SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5121

Ugradbena izvedba konvolucijskog modela za semantičku segmentaciju

Stjepan Močilac

Zagreb, lipanj 2017.

Zahvaljujem se mentoru prof. dr. sc. Siniši Šegviću na stručnim savjetima i pomoći pri izradi rada.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`***.***

SADRŽAJ

1. Uvod

Semantička segmentacija slike je podjela slike na koherentne i semantički smislene dijelove radi lakše analize slike. Preciznije definirano, semantička segmentacija slike je proces pridjeljivanja semantičke oznake svakom pikselu (eng. pixel). Semantička segmentacija je problem koji većina konvencionalnih metoda ne može riješiti.

Danas najbolje rezultate na području semantičke segmentacije ostvaruju duboke konvolucijske neuronske mreže. U ovom radu posebno ćemo se orjentirati na strogo nadzirano učenje gdje je u svakoj piknji skupa za učenje poznata informacija o pripadajućem semantičkom razredu. Parametre prednjeg dijela mreže inicijaliziramo parametrima koji su naučeni na klasifikacijskom problemu(eng. fine tuning). Primjer pogodne klasifikacijske arhitekture je vgg-16. Korištena je na ILSVRC natjecanjima, a dizajnirana je od strane Karena Simonyana i Andrewa Zissermana u Oxfordu [2].

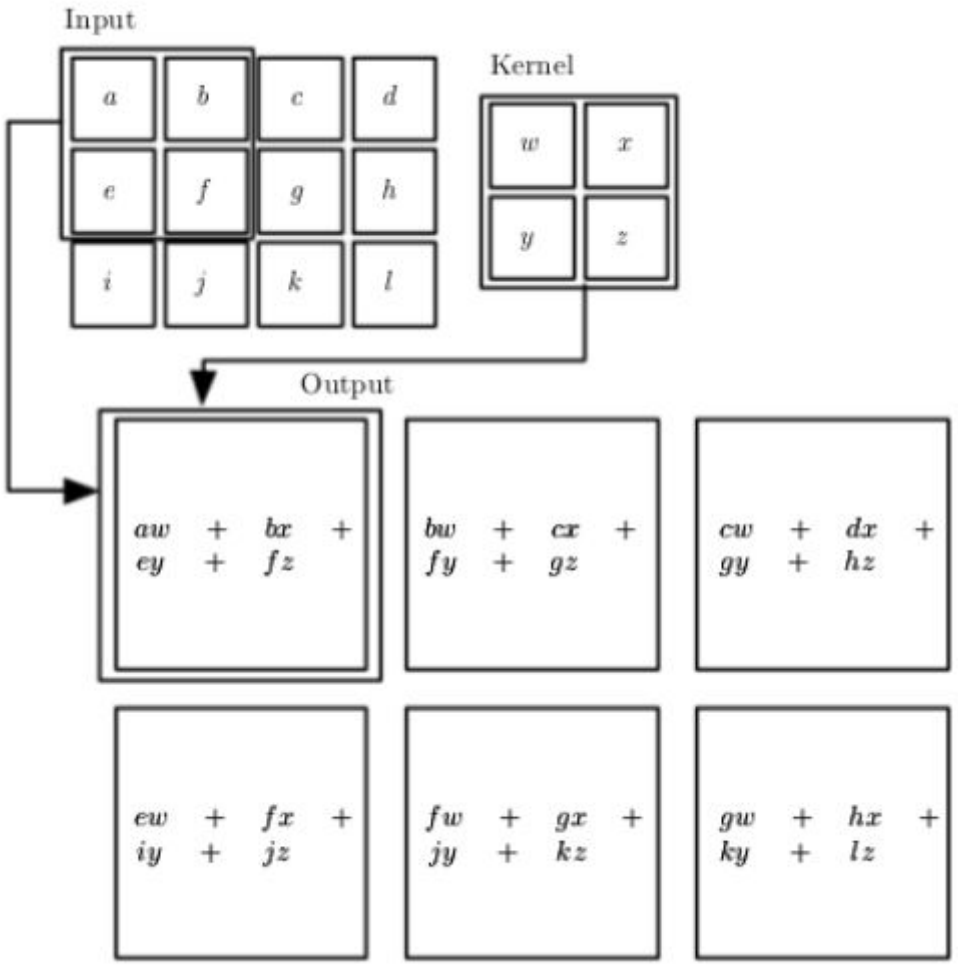
Postoje mnoge zanimljive primjene na području semantičke segmentacije prirodnih scena, čiji je problem neriješeni problem računalnog vida. Neke od primjena su: autonomni automobili, detekcija objekata, obrada slike u medicini, sustav za kontrolu prometa, … . U ovom radu glavna referenca će biti primjena segmentacije u autonomnim automobilima, odnosno u prometu.

U sljedećem poglavlju dan je opis arhitekture vgg-16, opis slojeva i algoritma učenja mreže. U trećem poglavlju stoji opis ugradbenog računala TX1. Četvrto poglavlje opisuje implementaciju u programskom jeziku python. Pojedinost biblioteka aravis i tensorflow su također opisane u četvrtom poglavlju. U petom poglavlju su rezultati eksperimentalnih testiranja i njihove usporedbe s obzirom na računalni sustav i metodi implementacije.

2. Algoritmi i metode strojnoga učenja

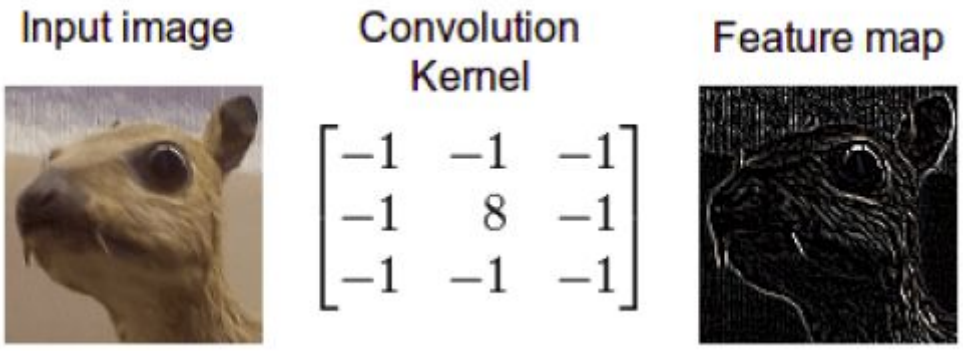
2.1. Konvolucija

Sažeto rečeno, konvolucija je procedura preplitanja dva izvora informacija. Kada konvoluciju primjenjujemo na slike, tada je prvi izvor informacije slika, a drugi izvor je konvolucijski filter. Konvolucija je slična potpuno povezanom sloju, no ona modelira samo lokalne interakcije. Možemo pretpostaviti dimenzije konvolucijskog filtra (eng. kernel) 2x2 i konvolucijski korak 1. Izlaz dobiven primjerom je prikazan na slici 3.



