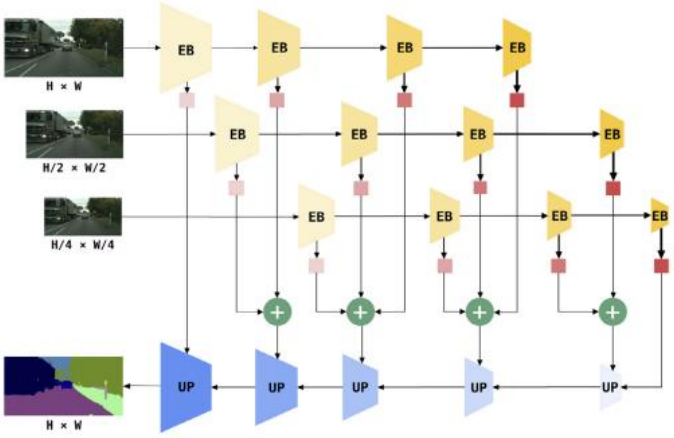


Slika 17 Arhitektura *singlescale* SwiftNeta iz [3]. Na ulazu je slika jedne (originalne) rezolucije koja se šalje u kodera od 4 bloka EB. Na kraju kodera se primjeni modul SPP čiji se izlaz šalje u dekoder. Dekoder se sastoji od 3 bloka naduzorkovanja UP. U pojedinom bloku UP se zbraja izlaz prethodnog bloka UP i odgovarajućeg bloka EB te se primijeni 3x3 konvolucija. Prije zbrajanja se izlaz prethodnog UP naduzorkuje na rezoluciju izlaza koji dolazi iz EB, a izlaz EB-a prolazi kroz konvoluciju 1x1 kako bi se dimenzija dovela na dimenziju dekodera. Izlaz dekodera su logiti 4x manjih dimenzija od dimenzija slike pa se bilinearno naduzorkuju.

Iako je *singlescale* SwiftNet dobar *baseline* model, *multiscale* SwiftNet s piramidalnom fuzijom tj. piramidalni SwiftNet (Slika 18) je još bolji. Predstavimo konačnu još njegovu ideju i objasnimo zašto je takav dizajn bolji. Standardna konfiguracija piramidalnog SwiftNeta ima 3 razine piramide (rezolucije H x W, H/2 x W/2 i H/4 x W/4). Na svakoj razini koder neovisno izlučuje značajke. Odgovarajuće instance blokova kodera EB na svim razinama dijele parametre (indicirano istom nijansom žute boje - Slika 18) što omogućava: i) prepoznavanje objekata različitih mjerila s istim parametrima, dok SPP ne omogućava kovarijantnost mjerila u koderu), i ii) povećava receptivno polje, što i SPP radi, no ovdje tako što kod predikcije značajke iz grubljih rezolucija stoje na raspolaganju. Uz EB, instance konvolucije 1x1 (crveni kvadrati - Slika 18) koje smanjuju dimenzionalnost značajki iz EB također dijele parametre kroz sve slojeve. Blokovi UP su ekvivalentni onima u osnovnom modelu. Na kraju se logiti projeciraju 4x bilinearno naduzorkuju jer su

rezolucije H/4 x W/4 kako bi došli na rezoluciju slike H x W.

Završni i ključni element je zbrajanje značajki iz blokova EB iste rezolucije s različitih razina rezolucijske piramide (zeleni krugovi s '+' - Slika 18) – to je piramidalna fuzija značajki. Piramidalna fuzija je korak prema još uspješnijem učenju pogotovo zato što spaja heterogene značajke različite semantike odnosno spaja reprezentacije različite razine apstrakcije, što nije uobičajeno. Također, s više razina piramide još se dodatno proširuje receptivno polje i pojačava regularizacijski učinak na koder. Konačno, piramidalna je varijanta SwiftNeta veće generalizacijske moći jer djeluju kao ansambl manjih modela.



Slika 18 Arhitektura piramidalnog (*multiscale*) SwiftNeta iz [3]. Na ulazu je rezolucijska piramida s 3 razine – slika originalnih, 2x i 4x manjih dimenzija. Na sve 3 razine se neovisno izlučuju značajke, instance blokova kodera EB i konvolucija 1x1 koje iza njih slijede međusobno dijele parametre što je prikazano istom nijansom boje. Debljina strelica indicira dimenziju značajki koja se povećava prema dubini kodera. Izlazi blokova EB iste dimenzije značajki provučeni kroz konvolucije 1x1 se zbrajaju piramidalnom fuzijom (zeleni +). Tako spojene značajke se prosljeđuju odgovarajućem bloku UP. Konačno se logiti 4x naduzorkuju.

Treba spomenuti na kraju još jednu komponentu piramidalnog SwiftNeta koja ublažava negativni efekt prenaučenosti na grubljim razinama rezolucijske piramide zbog kojih je točnost bila lošija na malim slikama – to je gubitak koji je „svjestan“ semantičkih granica: *boundary-aware loss*. Taj gubitak je negativna log-izglednost s prikladnim prilagodbama definiran u (10) desno. On prioritizira piksele blizu semantičkih granica tako što za piksele (i, j) koji se bliže granici (udaljenost d^{ij} je manja) faktor α^{ij} je veći ((10) desno) pa će piksel čija je vjerojatnost p_t^{ij} da pripada točnoj klasa t mala biti više kažnjen. Slično, eksponencijalni član je već kad je ta vjerojatnost manja.

$$\alpha^{ij} = \begin{cases} 8, & \text{if } d^{ij} \in [0, 15] \\ 4, & \text{if } d^{ij} \in [16, 63] \\ 2, & \text{if } d^{ij} \in [64, 127] \\ 1, & \text{if } d^{ij} > 127 \end{cases} \qquad L^{ij} = -\alpha^{ij} e^{\gamma(1-p_t^{ij})} \log p_t^{ij} \tag{10}$$

Ovaj gubitak i piramidalna fuzija čine SwiftNet modernim modelom za gustu predikciju.

3. Programska izvedba i vanjske biblioteke

3.1. Vanjske biblioteke

Najvažnije vanjske biblioteke koje se koriste u implementaciji (prilagođenom preuzetom kodu i kodu koji smo sami napisali) su:

- PyTorch [30] – radni okvir za automatsku difererencijaciju u Pythonu
- Numpy [31] – matematička biblioteka za Python
- Pillow [32] – biblioteka za obavljanje operacija nad slikama za Python
- Timm [33] – biblioteka za instanciranje *state-of-the-art* modela u CV-u za Python
- Mmseg [34] – biblioteka za semantičku segmentaciju napisana u PyTorchu
- Mmcv [35] – biblioteka za primjene u računalnom vidu napisana u PyTorchu

Svoje implementacije baziramo na službenim implementacijama dostupnim na službenim Github repozitorijima radova:

- SwiftNet [36]
- Segmenter [18]
- SWIN [19] i [37]

3.2. Programska izvedba

U ovom radu učili smo i evaluirali transformerske modele Segmenter i SWIN, konvolucijski model SwiftNet i hibridni model Swiftnet-Segmenter u obliku ansambala i združenih modela. Pritom smo razvili funkcije za mjerenje vremenske i prostorne složenosti kao i računske složenosti modela. Tako su sljedeća potpoglavlja organizirana po projektima (1 skupina modela – 1 projekt). Popisat ćemo glavne implementirane funkcije u svakom projektu. Za SwiftNet ćemo navesti kodove funkcija za mjerenja učinkovitosti, a za Segmenter i SWIN gdje imamo ekvivalentne funkcije nećemo ponavljati implementaciju koja je slična, no prilagođena pojedinom modelu. Za daljnje detalje implementacije se čitatelj upućuje na izvorni kod.

3.2.1. Projekt: SwiftNet

Implementirali smo sljedeće funkcije za mjerenje performansi evaluacije:

- *do_eval_step(model, device, im_sh, target_size_feats, bs)* – Slika 19
    - funkcija u swiftnet/eval.py
    - radi evaluaciju – 1 unaprijedni prolaz modelom *model* nad slikom dimenzija *im_sh* koji se nalazi na uređaju *device* uz veličinu minigrupe *bs*
    - pozivaju je druge funkcije za mjerenje performansi evaluacije
- *get_max_inputsize_eval(model, init_im_sh, init_target_size_feats)* – Slika 20
    - funkcija u swiftnet/eval.py
    - pronalazi najveću rezoluciju slike oblika H x 2H pri evaluaciji modela
- *get_max_batchsize_eval(model, im_sh, target_size_feats)* – Slika 21
    - funkcija u swiftnet/eval.py
    - pronalazi najveću veličinu minigrupe koju model može evaluirati za sliku dimenzija *im_sh*
- *measure_speed_eval_fps(model, bs, im_sh, target_size_feats, n)* – Slika 22
    - funkcija u swiftnet/eval.py
    - mjeri brzinu broja slika u sekundi (fps) za evaluaciju modela *model* na veličini grupe *bs* i sliku dimenzija *im_sh*; mjerenje ponavlja *n* puta i računa srednju brzinu
- *measure_speed_eval_wrt_inputsize(model, max_im_sh, max_target_size_feats, n)* – Slika 23
    - funkcija u swiftnet/eval.py
    - mjeri brzinu broja slika u sekundi (fps) za evaluaciju modela *model* na veličini grupe 1 i sliku dimenzija *max_im_sh* što je maksimalna veličina slike na kojoj se *model* može evaluirati; mjerenje ponavlja *n* puta i računa srednju brzinu
- *measure_macs_params(model)* – Slika 24

- funkcija u swiftnet/train.py
- mjeri računsku zahtjevnost (u MAC-ovima) i broj parametara modela na slici dimenzija 1024x2048

Implementirali smo sljedeće funkcije za mjerenje performansi učenja:

- *do_train_step (model, optimizer, im_sh, crop_size, num_classes, bs, batch, dist_trans_alphas, record_memory)* – Slika 25
  - funkcija u swiftnet/train.py
  - radi 1 korak učenja – 1 unaprijedni i unatražni prolaz modelom *model* optimizatorom *optimizer* nad slikom originalnih dimenzija *im_sh* i dimenzija isječka za učenje *crop_size*, na veličini grupe *bs*, model se nalazi na uređaju *device*, postoji *num_classes* klasa u podatkovnom skupu; u nekim slučajevim se predaje minigrupa *batch* umjesto instanciranja podataka unutar ove funkcije; ima opciju *record_memory* koja ako je *True*, vraća memorijsko zauzeće *max_mem* (zauzeto parametrima modela i momentima) i *mem_bwd* – *max_mem* (memorijsko zauzeće aktivacija)
  - pozivaju je druge funkcije za mjerenje performansi učenja
- *get_max_batchsize_train(model, optimizer, dist_trans_alphas, im_sh, crop_size, num_classes)* – Slika 26
  - funkcija u swiftnet/train.py
  - pronalazi najveću veličinu minigrupe na kojoj model može učiti za sliku originalnih dimenzija *im_sh* i veličinu isječka za učenje *crop_size*
- *measure_speed_train_fps(model, bs, optimizer, dist_trans_alphas, im_sh, crop_size, num_classes, n)* – Slika 27
  - funkcija u swiftnet/train.py
  - mjeri brzinu broja slika u sekundi (fps) za učenje modela *model* optimizatorom *optimizer* na veličini grupe *bs* i sliku originalnih dimenzija *im_sh* i veličinu isječka za učenje *crop_size*; mjerenje ponavlja *n* puta i računa srednju brzinu

- *train_one_epoch_duration(model, optimizer, dana_loader)* i blok koda pod „*elif args.measure_train_duration*“ granom u bloku „__main__“ – Slika 28
    - funkcija i blok koda u swiftnet/train.py
    - mjeri procijenjeno trajanje jedne epohe učenja modela *model* optimizatorom *optimizer* s podacima koje učitava iz *data_loader*; mjerenje ponavlja na *args.n_epoch* epoha i računa srednje trajanje epohe
- blok koda pod „*elif args.log_memory*“ granom u bloku „__main__“ – Slika 29
    - blok koda u swiftnet/train.py
    - mjeri zauzeće memorije (izraženo u GB) na jednoj isječku za učenje veličine *crop_size* modela *model* optimizatorom *optimizer* na slici originalnih dimenzija *im_sh* i skupu s *num_classes* klasa; mjerenje ponavlja 10 puta i uzima rezultat iz zadnje iteracije

```
def do_eval_step(model, device, im_sh, target_size_feats=None, bs=None):
    torch.cuda.empty_cache()
    im = torch.randint(low=0, high=256, size=(bs, 3, *im_sh)).float().to(device)

    logits = model.forward(im, target_size_feats, im_sh, ret_add=False)
    pred = torch.argmax(logits, dim=1)

    del logits, pred

    torch.cuda.synchronize()
```

Slika 19 Funkcija do_eval_step.

```python
def get_max_inputsize_eval(model, init_im_sh=(64,128), init_target_size_feats=(16,32)):
    model.eval()

    model_dev = next(model.parameters()).device

    double_dim = lambda shape_tup : tuple(2*d for d in shape_tup)
    half_dim = lambda shape_tup: tuple(d//2 for d in shape_tup)

    factor_feats = init_im_sh[0] // init_target_size_feats[0]
    scale_feats = lambda shape_tup: tuple(d//factor_feats for d in shape_tup)

    with torch.no_grad():
        cur_im_sh, cur_target_size_feats = init_im_sh, init_target_size_feats
        while True:
            try:
                print("cur_im_sh =", cur_im_sh, cur_target_size_feats)
                do_eval_step(model, model_dev, im_sh=cur_im_sh, target_size_feats=cur_target_size_feats, bs=1)

                cur_im_sh = double_dim(cur_im_sh)
                cur_target_size_feats = scale_feats(cur_im_sh)
            except RuntimeError as err:
                print("!!! err =", err)
                top_im_sh, top_target_size_feats = cur_im_sh, cur_target_size_feats
                bottom_im_sh = half_dim(cur_im_sh)
                bottom_target_size_feats = scale_feats(bottom_im_sh)
                break

        print("bottom =", bottom_im_sh, bottom_target_size_feats)
        print("top =", top_im_sh, top_target_size_feats)
        while (top_im_sh[0] - bottom_im_sh[0]) > 1:
            mid_im_sh = half_dim(tuple(d1+d2 for d1,d2 in zip(top_im_sh, bottom_im_sh)))
            mid_target_size_feats = scale_feats(mid_im_sh)
            print("mid_im_sh =", mid_im_sh, mid_target_size_feats)
            try:
                do_eval_step(model, model_dev, im_sh=mid_im_sh, target_size_feats=mid_target_size_feats, bs=1)
                bottom_im_sh, bottom_target_size_feats = mid_im_sh, mid_target_size_feats
            except RuntimeError:
                top_im_sh, top_target_size_feats = mid_im_sh, mid_target_size_feats

        print("bottom =", bottom_im_sh, bottom_target_size_feats)
        print("top =", top_im_sh, top_target_size_feats)
        try:
            do_eval_step(model, model_dev, im_sh=top_im_sh, target_size_feats=top_target_size_feats, bs=1)
            max_im_sh = top_im_sh
        except RuntimeError as err:
            print("!!! err =", err)
            max_im_sh = bottom_im_sh

    return max_im_sh
```

Slika 20 Funkcija get_max_inputsize_eval.

```python
def get_max_batchsize_eval(model, im_sh=(1024,2048), target_size_feats=(256,512)):
    model.eval()

    model_dev = next(model.parameters()).device

    with torch.no_grad():
        cur_bs = 2
        while True:
            try:
                print("cur_bs =", cur_bs)
                do_eval_step(model, model_dev, im_sh, target_size_feats=target_size_feats, bs=cur_bs)

                cur_bs *= 2
            except RuntimeError as err:
                print("!!! err =", err)
                top_bs = cur_bs
                bottom_bs = cur_bs // 2
                break

        print("bottom =", bottom_bs)
        print("top =", top_bs)
        while (top_bs - bottom_bs) > 1:
            mid_bs = (top_bs + bottom_bs) // 2
            print("mid_bs =", mid_bs)
            try:
                do_eval_step(model, model_dev, im_sh, target_size_feats=target_size_feats, bs=mid_bs)
                bottom_bs = mid_bs
            except RuntimeError:
                top_bs = mid_bs

        print("bottom =", bottom_bs)
        print("top =", top_bs)
        try:
            do_eval_step(model, model_dev, im_sh, target_size_feats=target_size_feats, bs=top_bs)
            max_bs = top_bs
        except RuntimeError as err:
            print("!!! err =", err)
            max_bs = bottom_bs

    return max_bs
```

Slika 21 Funkcija get_max_batchsize_eval.

```python
def measure_speed_eval_fps(model, bs, im_sh=(1024,2048), target_size_feats=(256,512), n=100):
    import torch
    import time

    model.eval()

    warm_up = 10

    model_dev = next(model.parameters()).device

    ###############
    torch.cuda.empty_cache()
    im = torch.randint(low=0, high=256, size=(bs, 3, *im_sh)).float().to(model_dev) # float32
    ###############

    with torch.no_grad():
        for i in range(warm_up):
            torch.cuda.empty_cache()
            logits = model.forward(im, target_size_feats, im_sh, ret_add=False)
            pred = torch.argmax(logits.data, dim=1)
            del logits, pred

    tot_time = 0
    with torch.no_grad():
        for _ in range(n):
            torch.cuda.empty_cache()

            torch.cuda.synchronize()
            start = time.time()

            logits = model.forward(im, target_size_feats, im_sh, ret_add=False)
            pred = torch.argmax(logits, dim=1)

            torch.cuda.synchronize()
            end = time.time()

            tot_time += (end - start)

            del logits, pred

    return (bs * n) / tot_time
```

Slika 22 Funkcija measure_speed_eval_fps.

```python
def measure_speed_eval_wrt_inputsize(model, max_im_sh=(1024, 2048), max_target_size_feats=(256,512), n=100):
    im_sh = (64, 128)
    max_im_h, max_im_w = max_im_sh
    double_dim = lambda shape_tup: tuple(2 * d for d in shape_tup)

    factor_feats = max_im_sh[0] // max_target_size_feats[0]
    scale_feats = lambda shape_tup: tuple(d//factor_feats for d in shape_tup)

    while im_sh[0] < max_im_h:
        print("im_sh =", im_sh)
        speed_result = measure_speed_eval_fps(model, bs=1, im_sh=im_sh, target_size_feats=scale_feats(im_sh), n=n)
        print("FPS = ", speed_result)

        im_sh = double_dim(im_sh)

    print("im_sh = MAX IM SH =", max_im_sh)
    speed_result = measure_speed_eval_fps(model, bs=1, im_sh=max_im_sh, target_size_feats=max_target_size_feats, n=n)
    print("FPS = ", speed_result)
```

Slika 23 Funkcija measure_speed_eval_wrt_inputsize.

```python
def measure_macs_params(model):
    from thop import profile
    model.eval()
    inp = torch.randn(1, 3, 1024, 2048).to(next(model.parameters()).device)
    macs, params = profile(model, inputs=(inp,(1024,2048),(1024,2048)))

    return macs, params
```

Slika 24 Funkcija measure_macs_params.

```python
def do_train_step(model, optimizer, im_sh=(1024, 2048), crop_size=768, num_classes=19, bs=None, batch=None, dist_trans_alphas=None, record_memory=False):
    if batch is None:
        torch.cuda.empty_cache()
        batch = {}
        batch['original_labels'] = torch.randint(low=0, high=num_classes+1, size=(bs, *im_sh))   #int64
        batch['labels'] = torch.randint(low=0, high=num_classes+1, size=(bs, crop_size, crop_size))   #int64
        batch['target_size'] = [crop_size,crop_size]
        batch['image'] = torch.randint(low=0, high=256, size=(bs, 3, crop_size, crop_size)).float() #float32

        if dist_trans_alphas is not None:
            section = crop_size//(len(dist_trans_alphas)+1)
            batch['label_distance_alphas'] = torch.zeros((bs, crop_size, crop_size),dtype=torch.float32) #float32
            for i in range(len(dist_trans_alphas)):
                batch['label_distance_alphas'][:,i*section:(i+1)*section,...] = dist_trans_alphas[i]

    optimizer.zero_grad()
    loss = model.loss(batch)

    loss.backward()

    mem_bwd = torch.cuda.max_memory_allocated()
    torch.cuda.reset_max_memory_allocated()

    optimizer.step()

    del loss
    torch.cuda.synchronize()

    if record_memory:
        max_mem = torch.cuda.max_memory_allocated(model_dev)
        return max_mem, mem_bwd - max_mem
```

Slika 25 Funkcija do_train_step.

```python
def get_max_batchsize_train(model, optimizer, dist_trans_alphas=None, im_sh=(1024,2048), crop_size=768, num_classes=19):
    model.train()

    cur_bs = 2
    while True:
        try:
            print("cur_bs =", cur_bs)
            do_train_step(model, optimizer, im_sh, crop_size, num_classes, bs=cur_bs, dist_trans_alphas=dist_trans_alphas,)

            cur_bs *= 2
        except RuntimeError as err:
            print("!!! err =", err)
            top_bs = cur_bs
            bottom_bs = cur_bs // 2
            break

    print("bottom =", bottom_bs)
    print("top =", top_bs)
    while (top_bs - bottom_bs) > 1:
        mid_bs = (top_bs + bottom_bs) // 2
        print("mid_bs =", mid_bs)
        try:
            do_train_step(model, optimizer, im_sh, crop_size, num_classes, bs=mid_bs, dist_trans_alphas=dist_trans_alphas)
            bottom_bs = mid_bs
        except RuntimeError:
            top_bs = mid_bs

    print("bottom =", bottom_bs)
    print("top =", top_bs)
    try:
        do_train_step(model, optimizer, im_sh, crop_size, num_classes, bs=top_bs, dist_trans_alphas=dist_trans_alphas)
        max_bs = top_bs
    except RuntimeError as err:
        print("!!! err =", err)
        max_bs = bottom_bs

    return max_bs
```

Slika 26 Funkcija get_max_batchsize_train.

```python
def measure_speed_train_fps(model, bs, optimizer, dist_trans_alphas=None, im_sh=(1024,2048), crop_size=768, num_classes=19, n=30):

    import torch
    import time

    model.train()

    warm_up = 10

    ###############
    torch.cuda.empty_cache()
    batch = {}
    batch['original_labels'] = torch.randint(low=0, high=num_classes + 1, size=(bs, *im_sh))  # int64
    batch['labels'] = torch.randint(low=0, high=num_classes + 1, size=(bs, crop_size, crop_size))  # int64
    batch['target_size'] = [crop_size, crop_size]
    batch['image'] = torch.randint(low=0, high=256, size=(bs, 3, crop_size, crop_size)).float()  # float32

    if dist_trans_alphas is not None:
        section = crop_size // (len(dist_trans_alphas) + 1)
        batch['label_distance_alphas'] = torch.zeros((bs, crop_size, crop_size), dtype=torch.float32)  # float32
        for i in range(len(dist_trans_alphas)):
            batch['label_distance_alphas'][:, i * section:(i + 1) * section, ...] = dist_trans_alphas[i]
    ###############

    torch.cuda.synchronize()
    for i in range(warm_up):
        torch.cuda.empty_cache()
        print("i",i)
        do_train_step(model, optimizer, im_sh, crop_size, num_classes, batch=batch, dist_trans_alphas=dist_trans_alphas)

    start = time.time()

    for i in range(n):
        torch.cuda.empty_cache()
        do_train_step(model, optimizer, im_sh, crop_size, num_classes, batch=batch, dist_trans_alphas=dist_trans_alphas)

    torch.cuda.synchronize()

    end = time.time()
    return (bs * n) / (end - start)
```

Slika 27 Funkcija measure_speed_train.

```python
def train_one_epoch_duration(model, optimizer, data_loader):
    batch_iterator = iter(enumerate(data_loader))

    for i, batch in batch_iterator:
        if i % 50 == 0:
            print("i", i)

        torch.cuda.empty_cache()
        optimizer.zero_grad()
        loss = model.loss(batch)

        loss.backward()
        optimizer.step()

        del loss

        torch.cuda.synchronize()
```

```
epoch_times = []
print("BATCH_SIZE===", conf.loader_train.batch_size)
print("*************** NUM EPOCHS =", args.n_epoch, "***************")
for ep in range(args.n_epoch):
    torch.cuda.synchronize()
    start = time.time()

    # train for one epoch
    train_one_epoch_duration(conf.model, conf.optimizer, conf.loader_train)

    torch.cuda.synchronize()
    end = time.time()

    epoch_time = (end - start)
    print(">>>>>>>>>>>>>>>> TIME =", epoch_time, ">>>>>>>>>>>>>>>>")

    epoch_times.append(epoch_time)

avg_epoch_time = 0.0
for i in range(args.n_epoch):
    print("epoch {} : {}".format(i, epoch_times[i]))
    avg_epoch_time += epoch_times[i]
avg_epoch_time /= args.n_epoch
print(">>>>>>>>>>>>>>>> AVERAGE EPOCH TIME =", avg_epoch_time, ">>>>>>>>>>>>>>>>")
```

Slika 28 Funkcija train_one_epoch_duration i blok koda koji je poziva *args.n_epoch* puta.

```
for i in range(10):
    param_moments_mem, activations_mem = do_train_step(conf.model, conf.optimizer, im_sh=(1024, 2048), crop_size=768,
        num_classes=19, bs=args.log_memory_batchsize, dist_trans_alphas=dist_trans_alphas, record_memory=True)

print("MEMORY:")
print("mem alloc before train:", mem_alloc_before_train)
GB = 1024.0 * 1024.0 * 1024.0
print("MEMORY REQUIREMENTS FOR TRAIN: (PARAMS + MOMENTS, ACTIVATIONS) = ({}, {}) =({} GB, {} GB)".format(
    param_moments_mem, activations_mem, param_moments_mem / GB, activations_mem / GB))
```

Slika 29 Blok koda za mjerenje memorijskog otiska modela pri učenju.

