Algoritam za procjenu pokreta pri kodiranju videa proširen informacijama o kretanju kamere

Barbir, Marko

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 644

ALGORITAM ZA PROCJENU POKRETA PRI KODIRANJU VIDEA PROŠIREN INFORMACIJAMA O KRETANJU KAMERE

Marko Barbir

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 644

ALGORITAM ZA PROCJENU POKRETA PRI KODIRANJU VIDEA PROŠIREN INFORMACIJAMA O KRETANJU KAMERE

Marko Barbir

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 644

Pristupnik:	**Marko Barbir (0036522110)**
Studij:	Računarstvo
Profil:	Računalno inženjerstvo
Mentor:	izv. prof. dr. sc. Daniel Hofman

Zadatak: **Algoritam za procjenu pokreta pri kodiranju videa proširen informacijama o kretanju kamere**

Opis zadatka:

Algoritmi za procjenu pokreta pri kodiranju videa traže najsličnije blokove u prethodnim slikama kako bi povećali stupanj kompresije. Kod kamera za koje znamo kako će se kretati moguće je smanjiti broj pretraživanih blokova koristeći informacije o smjeru kretanja kamere. Pri korištenju dronova moguće je koristiti informacije o kretanju drona te izračunati kretanje kamere i time smanjiti broj pretraživanja. Potrebno je proučiti algoritme za procjenu pokreta pri kodiranju videa. Proučiti kako se kreće dron u ovisnosti o signalima s kontrolera kojim se upravlja kretanje drona. Pronaći postojeće implementacije video kodera te modificirati dio vezan za procjenu pokreta. Testirati rad proširenog algoritma za procjenu pokreta koji uzima u obzir informacije s kontrolera. Usporediti rad osnovnog algoritma i proširenog algoritma. Predložiti načine poboljšanja algoritma za procjenu pokreta. Napraviti dokumentaciju i objaviti programski kôd na repozitoriju Git.

Rok za predaju rada: 28. lipnja 2024.

SADRŽAJ

1. Uvod **1**

2. Procjena pokreta u kompresiji videa **2**

2.1. Odabir enkodera 3

3. Statistička analiza rezultata procjene pokreta **5**

3.1. Procjena pokreta u Kvazaar HEVC enkoderu 5

3.2. Implementacija ekstrakcije statistike procjene pokreta 8

3.2.1. Dodavanje novog ME algoritma 10

3.2.2. Dodavanje potrebnih informacija novom ME algoritmu 11

3.2.3. Osiguravanje ispravnog funkcioniranja kod višedretvenosti . . 13

3.2.4. Kreiranje CSV datoteke i oslobađanje memorije 14

3.3. Procesiranje dobivenih podataka 15

4. Implementacija modela strojnog učenja za procjenu pokreta **21**

4.1. Prijedlog modela za ujedinjeno predviđanje vektora pokreta 21

4.1.1. LSTM - modeliranje pokreta 22

4.1.2. FCNN - predviđanje vektora pokreta 24

4.1.3. Ujedinjeno treniranje 25

4.2. Inferencija modela u Kvazaar enkoderu 28

4.2.1. Spremanje modela u ONNX formatu 28

4.2.2. Instaliranje ONNX Runtime biblioteke 29

4.2.3. Postavljanje okvira za korištenje modela 31

4.2.4. Kreiranje algoritma za procjenu pokreta koji koristi modele . 35

5. Rezultati i budući rad **36**

6. Zaključak **38**

Literatura **39**

1. Uvod

Bespilotne letjelice (eng. “Unmanned Aerial Vehicle” - "UAV", kolokvijalno “drone / dron”) danas imaju razne primjene u različitim industrijama poput - poljoprivredne (kontrola stanja oranica), vojne (izviđanje ili kao oružje) ili trgovine (dostava dobara). Postoji više načina na koje se dronovi mogu upravljati, a jedan od njih je ručno daljinsko upravljanje pri kojem operater pilotira dronom koristeći izravni video prijenos s kamere na letjelici, koji pruža pogled iz prvog lica (eng. “First Person View” - “FPV”). Kako se pilot koristi tom snimkom za navigaciju, od izrazite je važnosti što manje kašnjenje od trenutka snimanja do prikaza na korisnikovom ekranu. Moderni dronovi često koriste kamere visoke 4k rezolucije sa 60 slika u sekundi, što je visoka količina podataka koju često nije moguće prenijeti u izvornom obliku te se stoga koriste razni algoritmi kompresije videa kako bi se smanjila ukupna količina podataka koje je potrebno prenijeti s letjelice do pilota. Međutim, sam proces kompresije uvodi dodatno kašnjenje u sustav zbog svoje računarne složenosti.

Ovaj rad bavi se načinima video kompresije s dovoljno niskim kašnjenjem za uspješnu primjenu kod FPV dronova, te potencijalnog iskorištavanja upravljačkih naredbi prema letjelici za efikasniju i/ili bržu kompresiju inpirirano [4], gdje je autor prikazao način skraćivanja vremena trajanja tog algoritma kompresije koristeći dodatne informacije pilotovih naredbi dronu kako bi se pojednostavio korak procjene pokreta, jedan od resursno najzahtjevnijih koraka video kompresije. Također istražuje se mogućnost upotrebe strojnog učenja za altrenativni način iskorištavanja tih informacija od načina prikazanog u [4], te implementacija nadogradnje enkodera otvorenog koda Kvazaar [15] kako bi podržao izvođenje jednog takvog modela strojnog učenja razvijenog u programskom jeziku Python u programskom okviru Pytorch [1].

2. Procjena pokreta u kompresiji videa

Algoritmi za kompresiju videa koriste dvije bitne značajke video snimaka, a to su *prostorna* i *vremenska* redundancija.

Prostorna redundancija odnosi se na sličnost susjednih piksela u slici, koju možemo iskoristi kako bi prikazali skoro identičnu sliku s manjom količinom podataka. Ovim principom funkcioniraju i algoritmi kompresije slika poput JPEG [16] algoritma.