Slika 1. "Što je konvolucija?" [7]

Konvolucija primijenjena na sliku uz konvolucijski filter dimenzije 3x3 i korak filtra 1 prikazana je na slici 4.



Slika 2. Konvolucija slike uz filter koji detektira rubove [10]

Na slici 1 vidi se da konvolucija uzrokuje smanjenje dimenzije slike. Da bi se to izbjeglo koristi se nadopunjavanje nulama (slika 3) [5].

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	(width) x (height) x 3					0	0
0	0						0	0
0	0						0	0
0	0						0	0
0	0						0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Slika 3. Prikaz nadopunjavanja slike nulama za konvolucijski filter 3x3 i korak 1

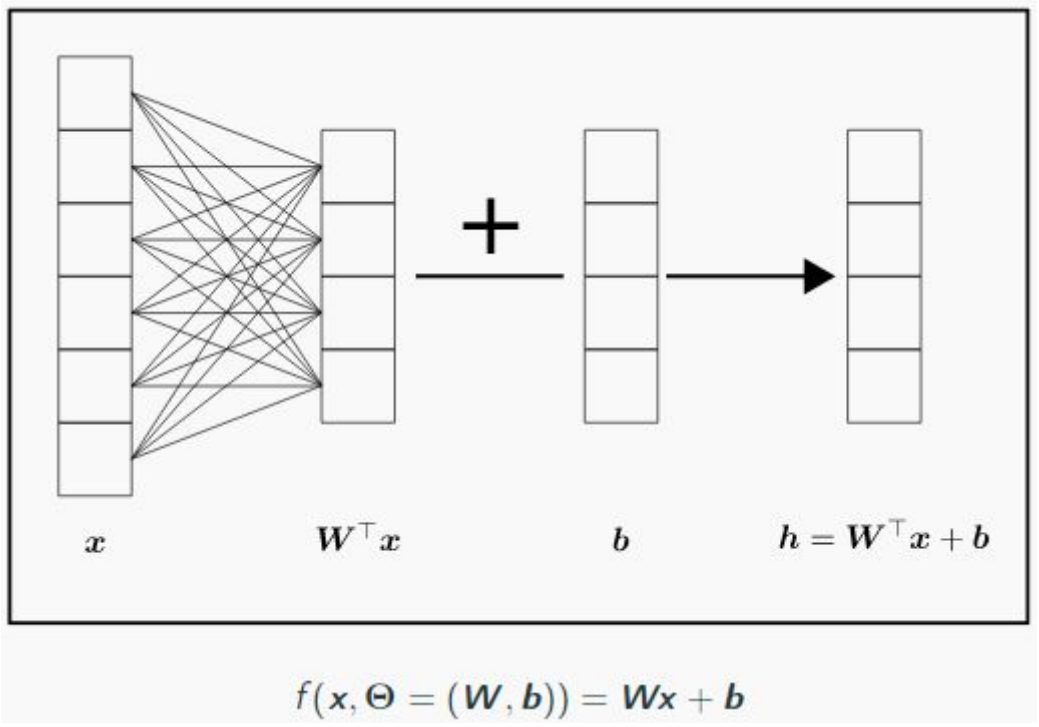
Općenito, dimenzije izlazne slike možemo odrediti prema formuli:

$$O = \frac{(W - K + 2P)}{S} + 1 .$$

O je visina/širina izlazne slike, W visina/širina ulazne slike, K veličina konvolucijskog filtra, P je broj nadopunjavanje redaka/stupava, a S je konvolucijski korak (eng. stride).

2.2. Potpuno povezani sloj

Potpuno povezani sloj (eng. fully connected layer) povezuje sve ulaze neurona u tom sloju sa svim izlazima aktivacijskih funkcija prethodnog sloja (slika 4).

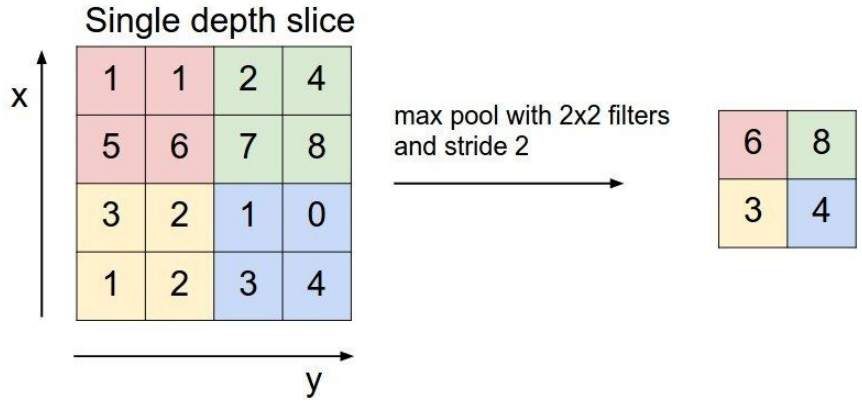


Slika 4. "Potpuno povezani sloj" [7]

Glavna razlika izmedju konvolucijskog i potpuno povezanog sloja je da konvolucijski sloj modelira samo lokalne interakcije. Može se primijetiti da neuroni u oba sloja rade skalarni produkt. Samim time ispada da je konvolucijski sloj moguće konvertirati u potpuno povezani uz odgovarajući skup pravila (i obrnuto).

2.3. Sloj sažimanja

Sažimanje odziva (eng. pooling layer) se često koristi u konvolucijskim neuronskim mrežama u svrhu progresivnog smanjivanja prostorne veličine reprezentacije kako bi se smanjila količina značajki i računalna složenost mreže. Funkcija sažimanja mapira skup prostorno bliskih značajki na ulazu u jednu značajku na izlazu [7]. Funkcija je najčešće neka statistička vrijednost skupa vrijednosti značajki na ulazu. Uobičajno se koristi sažimanje maksimalnim odzivom na način da se odabere najveći element iz prozora te ga se prosljeđuje idućem sloju [slika 5].