3.2.2. Projekt: Segmenter

Kako smo za Segmenter implementirali istovjetne funkcija onima za SwiftNet (uz prilagodbe za Segmenter), navedimo samo imena i potpise tih funkcija uz lokaciju.

Funkcije za mjerenje performansi evaluacije:

- *do_eval_step(model, variant, im_sh, bs, im)* – u segm/engine.py
- *get_max_inputsize_eval(model, variant, init_im_sh)* – u segm/engine.py
- *get_max_batchsize_eval(model, variant, im_sh)* – u segm/engine.py
- *measure_speed_eval_fps(model, variant, bs, im_sh, n)* – u segm/engine.py
- *measure_speed_eval_wrt_inputsize(model, variant, max_im_sh, n)* – u segm/engine.py

- *measure_macs_params(model)* – u segm/train.py

Funkcije za mjerenje performansi učenja:

- *do_train_step (model, criterion, optimizer, amp_autocast, loss_scaler, bs, crop_size, num_classes, im, seg_gt, record_memory)* – u segm/engine.py
- *get_max_batchsize_train(model, optimizer, amp_autocast, loss_scaler, crop_size)* – u segm/engine.py
- *measure_speed_train_fps(model, bs, optimizer, amp_autocast, loss_scaler, crop_size, num_classes, n)* – u segm/engine.py
- *train_one_epoch_duration(model, data_loader, optimizer, amp_autocast, loss_scaler)* – u segm/engine.py + blok koda pod „*elif measure_train_duration*“ granom u funkciji *main* – u segm/train.py
- blok koda pod „*elif log_memory*“ granom u funkciji *main* – u segm/train.py

3.2.3. Projekt: SWIN

Također, kao i za SWIN smo radi usporedbe sa Segmenterom i SwiftNetom implementirali iste funkcije kao i prije uz prilagodbe za SWIN.

Funkcije za mjerenje performansi evaluacije:

- *do_eval_step(model, bs, im_sh, imgs, img_metas)* – u tools/test.py
- *get_max_inputsize_eval(model, distributed, init_im_sh)* – u tools/test.py
- *get_max_batchsize_eval(model, distributed)* – u tools/test.py
- *measure_speed_eval_fps(model, bs, im_sh, distributed, n, inst_model)* – u tools/test.py
- *measure_speed_eval_wrt_inputsize(model, distributed, max_im_sh, n)* – u tools/test.py
- *measure_macs_params(model)* – u tools/test.py

Funkcije za mjerenje performansi učenja:

- *do_train_step (model, optimizer, crop_size, num_classes, bs, imgs, img_metas, gt_semantic_seg, record_memory)* – u tools/train.py

- *get_max_batchsize_train(model, cfg, distributed)* – u tools/train.py
- *measure_speed_train_fps(model, bs, cfg, distributed, crop_size, num_classes, n)* – u tools/train.py
- blok koda pod „*elif args.measure_train_duration*“ granom u funkciji *main* – u tools/train.py
- blok koda pod „*elif args.log_memory*“ granom u funkciji *main* – u tools/train.py

3.2.4. Projekt: Ansambli i arhitektura SwiftNet-Segmenter

Projekt evaluacije ansambala te učenja i evaluacije združenih modele SwiftNet-Segmenter smo implementirali u tri projekta. U jednom projektu je evaluacija ansambala i učenje i evaluacija združenih modela koji se uče sa zasebnim optimizatorima (SwiftNet se uči optimizatorom Adam sa svojim hiperparametrima, a Segmenter SGD-om sa svojim hiperparametrika). U 2. projektu je isto, ali se oba modela u združenom modelu uče samo s optimizatorom SGD s hiperparametrima Segmentera, a u 3. projektu je isto, ali se oba modela u združenom modelu uče samo s optimizatorom Adam s hiperparametrima SwiftNeta. Također, u svakom projektu gdje učimo združeni model imamo 2 varijante – 1) u Segmenter ulazi ulaz u 3. razinu SwiftNeta i 2) u Segmenter ulazi središnji isječak (slučajno pozicioniran i uvećan isječak koji se poklapa s mIoU od barem 50% (nevalidiran hiperparametar) sa središnjim isječkom iste veličine.

Prvo prikažimo glavne promjene u kodu gdje smo implementirali spoj Segmentera i SwiftNeta u združenom modelu (Slika 30). Prikazana funkcija *forward(self, image, seg_enc_out)* iz models/resnet/resnet_pyramid.py iz klase *ResNet* implementira unaprijedni prolaz modelom piramidalnog ResNeta koji piramidalni SwiftNet koristi kao kralješnicu u svim modelima koje smo učili. Prvi dio funkcije je isti kao kod te funkcije u kodu SwiftNeta, a drugi dio koji počinje od linije „*if seg_enc_out is None:*“ se razlikuje, uz također dodatni argument *seg_enc_out* ovoj funkciji. Taj argument je u slučaju ansambala jednak *None* pa se on ne koristi jer implementirani ansambli samo usrednjuju logite dobivene unaprijednim prolazim Segmenterom i SwiftNetom koji se odvijaju neovisno na istoj rezoluciji ulazne slike 1024x2048 (na Cityscapesu). Inače se radi o združenom modelu koji može biti u dvije varijante gdje se izlaz kodera Segmentera se spaja sa SwiftNetom 1) nakon (*self.add_seg_after = True*) ili 2) prije (*self.add_seg_after = False*) 1. bloka naduzorkovanja (UP) dekodera SwiftNeta. U oba slučaja se koristi 1x1

konvolucija *self.conv_seg_swift* (Slika 31) s brojem izlaznih kanala jednakim dimenziji značajki, a ulazni broj kanala te konvolucije je različiti: u 1) je jednak zbroju dimenzije kodera Segmentera i broju značajki jer se konvolucija primjenjuje na konkatenaciju izlaza kodera Segmentera i 1. bloka UP, a u 2) je to dimenzija kodera Segmentera. Dakle, u 1) se izlaz 1. bloka UP konkatenira s izlazom kodera Segmenter, primijeni se 1x1 konvolucija na tu konkatenaciju i dobiveno se normalno dalje šalje u 2. blok UP SwiftNeta, a u 2) se na izlaz kodera Segmentera primijeni 1x1 konvolucija, zatim se izlaz te konvolucije pribraja s ostala 2 pribrojnika koji ulaze u 1. blok UP piramidalnog SwiftNeta i dalje se ta suma normalno šalje u taj, 1. blok UP SwiftNeta.

Slika 32 prikazuje još promjenu u kodu Segmentera koja treba da se odredi treba li izlaz Segmentera biti izlaz njegovog kodera ili dekodera. To je kod metode *forward(self, im_return_masks)* u segm/model/segmenter.py koja radi unaprijedni prolaz modelom Segmentera. Ako se evaluira ansambl, izlaz će biti izlaz dekodera. U evaluaciji združenog modela, izlaz je izlaz kodera. U učenju združenog modela, izlaz je izlaz kodera ako se gubitak Segmentera ne koristi kao pomoćni gubitak pa dekoder Segmentera se ne koristi, a ako se gubitak Segmentera koristi kao pomoćni gubitak, dekoder Segmenter treba jer se iz njega računa gubitak pa je izlaz jednak izlazu dekodera u tom slučaju. O tome koristi li se pomoćni gubitak odlučuje vrijednost hiperparametra težine pomoćnog gubitka koja ako je jednaka 0, gubitak je jednak gubitku SwiftNeta pa se unatražni prolaz radi sam nad SwiftNetom, a inače je gubitak jednak zbroju gubitka SwiftNeta i gubitka Segmentera pomnoženog tim hiperparametrom pa se unatražni prolaz radi na ukupnim (zbrojenim) gubitkom tj. i nad koderom Segmentera.

Daljnje dijelove implementiranog koda čitatelj može promotriti u izvornom kodu jer su više tehničke prirode. Samo ćemo uputiti gdje se koja funkcija nalazi i što radi:

- *train(self)* – u train.py: provodi učenje združenog modela u varijanti gdje je ulaz u Segmenter jednak ulazu u 3. razinu piramide SwiftNeta (ulazna slika za Segmenter se dobiva na isti način kako SwiftNet dobiva ulaznu sliku za 3. razinu – bikubičnom interpolacijom slike na rezoluciju H/4 x W/4)
- *train_center_crop(self)* – u train.py: provodi učenje združenog modela u varijanti gdje je ulaz u Segmenter jednak slučajno uvećanom i slučajno pozicioniramo isječku ulazne slike koji se preklapa sa središnjim isječkom istih dimenzija s mIoU većim od ili jednakim pragu (pretpostavljeni prag mIoU = 50% - radi toga su

instancirana dvije instance skupa podataka u konfiguraciji piramidalnog ResNeta npr. u konfiguraciji configs/rn18_pyramid_centercrop.py i implementirana transformacija *RandomSquareCenterCropAndScale* u dana/transform/jitter.py po uzoru na *RandomSquareCenterCropAndScale*, samo s prilagodbom kako je navedeno da smo definirali središnji isječak koji šaljemo kao ulaz Segmenteru)

- *evaluate_semseg(model_swift, data_loader, class_info, model_seg, multiscale, window_size, window_stride, window_batch_size, observers, is_ensemble)* – u evaluation/evaluate.py: provodi evaluaciju ansambla (ili zasebnih modela)
- *evaluate_semseg_swift_seg(model_swift, data_loader, class_info, model_seg, img_dev, multiscale, observers)* – u evaluation/evaluate.py: provodi evaluaciju združenog modela gdje je Segmenter učen s ulazom jednakim ulazu u 3. razinu SwiftNeta
- *evaluate_semseg_swift_seg_center_crop(model_swift, data_loader, class_info, model_seg, img_dev, multiscale, observers)* – u evaluation/evaluate.py: provodi evaluaciju združenog modela gdje je Segmenter učen na središnjim isječcima

Na kraju, istovjetne funkcije su implementirane na istim lokacijama u svim trima projektima za učenje združenog modela – zasebni optimizator, optimizator SwiftNeta i optimizator Segmentera. Razlika je samo u tome koji se optimizatori instanciraju na početku i kako se njihove funkcije koraka učenja pozivaju u funkcijama za učenje *train* odnosno *train_center_crop*.

```
def forward(self, image, seg_enc_out=None):
    if isinstance(self.bn1[0], nn.BatchNorm2d):
        if hasattr(self, 'img_scale'):
            image /= self.img_scale
        image -= self.img_mean
        image /= self.img_std
    pyramid = [image]
    for l in range(1, self.pyramid_levels):
        if self.target_size is not None:
            ts = list([si // 2 ** l for si in self.target_size])
            pyramid += [
                F.interpolate(image, size=ts, mode=self.pyramid_subsample, align_corners=self.align_corners)]
        else:
            pyramid += [F.interpolate(image, scale_factor=1 / 2 ** l, mode=self.pyramid_subsample,
                                      align_corners=self.align_corners)]
    skips = [[] for _ in range(self.num_skip_levels)]
    additional = {'pyramid': pyramid}
    for idx, p in enumerate(pyramid):
        skips = self.forward_down(p, skips, idx=idx)
    skips = skips[::-1]
    x = skips[0][0]
    if self.detach_upsample_in:
        x = x.detach()

    if seg_enc_out is None:
        for i, (sk, blend) in enumerate(zip(skips[1:], self.upsample_blends)):
            x = blend(x, sum(sk))

        return x, additional

    seg_enc_out_repr_shape = seg_enc_out.shape[-2:]

    if self.add_seg_after:
        for i, (sk, blend) in enumerate(zip(skips[1:], self.upsample_blends)):
            x = blend(x, sum(sk))
            if x.shape[-2:] == seg_enc_out_repr_shape:
                concat_tens = torch.cat((x, seg_enc_out), dim=1)

                x = self.conv_seg_swift(concat_tens)
    else:
        seg_enc_out = self.conv_seg_swift(seg_enc_out)

        for i, (sk, blend) in enumerate(zip(skips[1:], self.upsample_blends)):
            if sk[0].shape[-2:] == seg_enc_out_repr_shape:
                sk.append(seg_enc_out)

            x = blend(x, sum(sk))

    return x, additional
```

Slika 30 Funkcija unaprijednog prolaza ResNeta – implementiran spoj Segmentera i SwiftNeta u združenom modelu.

```
if new_arch:
    if add_seg_after:
        conv_seg_swift_in_planes, conv_seg_swift_out_planes = seg_enc_d_model+num_features, num_features
    else:
        conv_seg_swift_in_planes, conv_seg_swift_out_planes = seg_enc_d_model, num_features
    self.add_seg_after = add_seg_after
    self.conv_seg_swift = nn.Conv2d(conv_seg_swift_in_planes, conv_seg_swift_out_planes, kernel_size=1, stride=1, padding=0, bias=False)
```

Slika 31 Konvolucijski sloj za prilagođavanje dimenzije izlaza kodera Segmentera u dekoderu SwiftNeta.

```
def forward(self, im, return_masks=True):
    H_ori, W_ori = im.size(2), im.size(3)
    im = padding(im, self.patch_size)
    H, W = im.size(2), im.size(3)

    x = self.encoder(im, return_features=True)

    # remove CLS/DIST tokens for decoding
    num_extra_tokens = 1 + self.encoder.distilled
    x = x[:, num_extra_tokens:]

    enc_out = rearrange(x, "b (h w) n -> b n h w", h=(H // self.patch_size))

    if not self.training and self.new_arch:
        return enc_out

    if not return_masks:
        return enc_out, None

    masks = self.decoder(x, (H, W))

    masks = F.interpolate(masks, size=(H, W), mode="bilinear")
    masks = unpadding(masks, (H_ori, W_ori))

    if self.new_arch:
        return enc_out, masks
    else:
        return masks
```

Slika 32 Funkcija unaprijednog prolaza Segmenter - prilagođena implementaciji združenog modela da vraća izlaz kodera ili dekodera.

4. Podatkovni skupovi

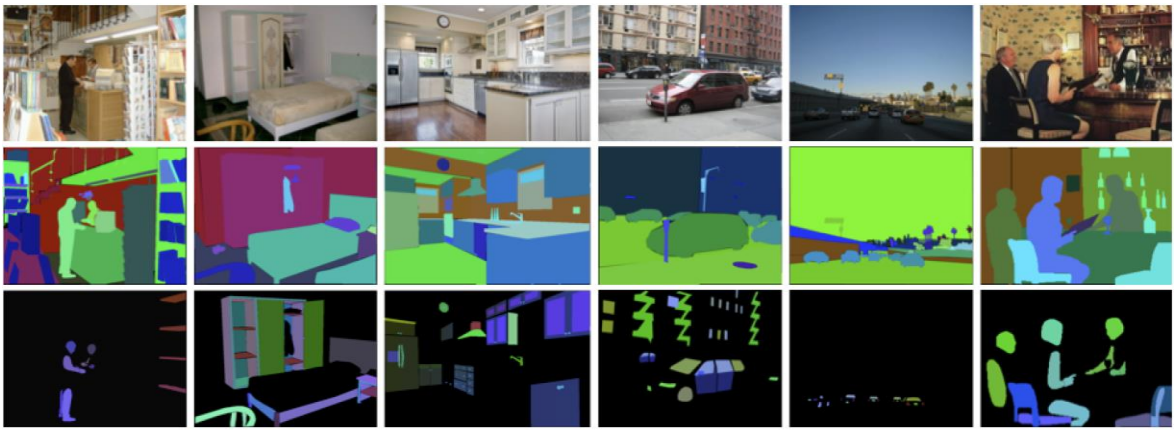
4.1. PASCAL Context

PASCAL Context [38] je skup od 60 klasa (59 klasa + klasa pozadine). Ima 4996 slika za učenje i 5104 za validaciju. Slika 33 daje primjer originalnih slika (1. red) i oznaka za segmentaciju (2. red).

Slika 33 Nekoliko primjera slika (1. red) i segmentacija (2. red) skupa Pascal Context [38].

4.2. ADE20K

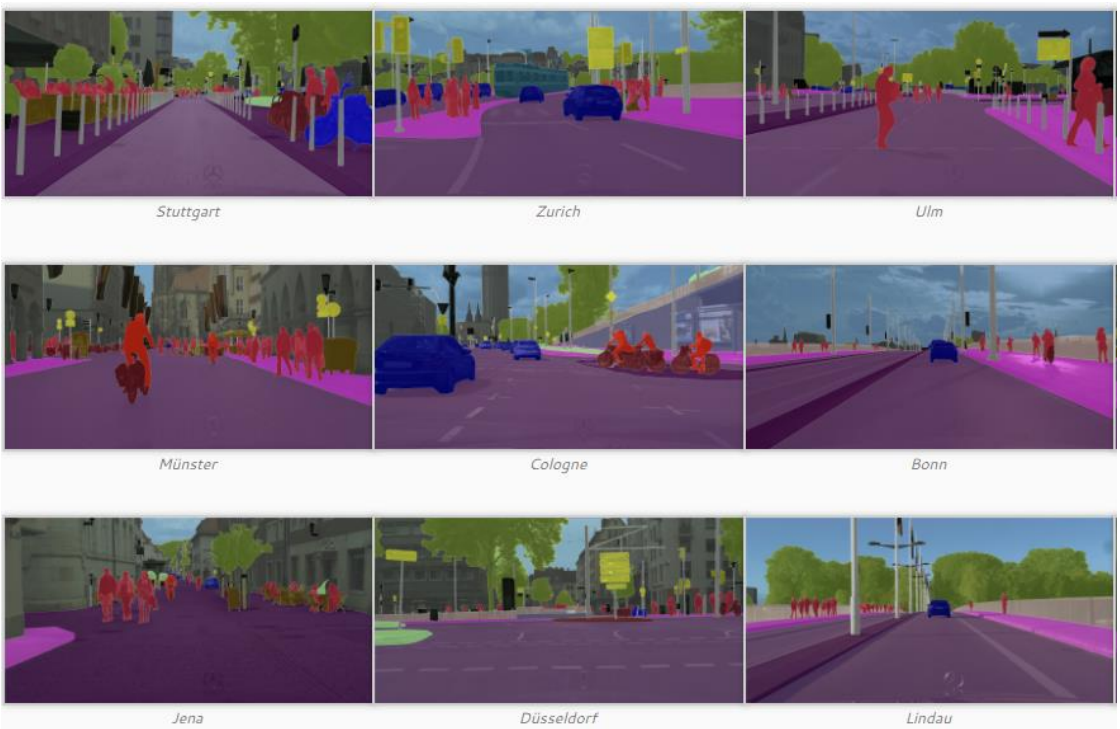
ADE20K [39] je skup od 150 klasa. Ima 20210 slika za učenje, 2000 za validaciju i 3352 za ispitivanje. Jedan je od najizazovnijih skupova za učenje. Medijan veličine slike je 512x480. Slika 34 daje primjer originalnih slika (1. red), oznaka za segmentaciju (2. red) i oznaka dijelova segmentacija (3. red).



Slika 34 Nekoliko primjera slika (1. red), segmentacija (2. red) i dijelova segmentacija (3. red) skupa ADE20K [39].

4.3. Cityscapes

Cityscapes [40] je skup od 30 klasa od čega se često koristi 19 klasa (ostalo je pozadina). Ima 2975 slika u skupu za učenje, 500 za validaciju i 1525 za ispitivanje. Slike su dimenzije 1024x2048. Slika 35 daje primjer oznaka za segmentaciju.



Slika 35 Nekoliko primjera segmentacija slika skupa Cityscapes [40].

5. Eksperimenti

Ovo poglavlje prikazuje eksperimente učenja i evaluacija odabranih transformerskih i konvolucijskih modela. Pojedini se eksperimenti djelomično preklapaju, nadopunjuju ili nadovezuju jedan na drugoga, no svaki donosi nova saznanja o istraživanim modelima.

5.1. Uvodni eksperimenti

5.1.1. Eksperimenti na skupu PASCAL Context

Najprije smo proveli analizu performansi učenja i zaključivanja na skupu Pascal Context na GPU-u NVIDIA GTX 1080 Ti. Rezultate daje Tablica 1. Napomene: i) svi stupci u tablici se odnose na ulaz 480x480, ii) memorijski otisak učenja ("train – memory“) je mjeren na maksimalnoj mogućoj veličini grupe pri učenju ("max bs train“) i izražen je kao par (memorija parametara i momenata, memorija aktivacija).

	max bs train (480x480)	train speed (bs = max bs train) / fps	max bs eval	eval speed (bs = max bs eval) / fps	eval speed (bs = 1) / fps	train - memory (PARAMS + MOMENTS, ACTIVATIONS) / GB per 480x480 crop	train - 1 epoch duration / min	assessment: train - total duration / hours
Segmenter-T-Mask/16	17	19.35	3816	41.39	31.29	(0.13, 0.46)	5	19.5
Swin-T	6	3.89	3656	14.37	12.66	(1.01, 1.39)	22	95.8
Segmenter-S-Mask/16	11	12.66	3791	41.01	27.99	(0.36, 0.77)	7	29.2
Swin-S	5	3.45	3623	11.55	11.01	(1.34, 1.73)	26	111.6
Segmenter-B-Mask/16	5	5.74	3649	17.62	11.74	(1.22, 1.39)	16	67.9
Swin-B	3	2.84	3565	9.88	9.26	(1.93, 2.03)	31	133.5
Segmenter-L-Mask/16	1	1.61	3332	6.59	6.03	(3.80, 3.05)	51	216.2
Swin-L	2	2.14	3377	6.68	5.99	(3.64, 2.61)	42	179.8
DeepLabV3+ RN50	5	3.87	3676	15.41	12.87	(0.54, 1.67)	21	88.2
DeepLabV3+ RN101	3	2.90	3656	10.12	8.93	(0.76, 2.37)	31	132.5
HRNetV2p-W18	12	12.72	3819	25.02	23.16	(0.11, 0.73)	7	28.9
HRNetV2p-W48	7	6.71	3698	23.50	15.09	(0.75, 1.25)	13	57.2
SNp-RN18-512	23	18.08	57	56.67	55.01	(0.38, 0.37)	5	20.3

Tablica 1 Vremensko-prostorna zahtjevnost Segmentera, SWIN-a i 3 konvolucijska modela (posebno važan piramidalni SwiftNet) na Pascal Contextu.