Algoritmi za kompresiju slike se tehnički također mogu primijeniti i na video snimke, na način da se svaki okvir videa (eng. "frame") kodira posebno, neovisno o ostalim okvirima. Međutim takav pristup ne koristi temporalnu redundanciju video zapisa - sličnost svake slike snimke sa susjednim slikama u vremenskom nizu (najčešće prethodna i sljedeća slika), kako bi dodatno značajno poboljšao efikasnost kompresije. Naime, ta zalihost koristi se kako bi se izbjeglo zapisivanje svake slike posebno već se, kad je to moguće umjesto toga zapisuje razlika trenutne slike s prethodnom (što je upravo i razlog zašto u dinamičnim trenutcima kvaliteta videa opada, sa istom količinom bitova moguće je prikazati manje korisnih informacija ako se mora koristiti samo prvi postupak).

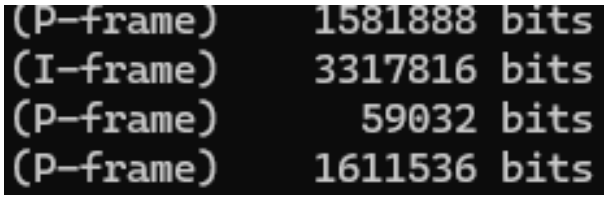
Ovisno o tome koje postupke kompresije koriste postoje 3 vrste slika u kodiranom videu:

I-frame: Potpuna slika kodirana isključivo principom prostorne redundancije, bez korištenja sličnosti s bilo kojom drugom slikom. Koriste se zbog olakšanog traženja (seek) kroz video (da ima samo jedan I-frame na početku ako bi korisnik htio preskočiti pola videozapisa dekoder bi morao procesirati sve slike do tražene pozicije), ili ako je promjena u susjednim slikama toliko velika da nije moguće iskoristiti njihovu sličnost.

P-frame: Predviđene slike, koriste sličnost s jednom ili više prethodnih referentnih slika kako bi smanjile količinu informacije potrebnu za predstavljanje videozapisa. U pravilu su P-frameovi značajno manji od I-frameova.

B-frame: Dvosmjerno predviđene slike, slično kao i P-frameovi s razlikom što re-

ferentne slike ne moraju prethoditi kodiranoj, već se B-frame može koristiti i slijedećim slikama. Za kodiranje videa s drona nisu primjenjive zbog dodanog kašnjenja kako u trenutku kodiranja slijedeće slike još ne postoje, uz minimalan benefit.



Slika 2.1: Razlika veličine P i I frameova

Postupak pronalaženja vektora pokreta (eng. motion vectors, tj. MV) između dviju slika koristi se za kodiranje predviđenih P-frameova, te se zove procjena pokreta (eng. motion estimation / ME). Svaki predviđeni vektor pokreta predstavlja pomak u pikselima određenog dijela slike relativno od referentne slike. Za dio slike koriste se blokovi piksela umjesto pojedinačnih piksela zbog povećane efikasnosti kompresije, uz veću efikasnost za veće blokove.

Procjena pokreta interesantan je dio kodiranja videa zbog toga što je računalno jedan od zahtijevnijih dijela procesa (na jednom primjeru kodiranja kratkog video isječka s drona taj dio kodiranja čini 19.15% procesorskog vremena) te zbog toga što možemo iskoristiti informacije o kretanju vozila za optimizaciju tog postupka kako je pokazano u [4].

2.1. Odabir enkodera

Od algoritama kompresije videa danas se najčešće koriste Advanced Video Codec (AVC, također poznat kao H.264 ili MPEG-4 Part 10) [13] i High Efficiency Video Coding (HEVC/H.265/MPEG-H Part 2) [14]. Noviji kodeci su Versatile Video Coding (VVC/H.266/MPEG-I Part 3/ISO/IEC 23090-3) [5] i AOMedia Video 1 (AV1) [6], čija popularnost raste zbog sve većeg broja uređaja koji imaju hardversku podršku za njega (npr. YouTube kodira videozapise koji pređu određeni broj pregleda AV1 kodekom zbog uštede na internet prometu). U ovom radu ćemo se fokusirati na implementaciju s HEVC kompresijom, kako se u [3] pokazala prikladnom za ovaj slučaj korištenja od H.264, a H.266 svoj napredak očituje u boljem stupnju kompresije pod cijenu većeg vremena potrebnog za kodiranje (čak i do 10 puta)[5] te nam zbog toga nije od interesa za ovakvu primjenu osjetljivu na kašnjenje.

Konkretni enkoder kojim će se ovaj rad baviti je Kvazaar [15], enkoder otvorenog koda dostupan na https://github.com/ultravideo/kvazaar/ pisan u C jeziku. Alternativna opcija bio bi x265 enkoder, također otvorenog koda ali napisan djelomično u C++, djelomično u asemblerskom jeziku. X265 najčešće je korišten CPU h.265 enkoder, ali je za svrhu ovog rada izabran Kvazaar zbog toga što je korišten u [4] te je lakše napraviti modifikacije kako pruža već postavljeni projekt za integrirano razvojno okruženje Visual Studio.

3. Statistička analiza rezultata procjene pokreta

Kako bi se mogli napraviti zaključci o ovisnosti rezultata algoritma procjene pokreta o ulaznim kontrolama drona ili imati podatke za potencijalno učenje modela strojnog učenja korisno je izvući te podatke u jednu datoteku za daljnju obradu. Za to postići potrebno je analizariti strukturu programskog koda Kvazaar enkodera za napraviti potrebne modifikacije.

3.1. Procjena pokreta u Kvazaar HEVC enkoderu

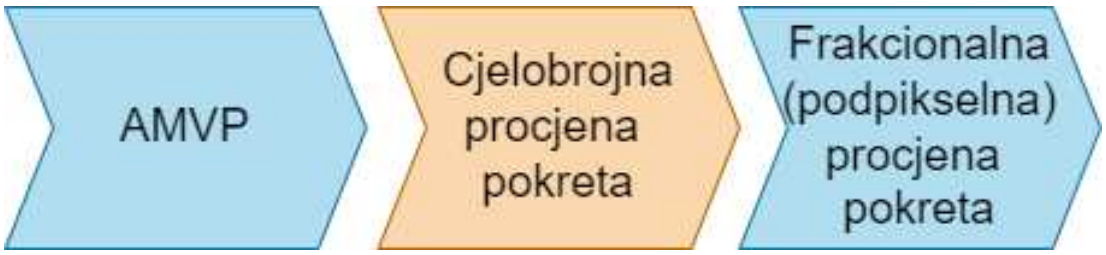
Procjena pokreta u Kvazaaru izvodi se u *search_inter.c* datoteci (inter - traženje vektora pokreta između slika za iskorištavanje vremenske zalihosti podataka, za razliku od intra - traženje unutar slike za korištenje prostorne zalihosti). Sam postupak procjene pokreta u HEVC-u sastoji se od 3 koraka:

1. Odabir najboljeg početnog vektora pokreta za pretragu od kandidata dobivenih iz već odabranih prostornih (susjedni dijelovi trenutne slike) i vremenskih (isti dio prethodnih ili slijedećh slika). Taj postupak se u HEVC naziva *AMVP* (advanced motion vector prediction) te nije fokus ovog rada, kako pristup informacijama o kretanju drona ne pomaže izravno ovom koraku.