Slika 5. Sažimanje maksimalnim odzivom uz filter 2x2 i korak 1 [11]

Kod vgg-16 glavni razlog sažimanja je smanjivanje parametara prvog potpuno povezanog sloja. Stoga, u mreži su četiri sloja sažimanja maksimalnog odziva, što ujedno znači da će izlaz biti 16 puta manjih dimenzija.

2.4. Adam

Za učenje mreže u ovom radu koristi se varijanta stohastičkog gradijentnog spusta poznata pod imenom Adam. Baziran je na adaptivnoj procjeni momenta(eng. ADAptive Moment estimation). Empirijski dobivenim rezultatima pokazalo se da Adam verzija funkcionira dobro u praksi [1, 2]. Ima sposobnost brze konvergencije i

računarski je dovoljno efikasan [1]. Pseudokod je dan u nastavku [7]:

Unesi:

1. η - faktor učenja
2. β_1 , β_2 - eksponencijalne stope propadanja za procjenu momenta (preporučeno 0.9 u 0.999 respektivno)
3. ε - pozitivna konstanta, korak učenja
4. δ - mala konstanta za numeričku stabilizaciju (preporučeno 10^{-8})

Inicijaliziraj:

1. θ , parametri
2. s, r $\leftarrow$ 0 , pomični presjeci
3. t $\leftarrow$ 0 , brojač iteracije

Ponavljaj dok nije kovergencija:

1. t $\leftarrow$ t + 1
2. Odredi lokalne gradijente prema
$$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$$
3. Ažuriraj s, r prema:
$$\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$$
$$\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$$
4. Ispravi pomake ulaza za pomične presjeke prema:
$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$$
$$\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$$
5. Izračunaj delta vrijednosti parametara:
$$\Delta\boldsymbol{\theta} = -\epsilon\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$$
6. Ažuriraj parametre:
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$$

2.5. Vgg-16

VGG-16 mreža je vrlo duboka umjetna neuronska mreža dubine 16 slojeva čiji je približan broj parametara 138 milijuna [6]. Prikaz arhitekture mreže dan je tablicom 1.

conv3-64 conv3-64
maxpool
conv3-128 conv3-128
maxpool
conv3-256 conv3-256 conv3-256
maxpool
conv3-512 conv3-512 conv3-512
maxpool
fc-4096 fc-4096 fc-1000
softmax

Tablica 1. Prikaz arhitekture mreže

Isječak koda koji definira mrežu VGG-16 napisan je uz pomoć biblioteka tensorflow i numpy u programskom jeziku python.

```
with tf.contrib.framework.arg_scope([layers.convolution2d],
                                    kernel_size=3, stride=1, padding='SAME', rate=1,
                                    activation_fn=tf.nn.relu,
                                    normalizer_fn=None, weights_initializer=None,
                                    weights_regularizer=layers.l2_regularizer(weight_decay)):
    net = layers.convolution2d(inputs, 64, scope='conv1_1')
    net = layers.convolution2d(net, 64, scope='conv1_2')
    net = layers.max_pool2d(net, 2, 2, scope='pool1')
    net = layers.convolution2d(net, 128, scope='conv2_1')
    net = layers.convolution2d(net, 128, scope='conv2_2')
    net = layers.max_pool2d(net, 2, 2, scope='pool2')
    net = layers.convolution2d(net, 256, scope='conv3_1')
    net = layers.convolution2d(net, 256, scope='conv3_2')
    net = layers.convolution2d(net, 256, scope='conv3_3')
    net = layers.max_pool2d(net, 2, 2, scope='pool3')
    net = layers.convolution2d(net, 512, scope='conv4_1')
    net = layers.convolution2d(net, 512, scope='conv4_2')
    net = layers.convolution2d(net, 512, scope='conv4_3')
    net = layers.max_pool2d(net, 2, 2, scope='pool4')
    net = layers.convolution2d(net, 512, scope='conv5_1')
    net = layers.convolution2d(net, 512, scope='conv5_2')
    net = layers.convolution2d(net, 512, scope='conv5_3')
with tf.contrib.framework.arg_scope([layers.convolution2d],stride=1,padding='SAME',
                                    weights_initializer=layers.variance_scaling_initializer(),
                                    activation_fn=tf.nn.relu,normalizer_fn=layers.batch_norm,
                                    normalizer_params=bn_params,
                                    weights_regularizer=layers.l2_regularizer(1e-3)):
    net = layers.convolution2d(net, 512, kernel_size=3, scope='conv6_1',rate=4)
logits = layers.convolution2d(net, 19, 1,padding='SAME', activation_fn=None,scope='unary_2',rate=2)
```

Aktivacijska funkcija je zglobnica (eng. rectifier linear unit). Ako je x ulaz u neuron, tada je ReLU definirana oblikom [2]:

$$f(x) = \max(0, x),$$

S ciljem vraćanja slike na početnu dimenziju provodi se bilinearna interpolacija sljedećim kodom:

logits=tf.image.resize_bilinear(logits,[height,width],name='resize_score')

Kako po kanalima imamo 19 mogućih razreda, pozivamo argmax koja vraća razred sa najvećom vjerojatnošću na tom mjestu.