Može se ustanoviti da varijante Segmentera T, S i B imaju manji memorijski otisak te veću brzinu učenja i zaključivanja od odgovarajućih varijanti SWIN-a, dok je najveća varijanta SWIN-L manjeg memorijskog otiska i ima veću brzinu učenja od Segmenter-L, a podjednaku brzinu zaključivanja. Uz to se pokazuje da varijante DeepLabV3+ s kralješnicom ResNet-50 odnosno ResNet-101 svojim memorijskim (max bs train i memorija za aktivacije) i vremenskim (brzina zaključivanja i brzina učenja) performansama otprilike odgovaraju modelima Segmenter-B i SWIN-B (parovi Segmenter-B – DeepLabV3+ RN50 i SWIN-B – DeepLabV3+ RN101). Jedino malo više odudara brzina učenja za par Segmenter-B – DeepLabV3+ RN101 (brzina Segmentera = 5.74 fps, brzina DeepLabV3+ RN50 = 3.87 fps), no ostale performanse su usporedive za oba para.

Piramidalni SwiftNet s kralješnicom ResNet-18 i 512 značajki u modulu za naduzorkovanje, SNp-RN18-512, može se usporediti sa Segmenter-T – imaju slične brzine učenja (Segmenter-T nešto veću brzinu 19.35 fps od brzine učenja SwiftNeta 18.08%), podjednake memorijske otiske iako je Swiftnet ipak nešto memorijski učinkovitiji (memorijsko zauzeće aktivacija SwiftNeta je 0.37GB po isječku 480x480, a Segmenter-T je 0.46GB; ekvivalentno se očituje u maksimalnoj veličini grupe pri učenju koja je za SwiftNet 23, a za Segmenter 17). Veća razlika ovih dvaju modela je u brzini zaključivanja gdje je SwiftNet bolji s 56.67 fps što je veće od brzine Segmentera od 41.39 fps.

Kako bismo dobili uvid u generalizacijsku točnost transformera, naučili smo nekoliko inačica Segmentera na Pascal Contextu na najvećoj mogućoj veličini grupe za dani GPU NVIDIA GTX 1080 Ti. Rezultate prikazuje Tablica 2.

	#epochs trained / total	crop size	batch size (max bs train)	mIoU (SS)	batch size (official)	mIoU (MS) (official)
Segmenter-T-Mask/16	64 / 256	480x480	17	44.49	16	-
Segmenter-S-Mask/16	88 / 256	480x480	11	51.16	16	-
Segmenter-B-Mask/16	256 / 256	480x480	5	51.82	16	55.00
Segmenter-L-Mask/16	256 / 256	480x480	1	51.45	16	59.00

Tablica 2 Točnosti inicijalno naučenih modela Segmentera na Pascal Contextu.

Točnosti nismo uspjeli reproducirati zbog manje veličine grupe na kojoj su modeli učeni (učeni su na maksimalnoj veličini grupe), no pokazuju generalizacijski potencijal Segmentera.

5.1.2. Eksperimenti na skupu Cityscapes

Sljedeće, kako bismo bolje usporedili transformere s konvolucijskim modelima, analizirali smo memorijske i vremenske zahtjeve transformera i varijanti SwiftNeta na Cityscapesu na GPU-u NVIDIA GTX 1080 Ti. Rezultati popisuje Tablica 3.

	max bs train (768x768)	train speed (bs = max bs train) / fps	max bs eval		eval speed (bs = 1) / fps		max inputsize eval (hxw; w = 2h)	train - memory (PARAMS + MOMENTS, ACTIVATIONS) / GB		train - 1 epoch duration / min
			768x768	1024x2048	768x768	1024x2048		768x768	1024x1024	
Segmenter-T	6	6.33	70	6	21.8	2.01	1528x3056	(0.13, 1.50)	(0.18, 4.08)	8.3
SWIN-T	2	1.46	1236	165	6.95	1.94	1752x3504	(1.01, 3.32)	(1.02, 5.87)	30.3
SwiftNetPyr-RN18	20	17.57	68	19	74.58	28.03	4464x8928	(0.21, 0.46)	(0.21, 0.82)	3.2
SwiftNet-RN18	21	21.66	78	22	109.63	36.44	4793x9586	(0.20, 0.45)	(0.21, 0.81)	2.5
SwiftNetPyr-RN34	17	12.06	67	19	50.15	18.71	4458x8916	(0.36, 0.54)	(0.36, 0.96)	4.4
SwiftNet-RN34	16	16.28	78	21	76.23	24.79	4792x9584	(0.36, 0.56)	(0.36, 1.01)	3.2

Tablica 3 Vremenska i prostorna zahtjevnost Segmenter-Ti, SWIN-T i piramidalnih SwiftNeta na Cityscapesu.

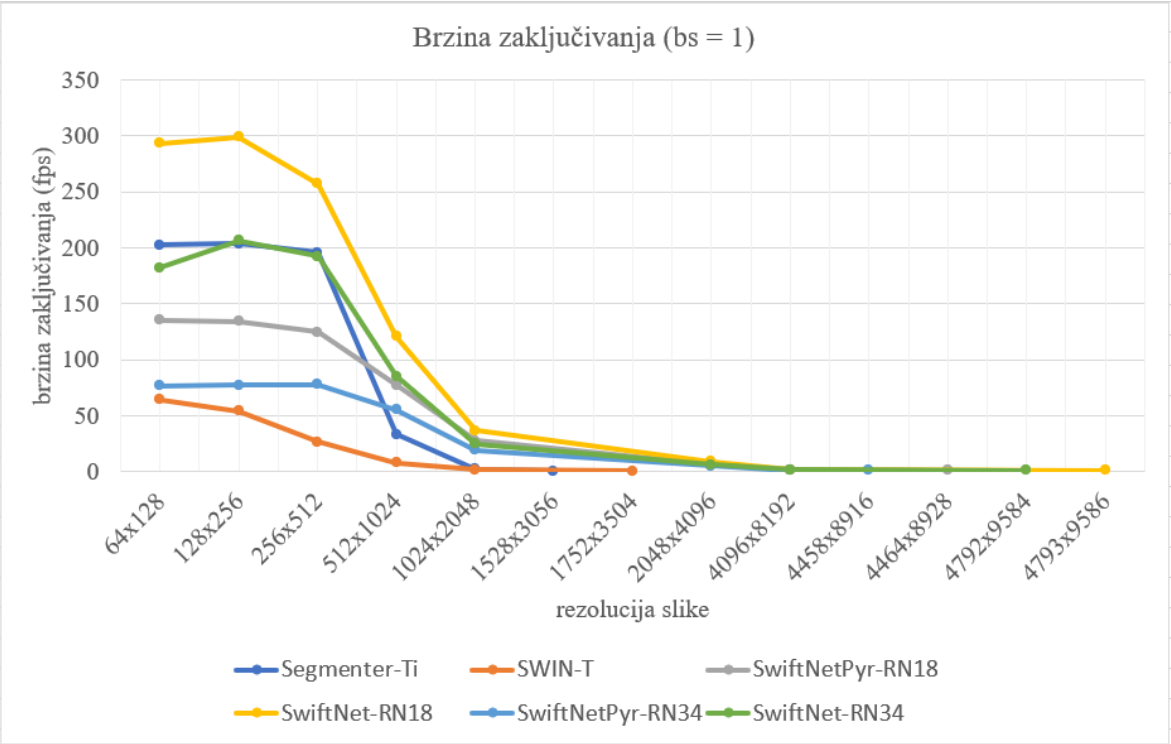
Slično kao na Pascalu, dobiveno je da je Segmenter u najmanjoj varijanti memorijski manje zahtjevan za učenje i brži u učenju i zaključivanju.

Osim toga, može se iščitati da su očito sve standardne varijante SwiftNeta brže u učenju i zaključivanju, imaju manje memorijske zahtjeve i mogu primiti slike veće rezolucije od transformera.

Odnosi memorijskih otisaka na dvjema veličinama isječka, 768x768 i 1024x1024, ukazuju na linearnu prostornu složenost učenja SWIN-a i SwiftNeta te na složenost Segmentera koja je veća od linearne. Slično vrijedi za odnose brzina zaključivanja na isječcima 768x768 i 1024x2048 – brzina zaključivanja linearno opada s porastom rezolucije slike za SWIN i SwiftNet, a brže nego linearno za Segmenter. Ovi odnosi ukazuju da Segmenter ima smisla koristiti prvenstveno na malim rezolucijama, što posebno vrijedi u slučajevima kad je memorija dosta ograničena.

5.2. Usporedno vrednovanje na skupu Cityscapes

Kako bismo dobili detaljniji uvid u ovisnost brzine zaključivanja ovih modela o rezoluciji ulazne slike, proveli smo eksperiment čiji su rezultati prikazanim grafom (Slika 36).



Slika 36 Ovisnost brzine zaključivanja Segmenter-Ti, SWIN-T i piramidalnih Swiftneta ovisno o rezoluciji ulazne slike na Cityscapesu.

S grafa se vidi da se na manjim rezolucijama SwiftNet-RN18 pokazao najbržim, Segmenter-Ti drugim najbržim, a SWIN-T najsporijim u zaključivanju, no na velikim su rezolucijama sve varijante SwiftNeta brže (Tablica 4). Tablica 4 prikazuje brzinu zaključivanja na najvećoj mogućoj rezoluciji dimenzije H x 2H za svaki model kao i broj piksela koje model može obraditi u jednoj sekundi. Odnosi maksimalne veličina ulazne slike odgovaraju broju piksela koje model može obraditi u sekundi te sam broj Mpix/s otkriva da SWIN-T može obraditi skoro 2x više piksela (3.68 Mpix/s za SWIN-T u odnosu na 1.92 Mpix/s za Segmenter-T), a sva 4 modela SwiftNeta su barem 8 puta brži od SWIN-T (model najvećeg kapaciteta, SwiftNetPyr-RN34, može obraditi najmanje piksela (29.88 Mpix/s) od prikazane 4 inačice SwiftNeta što je više od 8x više od 3.68 Mpix/s za SWIN-T.

	max inputsize eval	eval speed (bs = 1) / fps	Mpix/s
Segmenter-Ti	1528x3056	0.41	1.92
SWIN-T	1752x3504	0.60	3.68
SwiftNetPyr-RN18	4464x8928	1.06	42.43
SwiftNet-RN18	4793x9586	0.97	44.75
SwiftNetPyr-RN34	4458x8916	0.75	29.88
SwiftNet-RN34	4792x9584	0.82	37.72

Tablica 4 Brzina zaključivanja Segmenter-Ti, SWIN-T i piramidalnih SwiftNeta na najvećoj rezoluciji koju pojedini model prima.

Sljedeći dio ove analize vremensko-prostorne učinkovitosti odabranih transformera na Cityscapesu su mjerenja računske zahtjevnosti evaluacije (izražena u GMAC na isječku 1x3x1024x2048) i memorijskog otiska modela pri učenju (izražen u GB na isječku 1x3x1024x1024). Tablica 5 prikazuje rezultate.

	#param (M)	eval GMAC (1x3x1024x2048)	train GB: (PARAMS + MOMENTS, ACTIVATIONS) (1x3x1024x1024)
Segmenter-Ti	6.4	52.46	(0.18, 4.08)
Segmenter-B	100.4	822.61	(1.30, 16.30)
Segmenter-L	329.3	2697.69	(3.87, 36.73)
SWIN-T	59	1871.35	(1.02, 5.87)
SWIN-B	120	2348.08	(1.95, 8.60)
SWIN-L	231.8	3183.25	(3.65, 11.15)
SNp-RN18-128	12.1	128.89	(0.21, 0.82)

Tablica 5 Broj parametara, računska zahtjevnost i memorijski otisak učenja Segmentera i SWIN-a.

Iz tablice (Tablica 5) je vidljivo da SNp-RN18-128 ima najmanje parametara (12.1M), drugi je po redu najmanje računski zahtjevan s 128.89 GMAC na 2 Mpix tj. po isječku 1024x2048 (najmanje računski zahtjevan za evaluaciju je Segmenter-Ti s 52.46 GMAC po isječku iste veličine) i najmanjeg je memorijskog otiska učenja u vidu memorije aktivacije od 0.82 GB po isječku 1024x1024.

U vidu odnosa inačica Segmentera i SWIN-a, Seg-Ti i Seg-B imaju manje parametara (6.4M i 100.4M) od SWIN-T i SWIN-B (59M i 120M) te su manje računski zahtjevni (52.46 GMAC i 822.61 GMAC) od SWIN-T i SWIN-B (1871.35 GMAC i 2348.08). U najvećoj prikazanoj konfiguraciji (*large*) odnos računske zahtjevnosti je isti (Segmenter-L je manje računski zahtjevan (2697.69 GMAC) od SWIN-L (3183.25 GMAC), samo što

SWIN-L ima manje parametara (231.8M) od Segmenter-L (329.3M). Memorijski otisak učenja je za najmanje varijante (*tiny*) manji za Segmenter (Seg-Ti ima 4.08 GB aktivacija za razliku od SWIN-T s 5.87 GB aktivacija), a na većim konfiguracijama je SWIN memorijski učinkovitiji (SWIN-B s 8.60 GB i SWIN-L s 11.15 GB je ovdje bolji od Seg-B sa 16.30 GB i Seg-L s 36.73 GB aktivacija, uz napomenu da je SWIN-B zauzima više memorije za parametre i momente (1.95 GB) jer ima više parametara (120M) od Seg-B (1.30 GB memorije parametara i momenata, 100.4M parametara).

Posljednji dio analize vremensko-prostorne učinkovitosti odabranih transformera je analiza asimptotske vremenske složenosti učenja i zaključivanja i prostorne složenosti (primarno memorije aktivacija) Segmentera i SWIN-a na primjeru Seg-Ti i SWIN-T.

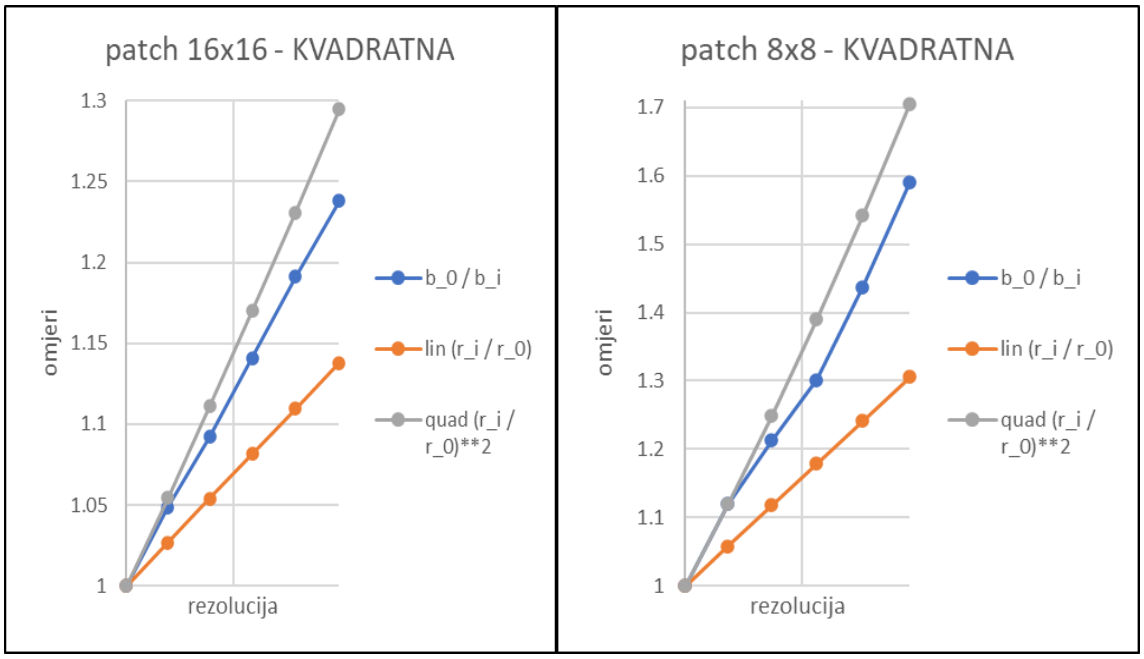
Prvo analiziramo složenost Segmentera. Radi detaljnije analize analiziramo dvije varijante Segmentera s različitom veličinom regije (*patch*): 16x16 (Slika 37 lijevo) i 8x8 (Slika 37 desno). Prije toga, Tablica 6 prikazuje način analize brzine učenja u ovisnosti o rezoluciji slike za Segmenter-Ti za patch 16x16; ekvivalentna analize je provedena za sve složenosti. U 1. stupcu (cropsize) pa je cropsize^2 jednako rezoluciji slike (kvadratni isječak veličine cropsize x cropsize), 2. stupac (brzina) je izmjerena vrijednost promatrane veličine (ovdje vrijednost brzine učenja). 3. stupac (r_i/r_0) daje omjer rezolucije i rezolucije u prvom retku (npr. u 2. retku je $r_1/r_0 = (1216/1200)^2 = 1.167$), a 4. stupac ($b_0/b_i$) analogno daje omjere izmjerenih vrijednosti promatrane veličine samo što daje omjer u suprotnom redoslijedu – omjer brzine u prvom retku i trenutne brzine (npr. u 2. retku je $b_0/b_1 = 1.224/1.167 = 1.027$). Stupac $(r_i/r_0)^2$ je kvadrirana vrijednost r_i/r_0 iz istog retka. Zadnja 2 stupca su omjeri na temelju kojih promatramo je li složenost bliža linearnoj ili kvadratnoj: stupac r_{i0}_lin/b_{0i} je omjer r_i/r_0 i b_0/b_i, stupac b_{0i}/r_{i0}_quad je omjer b_0/b_i i $(r_i/r_0)^2$. U svakom retku je žuto označen onaj omjer od ta 2 omjera koji je bliže 1 (omjer 1 bi bilo savršeno preklapanje složenosti s linearnom (omjer r_{i0}_lin/b_{0i}) odnosno kvadratnom (omjer b_{0i}/r_{i0}_quad)).

Analiziramo vremensku složenost učenja Seg-Ti. Prvo pogledajmo model s regijom 16x16. Tablica 6 prikazuje brzine učenja na najvećim rezolucijama gdje su mjerenja relevantna i najpouzdanija. Dobiva se da je složenost puno sličnija kvadratnoj što se u tablici (Tablica 6) vidi kao sve više obojanih omjera b_{0i}/r_{i0}_quad, a na grafu (Slika 37) lijevo se vidi da su omjeri brzina (b_0/b_i) puno bliži kvadratu omjera rezolucija (r_{i0}_quad, $(r_i/r_0)^2$). Slično je za Seg-Ti za regiju 8x8 – tablica nije prikazana jer je analiza provedena na isti način, a na grafu (Slika 37) desno se također vidi da je omjer brzina na većim rezolucijama blizak

kvadratu omjera rezolucija. Iz ovoga bi se eksperimentalno zaključilo da brzina učenja Segmenter na primjeru Segmenter-Ti kvadratno ovisi o broju piksela slike.

cropsize	**brzina**	r_i/r_0	b_0/b_i	$(r_i/r_0)^2$	r_{i0}_lin/b_{0i}	r_{i0}_quad/b_{0i}
1200	1.224	1.000	1.000	1.000	1.000	1.000
1216	1.167	1.027	1.049	1.054	0.979	1.005
1232	1.121	1.054	1.092	1.111	0.965	1.017
1248	1.073	1.082	1.141	1.170	0.948	1.025
1264	1.028	1.110	1.191	1.231	0.932	1.034
1280	0.989	1.138	1.238	1.295	0.919	1.046

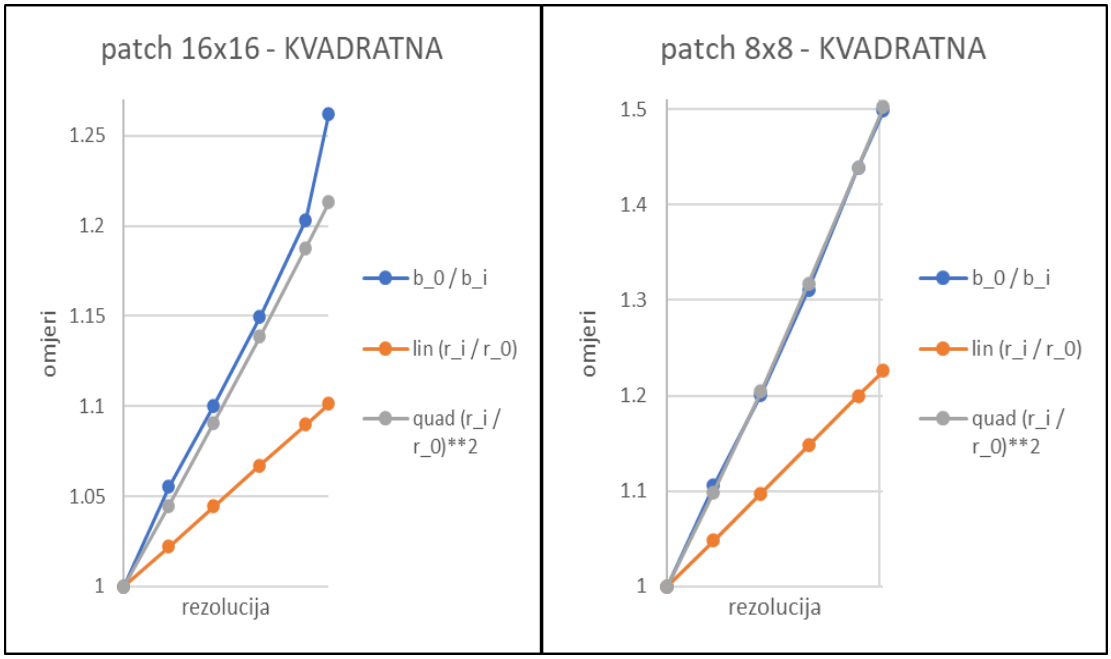
Tablica 6 Analiza omjera brzina učenja i rezolucija (ista tehnika primijenjena na analizu ostalih složenosti Segmentera i SWIN-a).



Slika 37 Ovisnost omjera brzina učenja Seg-T o rezoluciji slike za patch 16^2 (lijevo) i 8^2 (desno).

Sljedeća je analiza brzine zaključivanja ovisno o rezoluciji slike (Slika 38). Ovisnost brzine zaključivanja o broju piksela je slična kao ovisnost za brzine učenja o broju piksela: za oba modela omjeri brzina zaključivanja počinju su puno bliže kvadratnom (sivi graf) nego linearnom (narančasti graf) omjeru rezolucija. Može se još primijetiti da je ovdje ta ovisnost još „bliža“ kvadratnoj, no obje složenosti (vremenska složenost učenja i

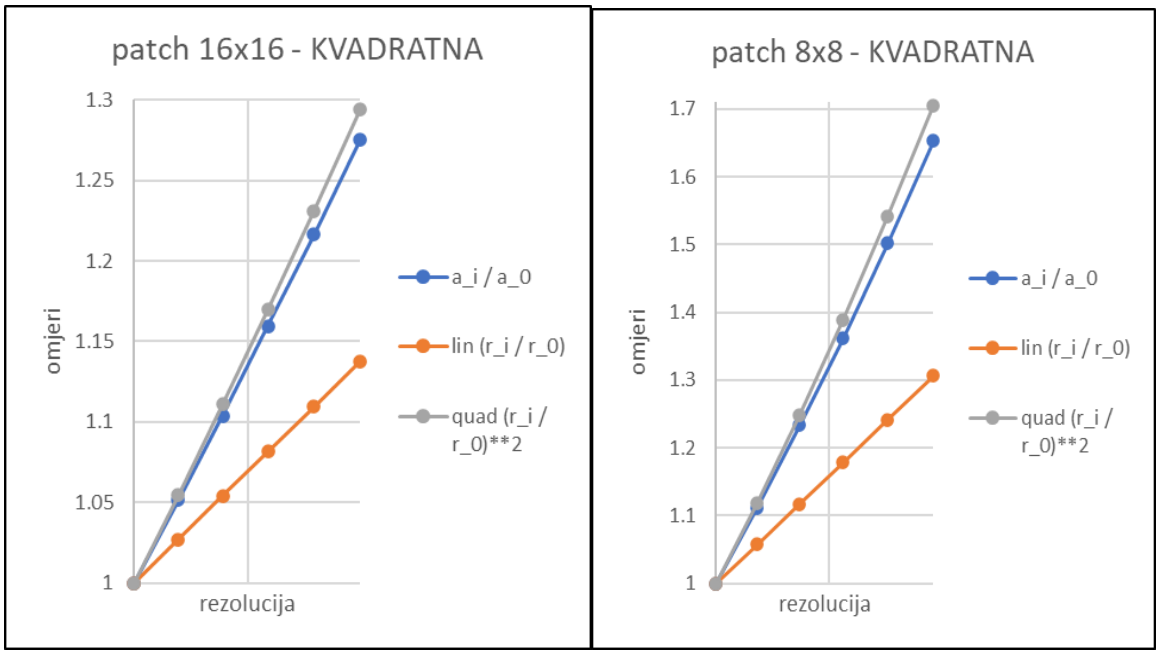
zaključivanja) asimptotski teže kvadratnoj složenosti. Oba modela za koje su izmjerene vrijednosti brzine zaključivanja (model s regijom 16x16 i s regijom 8x8) jasno je eksperimentalno pokazana kvadratna asimptotska vremenska složenost zaključivanja.