2. Sama pretraga najboljeg vektora pokreta, koristeći početni vektor pronađen u prvom koraku kao polazište. "Bolji" vektor pokreta se smatra oni čija je ukupna "cijena" (cost) manja. Ukupna cijena suma je *cost* i *bitcost* komponente, gdje je cost SAD (sum of absolute differences) - suma razlika vrijednosti piksela trenutnog dijela slike koji se kodira i bloka referentne slike na lokaciji trenutnog bloka pomaknutog za vektor pokreta; a bitcost predstavlja količinu bitova potrebnu za kodiranu reprezentaciju trenutnog bloka ako bi se kao vektor pokreta koristio trenutni vektor kandidat. Ovaj dio nam je najzanimljiviji zbog toga što koristeći

informacije o fizičkom pokretu kamere na dranu možemo postaviti određene pretpostavke na lokaciju najboljeg vektora pokreta. Npr. ako se dron okreće oko z osi u lijevo, očekivamo da je vektor pokreta najvjerojatnije također usmjeren lijevo.

3. Daljnje preciziranje vektora pokreta pronađenog u 2. koraku, u podpikselnoj rezoluciji (rezolucija $\frac{1}{4}$ piksela). I ovaj korak također nije relevantan za ovaj rad te ostaje nepromijenjen.



Slika 3.1: 3 koraka procjene pokreta.

Konkretno funkcija `search_pu_inter_ref` poziva algoritam procjene pokreta kojeg je korisnik specifirao pri pokretanju programa (ili Test Zone search algoritam, ako nikoji nije bio specificiran) za pronalaženje vektora pokreta najmanje jedinice na kojoj se odlučuje o tipu pretrage (inter ili intra), tzv. PU (prediction unit). PU-ovi mogu biti variabilne veličine od 4x4 piksela za intra pretragu ili 8x4/4x8 za inter pretragu sve do 64x64 pikelskih blokova. Podrška većih dimenzija blokova za predikciju pokreta je jedna od prednosti HEVC standarda nad prethodnim standardima poput AVC(h.264) koji ne podržavaju veće blokove od 16x16 piksela, te su takve veće dimenzije posebno korisne kod kodiranja videa visoke rezolucije, kao što je 4K snimka s kamere drona s kojom se bavimo u ovom radu.

Od algoritama za procjenu pokreta Kvazaar podržava sljedeće:

Full Search: Potpuna pretraga oko vektora (0, 0), početnog vektora i svih vektora kandidata u kvadratu veličine stranice 8, 16, 32 ili 64 piksela, nazivi algoritma koje je potrebno postaviti u parametru `--me` pri pozivu programa su redom textitfull8, textitfull16, textitfull32, textitfull64. Ako veličina nije specificirana (naziv algoritma *full*) koristi se kvadrat 32x32.

Hexagon Based Search: Naziv algoritma *hexbs*. Iterativna pretraga boljeg kandidata vektora pokreta po uzorku šesterokuta, postavljajući najbolji pronađeni vektor kao početni vektor sljedeće iteracije, sve dok takav postoji ili broj koraka

pređe zadani maksimalni broj koraka u CLI ("Command Line Interface") parametru `--me-steps`. Nakon čega se provede jedan finalni korak pretrage po manjem šesterokutu.

Test Zone Search: Naziv algoritma *tz*. Prvo se provodi pretraga po uzorku različitih radiusa (Kvazaar koristi uzorak dijamanta radiusa potencija broja 2, od 1 do 64 uključivo) oko vektora (0,0) i početnog vektora ako je različit od (0,0). Nakon toga se najbolji pronađeni vektor pokreta dalje precizira ponavljajući prethodni korak oko njega.

Diamond Search: Naziv algoritma *dia*. Jednako kao *Hexagon Based Search*, osim toga što se umjesto uzorka šesterokuta koristi dijamant (koordinate lijevo gore, desno i dolje od trenutnog kandidata).

Osim tih algoritama već podržanih u Kvazaaru dodao sam i:

Motion Dyamics Input Search (MDIS): Detaljno opisan u [4]. Modifikacija *Diamond Search*-a koja koristi korisnikove upute kontrole drona za procjenu tipa kretanja, kako bi se suzio izbor smjerova za pretragu, rezultirajući smanjenjim vremenom pretrage za prosječno 32%.

Primjer potpisa funkcije algoritma procjene pokreta je:

```
static void tz_search(inter_search_info_t *info,
                      vector2d_t extra_mv,
                      double *best_cost,
                      double* best_bits,
                      vector2d_t *best_mv)
```

Programski kod 3.1: Potpis funkcije Test Zone Search algoritma

Od tih parametara ulazni su:

- `inter_search_info_t *info` – koji sadrži potrebne informacije za kodiranje trenutnog okvira u slici poput koordinata i dimenzija tog okvira ili pokazivača na trenutnu i referentnu sliku, te trenutno stanje enkodera uključujući stanje specifično za trenutnu sliku ali i globalno stanje/stanje konfigurabilnih postavki
- `vector2d_t extra_mv` – početni vektor pokreta iz AMVP

,a izlazni parametri:

- `double *best_cost` – SAD najboljeg pronađenog vektora
- `double* best_bits` – broj bitova potrebnih za kodiranje trenutnog bloka

– `vector2d_t *best_mv` – najbolji pronađeni MV

Svi izlazni parametri su također inicionalno popunjeni odgovarajućim vrijednostima koje vrijede za vektor dobiven AMVP-om.