Parametar na koji ćemo se fokusirati kada govorimo o uspješnosti učenja biti će parametar IoU (eng. intersection over union):

$$IoU_i = \frac{TP_i}{TP_i + FN_i + FP_i}$$

Gdje vrijedi:

TP_i (eng. true positive) je broj uspješno klasificiranih piksela u razred i,

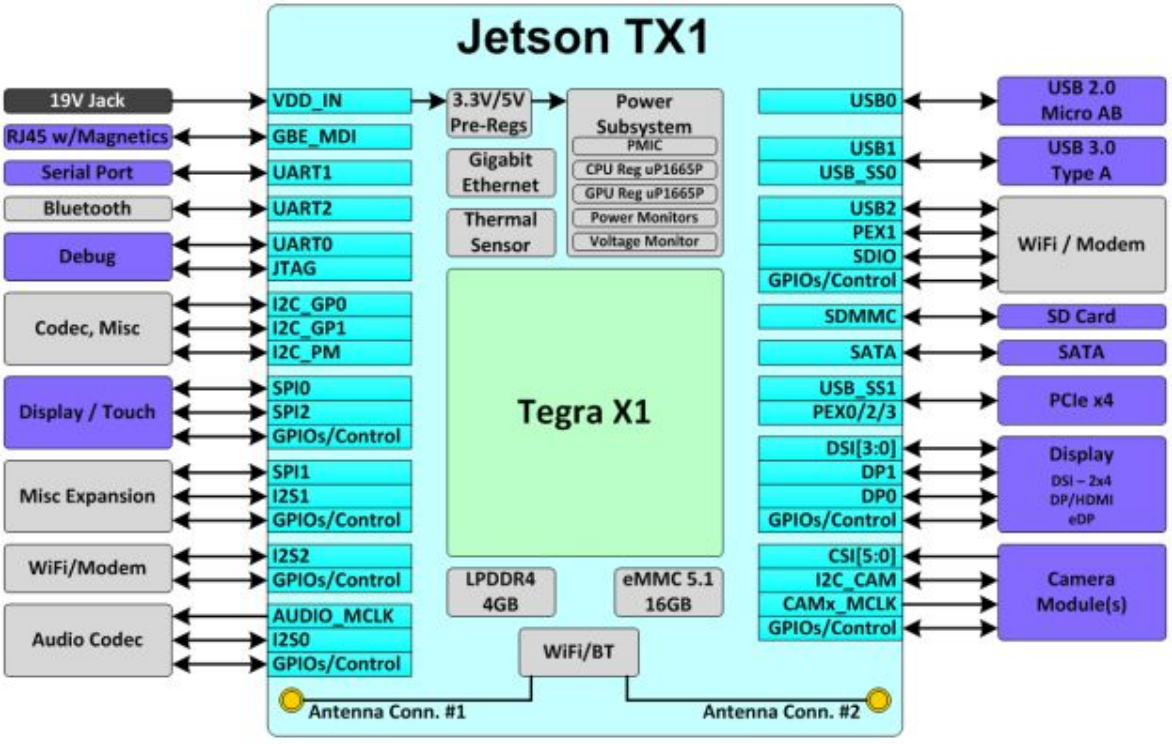
FP_i (eng. false positive) je broj pogrešno klasificiranih piksela u razred i,

FN_i (eng. false negative) je broj piksela koji pripadaju razredu i, ali nisu klasificirani u razred i [7].

3. Ugradbeno računalo TX1

TX1 je ugradbeni SoM (eng. System-on-Module) dizajniran za operacijski sustav Linux koji dolazi u paketu NVIDIA-ih proizvoda pod nazivom Jetson. Namijenjen je za kompleksne aplikacije na ugradbenim računalima i to na području računalnog vida. Performansa mu se kreće oko 10^12 FLOPs-a (eng. floating point operation per second) [8].

Jetson TX1 je zapravo omotač oko Tegra X1 SoC-a (eng. System on Chip) izvedenog u 20nm tehnologiji. Arhitektura modula prikazana je na slici 6. Procesor je 64-bitni četverojezgreni ARM Cortex-A57. Periferno sučelje TX1 omogućuje spajanje i do 6 MIPI CSI-2 kamera istovremeno (na dual ISP-u (eng. Image Signal Processor)). Potrošnja snage u praznom hodu iznosi oko 1W, u normalnom režimu rada se kreće oko 10W, dok je potrošnja na maksimalnom opterećenju 15W. Grafička jedinica temeljena je na arhitekturi Maxwell, te sadrži 256 CUDA jezgri. Omogućava dinamički paralelizam na NVidijinoj instrukcijskoj arhitekturi razine 5.3. U okviru instalacijskog paketa dolaze CUDA i cuDNN (više u poglavlju 4).



Slika 6. "Jetson TX1 block diagram" [8]

4. Programska implementacija

4.1. CUDA i cuDNN

Cuda je paralelna računarska platforma i programski model koji omogućuje korištenje GPU-a za računalnu upotrebu opće namjene. Programer i dalje koristi programske jezike s kojima je upoznat (npr. C, C++, Fortran ili bilo koji drugi podržani jezik) i koristi proširenja tih programskih jezika u obliku nekoliko osnovnih ključnih riječi. Pojednostavljen tok procesa u cudi je:

1. Kopiraj ulazne podatke iz CPU memorije u GPU memoriju
2. Učitaj GPU program i izvrši ga, podaci se spremaju u priručnu memoriju (eng. cache) radi bolje performanse
3. Kopiraj rezultat iz GPU memorije u CPU memoriju

NVIDIA kompajler za C/C++ poznat je pod nazivom nvcc.

CuDNN je biblioteka primitiva koja koristi GPU uz pomoć cude za ubrzani rad sa dubokim neuronskim mrežama (eng. Deep Neural Network). Ovu biblioteku koriste razni programski okviri za duboko učenje poput okvira Caffe, Caffe2, Torch, Theano i TensorFlow.

4.2. TensorFlow

Tensorflow je biblioteka otvorenog koda namjenjena izvođenju numeričkih proračuna definiranih računskim grafom. Ima fleksibilnu arhitekturu pa se može izvoditi na jednim ili više CPU-ova ili GPU-ova, serveru, pa i na mobilnom uređaju bez ponovnog pisanja koda. Jezgra TensorFlow biblioteke napisana je najvećim dijelom u visoko optimiziranoj kombinaciji programskog jezika C++ i CUDA-e, najvećim dijelom uz pomoć biblioteka Eigen i cuDNN.

Operacije koje definiramo u programskom jeziku python se ne izvode odmah, nego definiraju graf toka podataka (eng. dataflow graph). Grafom smo zapravo rekli da želimo uzeti određene ulazne podatke, primjeniti neke operacije nad njima, te opskrbiti ulaz iduće operacije sa izlazom prethodne. Na takav način smo omogućili da definiramo graf u pythonu, no kada se izvodi model, zapravo se većinom izvodi visoko optimizirani kod u C++-u, CUDA-i ili strojnom jeziku ciljnoga procesora.