Slika 38 Ovisnost omjera brzina zaključivanja Seg-T o rezoluciji slike za patch 16^2 (lijevo) i 8^2 (desno).

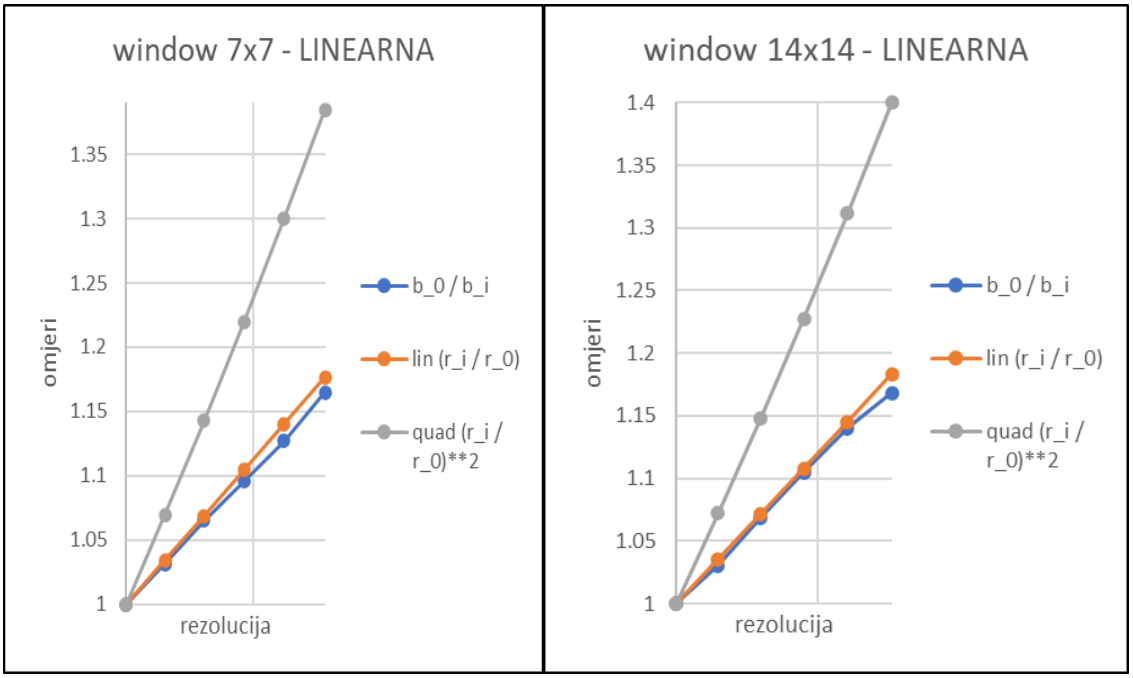
U analizi prostorne složenosti Seg-Ti promatrali smo ovisnost memorije zauzete aktivacijama o rezoluciji (Slika 39). Slično kao za vremenske složenosti, na najvećim rezolucijama omjer se vidljivo približio kvadratnom omjeru. Eksperimentalno se pokazalo da je i prostorna asimptotska složenost Segmentera na primjeru Segmenter-Ti kvadratna.

Prema navedenome, pokazalo se da su asimptotska vremenska složenost učenja i zaključivanja i prostorna složenost Segmentera kvadratne. To je u skladu s očekivanjima jer Segmenter preuzima ideju podjele slike u regije od ViT-a koja ne smanjuje asimptotski kvadratnu računsku složenost u linearnu, već samo veličinom regije regulira faktor s kojim se kvadratna ovisnost množi (za veću veličinu regije računska složenost se smanjuje, ali je i dalje asimptotski kvadratna). Pogledajmo stoga radi usporedbe kakve su asimptotske složenosti SWIN-a na primjeru također najmanje inačice SWIN-a – SWIN-T.



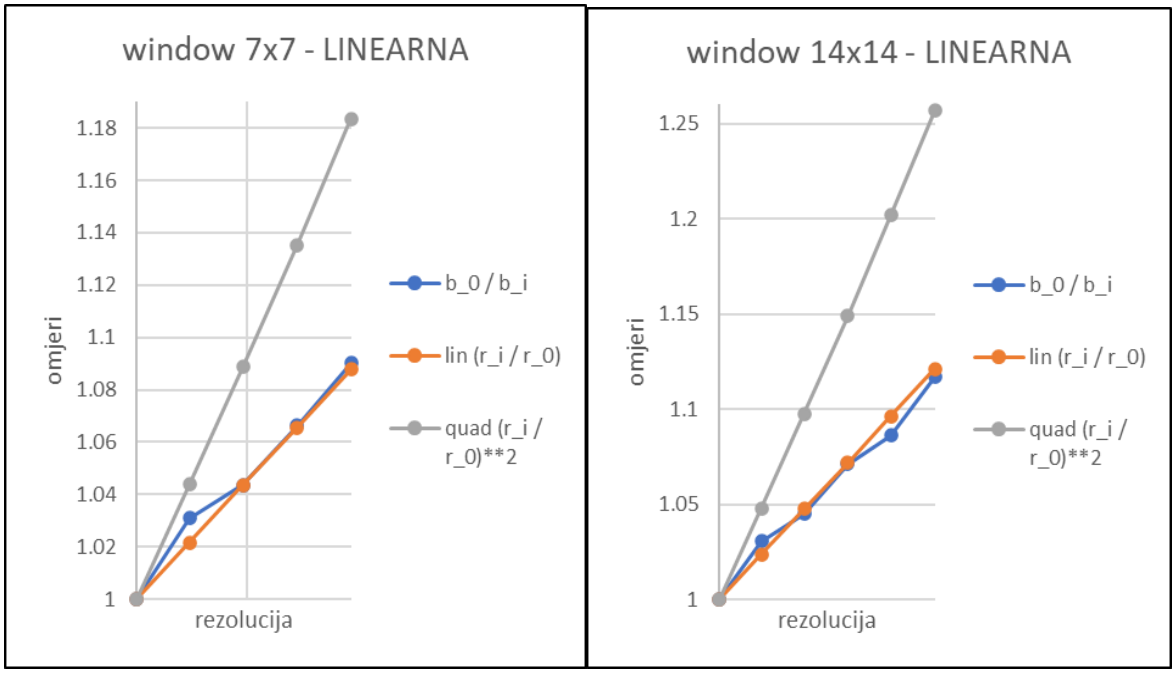
Slika 39 Ovisnost omjera memorije aktivacija Seg-T o rezoluciji slike za patch 16^2 (lijevo) i 8^2 (desno).

Asimptotske složenosti SWIN-a analiziramo za 2 inačice veličine prozora – 7x7 i 14x14. Prvu analizu vremenske složenosti učenja prikazuje Slika 40. Obje inačice na velikim rezolucijama imaju omjere brzina učenja bliske odgovarajućim linearnim omjerima rezolucija. Za obje inačice se pokazuje linearna asimptotska vremenska složenost učenja.



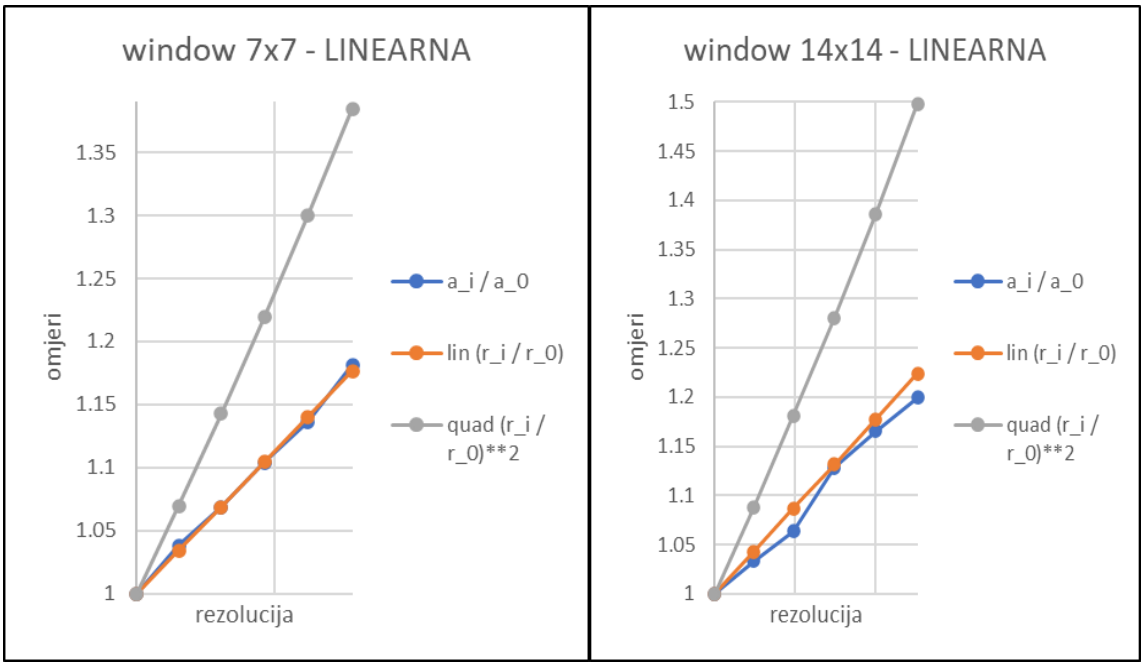
Slika 40 Ovisnost omjera brzina učenja SWIN-T o rezoluciji slike za prozor 7^2 (lijevo) i 14^2 (desno).

Slična promatranja kao za vremensku složenost učenja vrijede i za vremensku složenost zaključivanja SWIN-T (Slika 41). Dobiva se da brzina zaključivanja SWIN-a linearno opada s povećanjem broja piksela.



Slika 41 Ovisnost omjera brzina zaključivanja SWIN-T o rezoluciji slike za prozor 7^2 (lijevo) i 14^2 (desno).

Također, analogno se pokazuje (Slika 42) da je asimptotska prostorna složenost memorijskog zauzeća aktivacija SWIN-T linearna u ovisnosti o broju piksela.



Slika 42 Ovisnost omjera memorija aktivacija SWIN-T o rezoluciji slike za prozor 7^2 (lijevo) i 14^2 (desno).

Prema tome, eksperimentalno je pokazano da su vremenske i prostorna složenost SWIN-a linearne, na primjeru SWIN-T, što je u skladu s očekivanjima. SWIN je mehanizmom pomičnih prozora veličine MxM (gdje je M puno manje od dimenzija slike) omogućio računanje pažnje samo unutar relativno malih lokalnih prozora koji se ne preklapaju (unutar jednog sloja) pa je umjesto kvadratne ovisnosti o broju piksela dobio linearnu računsku složenost tj. linearnu ovisnost o broju piksela s faktorom M^2 što daje i linearne asimptotske vremenske i prostorne složenosti SWIN-a.

Pogledajmo sada još pregled transformera i SwiftNeta koje smo naučili na Cityscapesu (Tablica 7). Modele smo učili na GPU-u NVIDIA RTX A6000, a vremensko-prostorne performanse mjerili na NVIDIA RTX A5000. Usporedit ćemo segmentacijske točnosti zaključivanja SS (*singlescale*) (zaključivanje na originalnoj slici) i MS+flip (*multiscale* + zrcaljenje lijevo-desno) (zaključivanje na standardnih 6 rezolucija slike (faktori skaliranja: 0.5, 0.75, 1.0, 1.25, 1.5, 1.75) uz dodatak zrcaljenja slike lijevo-desno; 12 dobivenih logita se usrednjuju).

	crop	GPU RAM (GB/crop)	bs train	fps@2Mpx	GMAC@2Mpx	init	MS	mIoU	napomena
SLUŽBENA MJERENJA									
Segmenter-Ti	768x768		8			ViT-Ti patch=16 384x384			
Segmenter-B	768x768		8			DeiT-B distilled patch=16 384x384	da	80.60	
SWIN-T									
SWIN-B									
SNp-RN18	768x768		14	67.32	128	Imagenet-1k 224x224	ne	76.40	fps = **34** (GPU: **GTX 1080 Ti**); procjena omjera fps iz fps na GTX 1080 Ti i RTX A5000
PROVEDENA MJERENJA									
Segmenter-Ti	768x768	(0.13, 1.50)	8	6.44	52.46	ViT-Ti patch=16 384x384	ne	73.16	
							da	75.41	
		(0.13, 1.50)	12	6.44		ViT-Ti patch=16 384x384	ne	73.47	
Segmenter-B	768x768	(1.23, 5.81)	7	1.57	822.61	DeiT-B distilled patch=16 384x384	ne	78.26	
							da	80.52	
SWIN-T	768x768	(1.01, 3.32)	12	4.29	1871.35	Imagenet-22k 384x384	ne	73.59	
SWIN-B	769x769	(1.97, 4.88)	9	3.21	2348.08	Imagenet-1k 224x224	ne	79.02	učeno 40k/160k iteracija
							da	80.45	
	769x769	(1.97, 4.88)	9	3.21	2348.08	Imagenet-22k 384x384	ne	81.89	
							da	83.11	
SNp-RN18	768x768	(0.21, 0.46)	14	55.6	128.89	Imagenet-1k 224x224	ne	76.48	
							da	77.54	

Tablica 7 Točnosti naučenih modela Segmenter, SWIN i SwiftNet na Cityscapesu: rezultati iz literature (gore) i provedena mjerenja (dolje).

Uspjeli smo reproducirati točnost Segmenter-B mIoU = 78.26% (SS) odnosno 80.52% (MS) što je blizu službenih 80.60% (MS) i točnost SNp-RN18 mIoU = 76.48% (SS) što je blizu službenih 76.40% (SS), uz dodatno izmjerenu MS točnost od 77.54%. Također smo naučili sljedeće modele čija točnost nije službeno dostupna: Segmenter-Ti mIoU = 73.16% (SS) odnosno 75.41% (MS) (i dodatno mjerenje na bs=12 73.47% (SS)), SWIN-T na bs=12 mIoU = 73.59% (SS) i SWIN-B na bs = 9 na 160k iteracija mIoU = 81.89% (SS) odnosno 83.11% (MS). Ovdje treba napomenuti da su prvi rezultati za SWIN-B (mIoU = 79.02% (SS) i 80.45% (MS)) dobiveni učenjem istog modela samo s nešto slabijom inicijalizacijom (ImageNet-1k 224x224) i na manjem broju iteracija (40k), a nakon toga smo učenjem na najjačoj dostupnoj inicijalizaciji što je model SWIN-B prednaučen za klasifikaciju na ImagNet-22k na rezoluciji 384x384. Prešavši na tu bolju inicijalizaciju smo unutar 40k iteracija vidjeli da učenje brže napreduje pa smo nastavili učenje tog modela. Također, treba reći da ne postoji službeno dostupna konfiguracija za učenje SWIN-a na Cityscapesu zbog čega smo odabrali najveću moguću veličinu grupe bs = 9 za učenje na danom GPU-u, te da broj iteracija od 160k nije validiran kao ni veličina minigrupe odnosno svi hiperparametri učenja su preuzeti iz konfiguracije za ADE20k bez validacije.

Prema navedenome se vidi da na Cityscapesu SWIN-T ima sličnu točnost kao Segmenter-T - na bs=12 je SS točnost SWIN-T 73.59% i Seg-T 73.47%. U vidu učinkovitosti, Seg-T ima manji memorijski otisak učenja (zauzima 1.50 GB za aktivacije nasuprot SWIN-T 3.32

GB), brže zaključuje (6.44 fps nasuprot 4.29 fps za SWIN-T na 2 Mpx) i zahtijeva puno manje računa (52.46 GMAC nasuprot 1871.35 GMAC za SWIN-T na 2 Mpx).

S druge strane, naučeni modela SWIN-a najvećeg kapaciteta (SWIN-B) ima veću točnost (83.11% (MS)) od naučenog Segmenter-B (80.52% (MS)), što je dodatno poboljšano odabirom bolje inicijalizacije. Uz to, validacijom hiperparametara učenja SWIN-a na Cityscapesu potencijalno bi se dobile još bolje točnosti SS i MS. Gledajući prostorno-vremenske performanse, SWIN-B ima i manje memorijsko zauzeće aktivacija (4.88 GB za razliku od 5.81 GB za Seg-B) i sukladno tome veću najveću veličinu grupe pri učenju (9 za razliku od 7 za Seg-B). Također, SWIN-B više nego 2x brže zaključuje s 3.21 fps na rezoluciji od 2 Mpx za razliku od 1.51 fps (Seg-B). Ipak, SWIN-B je s 2348.08 GMAC računski zahtjevniji od Seg-B s 3x manjim brojem operacija – 822.61 GMAC na 2 Mpx.

Još smo proveli mjerenja za piramidalni SwiftNet SwiftNetPyr-RN18. Njegova MS točnost od 77.54% je između Seg-T (75.41%) i Seg-B (80.52%) te također između SWIN-T (73.59%) i SWIN-B (83.11%), a prostorno i vremenski je učinkovitiji od svih navedenih modela Segmentera i SWIN-a: i) ima 0.46 GB aktivacija što je oko 3x manji memorijski otisak od Seg-Ti koji ima 1.50 GB aktivacija, a više od 3x manji otisak od ostalih modela koji imaju puno veće memorijsko zauzeće i ii) brzina zaključivanja na 2 Mpx je 55.6 fps što je blisko službenom mjerenju 67.32 fps (preračunato iz vrijednosti od 34 fps na GTX 1080 Ti u brzinu na RTX A5000 pomoću omjera brzina SNp-RN18 koje smo dobili na ta 2 GPU-a) i što je oko 9x brže od najbržeg prikazanog transformera (Seg-T sa 6.44 fps). Jedino ima nešto veću računsku zahtjevnost od SWIN-T (128.89 GMAC što odgovara službenom mjerenju od 128 GMAC, nasuprot 52.46 GMAC Seg-T na 2 Mpx).

Zaključno, promatrani transformeri imaju suprotne odnose memorijske i vremenske učinkovitosti u T i B varijanti (Segmenter je učinkovitiji u T, a SWIN u B varijanti), dok oba modela Segmentera (Seg-T i Seg-B) su manje računski zahtjevni. Promatrani konvolucijski model (SNp-RN18) apsolutno dominira u vremensko-prostornoj učinkovitosti, osim nešto veće računske složenosti od Seg-T.

5.3. Usporedno vrednovanje na skupu ADE20K

Ovaj odjeljak uspoređuje SWIN i Segmenter na skupu ADE20k (Tablica 8). Modele smo učili na GPU-u NVIDIA RTX A6000, a prostorne i vremenske performanse mjerili na RTX A5000. Usporedimo segmentacijske točnosti u SS i za neke i MS, u kontekstu

izmjerenih vrijednosti mjera prostorno-vremenske učinkovitosti, isto kako su prezentirana mjerenja na Cityscapesu. Zaključivanje *multiscale* se provodi na istim rezolucijama kao i na skupu Cityscapes uz isto zrcaljenje.

	crop	GPU RAM (GB/crop)	bs train	fps@2Mpx	GMAC@2Mpx	init	MS	mIoU	napomene
SLUŽBENA MJERENJA									
Segmenter-Ti	512x512		8			ViT-Ti patch=16 384x384	ne	38.1	
SWIN-T	512x512		16	9.25	1888.14	Imagenet-1k 224x224	ne	44.51	* fps za GPU **V100**: fps@1Mpx = **18.5** (na 512x2048) -> fps@2Mpx ≈ 9.25 * GMAC@1Mpx = 945; procjena omjera GMAC iz GMAC na 1Mpx i 2Mpx (omjer ≈ 2)
SWIN-B	512x512		16	7.05	2370.39	Imagenet-1k 224x224	ne	48.13	* fps za GPU **V100**: fps@1Mpx ≈ **14.1** (na 512x2048) -> fps@2Mpx ≈ 7.05 * fps ≈ 14.1 procijenjen iz fps SWIN-T i SWIN-B 640x640 pomoću omjera GMAC SWIN-T, SWIN-B i SWIN-B 640x640 (pretpostavka: omjer GMAC i fps je ugrubo približan) * GMAC@1Mpx = 1188; procjena omjera GMAC iz GMAC na 1Mpx i 2Mpx (omjer ≈ 2)
							da	49.72	
PROVEDENA MJERENJA									
Segmenter-Ti	512x512	(0.23, 0.83)	8	8.48	52.58	ViT-Ti patch=16 384x384	ne	38.3	
SWIN-T	512x512	(1.01, 1.84)	16	4.09	1880.14	Imagenet-1k 224x224	ne	41.76	učeno 64k/160k iteracija
SWIN-B	512x512	(1.94, 2.53)	14	3.11	2356.88	Imagenet-1k 224x224	ne	47.42	
							da	48.88	

Tablica 8 Točnosti naučenih modela Segmenter i SWIN na ADE20K: rezultati iz literature (gore) i provedena mjerenja (dolje).

Ovdje smo uspjeli reproducirati točnost Segmenter-Ti mIoU = 38.3% (SS) (blizu točnosti od 38.1% (SS) koja je dostupna u tablici ADE20K u repozitoriju [18]).

Osim toga smo proveli i eksperiment učenja SWIN-B na ADE20k na bs = 14 (max bs train na korištenom GPU-u A6000) umjesto bs = 16 i dobili mIoU = 47.42% (SS) odnosno 48.88% (MS). Ovdje zbog manje veličine grupe nismo uspjeli reproducirati službenu točnost mIoU = 48.13% (SS) odnosno 49.72% (MS) (dostupno u tablici ADE20K na repozitoriju [19]). Slično, započeli smo učenje SWIN-T na ADE20k i na naučenih 64k od 160k iteracija dobili mIoU = 41.76% (SS), dok je službeni rezultat (na svih 160k iteracija) mIoU = 44.51%. Vjerujemo da bismo učenjem na svih 160k iteracija uspjeli reproducirati rezultat.

U vidu učinkovitosti, slično kao na Cityscapesu, na ADE20K Segmenter-T ima manji memorijski otisak (0.83 GB aktivacija nasuprot 1.84 GB aktivacija SWIN-T), više nego 2x brže zaključuje (8.48 fps na 2 Mpx nasuprot 4.09 fps SWIN-T) i puno manje je računski zahtjevan (52.58 GMAC nasuprot 1880.14 GMAC na 2 Mpx). Jedina znatnija razlika u odnosu na Cityscapes je što ovdje SWIN-T nema sličnu, nego zamjetno veću točnost od Seg-T – 41.76% (SS) nasuprot 38.3% Seg-T. Pritom navedena točnost od 41.76% je dobivena učenjem na samo dijelu (64k/160k) iteracija što znači da bi potencijalna točnost bila još i veća.