3.2. Implementacija ekstrakcije statistike procjene pokreta

Sljedeći korak je definiranje podataka koje želimo zapisati za svaku instancu izvođenja algoritma procjene pokreta, u isječku koda 3.2 definiriana je struktura u kakvoj sam odlučio pohraniti te podatke. Radi se o običnoj jednostruko povezanoj listi, gdje je element liste struktura `search_inter_statistic_t`, a posebna struktura glava liste `search_inter_statistic_list_t`. U glavi liste nalaze se pokazivači na prvi i posljednji element liste, zbog toga što će slučajevi korištenja liste biti: dodavanje novog elementa (potreban pokazivač na posljednji element) i ispis cijele liste u .csv datoteku na kraju izvršenja programa (potreban pokazivač na prvi element i iterativno ispisivanje sljedećeg elementa dok se ne dođe do NULL pokazivača). U nikojem trenutku nije potreban prethodni element, te nije potrebna dvostruko povezana lista već je dovoljna jednostruka.

Podatci koje sam odlučio zapisati su:

– cur_idx – identifikator izvorišne slike koja se kodira

– ref_idx – identifikator referentne slike za trenutnu sliku, indeksirano tako da 0 predstavlja prethodnu sliku, 1 sliku prije te itd.

– origin – ishodišna točka bloka slike koji se trenutno kodira

– width – širina trenutno kodiranog bloka slike

– height – visina trenutno kodiranog bloka slike

– pitch, roll, throttle i yaw – ulazne (input) naredbe vozilu u trenutku snimanja trenutne slike

– direction – procijenjeni smjer i način kretanja u trenutku snimanja trenutne slike, procijenjen python skriptom kako je opisano u [4]

– start_mv – početni vektor pokreta, dobiven AMVP; argument extra_mv u pozivu algoritma

– best_mv – najbolji vektor pokreta kojeg je pronašao algoritam procjene pokreta

– best_cost i best_bits – cijene za najbolji pronađeni MV

Isječak koda 3.3 prikazuje API za upravljanje listom. `create_head` stvara glavu liste,
`append_next` dodaje predanoj listi novi element i vraća ga pozivatelju,
`write_statistics_file` ispisuje listu `search_inter_statistic_list_t *stat_list` u datoteku `FILE *out_file`, a `free_statistics_list` iterativno oslobađa memoriju svih elemenata liste `search_inter_statistic_list_t *stat_list`.

```
typedef struct
   search_inter_statistic_t
    {
  int32_t cur_idx;
  int32_t ref_idx;

  point_t origin;
  int32_t width;
  int32_t height;

  double pitch; // RSV
  double roll; // RSH
  double throttle; // LSV
  double yaw; // LSH
  int direction;

  point_t start_mv;
  point_t best_mv;
  double best_cost;
  double best_bits;

  struct
     search_inter_statistic_t
      *next_stat;
} search_inter_statistic_t
   ;

typedef struct {
  search_inter_statistic_t
      *first_stat;
  search_inter_statistic_t
      *last_stat;
}
   search_inter_statistic_list_t
   ;
```

Programski kod 3.2: Struktura podataka izvučenih iz rezultata procjene pokreta

```
search_inter_statistic_list_t* create_head();

search_inter_statistic_t* append_next(
   search_inter_statistic_list_t *stat_list);

int write_statistics_file(const
   search_inter_statistic_list_t *stat_list, FILE *
   out_file);

int free_statistics_list(search_inter_statistic_list_t
   *stat_list);
```

Programski kod 3.3: API za upravljanje strukturom podataka statustike procjene pokreta

Za implementirati prikupljanje ovih podataka potrebno je postići par stvari:

1. Dodati novi ME algoritam koji će dodati novi zapis u listu statistike, izvršiti procjenu pokreta te upisati potrebne podatke u napravljeni zapis

2. Osigurati dostupnost liste u koju spremamo podatke, te podataka o kretanju/input kontrolama novododanom algoritmu. Ostali podatci su već dostupni ili kao rezultat samog postupka, ili iz već zadanih parametara poziva funkcije (3.4).

3. Osigurati uspješno funkcioniranje liste, ako je više dretvi pokuša koristiti istovremeno. Ovo tehnički nije nužno, ali značajno skraćuje vrijeme prikupljanja podataka ako možemo koristiti višedretene sposobnosti enkodera.

4. Na kraju izvršenja programa ispisati statističke podatke u CSV formatu, te osloboditi sve korištene strukture podataka.

```
cur_stat->cur_idx = info->state->frame->num;
cur_stat->ref_idx = info->ref_idx;
cur_stat->width = info->width;
cur_stat->height = info->height;

cur_stat->origin.x = info->origin.x;
cur_stat->origin.y = info->origin.y;
```

Programski kod 3.4: Zapisivanje već dostupnih podataka iz ulaznih parametara funkcije

3.2.1. Dodavanje novog ME algoritma

Kako bi se postigla prva stavka potrebno je napraviti par koraka. Prvo sam dodao novu funkciju `static void search_mv_full_with_logs( inter_search_info_t* info, vector2d_t extra_mv, double * best_cost, double* best_bits, vector2d_t* best_mv)` za taj algoritam u *search_inter.c* datoteku. Ta funkcija dodaje novi zapis na kraj liste pozivanjem `append_next`, delegira procjenu pokreta već implementiranom algoritmu *Test Zone Search* te upisuje sve potrebne podatke u kreirani zapis.
Zatim, kako `search_pu_inter_ref` poziva odgovarajući algoritam uspoređujući vrijednost u konfiguraciji s enumeracijom `enum kvz_ime_algorithm`, potrebno je dodati novu vrijednost u enumeraciju za slučaj novog algoritma i ako je konfigurirana vrijednost jednaka tome pozvati odgovarajuću vrijednost.

```
enum kvz_ime_algorithm {
//   ...
    KVZ_IME_LOG = 9,
};
```

Programski kod 3.5: Nova vrijednost u enumeraciji koja odgovara ME algoritmu koji zapisuje podatke

```
switch (cfg->ime_algorithm) {
  // ... ostali algoritmi
    case KVZ_IME_LOG:
      search_mv_full_with_logs(info, search_range,
        best_mv, &best_cost, &best_bits, &best_mv);
      break;
  // ...
}
```