U usporedbi sa drugim bibliotekama za strojno učenje, TensorFlow ima puno prednost. Omogućuje jednostavno korištenje više grafičkih procesorskih jedinica (GPU), za razliku od primjerice biblioteke Theano. Vrlo jednostavno programsko sučelje (eng. API), mnogi korisni operatori poput reda (eng. queue), te alati za otklanjanje pogrešaka (eng. debugging tools) i nadgledanje (Tensorboard) također su razlozi ekspanzije TensorFlowa. Zanimljiva značajka koju ova biblioteka omogućuje su kontrolne točke modela (eng. model checkpointing). Kontrolnim točkama moguće je privremeno spremiti model, zaustaviti treniranje, te kasnije učitati spremljeni model i nastaviti sa treniranjem. Performanse i korištenje memorije na GPU slične ostalim konkurentnim bibliotekama (npr. Theano).

Instalacija TensorFlowa na ugradbenom računalu TX1 pojednostavljena nakon objave uspješno generirane wheel datoteke za python 3.5 na službenom NVIDIA forumu. Nakon preuzimanja datoteke, otvaranja ljuske i pozicioniranja u direktorij preuzimanja, potrebno je pokrenuti sljedeću naredbu:

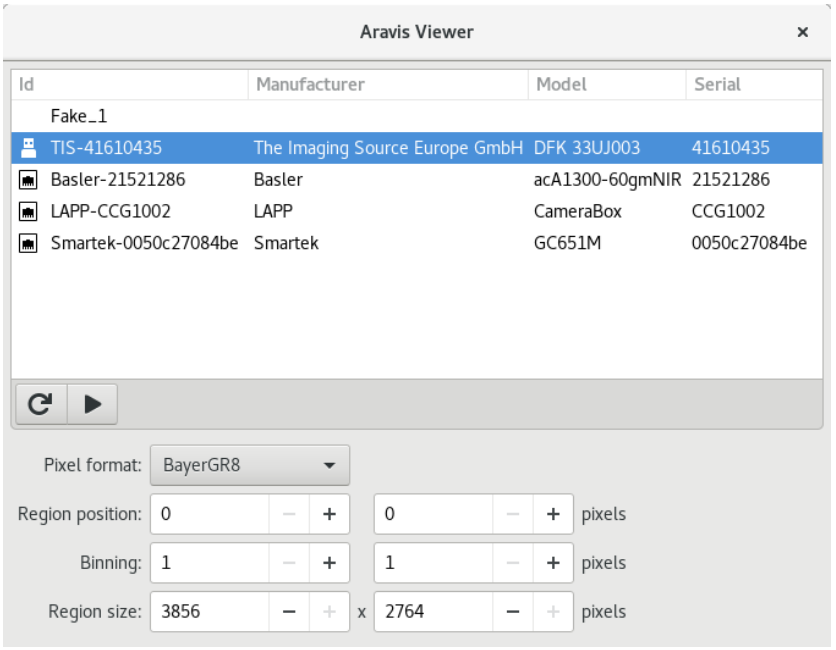
pip install tensorflow-1.0.1-cp35-cp35m-linux_aarch64.whl

Detaljnije upute za instalaciju TensorFlowa na ugradbenom računalu TX1 mogu se pronaći među priloženim datotekama ovog projekta ili na službenim stranicama NVIDIA foruma, dok se dokumentacija nalazi na službenim stranicama TensorFlowa: https://www.tensorflow.org [4].

4.3. Aravis

Aravis je bibilioteka bazirana na okviru glib/gobject i služi za dohvaćanje videa iz Genicam kamera. Podržana je simulacija kamere kao i GUI AravisViewer u

kojem se dohvaća video u realnom vremenu i prikazuje u zasebnom prozoru (uz neke dodatne funkcionalnosti). Instalacija Aravisa objašnjena je na github stranici projekta [3]. Za potrebe rada korištena je verzija Aravis 0.6.



Slika 7. Prikaz grafičkog sučelja programa "Aravis viewer" [3]

4.4. Cityscapes

Skup podataka Cityscapes namijenjen je istraživanjima u području semantičke segmentacije prirodnih scena [9]. Cityscapes je skup podataka velikih razmjera koji sadrži raznovrstan skup stereo video sekvenci zabilježenih u 50ak gradova sa 5000 visoko rezolucijskih slika označenih na razini piksela.

Slika 8. Primjer segmentirane slike iz skupa oznaka skupa podataka Cityscapes

4.5. Evaluacija jedne slike

Implementacija u realnom vremenu zahtjeva velike brzine obrade jednog slikovnog okvira (eng. frame). U svrhu određivanja vremena za različite rezolucije napisan je kod koji ispisuje vrijeme obrade jednog slikovnog okvira na standardni izlaz:

```
def evalone(model, resume_path, image):
    k=8
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        with tf.variable_scope('model'):
            img = tf.placeholder(tf.float32, shape=[1, 256//k, 512//k, 3])
            logits = model.build(img)
        print('\nRestoring params from:', resume_path)
        saver = tf.train.Saver(tf.global_variables(),sharded=False)
        saver.restore(sess, resume_path)
        sess.run(tf.local_variables_initializer())
        t0=time.time()
        out_logits = sess.run(logits, feed_dict = {img : image})
        print(time.time()-t0) # print eval time for one frame
        sess.close()
```

Da bismo mogli evaluirati slikovni okvir definiramo placeholder. Njega nije potrebno inicijalizirati, no da bismo mogli ispravno modelirati graf potrebno je definirati veličinu i tip. U trenutku izvođenja (eng. run-time) definiramo izvor iz kojeg ćemo uzeti podatke. Argument kojim ostvarujemo takvu funkcionalnost unutar metode sess.run zove se feed_dict. Kod je prikazan u nastavku.

```
( … )
   img = tf.placeholder(tf.float32, shape=[1, 256//k, 512//k, 3])
( … )
   out_logits = sess.run(logits, feed_dict={img: image})
```

Oblik (eng. shape) ulaznog podatka je [1, 256//k, 512//k, 3] gdje je konvencija takva da je parametar k potencija broja 2. Prije pokretanja izvođenja grafa potrebno je izgraditi model mreže i obnoviti parametre iz tekstualne datoteke:

```
(...)
   logits = model.build(img)
print('\nRestoring params from:', resume_path)
saver = tf.train.Saver(tf.global_variables(),sharded=False)
saver.restore(sess, resume_path)
(...)
```

4.6. Višedretvena evaluacija slike iz RAM-a

Višedretvena evaluacija slike implementirana je u svrhu redukcije latencije uz pomoć spremnika podataka red (eng. queue). U prethodnom potpoglavlju najprije CPU čeka dohvat slike iz RAM-a, zatim se vrijeme gubi na kopiranju, odnoso prijenosu podatka u GPU memoriju, obradi slike te kopiranju nazad u CPU memoriju. Tek nakon što je to sve odrađeno kreće se u dohvat nove slike. Očito je da imamo ciklus, zbog kojeg se javlja velika latencija. Sada je ideja da dok jedna dretva pribavlja slike iz kamere i prenosi ju na GPU, druga dretva obrađuje sliku.