SWIN-B kao model većeg kapaciteta ima normalno veće memorijske zahtjeve od SWIN-T (2.53 GB aktivacija nasuprot 1.84 GB), sporije zaključuje (3.11 fps nasuprot 4.09 fps na 2 Mpx) i računski je zahtjevniji (2356.88 GMAC (blizu službenih 2370.39 GMAC) nasuprot 1880.14 GMAC (blizu službenih 1888.14 GMAC) na 2 Mpx). Napomene u vezi službeno dostupnih brzina zaključivanja: i) službena mjerenja su provedena na drugom GPU-u (V100) pa su brzine drugačije, ali omjer brzina SWIN-T i SWIN-B je otprilike isti kao za provedena mjerenja – 9.25 fps (službeno SWIN-T) / 7.05 fps (službeno SWIN-B) ≈ 4.09 fps (provedeno SWIN-T) / 3.11 fps (provedeno SWIN-B) ≈ 1.05, ii) službeno dostupne brzine su zapravo navedene u fps@1Mpx pa su ovdje radi usporedbe preračunate u fps@2Mpx dijeljenjem s 2 (približni omjer) – iz 18.5 fps i 14.1 fps u 9.25 fps i 7.05 fps (SWIN-T i SWIN-B), iii) brzina SWIN-B zapravo nije službeno dostupna, nego je radi usporedbe aproksimirana služeći se brzinama zaključivanja (fps) i računskim zahtjevnostima (MAC) modela SWIN-T (18.5 fps na V100, 945 GMAC@1Mpx) i SWIN-B 640x640 (8.7 fps na V100, 1841 GMAC@1Mpx) učen na isječcima 640x640, uz službeno mjerenje računske zahtjevnosti SWIN-B (1188 GMAC@1Mpx), pod pretpostavkom da se MAC i fps odnose otprilike približno.

Ovim svim rezultatima, od koje smo neke uspjeli reproducirati, a nekima se približiti, dodatno smo dokazali generalizacijsku moć transformera u gustoj predikciji jer smo je pokazali na dvama skupovima: Cityscapes i ADE20k.

5.4. Ansambli SwiftNeta i Segmentera

Predstavimo rezultate ansambala SwiftNeta i Segmentera na Cityscapesu. Rezultate daje Tablica 9. Napomene: i) ansambli se evaluiraju na slikama 1024x2048 koje su ulaz u oba modela od koji se ansambl sastoji, ii) memorijski otisak učenja i brzina učenja za ansamble nisu dostupni (oznaka „N/A“) jer se ansambl sastoji od naučenih modela koje samo evaluiramo i predikciju donosimo usrednjivanjem logita oba modela, i iii) brzine učenja i zaključivanja mjerili smo na NVIDIA GTX 1080Ti što je GPU na kojem smo mjerili i brzine svih prethodnih modela te zbog nedostatka CUDA memorije na danom GPU-u za Seg-B nisu dostupne brzine.

Očekivano ansambli s *multiscale* varijantom SwiftNeta imaju bolju točnost od odgovarajućih *singlescale* varijanti, isto kao što ansambli sa SwiftNetom s kralješnicom ResNet-34 daju višu točnosti od istovjetnih s ResNet-18. Dobro je primijetiti da svi

ansambli daju veću točnost od točnosti oba modela od kojih su sagrađeni. Nama je od predmeta interesa ansambl SwiftNetPyr-RN18 + Seg-T koji ima točnost od mIoU = 77.43% (SS) odnosno mIoU = 78.29% (MS).

Općenito se može primijetiti da sljedeće vrijedi za ansamble: i) računska zahtjevnost (GMAC) ansambla jednaka je zbroju računskih zahtjevnosti građevnih modela i ii) brzina zaključivanja je malo manja, ali bliska brzini sporijeg modela što je u svim ansamblima Seg-T. Varijacije u brzini zaključivanja ansambala dolaze ovisno o korištenom modelu SwiftNeta i vidljive su za zaključivanju na manjoj rezoluciji (768x768), dok su na velikim slikama (1024x2048) sva 4 ansambla skoro podjednako brzi.

Također je zanimljivo usporediti SwiftNetPyr-RN18 + Seg-T sa SwiftNetPyr-RN34. SNp-RN34 ima veću točnost (78.21% (SS) nasuprot 77.43% (SS)) i puno veću brzinu zaključivanja (50.15 fps nasuprot 16.48 fps na 768x768, odnosno 18.71 fps nasuprot 1.81 fps), no računski je zahtjevniji od odabranog ansambla (230.64 GMAC nasuprot 181.26 GMAC na 2 Mpx).

	mIoU (SS) (1024x2048)	mIoU (MS) (1024x2048)	eval GMAC (1x3x1024x2048)	train speed (bs = max bs train) 768x768 / fps	eval speed (bs = 1) / fps		train - memory (PARAMS + MOMENTS, ACTIVATIONS) / GB	
					768x768	1024x2048	768x768	1024x1024
Seg-T	73.16	75.41	52.46	6.33	21.80	2.01	(0.13, 1.50)	(0.18, 4.08)
SN-RN18	75.54	76.9	104.2	21.66	109.63	36.44	(0.20, 0.45)	(0.21, 0.81)
ens SN-RN18 + Seg-T	76.72	77.71	156.56	N/A	18.03	1.83	N/A	N/A
SNp-RN18	76.48	77.54	128.89	17.57	74.58	28.03	(0.21, 0.46)	(0.21, 0.82)
ens SNp-RN18 + Seg-T	77.43	78.29	181.26	N/A	16.48	1.81	N/A	N/A
SN-RN34	77.05		181.72	16.28	76.23	24.79	(0.36, 0.56)	(0.36, 1.01)
ens SN-RN34 + Seg-T	77.82		234.08	N/A	16.72	1.79	N/A	N/A
SNp-RN34	78.21		230.64	12.06	50.15	18.71	(0.36, 0.54)	(0.36, 0.96)
ens SNp-RN34 + Seg-T	78.79		283	N/A	15.07	1.74	N/A	N/A
Seg-B	78.26	80.52	822.61	-	-	-	(1.23, 5.81)	(1.30, 16.30)

Tablica 9 Točnosti i vremensko-prostorni zahtjevi Segmentera, SwiftNeta i ansambala i naučenih modela Segmentera na Cityscapesu.

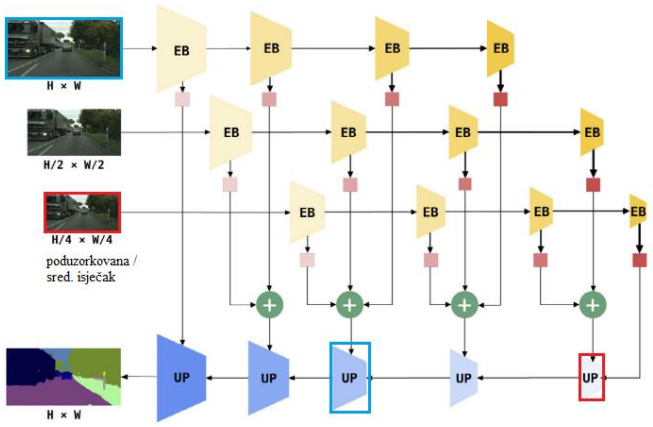
5.5. Združeno učenje Swiftneta i Segmentera

Ovaj odjeljak opisuje implementaciju združenih modela dobivenih učenjem arhitekture koja uklapa Segmenter u SwiftNet što je hibridni transformersko-konvolucijski model koji

nazivamo SwiftNet-Segmenter. U eksperimentima je naglasak stavljen na korištenje piramidalnog SwiftNeta s kralješnicom ResNet-18 (SN-RN18-pyramid) te najmanje varijante Segmentera (Seg-T) radi inicijalnog isprobavanja većeg broja konfiguracija. Nažalost u većini eksperimenata dobivene točnosti bile su podjednake ili lošije od točnosti *baseline* modela SwiftNeta i Segmenter, stoga ne prikazujemo rezultate, već ćemo samo ukratko objasniti zašto provedeni eksperimenti nisu uspjeli i na kraju spomenuti potencijalni smjer nastavka učenja ovih modela.

Slika 43 prikazuje moguće ulaze u model Segmentera unutar združenih modela – crveno uokvirena ulazna slika je poduzorkovana slika ili središnji isječak koji rezolucijom odgovara ulaznoj slici u 3. razinu piramide SwiftNeta, a plavo uokvirena slika je ulazna slika koja ulazi u 1. razinu piramide SwiftNeta. Ovisno o tome koja rezolucija slike od ove dvije se odabire za ulaz Segmenteru, izlaz kodera Segmentera se kombinira sa SwiftNetom kod bloka UP u modulu za naduzorkovanje (dekoderu SwiftNeta) koji prima značajke te prostorne rezolucije. Prvo je implementirana arhitektura (arh. v1) koja odgovara crveno označenom ulazu, gdje je grana Segmentera paralelna 3. razini piramide SwiftNeta, no nakon iscrpnog ispitivanja točnosti su se pokazale nedovoljnima te je ustanovljeno da je uzrok tome bio neprikladan dizajn te arhitekture. Stvar je da se Segmenter učio na malim slikama pa zbog toga se nije dobro naučio tj. već u početku je bio uskraćen za generalizacijsku moć koju bi mogao imati da se učio na čitavim slikama izvorne rezolucije. Stoga je dizajn promijenjen u suprotan gdje Segmenter prima ulaznu sliku HxW i u tom slučaju grana Segmentera je paralelna s 1. razinom piramide SwiftNeta (arh. v2). Međutim, ni taj dizajn nije dao dobre rezultate jer iako je u toj inačici arhitekture Segmenter učen na cijelim slikama, validiran je ponovo na malim slikama (poduzorkovanim slikama ili središnjim isječcima). Jedina vrijedna spoznaja koju smo dobili je da združene modele arh. v2 nije pogodno učiti zasebnim optimizatorima (SGD za Segmenter, Adam za SwiftNet) što je pokazala točnost manja od 50% za početni eksperiment koji smo proveli na toj inačici arhitekture. Na kraju je još započeto učenje združenih modela arh. v2 uz razliku što su Segmenteri učeni na cijelim slikama i validirani na cijelim slikama (umjesto smanjenim slikama) što se može nazvati arh. v2.1. Naučili smo samo jedan model arhitekture v2.1 koji je dao točnost približnu točnosti SwiftNeta, ali ne veću. Stoga bi sljedeći korak u budućem radu, koji nismo stigli izvesti, bio svakako naučiti više združenih modela SwiftNeta i Segmentera arh. v2.1, validirati hiperparametre (poziciju spoja Segmentera i Swiftneta

(prije/poslije), faktor pomoćnog gubitka, vrstu i hiperparametre optimizatora za učenje SwiftNeta i Segmentera u združenom modelu i dr.).



Slika 43 SwiftNet s označenim mjestima spoja sa Segmenterom (crveno i plavo – ulaz u Segmenter za 2 varijante arhitekture SwiftNet-Segmenter); slika prilagođena iz [3].

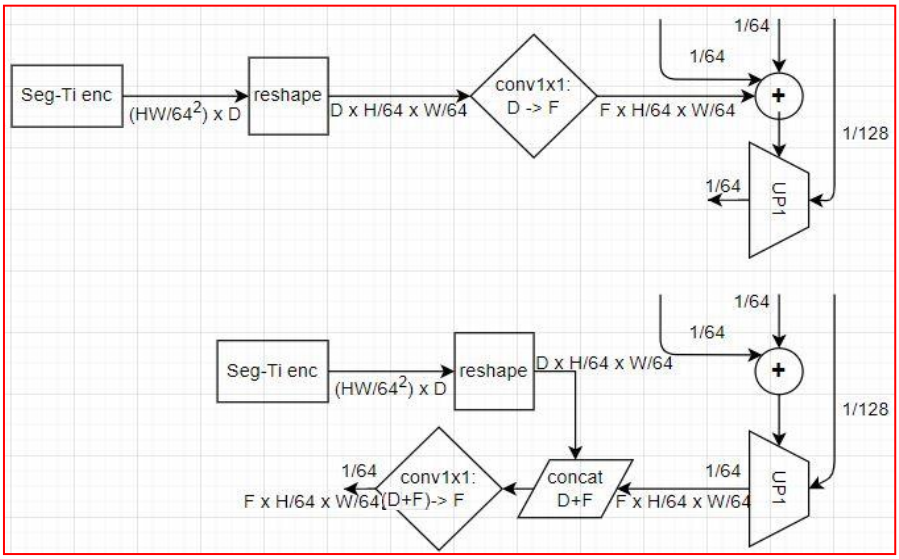
Slika 44 prikazuje su implementacije spoja SwiftNeta i Segmentera u inicijalnoj arhitekturi združenog modela arh. v1, gdje je ulaz u Segmenter rezolucije ulazne slike u 3. razinu piramide (gore – crveni okvir), i sljedećoj inačici arhitekture arh. v2 gdje je ulaz u Segmenter jednak ulaznoj slici u 1. razinu piramide (dolje – plavi okvir). U oba okvira je prikazana varijanta spoja prije (gore) i poslije (dolje) bloka UP koji prima značajke odgovarajuće rezolucije.

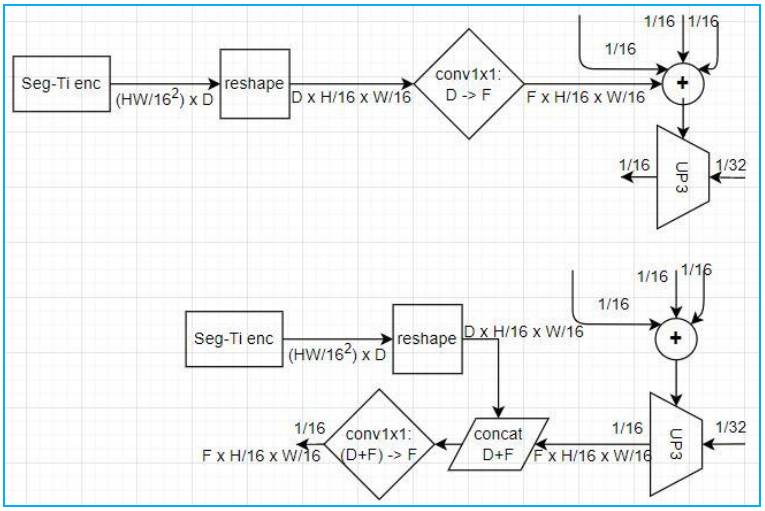
U arh. v1 je spoj kod 1. bloka UP (UP1) jer Segmenter koji na ulazu dobiva H/4 x W/4 na kraju kodera daje značajke dimenzije $HW/64^2$ x D (jer se ulazna slika H/4 x W/4 poduzorkuje 16x – 16 je (*default*) veličina *patcha* učenog Seg-Ti) koje se zatim preoblikuju (*reshape*) u oblik D x H/64 x W/64. Krajnji korak se razlikuje ovisno o tome je li spoj prije ili poslije bloka UP1.

Ako je spoj prije (gore u crvenom okviru), onda se preoblikovani izlaz provuče kroz 1x1 konvoluciju koja D (dimenzija modela Segmentera, za Seg-Ti je D = 192) ulaznih kanala prevodi u F (dimenzija značajki u *upsampling* modulu SwiftNeta, pretpostavljeno je F = 128) izlaznih kanala. Zatim se tako dobivene značajke zbroje kao 3. pribrojnik u piramidalnoj fuziji značajki na ulazu UP1 čime se izlaz kodera zapravo „tretira“ kao značajke SwiftNeta koje su iste rezolucije (na faktoru prostornog poduzorkovanja 1/64). Dalje se, normalno kao i u piramidalnom SwiftNetu, rezultat zbroja šalje u UP1 zajedno sa značajkama poduzorkovanim na 1/128 i rezultat UP1 je na 1/64.

Ako je spoj poslije (dolje u crvenom okviru), onda se preoblikovane značajke s izlaza kodera Segmentera ne šalju u piramidalnu fuziju, nego se spajaju s izlazom bloka UP1. Ovdje u piramidalnu fuziju ulaze samo značajke SwiftNeta i izlaz UP1 je opet na 1/64. Nakon UP1 se izlaz UP1 dimenzije F x H/64 x H/64 konkatentira po 1. dimenziji s izlazom Segmentera D x H/64 x W/64 čime se dobiva tenzor dimenzija (D+F) x H/64 x W/64 što su značajke iste prostorne rezolucije kao što je rezolucija oba konkatenirana tenzora, ali je dimenzije jednaka zbroju dimenzije značajki Segmenter i izlaza UP1 – za korištene D i F je to D+F = 192+128 = 320. Zatim se, slično kao u varijanti sa spojem prije, na dobivene značajke primijeni 1x1 konvolucija s F izlaznih, ali ovdje D+F ulaznih kanala. Rezultate konvolucije je poduzorkovan na 1/64 tj. jednakih je dimenzija kao izlaz UP1 (F x H/64 x W/64) pa se sad šalje dalje normalno kao prvi ulaz bloku UP2.

Ekvivalente implementacije u obje varijante spoja SwiftNeta i Segmentera (prije/poslije) bloka UP su izvedene za arh. v2 uz razliku da je izlaz Segmentera dimenzija $HW/16^2$ x D (jer se ulazna slika H x W poduzorkuje 16x) i zbog toga se spoj radi kod bloka UP3 koji prima značajke na 1/16 iz odgovarajućih instanci kodera i 1/32 iz prethodnog bloka dekodera (UP2). Prema tome, opisi svih komponenti implementacije spoja SwiftNeta i Segmentera dani za arh. v1 vrijede i ovdje za arh. v2 uz promjenu ovih dimenzija.





Slika 44 Implementacija spoja SwiftNeta i Segmentera u arh. v1 (gore) i arh. v2 (dolje). Za svaku inačicu arhitekture su prikazane 2 implementirane opcije spoja: prije (gore) i poslije (dolje) odgovarajućeg bloka UP. Za opis implementacije vidjeti tekst iznad slike.

5.6. Združeno učenje piramide Segmentera

Slično kao u prethodnom poglavlju gdje učili združene modele Segmenter i SwiftNeta u piramidi piramidalnog SwiftNeta, na kraju smo krenuli učiti slične piramide s 3 razine, ali ovdje s 3 modela Segmentera na istim rezolucijama (faktori poduzorkovanja ulaznih slika: 1/1, 1/2, 1/4). Odlučili smo se za taj korak jer smo očekivali dobiti možda bolje točnosti nego u združenim modelima SwiftNeta i Segmenter, a imali smo na raspolaganju i dovoljno GPU memorije kako bi se sva 3 Segmentera združeno učili u jednom združenom modelu. Nažalost, ni ovdje nismo za nijedan model dobili točnosti bolje od Segmentera (konkretno Seg-Ti jer smo učili piramide od 3 Seg-Ti) pa ih ne prikazujemo.

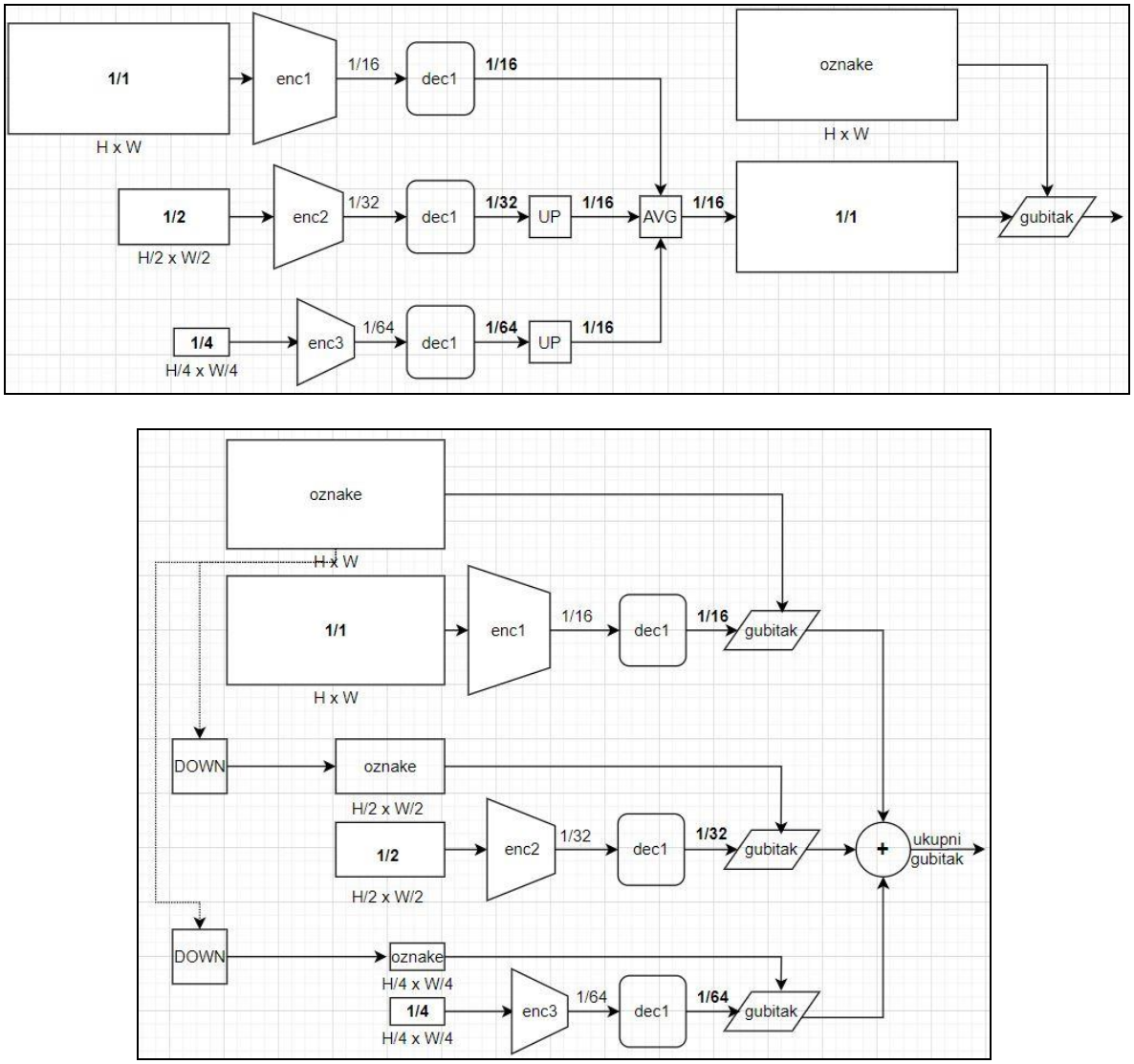
Prije svega smo isprobali 2 *baseline* modela tih piramida (Slika 45). Baseline 1) (Slika 45 gore) je 1 klasifikacija – segmentacijske mape (predikcije) dobivene iz kodera Segmentera na 2. i 3. razini, koje su na 1/32 i 1/64, se naduzorkuju na 1/16 (što odgovara razini poduzorkovanja mape dobivene iz Segmentera na 1. razini), tako naduzorkovane te 3 mape se usrednjuju (zbroje i podijele s 3) i nad tim mapama se konačno računa 1 gubitak. Baseline 2 (Slika 45 dolje) su 3 klasifikacije – na svakoj razini se računa gubitak svakog

modela i krajnji gubitak je zbroj ta 3 gubitka. Pritom kod računanja gubitaka treba poduzorkovati segmentacijske oznake na 2. i 3. razini faktorom 1/2 odnosno 1/4. Nešto veću točnost dobili smo očekivani za slučaj 1 umjesto 3 klasifikacije iako su dobivene točnosti bile vrlo bliske zbog čega bi dalje bilo preporučljivo naučiti oba *baseline* modela, no na većem broju epoha kako bi iskonvergirali (u eksperimentima nismo validirali broj epoha pa je moguće da bi se dobile veće razlike točnosti na većem broju epoha).