Programski kod 3.6: Pozivanje nove funkcije u `search_pu_inter_ref`

Također je potrebno omogućiti konfiguriranje odabira novog algoritma, varijabla koja predstavlja odabrani algoritam nalazi se u varijabli `enum kvz_ime_algorithm ime_algorithm;` `kvz_config` strukture. Ta varijabla postavlja se u *cfg.c* datoteci gdje se procesiraju CLI argumenti. Kako bi se uspješno postavila vrijednost enumeracije, potrebno je dodati odabrano ime novododanog algoritma u listu imena algoritama `static const char * const me_names[]` koja se nalazi u toj datoteci, naziv je potrebno dodati na način da indeks pozicije u listi odgovara vrijednosti algoritma u `enum kvz_ime_algorithm` kao što je napravljeno u 3.8.

```
{ "hexbs", "tz", "full", "full8", "full16", "full32", "
  full64", "dia", "uis", NULL };
```

Programski kod 3.7: Lista imena algoritama prije promjene

```
{ "hexbs", "tz", "full", "full8", "full16", "full32", "
  full64", "dia", "uis", "log", NULL};
```

Programski kod 3.8: Lista imena algoritama poslije promjene

S ovim modifikacijama, ako se prilikom pozivanja programa navede argument `--me` s vrijednosti *log* koristit će se algoritam s praćenjem statistike.

3.2.2. Dodavanje potrebnih informacija novom ME algoritmu

Podatci koji nedostaju algoritmu su:

- Podatci o korisničkim kontrolama i procijenjenjenom smjeru kretanja u trenutku snimanja trenutne slike
    - Ovi podatci nalaze se u posebnim tekstualnim datotekama koje imaju za svaku sliku u videozapisu spremljenu po jedan redak s odgovarajućom vrijednosti. Stoga je potrebno dodati CLI argumente koji specificiraju imena tih datoteka (3.9) te pročitati sljedeću vrijednost iz tih datoteka prije početka procesiranja svake slike (3.11). Sve nove CLI argu-

mente potrebno je registrirati u listu opcija `static const struct option long_options[]` u *cli.c* datoteci (3.10).

– Pokazivač na listu statistike. Njega sam odlučio inicializirati u *cfg.c* pri parsiranju CLI argumenata u trenutku kada se odabere "*log*" algoritam cjelobrojne procjene pokreta. U tom slučaju poziva se funkcija `create_head` iz *search_inter_statistics.h* te se kreirana lista sprema u konfiguraciju `struct kvz_config` (3.9) koja se u Kvazaaru automatski kopira u stanje ekodera svake slike pa je dostupna našem algoritmu (3.12).

```
else if OPT("me") {
  // ...
  if (cfg->ime_algorithm == KVZ_IME_LOG) {
    cfg->inter_stat_list = create_head();
  }
}
else if OPT("pitch") {
  cfg->pitch_file = fopen(value, "r");
}
else if OPT("roll") {
  cfg->roll_file = fopen(value, "r");
}
// ... isto za ostale datoteke
```

Programski kod 3.9: Kreiranje liste i otvaranje ulaznih datoteka s informacijama o kretanju u konfiguraciju (datoteka *cfg.c*)

```
static const struct option long_options[] = {
  // ...
  { "uis-dir",            required_argument, NULL, 0 },
  { "pitch",            required_argument, NULL, 0 },
  { "roll",           required_argument, NULL, 0 },
  { "throttle",           required_argument, NULL, 0
    },
  { "yaw",          required_argument, NULL, 0 },
  { "stat-out",           required_argument, NULL, 0
    },
  {0, 0, 0, 0}
};
```

Programski kod 3.10: Registrirani novi argumenti u *cli.c*

```
    if (state->encoder_control->cfg.uis_dir_file) {
      state->frame->uis_dir = fgetc(state->
         encoder_control->cfg.uis_dir_file);
      if (state->frame->uis_dir == EOF) {
        state->frame->uis_dir = '0';
      }
```

```
    }
    if (state->encoder_control->cfg.ime_algorithm ==
       KVZ_IME_LOG) {
      if (!fscanf(state->encoder_control->cfg.
         pitch_file, "%lf", &state->frame->pitch)) {
        state->frame->pitch = 0.0;
      }
      if (!fscanf(state->encoder_control->cfg.roll_file
         , "%lf", &state->frame->roll)) {
        state->frame->roll = 0.0;
      }
      // ...
    }
```

Programski kod 3.11: Čitanje i pripremanje u stanje podataka o kretanju, `kvazaar_encode` metoda *kvazaar.c* datoteke.

```
  search_inter_statistic_t* cur_stat = append_next(info
     ->state->encoder_control->cfg.inter_stat_list);
```

Programski kod 3.12: Pristupanje listi statistike iz algoritma.

3.2.3. Osiguravanje ispravnog funkcioniranja kod višedretvenosti

Prilikom izvođenja algoritma svaka instanca algoritma zapisuje podatke u samo jedan, svoj element liste. Jedini slučaj u kojem može doći do grešaka je ako dvije dretve istovremeno dodavaju novi element na kraj liste, tada se mogu pomiješati pokazivači na prvi/posljednji/sljedeći element. Za spriječiti taj slučaj dovoljno je zaštiti proces dodavanja novog elementa korištenjem `Windows.h` API-a za upravljanje kritičnim odsječcima.

```
search_inter_statistic_list_t* create_head()
{
  InitializeCriticalSection(&cs);

  // ...
}
```

Programski kod 3.13: Postavljanje kritičnog odsječka pri kreiranju liste.

```
search_inter_statistic_t* append_next(
   search_inter_statistic_list_t* stat_list)
{
  EnterCriticalSection(&cs);

  // ... dodaj novi element
```

```
  LeaveCriticalSection(&cs);
  return next;
}
```

Programski kod 3.14: Zaštita dodavanja novog elementa kritičnim odsječkom.

```
int free_statistics_list(search_inter_statistic_list_t
   *stat_list)
{
  DeleteCriticalSection(&cs);

  // ...
}
```