Podaci za potrebe testiranja su u RAM-u, konkretno u varijabli all_data koja je tipa numpy array. Najprije je potrebno definirati spremnik pod nazivom FIFO (eng.

first in first out) red veličine jedan. Veličina jedan nam je potrebna jer je nama potrebna "najsvježija" slika, odnosno želimo obrađivati najaktualniju sliku. Uz pomoć biblioteke TensorFlow to smo napravili pozivom metode tf.FIFOQueue(1,tf.float32,shape=[myShape]).
Red ćemo punit čitajući podatke iz placeholdera pod imenom feature_input, dok ćemo podatak iz reda dohvaćati u varijablu naziva inputs.

Definiramo metodu load_and_run() koju će nova dretva pozvati nakon poziva thread.start(). Zadatak te dretve je da podatak dodaje u red. U glavnoj dretvi iterativno obrađujemo sliku za slikom. Za potrebe mjerenja vremena izvođenja broj iteracija je sveden na 25. Kompletan kod u pythonu:

```
def evaluate(model, resume_path, image):
        all_data = np.random.random(size=(30,1, 256//k,512//k, 3))
        all_data = all_data.astype(dtype='float32')
        with tf.Session() as sess:
            feature_input = tf.placeholder(tf.float32, shape=myShape)
            queue = tf.FIFOQueue(1, tf.float32, shapes=[myShape])
            enqueue_op = queue.enqueue([feature_input])
            inputs = queue.dequeue()
            sess.run(tf.global_variables_initializer())
            with tf.variable_scope('model'):
                logits = model.build(inputs)
            saver = tf.train.Saver(tf.global_variables(),sharded=False)
            saver.restore(sess, resume_path)
            sess.run(tf.local_variables_initializer())
            def load_and_run():
                for i in range(25):
                    sess.run(enqueue_op, feed_dict={feature_input: all_data[i]})
            t = threading.Thread(target=load_and_run)
            t.start()
            for _ in range(25):
                l = sess.run(logits)
            sess.close()
```

4.7. Komunikacija s kamerom uz pomoć biblioteke Aravis

Generička kontrola nad kamerom može se ostvariti uz pomoć instance razreda ArvCamera. Instanca se može sigurno dohvatiti sljedećim blokom:

```
Aravis.enable_interface ("Fake")
   try:
      if len(sys.argv) > 1:
         camera = Aravis.Camera.new (sys.argv[1])
      else:
         camera = Aravis.Camera.new (None)
   except:
      print ("No camera found")
      exit()
```

Poziv camera.get_payload() je potreban za kreiranje međuspremnika toka podataka (eng. stream) jer vraća potrebnu veličinu buffera u koji se slika pohranjuje:

```
stream = camera.create_stream (None, None)
camera.set_acquisition_mode(Aravis.AcquisitionMode.CONTINUOUS)
device = camera.get_device()
payload = camera.get_payload()
buffer = Aravis.Buffer.new_allocate(payload)
```

Aravis nudi mogućnost dohvaćanja jednog slikovnog okvira (AcquisitionMode SINGLE_FRAME), kao i kontinuirano dohvaćanje slika. Nakon što smo kreirali instance klasa ArvCamera, ArvStream i ArvBuffer, imamo dovoljno za dohvaćanje jednog slikovnog okvira (eng. frame):

```
camera.start_acquisition ()
stream.push_buffer(buffer)
buffer = stream.pop_buffer()
camera.stop_acquisition()
```

Komunikacija kreće pozivom metode camera.start_acquisition(), a završava pozivom metode camera.stop_acquisition(). Metoda stream.push_buffer() zapravo dohvaća podatak sa kamere, puni predani buffer, te ga gura u interni red. Metodom stream.pop_buffer() skidamo taj podatak, odnosno sliku, sa interne strukture red (eng. queue) objekta pod nazivom stream. Ako nakon poziva metode stream.push_buffer(), a prije poziva stream.pop_buffer() driver primi više slika, pozivom metode stream.pop_buffer() dobit ćemo prvu dohvaćenu sliku.

Između ostalog, treba obratiti pozornost na to da Aravis ne omogućava višedretvenost (eng. is not thread safe), odnosno nije moguće instancirati više objekata iste kamere u zasebnim dretvama i dohvaćati slike bez uporabe mutexa[].

4.8. Višedretvna evaluacija slike iz kamere u realnom vremenu

Evaluacija čitanjem zapisa iz RAM-a prikazana je u poglavlju 5.5 . Sada je ideja testirati stvarno okruženje. U jednoj dretvi dohvaća se slika iz kamera, slika se stavlja u red ako je on prazan. Glavna dretva skida podatak sa reda i obrađuje ga, nakon čega zahtjeva novi podatak. Takva funkcionalnost ostvarena je sljedećim dijelovima koda:

```
def nextFrame():
    global buffer
    stream.push_buffer(buffer)
    buffer = stream.pop_buffer()
    data = buffer.get_data()
    imgData = np.ndarray(buffer=data, dtype=np.uint8, shape=(height,width,1))
    img = cv2.cvtColor(imgData, cv2.COLOR_BAYER_RG2RGB)
    img = transform.resize(img, (256//k, 512//k,3), order=3)
    img = img.reshape((1,256//k, 512//k, 3))
    return img

def evaluate(model, resume_path):
    active = True
    camera.start_acquisition()
    with tf.Session() as sess:
        ( .... )
        def load_and_run():
            while active:
                sess.run(enqueue_op, feed_dict={feature_input: nextFrame()})
        ( .... )
        active = False
        time.sleep(1)
        sess.close()
    camera.stop_acquisition()
```

Placeholder u ovom slučaju očekuje podatak od funkcije nextFrame(). U navedenoj funkciji moramo iz međuspremnika (eng. buffer) dobiti sliku odgovarajućih dimenzija. Najprije uz pomoć biblioteke opencv dobijemo RGB sliku, zatim ju skaliramo na željenu rezoluciju bikubičnom interpolacijom uz pomoć biblioteke skimage. Na kraju dodajemo jednu dimenziju koja modelu mreže predstavlja batch_size.