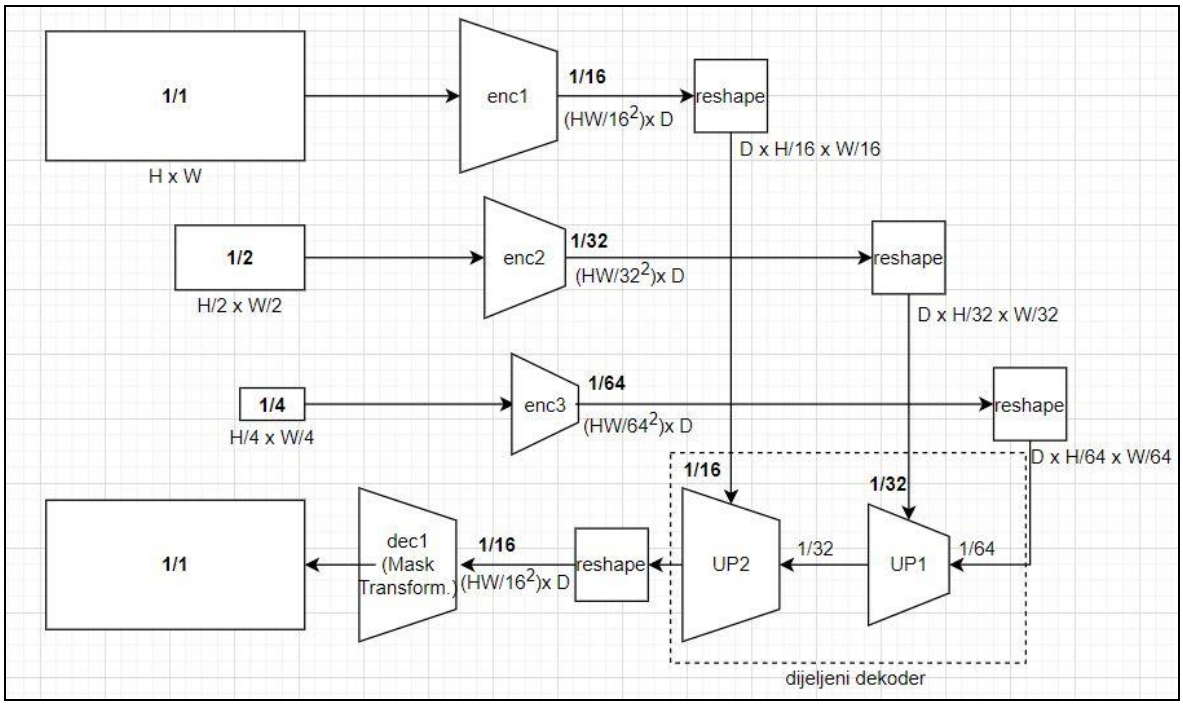
Zatim smo validirali utjecaj: i) dijeljenja parametara 3 Segmentera i dobivena je veća točnost za slučaj bez dijeljenja parametara, ii) *boundary-aware* fokalni gubitak i dobivena je niža točnost nego s unakrsnom entropijom (iako bi se očekivalo suprotno) i iii) zajednički dekoder (po uzoru na SwiftNet) koji rekombinira značajke koje izlaze iz kodera Segmentera i ovdje je dobivena znatno niža točnost (iako bi se očekivalo suprotno). Također smo model s *boundary-aware* fokalnim gubitkom i model sa zajedničkim dekoderom naučili na 2x većem broju epoha čime su se točnosti poboljšale, *boundary-aware* fokalni gubitak je dao bolju točnost nego unakrsna entropija (no nismo naučili model s unakrsnom entropijom na 2x većem broju epoha zbog čega ova tvrdnja možda ne vrijedi), model sa zajedničkim dekoderom je i dalje dao manju točnost od modela sa zasebnim dekoderima. Ova arhitektura bi imala poseban potencijal za daljnju validaciju i možda promjenu odluka dizajna kako bi se dobile točnosti bolje od jednog Segmentera.

Opišimo još ukratko kako smo implementirali piramidu Segmentera s dijeljenim dekoderom (Slika 46). Izlazi kodera Segmentera na 1/16, 1/32, 1/64 se samo preoblikuju (blok *reshape* u implementacijskom dijagramu radi ekvivalentno kao *reshape* u opisu implementacija arhitekture združenih modela SwiftNeta i Segmentera u prethodnom poglavlju) u oblik koji prima odgovarajući UP blok (blok dekodera). Blok UP1 prima značajke najmanje rezolucije (H/64 x W/64) iz (preoblikovanog) izlaza kodera na 3. razini (taj koder Segmentera prima sliku na 1/4 i poduzorkuje je 16x pa se dolazi do faktora poduzorkovanja od 1/64 – slično je opisano prije u ovom poglavlju i u prethodnom poglavlju). Također, UP1 prima i značajke 2x veće rezolucije (faktor 1/32) na istoj dimenziji (D – dimenzija modela), što je izlaz kodera Segmentera s 2. razine. Značajke na 1/64 naduzorkuje na 1/32, zatim zbraja tenzore značajki koji su sad oba na 1/32 te konačno provodi 3x3 konvoluciju. Takvu implementaciju bloka naduzorkovanja smo preuzeli iz piramidalnog SwiftNeta, samo što ovdje nemamo značajke iste rezolucije s različitih razina rezolucijske piramide pa nema piramidalne fuzije prije UP bloka. Zatim, blok UP2 prima izlaz UP1 (koji je poduzorkovan na 1/32) i izlaz kodera Segmentera s 1. razine (koji je na

1/16) i redom provodi naduzorkovanje ulaza koji je na 1/32, zbrajanje i konvoluciju i konačno se dobiva izlaz zajedničkog dekodera (koji se sastoji od 2 bloka naduzorkovanja – UP1 i UP2). Izlaz zajedničkog dekodera se preoblikuje (*reshape*) iz D x H/16 x W/16 u (HW/16^2) x D (što je preoblikovanje u suprotnom smjeru nego što su radili dosad opisani blokovi *reshape*) što su dimenzije značajki koje prima dekoder tipa Mask Transformer koji konačno donosi predikciju u obliku segmentacijske mape koja se na kraju 16x (veličina regije) naduzorkuje, kao što to normalno radi Segmenter.



Slika 45 Baseline modeli piramide Segmenter 1/1, 1/2, 1/4 - baseline 1 (gore – 1 klasifikacija) i baseline 2 (dolje – 3 klasifikacije).



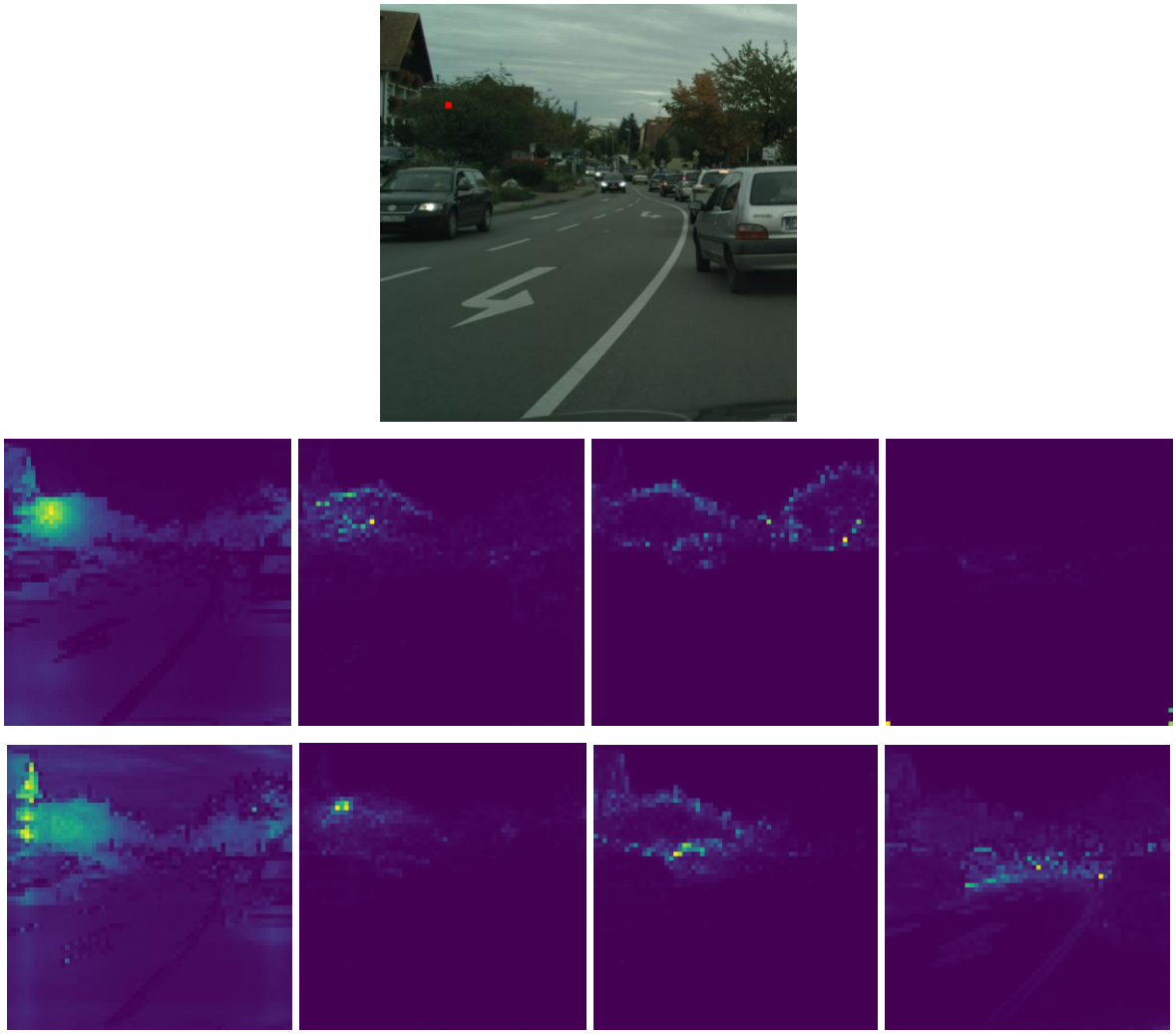
Slika 46 Piramida Seg-Ti 1/1, 1/2, 1/4 sa zajedničkim dekoderom koji rekombinira značajke iz kodera Segmentera.

5.7. Interpretiranje odluka Segmentera

U ovom eksperimentu vizualiziramo mape pažnje dobivene učenjem modela Segmenter-Ti i Segmenter-B kako bismo još bolje mogli interpretirati odlučivanje Segmentera. Svaki od sljedećih primjera bit će prikazan na slici gdje je: gore – slika s crveno označenom kvadratnom regijom (*patch*) 16x16 za koju se prikazuje mapa pažnje, sredina – mape pažnje za koder modela Seg-Ti (ViT-Ti) i dolje – mape pažnje za koder modela Seg-B (DeiT-Ti). Slika rezolucije 1024x2048 se isječe na veličinu 1024x1024 i od nje se danim modelom računaju mape pažnje koje su onda rezolucije 64x64 i zatim se naduzorkuju faktorom 16 (veličina regije) na rezoluciju slike 1024x1024. Prikazane su 4 mape pažnje – redom za 1., 4., 8. i 11. sloj kodera koji ima 12 slojeva.

Prvi primjer (Slika 47) je regija na krošnji stabla gore lijevo. Već nakon 1. sloja kodera oba modela pridaju pažnju ne samo označenoj regiji na stablu već cijeloj krošnji tog i okolnih stabala. Uz to, Segmenter-B više širi to receptivno polje mape pažnje i nešto jače aktivira druga stabla (na desnom dijelu slike), jedino što jače i aktivira objekte koje ne bi trebao (kuća na lijevoj strani). Nakon 4. sloja za oba modela pažnja „slabi“ u okolici označene regije, a „finije“ se raspoređuje na drvored na lijevoj strani. Nakon 8. sloja se pažnja počinje davati i drvoredu na desnoj strani, što je posebno naglašeno u mapama Segmenter-

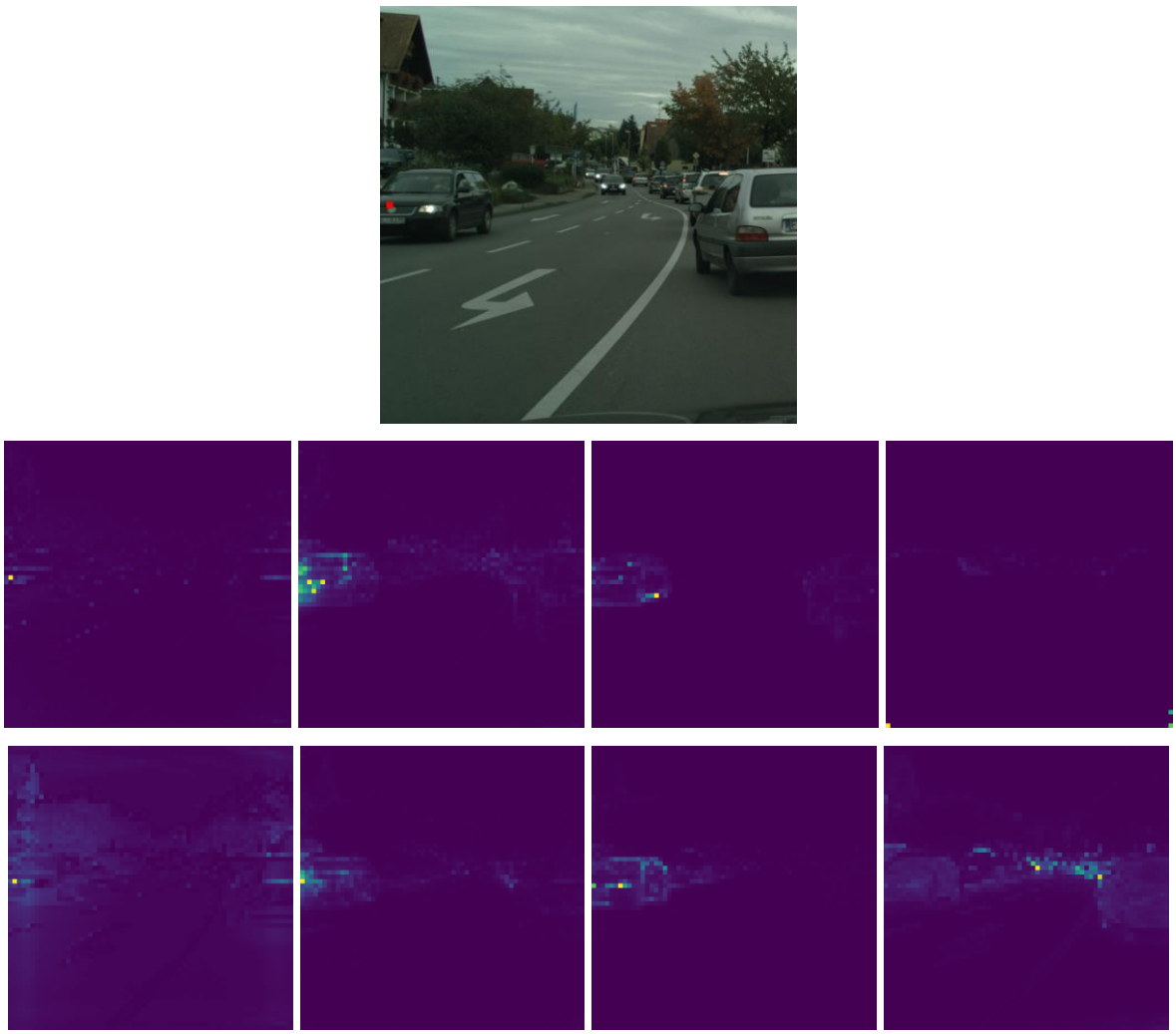
Ti iako bi se očekivalo da Segmenter-B kao veći model daje bolju mapu. Ipak, nakon 11. (predzadnjeg) sloja kodera situacija se dosta mijenja u korist Segmenter-B jer Segmenter-Ti ne daje više smislenu mapu pažnje, a Segmenter-B ipak zadržava dovoljno pažnje koja je sad raspoređena i po sredini i na lijevoj i desnoj strani slike po stablima.



Slika 47 Ulazna slika s crveno označenom regijom (gore), mape pažnje kodera Seg-Ti (sredina) i mape pažnje kodera Seg-B (dolje) nakon 1., 4., 8. i 11. sloja kodera (objašnjenje u tekstu).

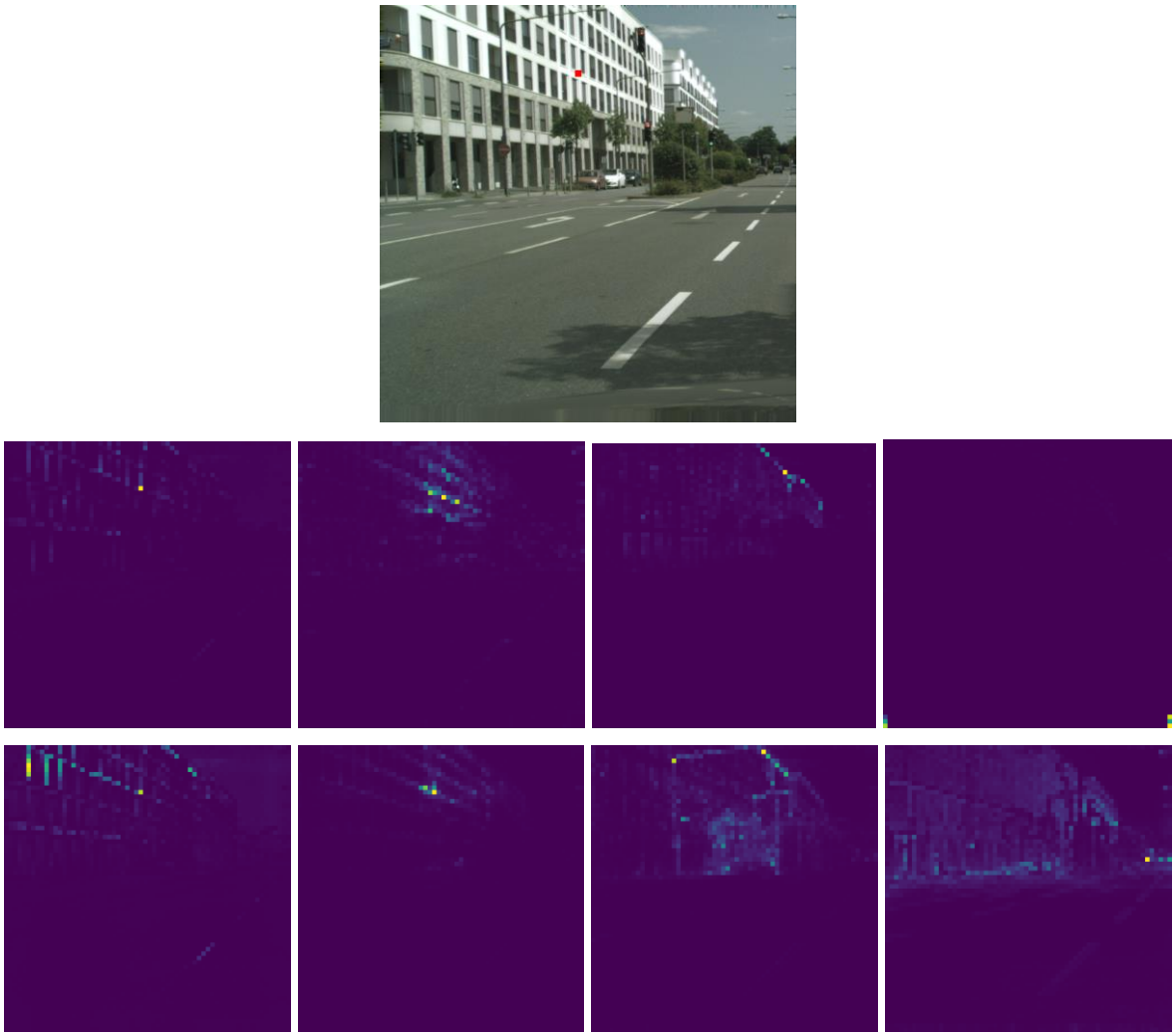
Drugi primjer (Slika 48) je regija na prednjem dijelu crnog auta na lijevoj strani slike. Nakon 1. sloja je pažnja usmjerena na označenu regiju i, za razliku od prethodnog primjera, još nije proširena na okolnu regiju iako Segmenter-B daje nešto više pažnje okolnoj regiji. Poslije 4. sloja oba modela daju pažnju cijelom autu na kojem je označena regija, čak Segmenter-Ti sada bolje raspoređuje pažnju po cijelom autu. Nakon 8. sloja nijedan model još ne daje pažnju drugim autima, ali sada Segmenter-B bolje daje pažnju autu od Segmenter-Ti. Nakon 11. sloja, slično kao u prethodnom primjeru, pažnja

Segmenter-Ti nije dobro raspoređena, ali Segmenter-B sada daje pažnju autima u pozadini i desno, a pažnja na autu lijevo se smanjuje.



Slika 48 Ulazna slika s crveno označenom regijom (gore), mape pažnje kodera Seg-Ti (sredina) i mape pažnje kodera Seg-B (dolje) nakon 1., 4., 8. i 11. sloja kodera (objašnjenje u tekstu).

Sljedeći je primjer (Slika 49) regija na središnjem dijelu zgrade. Nakon 1. sloja oba modela daju pažnju regiji i sličnim okomitim regijama na zgradi, s time da je Segmenter-B bolji. Nakon 4. sloja situacija je slična, a prema kraju kodera Segmenter-B sve više raspoređuje i širi pažnju na prostor cijele zgrade, a Segmenter-Ti nije baš u tome uspješan. Nakon 11. sloja pažnja Segmenter-B je dobro raspoređena po cijeloj zgradi, jedino što je jače aktivirana regija među stablima desno od zgrade koja ne bi trebala biti.

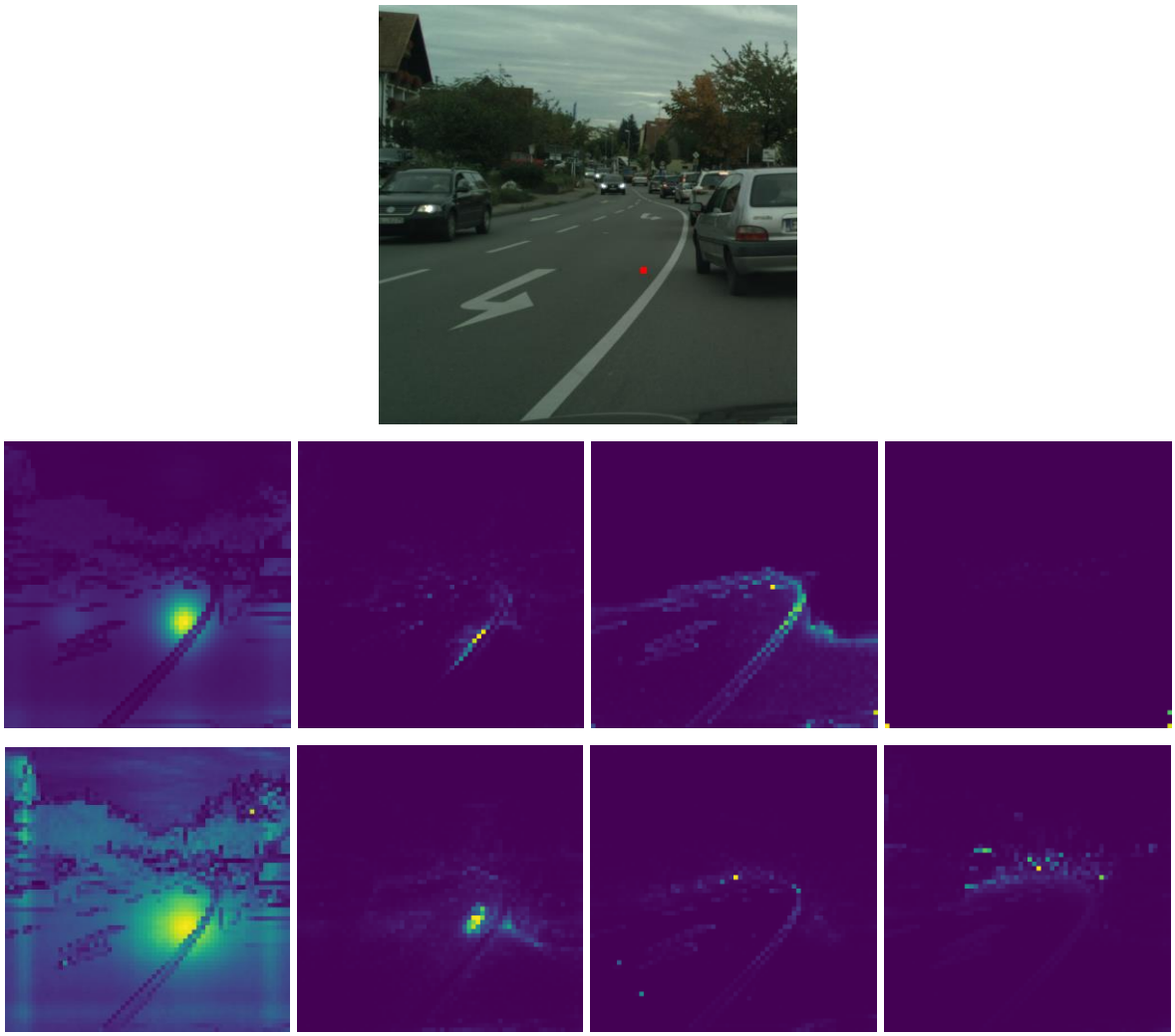


Slika 49 Ulazna slika s crveno označenom regijom (gore), mape pažnje kodera Seg-Ti (sredina) i mape pažnje kodera Seg-B (dolje) nakon 1., 4., 8. i 11. sloja kodera (objašnjenje u tekstu).