Programski kod 3.15: Brisanje kritičnog odsječka na kraju izvršenja programa

3.2.4. Kreiranje CSV datoteke i oslobađanje memorije

U `search_inter_statistics.c` implementirane su metode `write_statistics_file(const search_inter_statistic_list_t *stat_list, FILE *out_file);` koja ispisuje sadržaj liste u CSV formatu u `out_file` te `free_statistics_list(search_inter_statistic_list_t *stat_list)` koja oslobođuje svu memoriju zauzetu listom. Njih je potrebno pozvati na kraju izvođenja programa, jedno od mogućih mjesta implementiranja toga je `kvz_config_destroy` metoda u *cfg.c* datoteci gdje se brišu svi resursi zauzeti u Kvazaar konfiguracijskoj strukturi (`kvz_config`).

```
int kvz_config_destroy(kvz_config *cfg)
{
    // ...
    if (cfg->ime_algorithm == KVZ_IME_LOG) {
      write_statistics_file(cfg->inter_stat_list, cfg->
         stat_out_file);
      free_statistics_list(cfg->inter_stat_list);
    }
    if (cfg->pitch_file) fclose(cfg->pitch_file);
    if (cfg->roll_file) fclose(cfg->roll_file);
    if (cfg->throttle_file) fclose(cfg->throttle_file);
    if (cfg->yaw_file) fclose(cfg->yaw_file);
    if (cfg->stat_out_file) fclose(cfg->stat_out_file);
    if (cfg->uis_dir_file) fclose(cfg->uis_dir_file);
  }
  // ...
}
```

Programski kod 3.16: Ispisivanje i oslobađanje statističke liste, te zatvaranje svih uvedenih novih datoteka.

3.3. Procesiranje dobivenih podataka

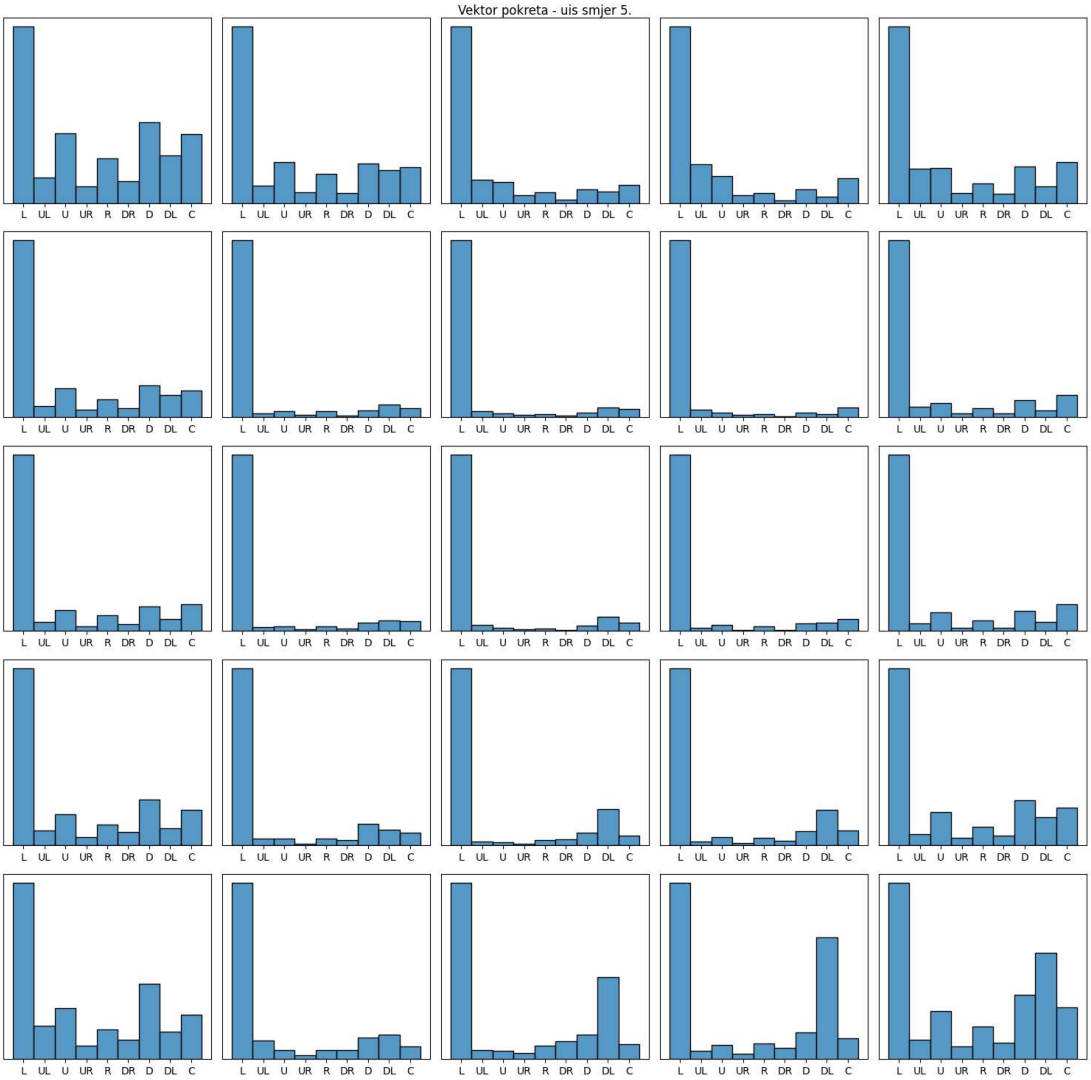
S modifiranim Kvazaar enkoderom možemo prikupiti podatke o vektorima pokreta pozivajući Kvazaar enkoder sa sljedećim CLI argumentima:

```
-i drone_video.yuv --input-res 3840x2160 -o out.hevc --
   input-fps 60 --ref 1 --me log --me-early-termination
    off --pitch Pitch.txt --roll Roll.txt --throttle
   Throttle.txt --yaw Yaw.txt --stat-out
   search_inter_statistics.csv --uis-dir
   FinalDirections.txt
```

Programski kod 3.17: CLI argumenti poziva Kvazaar enkodera za prikupljanje statističkih podataka.