Latencija je aproksimirana na način da smo izmjerili vremensku oznaku(eng. timestamp) dohvaćene slike. Naredba koja omogućava tu funkcionalost je: buffer.get_timestamp()
Vremenska oznaka proslijeđena je modelu mreže, koji taj podatak samo prosljeđuje glavnoj dretvi. Isječak koda:

```
def build(inputs, timestamp):
    (...)
    return logits, timestamp
```

Nakon što je podatak obrađen, i u glavnoj dretvi imamo vremenski otisak upravo obrađe slike, dohvaćamo sadašnju vremensku oznaku te ju oduzimamo sa otiskom slike, te na taj način možemo dovoljno dobro aproksimirati latenciju.

5. Rezultati

5.1. Utjecaj haube

Cityscapes slike su rezolucije 2048x1024. Svaki piksel u slici oznaka predstavlja jedan od 19 mogućih razreda pa kažemo da su oznake zadane jednojediničnim kodom (eng. one-hot notation). Ako bismo uzeli visinu slike 900, odnosno rezoluciju 2048x900 imali bi smo sliku bez haube.

Slika 9. Lijevo - slika bez haube, desno - slika sa haubom

U svrhu provjere utjecaja haube na rezultate učenja semantičke segmentacije scene napravljena su mjerenja dana tablicom 2.

Rezolucija	Hauba uključena	Broj piksela	IoU
312x144	NE	44928	53.71%
256x128	DA	32768	50.02%
288x144	DA	41472	52.32%
320x160	DA	51200	54.25%

Tablica 2. Eksperimentalni rezultati utjecaja haube na učenje mreže

Prikazanim eksperimentom aproksimiran je utjecaj haube na rezultate učenja. Zaključak je da je utjecaj haube na rezultate vrlo malen.

5.2. Utjecaj metoda skaliranja ulaznih slika

Skaliranje slika odvija se interpolacijom. Ovim eksperimentom želimo saznati koliki je utjecaj zaglađivanja slike prije provođenja bikubične interpolacije. Metoda skimage.resize omogućuje skaliranje slike interpolacijom, dok metoda skimage.pyramid_reduce(...) interno poziva metodu skimage.resize(...), ali prije toga radi zaglađivanje temeljeno na Gaussovom filtru:

```
def pyramid_reduce(image, downscale=2, sigma=None, order=1,
                                    mode='reflect', cval=0, multichannel=None):
        ( … )
        smoothed = _smooth(image, sigma, mode, cval, multichannel)
        out = resize(smoothed, out_shape, order=order, mode=mode, cval=cval)
        ( … )
```

Slika 10. Prikaz skaliranih slika (lijevo - bez zaglađivanja, desno - uz zaglađivanje)

Usporedba ove dvije metode skaliranja dana je eksperimentom u tablici 3. Iz eksperimenta se može zaključiti da zaglađivanje nema pozitivan utjecaj na učenje mreže.

Metoda	Rezolucija	Interpolacija	IoU
resize	512x256	bi-cubic	60.29 %
pyramid_reduce	512x256	bi-cubic	59.96 %

Tablica 3. Eksperimentalni rezultati utjecaja metode skaliranja ulazne slike

5.3. Utjecaj metoda skaliranja oznaka

Oznake su zadane jediničnim kodom, pa skaliranje labela nikako ne može biti dobiveno bikubičnom interpolacijom kao u prethodnom primjeru. Metode koje su primjenjive kod skaliranja oznaka su metoda najbližeg susjeda i najčešćeg razreda. Metoda najbližeg susjeda radi upravo ono što naziv sugerira, uzima se najbliži susjed, odnosno vrijednost piksela koja je nabliža sukladnoj novoj poziciji. Metoda najčešći razred uzima najčešće prisutan razred po dijelovima skaliranja, tj. prozorima. Eksperimentalni rezultati dani su tablicom 4.

Slika 11. Prikaz skaliranih oznaka (lijevo - najbliži susjed, desno - najčešći razred)

Metoda	Rezolucija	Interpolacija	IoU
Najčešći razred	512x256	bi-cubic	60.29 %
Najbliži susjed	512x256	bi-cubic	59.32 %

Tablica 4. Prikaz eksperimentalnih rezultata utjecaja skaliranja oznaka

5.4. Eksperimentalni rezultati evaluacije jedne slike

Eksperimentalni rezultati evaluacije jedne slike nisu pokazali zavidne rezultate [tablica 5]. Vrijeme evaluacije čak i najmanje slike na TX1 je iznad 3 sekunde, čime implementacija u realnom vremenu ovakvom obradom nikako ne može biti ostvarena. Zanimljivo je da vrijeme obrade u slučajevima kada je prisutna

grafička jedinica (gtx-970 i TX1) traje otprilike isto za rezoluciju 256x128 i 64x32 i da je vrijeme izvođenja za sliku na rezoluciji 128x64 najbrže ako se evaluacija odvija samo na CPU.

Evaluacija jedne slike (rezolucija = (512/k)x(256/k))			
k	**CPU (i5-4200U CPU @ 1.60GHz)**	**TX1**	**gtx-970**
1	2.53 s	5.79 s	0.93 s
2	0.71 s	3.17 s	0.56 s
4	0.20 s	3.13 s	0.80 s
8	0.06 s	3.26 s	0.48 s

Tablica 5. Eksperimentalni rezultati evaluacije jedne slike

5.5. Eksperimentalni rezultati višedretvene evaluacije slika dohvaćenih iz RAM-a

Paralelnim dohvaćanjem i obradom podataka značajno su se poboljšala vremena na gtx-970 i TX1, dok je evaluacija na CPU ostala nepromjenjena. Izmjereno je potrebno vrijeme za 25 iteracija, te je naknadno podijeljeno sa brojem iteracija, čime se dobiju rezultati prikazani tablicom 6.