Naredni je primjer drugačiji od prethodnih jer promatra mali objekt – semafor (Slika 50). Nakon 1. sloja oba modela daju pažnju označenoj regiji semafora (crveno svjetlo) iako Segmenter-Ti daje pažnju i okomitim uzorcima na zgrade što ne bi trebao. Nakon 4. sloja se pažnja širi već na cijeli semafor (ne više samo crveno svjetlo) za oba modela i također, na donji semafor s crvenim svjetlom na istom stupu dolje – za oba modela iako Segmenter-B daje jaču pažnju tom donjem semaforu. Nakon 8. sloja je pažnja još više proširena i to na semafor sa zelenim svjetlom na nižem stupu prema desno iza. Nakon 11. sloja pažnja Segmenter-Ti slabi, a Segmenter-B sad naglašava sva 3 dosad spomenuta semafora i još semafore na lijevoj strani slike kod zgrade.

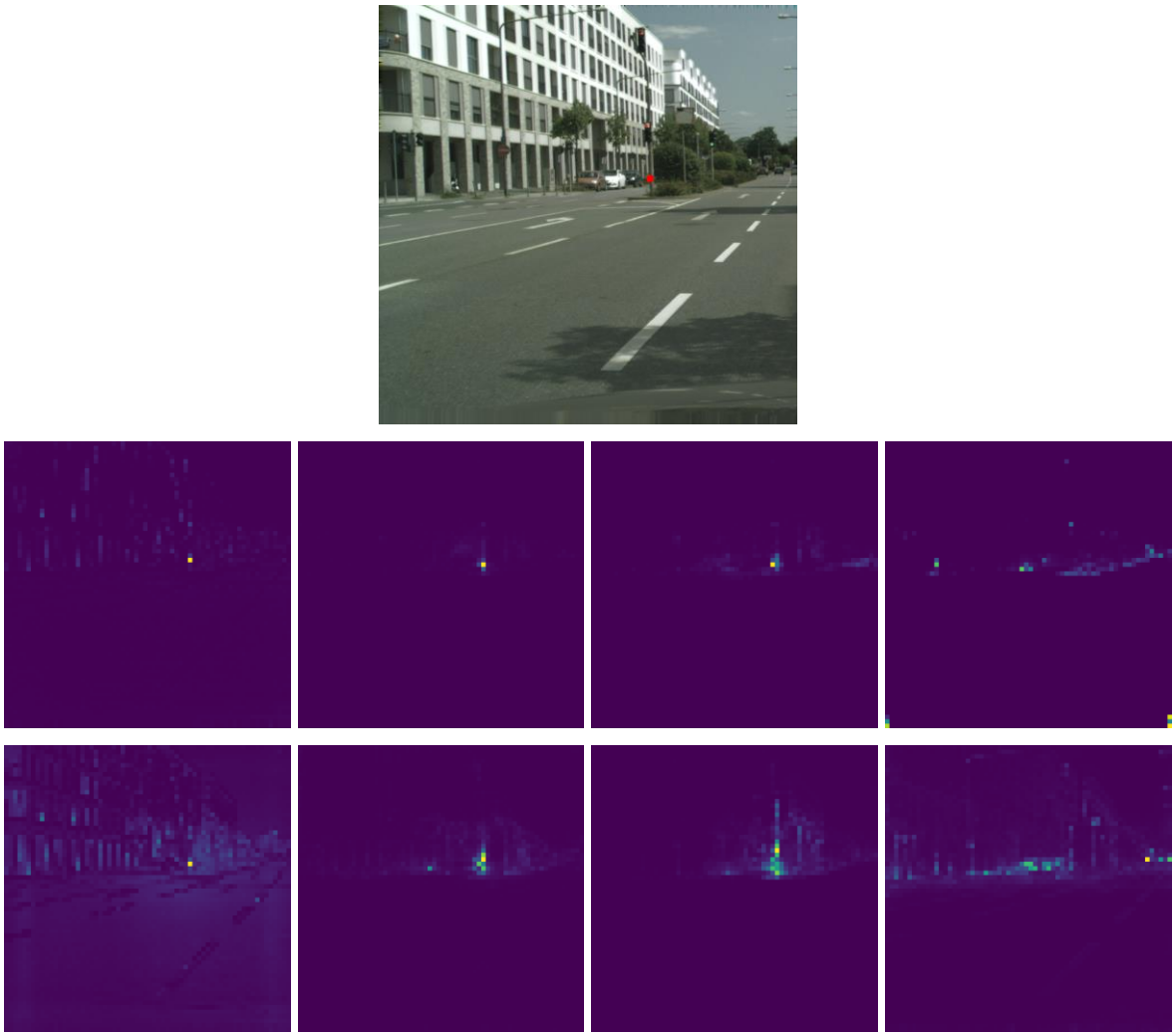
Slika 50 Ulazna slika s crveno označenom regijom (gore), mape pažnje kodera Seg-Ti (sredina) i mape pažnje kodera Seg-B (dolje) nakon 1., 4., 8. i 11. sloja kodera (objašnjenje u tekstu).

Sljedeći je primjer (Slika 51) regija ceste naprijed u sredini blizu bijele linije. Već nakon 1. sloja pažnja je proširena na okolicu regije, Segmenter-B širi na područje cijele ceste i stabala. Nakon 4. i 8. sloja se pažnja kod Segmenter-Ti pridaje skoro pa isključivo središnjoj bijeloj liniji na cesti iako to nije toliko neočekivano jer se označena regija nalazi vrlo blizu bijele linije. Donekle slično se ponaša Segmenter-B. Pri kraju kodera ponovo Segmenter-B ostavlja pažnju, no samo na dijelu ceste, a pažnja se gubi na prednjem dijelu.



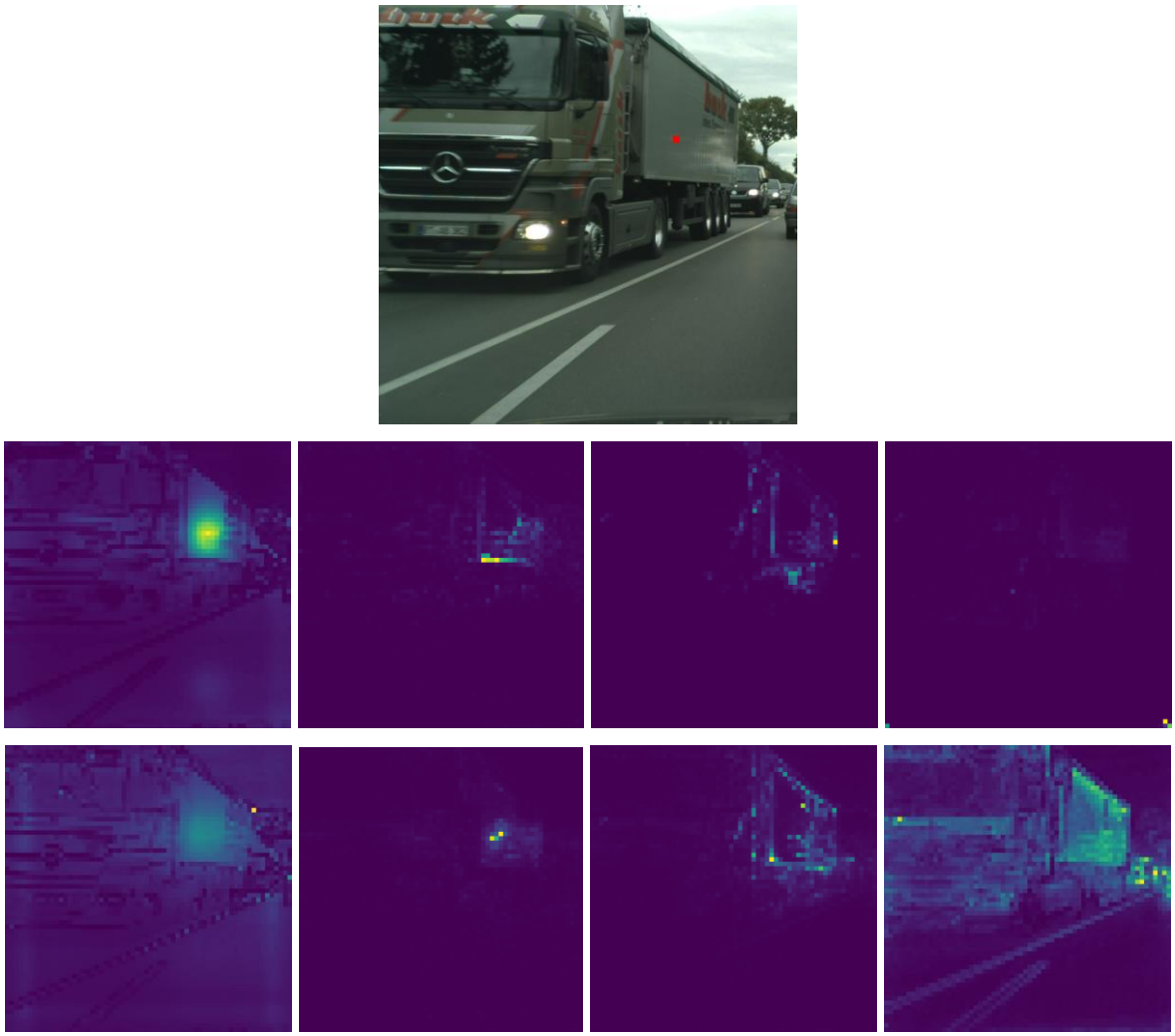
Slika 51 Ulazna slika s crveno označenom regijom (gore), mape pažnje kodera Seg-Ti (sredina) i mape pažnje kodera Seg-B (dolje) nakon 1., 4., 8. i 11. sloja kodera (objašnjenje u tekstu).

Iduće imamo (Slika 52) primjer sličan prethodnom – donji dio stupa s crvenim semaforima. Nakon 1. sloja oba modela daju pažnju toj regiji stupa, ali i sličnim okomitim uzorcima na zgradi što je malo očekivano jer označena regija zahvaća osim stupa i dio zgrade iza. Nakon 4. sloja se iz pažnje izbacuju ti elementi zgrade, a nakon 8. sloja se pažnja širi, posebno u Segmenter-B na cijeli stup i okolne stupove na desnoj strani. Nakon 11. sloja je zanimljiva situacija gdje sada oba modela daju pažnju stupovima na slici. Iako ne toliko dobro, pažnja Segmenter-Ti tu nije toliko „oslabljena“ kao u prethodnim primjerima, a Segmenter-B daje pažnju stupovima, ali ipak ponovo i okomitim regijama na zgradi, no samo u donjem dijelu zgrade (blizu cesti gdje bi se moguće nalazili stupovi).



Slika 52 Ulazna slika s crveno označenom regijom (gore), mape pažnje kodera Seg-Ti (sredina) i mape pažnje kodera Seg-B (dolje) nakon 1., 4., 8. i 11. sloja kodera (objašnjenje u tekstu).

Pogledajmo još jedan primjer (Slika 53) gdje je označena regija na sivoj regiji na zadnjem dijelu kamiona. Nakon 1. sloja se pažnja modela očekivano širi na cijeli zadnji (sivi) dio kamiona. Nakon 4. sloja se područje pažnje smanji, a zatim nakon 8. sloja se poveća, s time da nijedan model još ne daje pažnju prednjem dijelu kamiona, te Segmenter-B daje jaču pažnju zadnjem dijelu kamiona. Nakon 11. sloja se vidi očita razlika ovih modela. Iako Segmenter-B daje pažnju i autima iza kamiona, on vrlo precizno daje jaku pažnju cijelom kamionu.



Slika 53 Ulazna slika s crveno označenom regijom (gore), mape pažnje kodera Seg-Ti (sredina) i mape pažnje kodera Seg-B (dolje) nakon 1., 4., 8. i 11. sloja kodera (objašnjenje u tekstu).

Pregledom ovih mapa pažnje može se vidjeti da transformer poput Segmentera već u ranoj fazi unutar kodera počinje davati pažnju širem području i brzo širiti receptivno polje mapa pažnji. Očekivano se pritom bolje mape pažnje daje veći model koji prema kraju sve bolje usmjerava pažnju na objekte od interesa. Treba primijetiti da to radi dobro i za velike objekte poput ceste, zgrade i kamiona, srednje velike objekte poput auta i čak male objekte poput semafora ili uske objekt kao što je stup. Najzanimljivije je pritom kako se pažnja širi ne samo na cijeli objekt na kojem je regija čije se mape pažnje vizualiziraju, nego i na sve druge objekte iste klase, što se najbolje može vidjeti na primjeru semafora gdje je označena regija semafora s crvenim svjetlom, a pažnja se pridaje onda postupno i svim ostalim semaforima (s crvenim ili zelenim svjetlom ili bez svjetla).

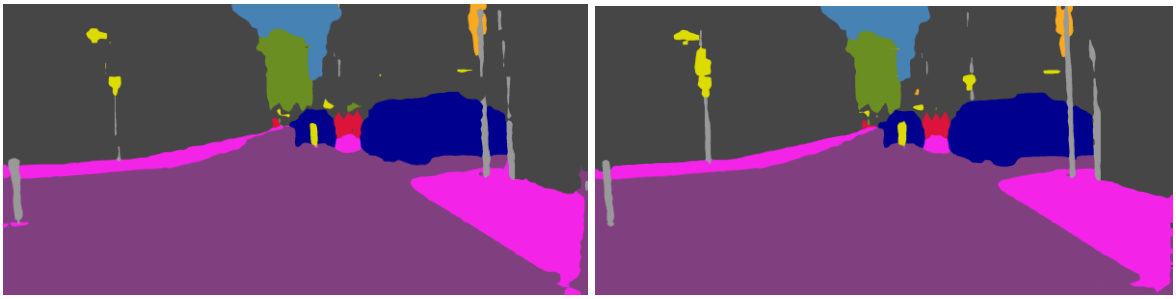
5.8. Kvalitativni eksperimenti

Prikažimo sada kvalitativne rezultate nekoliko naučenih modela vizualizacijom segmentacijskih mapa gdje ćemo redom gledati mape modela: Segmenter-Ti i Segmenter-B, SWIN-B, SwiftNet-ResNet-18-ss i SwiftNet-ResNet-18-pyr, ansambl SwiftNet-ResNet-18-pyramid + Segmenter-Ti, te arhitektura SwiftNet-ResNet-18-pyr + Segmenter-Ti sa zasebnim optimizatorima (1. eksperiment) u varijanti gdje je ulaz u Segmenter jednak ulazu u 3. razinu piramide SwiftNeta i gdje je jednak središnjem isječku. Prije mapa dana je ulazna slika i točne (*ground truth*) oznake.

Prvi primjer daje Slika 54.

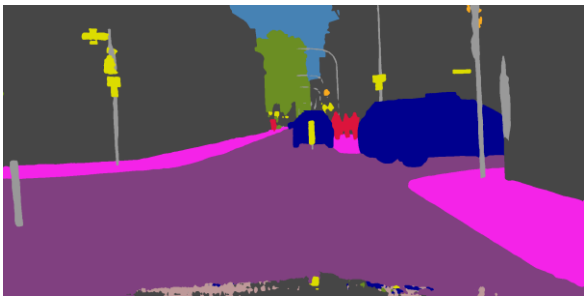
Slika 54 Slika (lijevo) i GT segmentacijske oznake slike (desno).

Mape Segmentera (Slika 55). Oba modela segmentiraju većinu objekata, a Segmenter-B je bolji posebno na uskim objektima (stupovima). Iako ni Segmenter-B ne uspijeva segmentirati sve stupove u pozadini, većinu ih dobro označi. Ovdje se dogodila i zabuna kod oba modela jer se rub zgrade desno naprijed segmentira kao stup. Ostali objekti su dobro segmentirani, Segmenter-B očekivano bolje označava granice većine objekata.



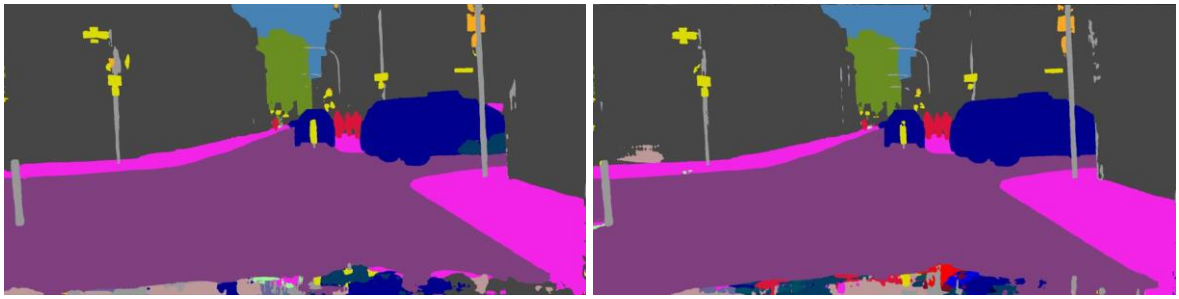
Slika 55 Segmentacijska mapa modela Seg-Ti (lijevo) i Seg-B (desno).

Mape SWIN-a (Slika 56). SWIN-B vrlo uspješno označava sve objekte pa čak i male uske stupove u pozadini, jedino se i ovdje događa zabuna ruba zgrade desno za stup. Na ovom primjeru SWIN-B bolje segmentira od Segmenter-B, što se vidi i na primjeru troje ljudi (crveno oznaka) čije granice bolje pristaju.



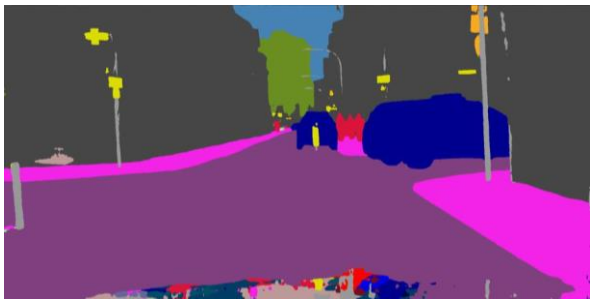
Slika 56 Segmentacijska mapa modela SWIN-B.

Mape SwiftNeta (Slika 57). Oba modela SwiftNet-RN-18 daju dosta dobre i slične segmentacije. Ipak, piramidalna varijanta je nešto bolja – bolje segmentira auto (donji desni dio auta naprijed) i stup desno skroz u pozadini iako „prepoznaje“ stup na lijevoj strani slike iza gdje nema stupa.



Slika 57 Segmentacijska mapa modela SwiftNet-RN18-ss (lijevo) i SwiftNet-RN18-pyr (desno).

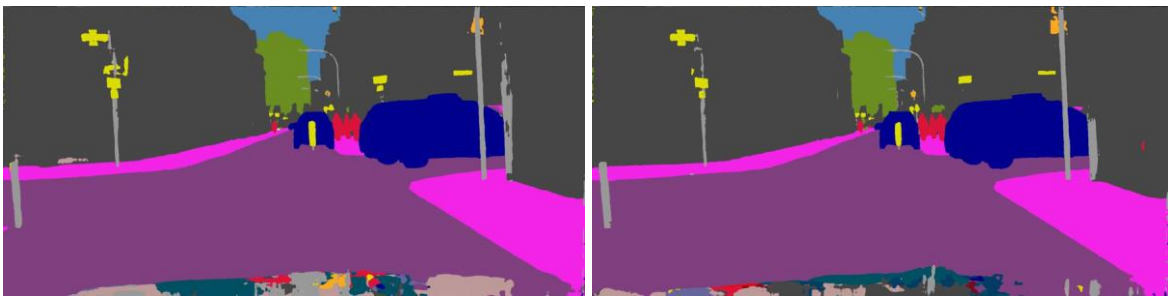
Mape ansambla SwiftNet-Segmenter (Slika 58). Ansambl očekivano ima bolje rezultate od modela SwiftNeta i Segmenter pojedinačno pa su i segmentacije vidljivo bolje. Bolje segmentira od SwiftNeta (SN-RN18-pyr), još bolje od Segmenter-T. Segmentacijska mapa je vrlo slična mapi SwiftNeta, no malo bolja jer npr. stup koji je SwiftNet prepoznao gdje ga nema, ovdje ne postoji, već je taj dio ispravno segmentiran oznakom građevine.



Slika 58 Segmentacijska mapa ansambla SwiftNet-RN18-pyr i Seg-Ti.

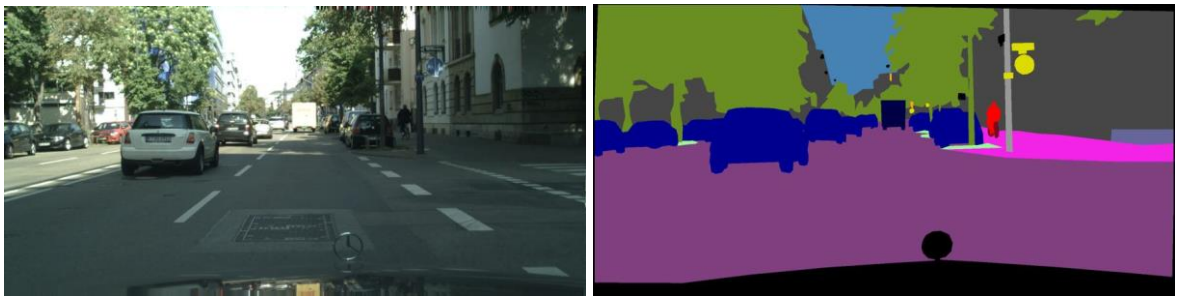
Mape arhitekture SwiftNet-Segmenter (Slika 59). Mape su vrlo slične kako su i segmentacijske točnosti ovih dvaju modela podjednake, a nešto lošije od mapa SwiftNet-

RN-18-pyr jer je točnost ovih modela niža. Obje mape su dosta dobre, no postoji prije primijećena zamjena ruba zgrade za stup što je zabuna puno manje izražena kod SwiftNeta.



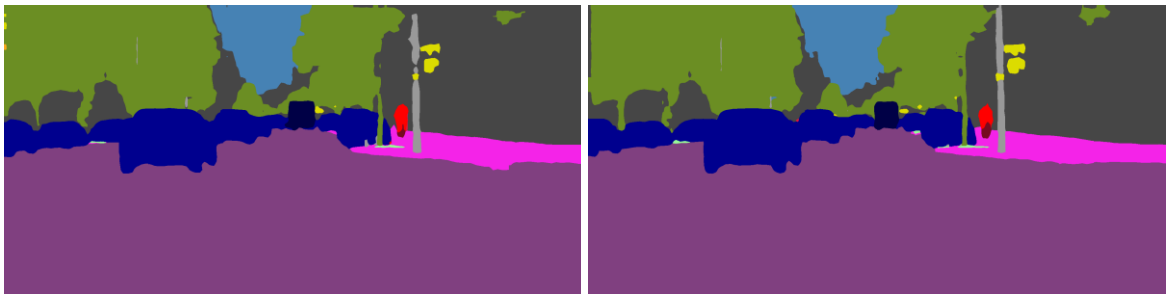
Slika 59 Segmentacijska mapa združenog modela SwiftNet-RN18-pyr i Seg-Ti. Lijevo je za model u kojem Segmenter prima isti ulaz kao ulaz u 3. razinu piramide SwiftNeta, a desno je za model u kojem Segmenter prima središnji isječak slike.