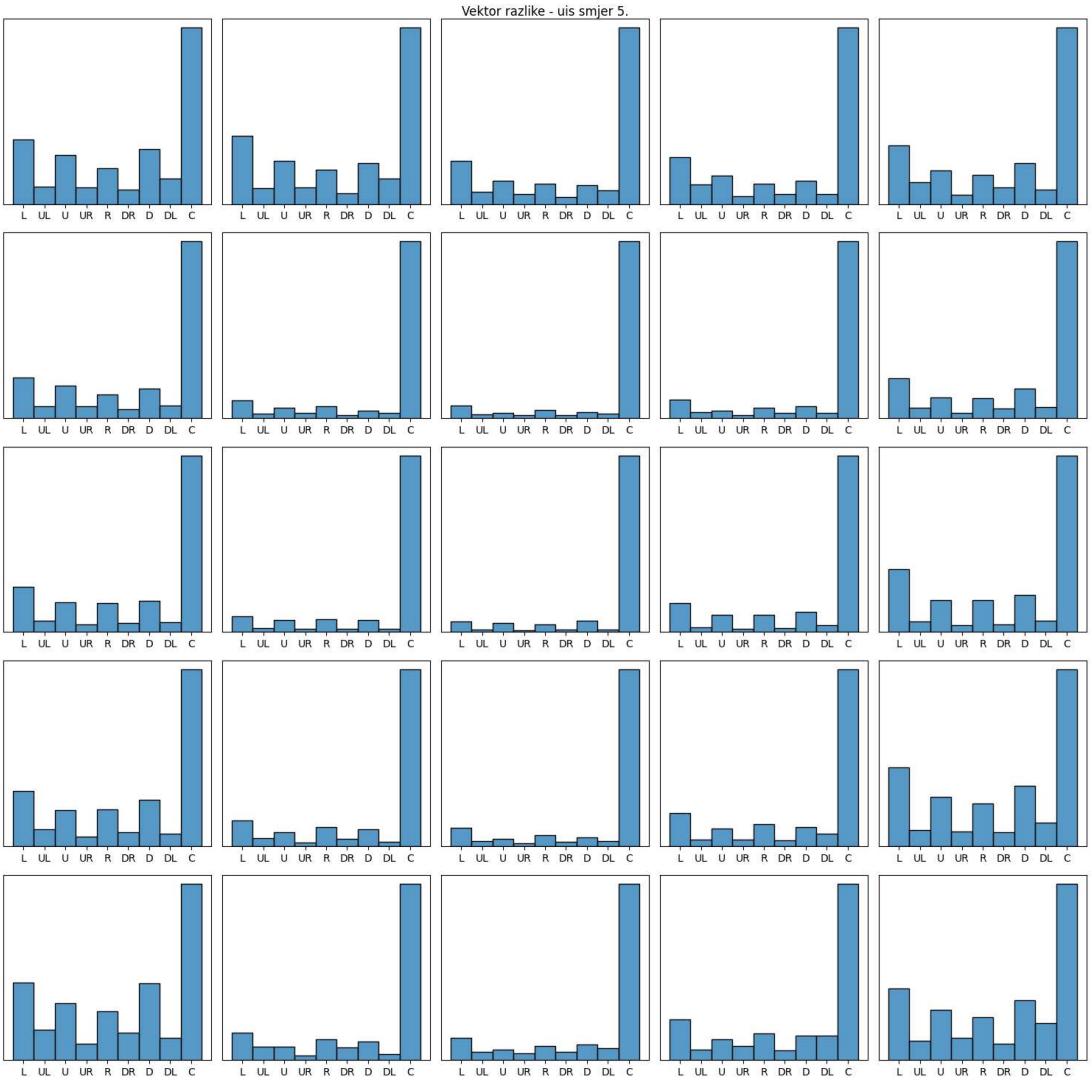
Gdje je `-i` ime datoteke ulaznog videozapisa, `--input-res` rezolucija u pikselima ulaznog videozapisa, `-o` ime izlazne kodirane datoteke, `--input-fps` broj slika u sekundi ulaznog videozapisa, `--ref` broj referentnih slika koje može koristiti jedan P-frame (koristi se vrijednost 1, kako je u [2] pokazano da veće vrijednosti utječu značajno na trajanje kodiranja s minimalnim benefitom), `--me` ime odabranog algoritma, `--pitch`, `--roll`, `--throttle`, `--yaw` i `--uis-dir` imena ulaznih datoteka s informacijama o kretanju vozila, te `--stat-out` ime izlazne CSV datoteke.
Tako dobivena datoteka učitava se za daljnju obradu u Python Jupyter [10] bilježnicu pomoću Pandas[12] biblioteke. Međutim, kako rezultantna CSV datoteka ima 120 milijuna redaka (prosjek 40000 redaka po slici videozapisa), problem je učitati ju kao Pandas okvir podataka (eng. datafatrame) u memoriju na računalu s 16GB radne memorije. Stoga sam odlučio uzeti nasumično uzorkovan podskup veličine 10% originalnog skupa podataka.
Ako koristeći koordinate najboljeg vektora dobijemo kut tog vektora iz izhodišta, te odredimo smjer kretanja pomoću tog kuta (između -22.5°i 22.5°desno, između 22.5°i 67.5°gore desno, između 67.5°i 112.5°gore, itd.), možemo grafički prikazati smjerove MV-a za različite načine tj. smjerove kretanja (stupac '*direction*").
Na slici 3.2 prikazana je ovisnost smjera vektora pokreta o lokaciji bloka piksela unutar slike. Slika i graf su oboje podijeljeni na 25 sekcija, te je u grafu na svakoj takvoj sekciji prikazan broj vektora pokreta koji imaju određeni smjer, ali samo uzimajući u obzir vektore pokreta čiji blokovi piksela pripadaju toj sekciji slike. Graf se odnosi

na kretanje u lijevo (“direction”=5) te se očituje velika predvidivost smjera vektora pokreta za taj tip kretanja, kao što je očekivano prema [4]. No, također ako pogledamo graf 3.3 koji prikazuje istu stvar kao prethodni, ali umjesto za najbolji vektor pokazuje smjer vektora razlike početnog i najboljeg vektora. Na tom grafu velika većina vektora razlike imaju smjer “centra”, što znači da su je razlika jednaka (0,0), tj. da je početni vektor dobiven AMVP-om ujedno bio i najbolji pronađeni. To pokazuje koliko je zapravo učinkovit AMVP, kad već ima vektore kandidate. Sličnu uspješnost vidimo i u grafu 3.4 koji prikazuje situaciju za cijeli skup podataka, neovisno o tipu kretanja.