Queue evaluacija jedne slike (rezolucija = (512/k)x(256/k))			
k	**CPU (i5-4200U CPU @ 1.60GHz)**	**TX1**	**gtx-970**
1	2.84 s	0.63 s	0.065 s
2	0.69 s	0.30 s	0.033 s
4	0.18 s	0.25 s	0.039 s
8	0.05 s	0.23 s	0.025 s

Tablica 6. Eksperimentalni rezultati višedretvene evaluacije slika iz RAM-a

5.6. Eksperimentalni rezultati višedretvene evaluacije slika dohvaćenih iz kamere

Konačno, sličnim pristupom kao u prethodnom poglavlju dohvaćane su slike sa kamere, te su izmjerena vremena u realnoj obradi. Konačni rezultati prikazani su u tablici 7.

Queue evaluacija jedne slike (rezolucija = (512/k)x(256/k)) u realnom vremenu	
k	**TX1**
1	0.81 s
2	0.40 s
4	0.37 s
8	0.32 s

Tablica 7. Eksperimentalni rezultati višedretvene evaluacije slika iz kamere

5.7. Mjerenje latencije

Eksperimentalnim rezultatima pokazalo se da smo višedretvenim pristupom uz pomoć spremnika podataka red (eng. queue) uspjeli minimizirati utjecaj latencije. Rezultati su prikazani tablicom 8. Prikazano vrijeme u sekundama predstavlja "starost" slike koju smo upravo obradili.

slika br.	vrijeme "starosti" nakon obrade [s]
0	5.890
1	6.022
2	5.885
3	0.572
4	0.659
5	0.354
6	0.835
7	0.488
8	0.370
9	0.825
10	0.479
11	0.360
12	0.818
13	0.504
14	0.355
15	0.820
16	0.470
17	0.366
18	1.113
19	0.490

Tablica 8. Eksperimentalni rezultati mjerenja latencije

6. Zaključak

Ovim projektom u prvom dijelu prikazani su rezultati semantičke segmentacije prirodne scene u ovisnosti o skaliranju slike, rezoluciji i utjecaju haube. Zaključak prvog dijela je da je osnovni razlog lošijih rezultata učenja bilo smanjivanje rezolucije, ali i da skaliranje slikovnih oznaka i slika igra ulogu u konačnom rezultatu.

U drugom dijelu pokazalo se da pametnim pristupom u optimizaciji možemo postići značajno bolje rezultate. Uz frekvenciju 2 frame/sec mogla bi se dobavljati slika s kamere te prikazivati semantički segmentirana slika na rezoluciji 256x128 na ugradbenom računalnom sustavu TX1.

Prijedlog za poboljšanje dosadašnjih rezultata bilo bi promjeniti oznake na način da više ne budu zadane jednojediničnim kodom, nego da budu distribucija po razredima. Oznaka za jedan piksel bi bila vektor kapaciteta 19 elemenata. Na indeksu razreda u koji bi se taj piksel zaista trebao evaluirati bila bi vrijednost jedan, dok na svim ostalim elementima (razredima) bila bi nula (suma vjerojatnosti je jedan). Na izlazu bismo sada imali još jednu dimenziju veličine 19, pa bismo za svaki piksel imali distribuciju, odnosno vjerojatnost po razredima.

Također, za potencijalnu uštedu na vremenu može se razmotriti specijalizirana višedretvena evaluacija obzirom na dobivene rezultate, poznata kašnjenja i na činjenicu da je potreban red (eng. queue) veličine jedan.

LITERATURA

[1] Diederik P. Kingma, University of Amsterdam, Jimmy Lei Ba, University of Toronto *A method for stochastic optimization*, 2015. URL https://arxiv.org/pdf/1412.6980v8.pdf

[2] Karen Simonyan, Andrew Zisserman, University of Oxford, *Very Deep Convolutional Networks for Large-Scale Visual Recognition*, 2014. URL http://www.robots.ox.ac.uk/~vgg/research/very_deep/

[3] *Aravis library and documentation*. URL https://github.com/AravisProject/aravis

[4] *TensorFlow library and documentation*. URL https://www.tensorflow.org/

[5] Tingwu Wang University of Toronto, *Semantic Segmentation.* URL http://www.cs.toronto.edu/~tingwuwang/semantic_segmentation.pdf

[6] Karen Simonyan, Andrew Zisserman, University of Oxford, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015. URL https://arxiv.org/pdf/1409.1556v6.pdf

[7] Siniša Šegvić, Fakultet elektrotehnike i računarstva, *Neslužbene stranice predmeta Duboko učenje*. URL http://www.zemris.fer.hr/~ssegvic/du/

[8] *NVIDIA embedded computing. URL* https://developer.nvidia.com/embedded-computing

[9] *Cityscapes dataset. URL https://www.cityscapes-dataset.com/*

[10] Tim Dettmers, *Understanding convolution*. *URL http://timdettmers.com/2015/03/26/convolution-deep-learning/*

[11] *URL https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks*

Ugradbena izvedba konvolucijskog modela za semantičku segmentaciju

Sažetak

Ovaj rad prikazuje semantičku segmentaciju prirodnih scena učenu na modelu VGG-16. Upotrebljen je strogo nadzirani pristupi gdje u svakoj piknji skupa za učenje na raspolaganju imamo informaciju o pripadnom semantičkom razredu. Za potrebe ovog rada koristio se predtrenirani model na većoj rezoluciji slike. Korišten je programski okvir Tensorflow te biblioteke programskog jezika Python za rukovanje matricama i slikama. Prikazana je potencijalna evaluacija slika pribavljenih iz kamere u realnom vremenu. Izvedba je prilagođena radu na ugradbenom računalnom sustavu TX1.

Ključne riječi: semantička segmentacija, računalni vid, duboko učenje, vgg-16, tensorflow, tx1, aravis

Embedded implementation of a convolutional model for semantic segmentation

Abstract

This paper shows the semantic segmentation of natural scenes learned from the VGG-16 model. Strictly supervised approaches are used where we have information on the corresponding semantic class for each of the pixel of the learning set. For the purposes of this paper, a predetermined model was used at a higher resolution of the image. The Tensorflow program library and the Python programming language library for the use of matrices and images were used. A potential evaluation of images obtained from the camera in real-time is shown. The design is tailored to work on the TX1 embedded computer system on module.

Keywords: semantic segmentation, computer vision, deep learning, vgg-16, tensorflow, tx1, aravis