Drugi primjer prikazuje Slika 60.



Slika 60 Slika (lijevo) i GT segmentacijske oznake slike (desno).

Mape Segmentera (Slika 61). Oba modela dosta dobro segmentiraju, a Segmenter-B bolje segmentira deblo stablo lijevo od stupa, nogostup na desnoj strani i stup naprijed.



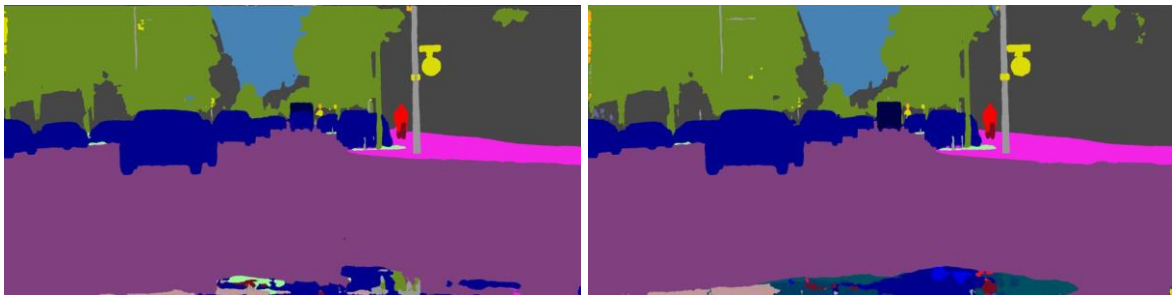
Slika 61 Segmentacijska mapa modela Seg-Ti (lijevo) i Seg-B (desno).

Mape SWIN-a (Slika 62). SWIN-B ponovo je vrlo dobar i bolje segmentira od Segmenter-B što se vidi na deblu stabla i stupu koji sad imaju još oštrije granice, okrugli prometni znak na stupu također je bolje segmentiran te posebno se poboljšanje primjećuje na glavi i nogi čovjeka na biciklu koji je bolje segmentiran. Jedino se može primijetiti detalj na

krošnjama stabala lijevo gdje se vidi artefakt „stupa“ koji je *false positive*, koji je jače izražen nego kod Segmentera, slično kao u prethodnom primjeru.

Slika 62 Segmentacijska mapa modela SWIN-B.

Mape SwiftNeta (Slika 63). Međusobno su slične, dosta dobre, iako slično kao SWIN imaju izražen artefakt „stupa“ u krošnji lijevo, a piramidalni SwiftNet je bolji što se vidi na primjeru kamiona koji *singlescale* model djelomično (većinski) segmentira kao auto, a piramidalni model cijeli objekt kao kamion, što je točno.

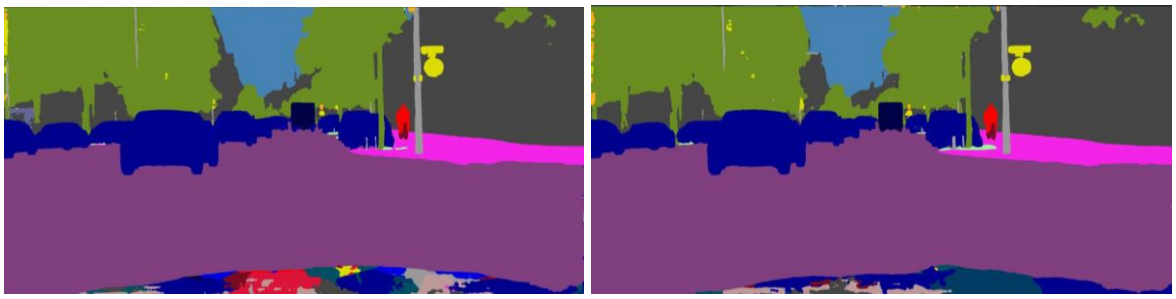


Slika 63 Segmentacijska mapa modela SwiftNet-RN18-ss (lijevo) i SwiftNet-RN18-pyr (desno).

Mape ansambla SwiftNet-Segmenter (Slika 64). Segmentacija je vrlo slična onoj od SN-RN18-py sa zanemarivim razlikama – npr. artefakt „stupa“ je malo tanji.

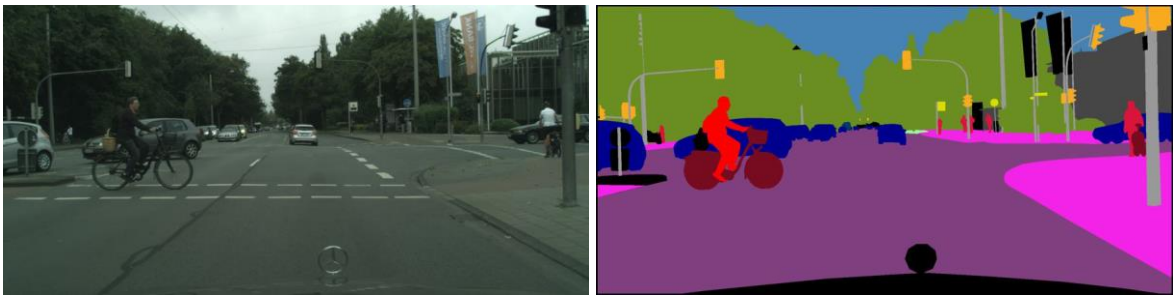
Slika 64 Segmentacijska mapa ansambla SwiftNet-RN18-pyr i Seg-Ti.

Mape arhitekture SwiftNet-Segmenter (Slika 65). Mape su opet vrlo slične međusobno, manja razlika je što donji dio kamiona model „ulaz u 3. razinu piramide“ segmentira kao auto, a model „središnji isječak“ kao kamion. Mape su približne mapama SwiftNeta.



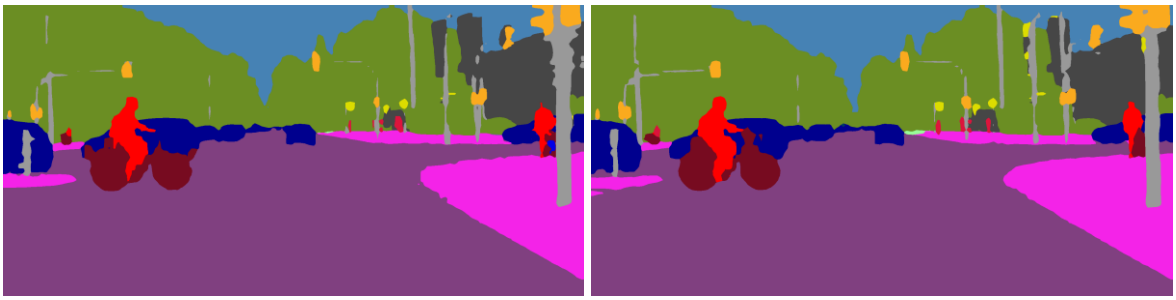
Slika 65 Segmentacijska mapa združenog modela SwiftNet-RN18-pyr i Seg-Ti. Lijevo je za model u kojem Segmenter prima isti ulaz kao ulaz u 3. razinu piramide SwiftNeta, a desno je za model u kojem Segmenter prima središnji isječak slike.

Treći primjer daje Slika 66.



Slika 66 Slika (lijevo) i GT segmentacijske oznake slike (desno).

Mape Segmentera (Slika 67). Općenito su dobre segmentacije, i ovdje se ipak vidi veći problem Segmentera sa segmentacijom uskih stupova, objekte bolje segmentira Segmenter-B (oba čovjeka i bicikla, stupovi, nogostup).

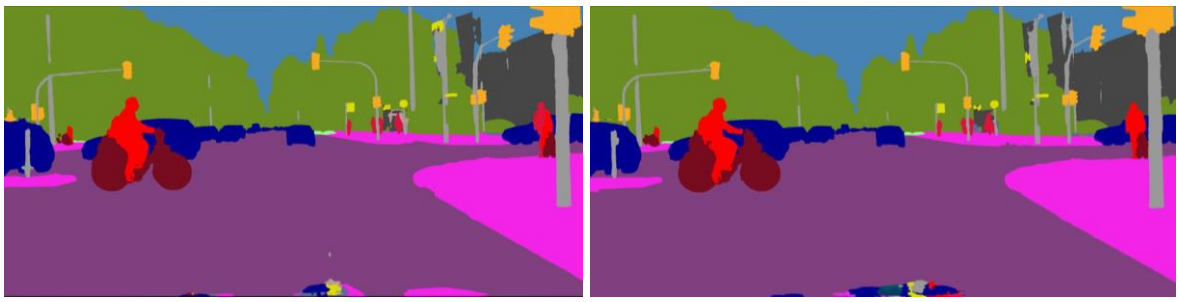


Slika 67 Segmentacijska mapa modela Seg-Ti (lijevo) i Seg-B (desno).

Mape SWIN-a (Slika 68). SWIN-B daje bolje granice od Seg-B i bolje segmentira: male (ljudi iza), srednje (bicikli i biciklisti), uske (stupovi) i velike (cesta, nogostup) objekte.

Slika 68 Segmentacijska mapa modela SWIN-B.

Mape SwiftNeta (Slika 69). Obje mape su vrlo dobre, piramidalni model malo bolje segmentira – npr. područje oko ljudi u pozadini gdje *singlescale* model segmentira dijelove kao stupove što su lažni pozitivi.

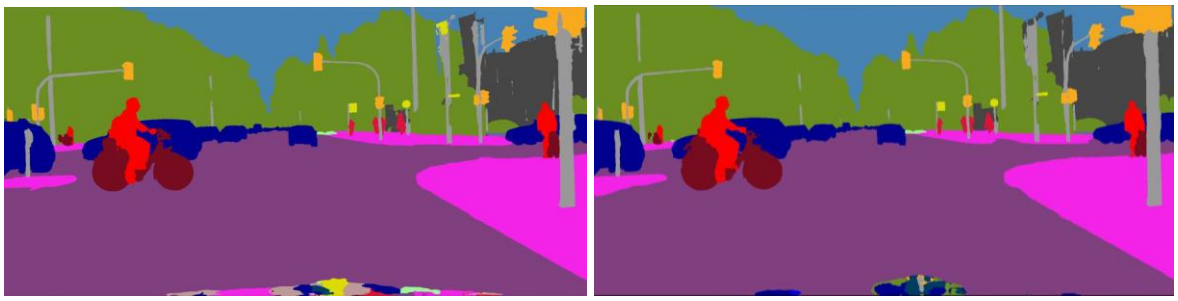


Slika 69 Segmentacijska mapa modela SwiftNet-RN18-ss (lijevo) i SwiftNet-RN18-pyr (desno).

Mape ansambla SwiftNet-Segmenter (Slika 70). Mape su slične mapama SN-RN18-pyr.

Slika 70 Segmentacijska mapa ansambla SwiftNet-RN18-pyr i Seg-Ti.

Mape arhitekture SwiftNet-Segmenter (Slika 71). Segmentacije su vrlo dobre i slične.



Slika 71 Segmentacijska mapa združenog modela SwiftNet-RN18-pyr i Seg-Ti. Lijevo je za model u kojem Segmenter prima isti ulaz kao ulaz u 3. razinu piramide SwiftNeta, a desno je za model u kojem Segmenter prima središnji isječak slike.

Zaključak

U ovom smo radu proučili transformerske modele za gustu predikciju Segmenter i SWIN te ih usporedili s konvolucijskim modelima, posebno SwiftNetom. Prvom usporedbom vremensko-prostorne učinkovitosti transformera i konvolucijskih modela na PASCAL Contextu dobili smo podlogu za daljnja mjerenja na drugim skupovima i modelima. Ustanovili smo da za pojedine konvolucijske modele postoje transformerski modeli slične učinkovitosti npr. Segmenter-B i DeepLabV3+ RN50 odnosno SWIN-B i DeepLabV3+ RN101, a zanimljiv par je bio Segmenter-Ti i SwiftNetPyr-RN18-512 sa sličnim memorijskim otiskom i brzinom učenja, no SwiftNet puno brže zaključuje. Idući korak je bio uvodno isprobati generalizacijsku moć transformera pa smo naučili modele Segmentera na punom ili manjem broju epoha. Dobivene su točnosti zbog ograničenih resursa (korištena je veličina grupe 5 za Seg-B i 1 za Seg-L umjesto 16 za oba modela) na odabranom GPU-u bile manje od službenih: za Seg-B je dobiveno 51.85% (SS) i Seg-L 51.45% (SS) za razliku od službenih 55% (MS) i 59% (MS), no za početak je bilo dovoljno da pokaže potencijal učenja transformera za gustu predikciju, konkretno na primjeru Segmentera, a u kasnijim mjerenjima smo naučili više modela transformera kao i konvolucijskih modela s ciljem usporedbe njihove generalizacijske moći. U nastavku sličnu smo usporedbu proveli na Cityscapesu. Od transformera smo odabrali najmanje modele i za predstavnika konvolucijskih modela odabrali SwiftNet. Modeli arhitekture SwiftNet bili su u svakoj metrici bolji od transformera usprkos manjem memorijskom otisku i bržem učenju i zaključivanju. Značajan rezultat je bila točnost SWIN-B od 83.11% (MS) što je veće od točnosti Segmenter-B od 80.52%, usto SWIN-B ima manji memorijski otisak i brže zaključuje, no računski je zahtjevniji. Zanimljivo je da su SWIN-T i Segmenter-T imali podjednake točnosti (73.59% i 73.47% redom), a suprotan odnos memorijskog otiska i brzine zaključivanja od odnosa SWIN-B i Segmenter-B, a SWIN-T bio je računski zahtjevniji od Seg-T. To je pokazalo da se već u odabiru inačica transformera očituje kompromis točnosti i učinkovitosti. Za SwiftNet SNp-RN18 smo uspjeli reproducirati točnost od 76.48% i utvrdili da je ona između točnosti Seg-T i Seg-B odnosno SWIN-T i SWIN-B, a da ima manji memorijski otisak, brže zaključuje i manje je računski zahtjevan od svih isprobanih transformera (jedino računski nešto zahtjevniji od Seg-T). Dalje smo proveli sličnu, kraću usporedbu na ADE20k. Dobiven je sličan odnos performansi SWIN-T i Seg-T, samo što je točnost SWIN-T od 41.76% (učeno na dijelu

iteracija) bila veća od točnosti Seg-T od 38.1%. Još smo za transformere proveli i analizu asimptotske vremenske složenosti učenja i zaključivanja i prostorne složenosti memorije aktivacija o broju piksela te eksperimentalno pokazali očekivani ishod da brzine i memorija Segmentera kvadratno, a SWIN-a linearno ovisi o broju piksela. Konačno smo implementirali evaluaciju ansambala SwiftNeta i Segmentera Seg-T gdje je ulaz u oba modela cijela slika 1024x2048 iz Cityscapesa i očekivano dobili da su oni bolji od pojedinog modela od kojeg su građeni. Za ansambl SNp-RN18 + Seg-T je dobiven mIoU = 77.43% (SS) odnosno 78.29% (MS) što je veće od točnosti SNp-RN18 76.48% (SS) / 77.54% (MS), a time veće i od točnosti Seg-T (73.16% (SS) / 75.41% (MS)).

Zaključno, ovaj rad je kroz niz eksperimenata pokazao kako *state-of-the-art* modeli zasnovani na pažnji i konvolucijski modeli daju vrlo slične rezultate iako su dizajnom vrlo različiti, temelje se na različitim idejama i dolaze iz različitih polja strojnog učenja. Stoga i transformeri i konvolucijski modeli trebaju biti fokus istraživanja. U budućem radu posebno bi zanimljivo bilo detaljno istražiti kombinacije tih modela u obliku piramida – po uzoru piramidalni SwiftNet koristi tri razine kodera s ulazima različite rezolucije, mogle bi se istražiti opcije: i) višerazinske piramide transformera – npr. trorazinska piramida Segmentera na istim rezolucijama kao SwiftNet (1/1, 1/2 i 1/4) i ii) heterogene piramide konvolucijskih modela i transformera – npr. trorazinska piramida SwiftNeta u kojoj se na jednoj grani (npr. grani najviše rezolucije 1/1) ili na sve 3 grane paralelno isti ulaz šalje u transformer npr. SWIN. Pritom bi bilo dobro isprobati bi li fokalni gubitak svjestan granica imao pozitivan utjecaj na takve piramide kao što to ima na piramidalni SwiftNet. Radi daljnjeg poboljšanja korisno bi bilo i validirati broja razina piramide te utvrditi koji je broj razina optimalan za segmentacijsku točnost, istovremeno da vremensko-prostorne performanse budu prihvatljive. Bilo bi zanimljivo ispitati ovisnost točnosti o rezoluciji ulazne slike za transformere i konvolucijske modele kao i za piramidalne modele čime bi se na temelju danih slika mogao odabrati model koji bi optimalno radio na slikama te rezolucije. Slično bi se moglo još detaljnije analizirati učinkovitost ovisno o rezoluciji slike nego što smo to proveli u ovom radu. Dalje, ovaj rad je pokazao učinkovitost na različitim podatkovnim skupovima, a korak dalje bi bio ocijeniti utjecaj broja klasa u podatkovnom skupu na učinkovitost. Svi ovi mogući smjerovi bi mogli rezultirati važnim spoznajama koje bi bile ključne za odluku koji model primijeniti u kojoj primjeni – koji su modeli optimalni za rad u realnom vremenu, a koji imaju najvišu segmentacijsku točnost.

Literatura

[1] Strudel, R., Garcia, R., Laptev, I., & Schmid, C. (2021). Segmenter: Transformer for semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 7262-7272).

[2] Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., ... & Guo, B. (2021). Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 10012-10022).

[3] Oršić, M., & Šegvić, S. (2021). Efficient semantic segmentation with pyramidal fusion. *Pattern Recognition*, *110*, 107611.

[4] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

[6] Staudemeyer, R. C., & Morris, E. R. (2019). Understanding LSTM--a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*.

[7] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*.

[8] Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., & Jégou, H. (2021, July). Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning* (pp. 10347-10357). PMLR.

[9] Wang, S., Li, B. Z., Khabsa, M., Fang, H., & Ma, H. (2020). Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*.

[10] Xiong, Y., Zeng, Z., Chakraborty, R., Tan, M., Fung, G., Li, Y., & Singh, V. (2021, May). Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 35, No. 16, pp. 14138-14148).

[11] Chen, L. C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2017). Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, *40*(4), 834-848.

[12] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431-3440).

[13] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

[14] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[15] Krešo, I., Krapac, J., & Šegvić, S. (2020). Efficient ladder-style densenets for semantic segmentation of large images. *IEEE Transactions on Intelligent Transportation Systems*, *22*(8), 4951-4961.

[16] Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700-4708).

[17] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, *37*(9), 1904-1916.

[18] Segmenter – službeni Github repozitorij: https://github.com/rstrudel/segmenter

[19] SWIN – službeni Github repozitorij: https://github.com/SwinTransformer/Swin-Transformer-Semantic-Segmentation

[20] Yu, F., & Koltun, V. (2015). Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*.

[21] Chen, L. C., Zhu, Y., Papandreou, G., Schroff, F., & Adam, H. (2018). Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 801-818).

[22] Zhao, H., Shi, J., Qi, X., Wang, X., & Jia, J. (2017). Pyramid scene parsing network. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2881-2890).

[23] Lin, T. Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2117-2125).

[24] Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention* (pp. 234-241). Springer, Cham.

[25] Huang, G., Liu, Z., Pleiss, G., Van Der Maaten, L., & Weinberger, K. (2019). Convolutional networks with dense connectivity. *IEEE transactions on pattern analysis and machine intelligence*.

[26] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

[27] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4510-4520).

[28] Child, R., Gray, S., Radford, A., & Sutskever, I. (2019). Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*.

[29] Beltagy, I., Peters, M. E., & Cohan, A. (2020). Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

[30] Biblioteka PyTorch – dokumentacija: https://pytorch.org/

[31] Biblioteka NumPy – dokumentacija: https://numpy.org/

[32] Biblioteka Pillow – dokumentacija: https://pillow.readthedocs.io/en/stable/

[33] Biblioteka Timm – dokumentacija: https://timm.fast.ai/

[34] Biblioteka mmseg – dokumentacija: https://mmsegmentation.readthedocs.io/en/latest/

[35] Biblioteka mmcv – dokumentacija: https://mmcv.readthedocs.io/en/latest/

[36] SwiftNet – službeni Github repozitorij: https://github.com/orsic/swiftnet

[37] SWIN – službeni Github repozitorij: https://github.com/microsoft/Swin-Transformer

[38] Mottaghi, R., Chen, X., Liu, X., Cho, N. G., Lee, S. W., Fidler, S., ... & Yuille, A. (2014). The role of context for object detection and semantic segmentation in the wild. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 891-898).

[39] Zhou, B., Zhao, H., Puig, X., Fidler, S., Barriuso, A., & Torralba, A. (2017). Scene parsing through ade20k dataset. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 633-641).

[40] Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., ... & Schiele, B. (2016). The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3213-3223).

Gusta predikcija primjenom slojeva pažnje

Gusta predikcija je izazov koji sve više rješavaju transformeri. Motivirani time uspoređujemo generalizacijsku moć i vremensko-prostornu učinkovitost transformera i konvolucijskih modela u gustoj predikciji. Od transformera odabiremo Segmenter i SWIN, a za konvolucijski model piramidalni SwiftNet. Inicijalno učenje Segmentera na PASCAL Contextu pokazalo je generalizacijsku moć transformera. Na Cityscapesu smo pokazali da su transformeri i konvolucijski modeli usporedive točnosti, no da su transformeri znatno zahtjevniji za učenje. Također smo ustanovili da SWIN-B bolje generalizira od Segmenter-B, te zatim na ADE20K otkrili da je SWIN-T bolje točnosti od Segmenter-Ti. Evaluacijom ansambala SwiftNeta i Segmentera dobili smo bolje točnosti od osnovica. Također smo implementirali i započeli validaciju piramida SwiftNet-Segmenter i Segmenter-Segmenter, no bez uspjeha. Najzanimljivija spoznaja je da su transformeri i konvolucijski modeli, iako dizajnom vrlo različiti, dali vrlo slične rezultate. U budućem bi radu bilo važno istražiti piramide modela radi još boljih točnosti od ansambala.

Ključne riječi: duboko učenje, računalni vid, gusta predikcija, transformeri, pažnja, konvolucijski modeli, segmentacijska točnost, vremenska učinkovitost, prostorna učinkovitost, Segmenter, SWIN transformer, piramidalni SwiftNet, PASCAL Context, ADE20K, Cityscapes.

Dense prediction by means of self-attention layers

Dense prediction is a challenge increasingly being solved by transformers. Motivated by this we compare generalization power and spatio-temporal efficiency of transformers and convolutional models in dense prediction. From transformers we choose Segmenter and SWIN, and for convolutional model pyramidal SwiftNet. The initial learning of Segmenters on PASCAL Context indicated generalization power of transformers. We showed on Cityscapes that transformers and convolutional models are of comparable accuracy but transformers are considerably more demanding to learn. We also found that SWIN-B generalizes better than Segmenter-B, and then also on ADE20K that SWIN-T has better accuracy than Segmenter-Ti. Having evaluated ensembles of SwiftNet and Segmenter we got better accuracies than baselines'. We also implemented and started validation of SwiftNet-Segmenter and Segmenter-Segmenter pyramids but without success. The most interesting finding is that transformers and convolutional models, though very different in design, gave very similar results. In future work it would be important to research model pyramids for even better accuracies than ensembles'.

Keywords: deep learning, computer vision, dense prediction, transformers, attention, convolutional models, segmentation accuracy, time efficiency, spatial efficiency, Segmenter, SWIN transformer, pyramidal SwiftNet, PASCAL Context, ADE20K, Cityscapes.