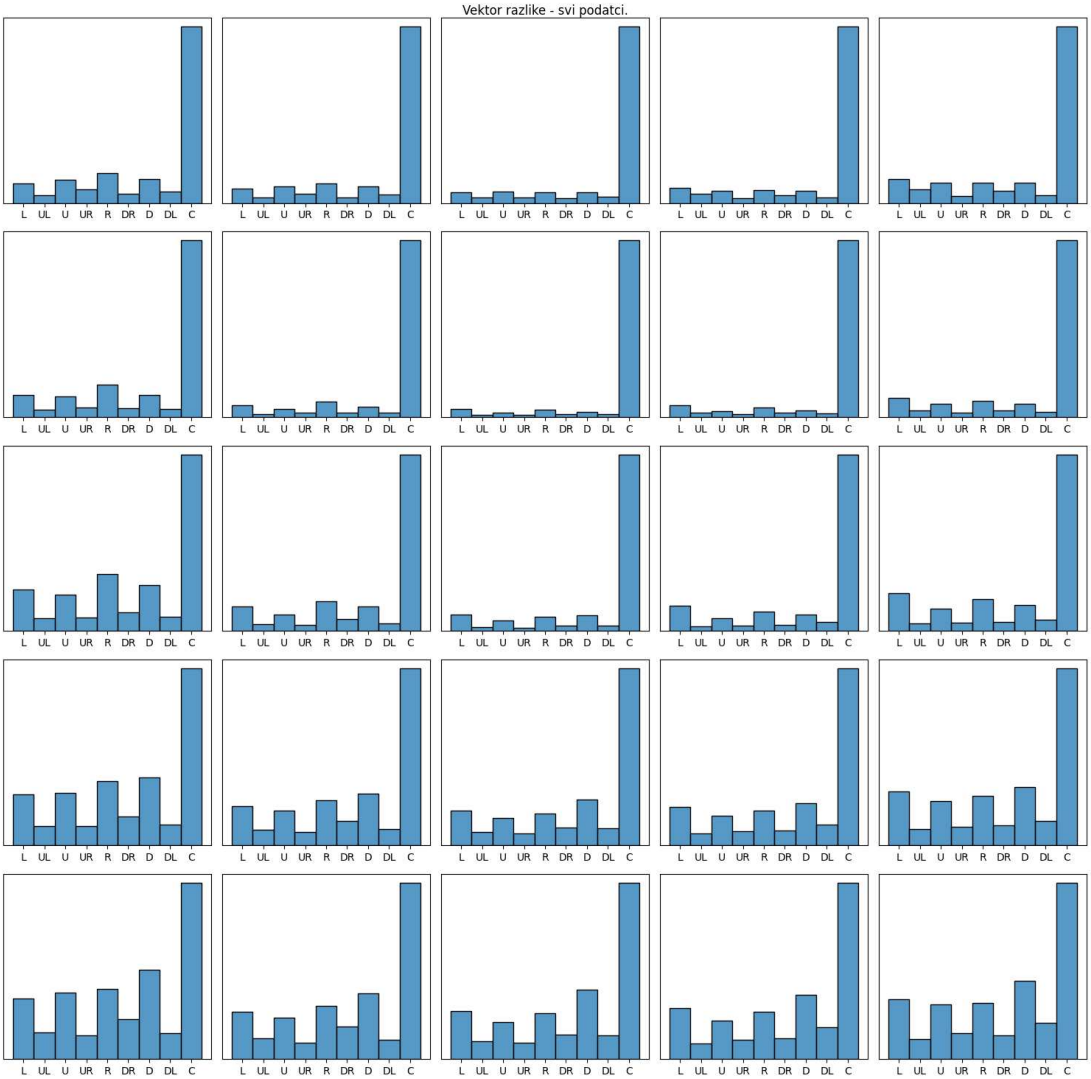


Slika 3.2: Graf ovisnosti smjera vektora pokreta o lokaciji bloka piksela unutar slike za sve slike gdje se dron kretao oko z osi u lijevo (smjer 5).

```
df['diff_mv_x'] = df['best_mv_x'] - df['start_mv_x']
df['diff_mv_y'] = df['best_mv_y'] - df['start_mv_y']

df['angle'] = np.degrees(np.arctan2(df['best_mv_y'], df
    ['best_mv_x']))
df['angle_diff'] = np.degrees(np.arctan2(df['diff_mv_y'
    ], df['diff_mv_x']))

bins = [-180, -157.5, -112.5, -67.5, -22.5, 22.5, 67.5,
     112.5, 157.5, 180]
labels = ['L', 'DL', 'D', 'DR', 'R', 'UR', 'U', 'UL', '
    L']

df['mv_direction'] = pd.cut(df['angle'], bins=bins,
    labels=labels, right=True, ordered=False)
```



Slika 3.3: Graf ovisnosti smjera vektora razlike najboljeg i početnog MV o lokaciji bloka piksela unutar slike za sve slike gdje se dron kretao oko z osi u lijevo (smjer 5).



Slika 3.4: Graf ovisnosti smjera vektora razlike najboljeg i početnog MV o lokaciji bloka piksela unutar slike za sve slike.

```
df.mv_direction = df.mv_direction.cat.add_categories(['
    C']).cat.reorder_categories(['L', 'UL', 'U', 'UR', '
    R', 'DR', 'D', 'DL', 'C'])
df.loc[(df.best_mv_x==0) & (df.best_mv_y==0), '
    mv_direction'] = 'C'

df['diff_mv_direction'] = pd.cut(df['angle_diff'], bins
    =bins, labels=labels, right=True, ordered=False)
df.diff_mv_direction = df.diff_mv_direction.cat.
    add_categories(['C']).cat.reorder_categories(['L', '
    UL', 'U', 'UR', 'R', 'DR', 'D', 'DL', 'C'])
df.loc[(df.diff_mv_x==0) & (df.diff_mv_y==0), '
    diff_mv_direction'] = 'C'
```

Programski kod 3.18: Postupak računanja smjera najboljeg vektora pokreta i vektora razlike.

```
grid_size = 5

cell_width = WIDTH / grid_size
cell_height = HEIGHT / grid_size

col_indices = (df.origin_x // cell_width).astype(int)
row_indices = (df.origin_y // cell_height).astype(int)

df['quadrant'] = (row_indices * grid_size + col_indices
    + 1)
```

Programski kod 3.19: Postupak računanja pripadanja jednoj od 25 sekcija ekrana.

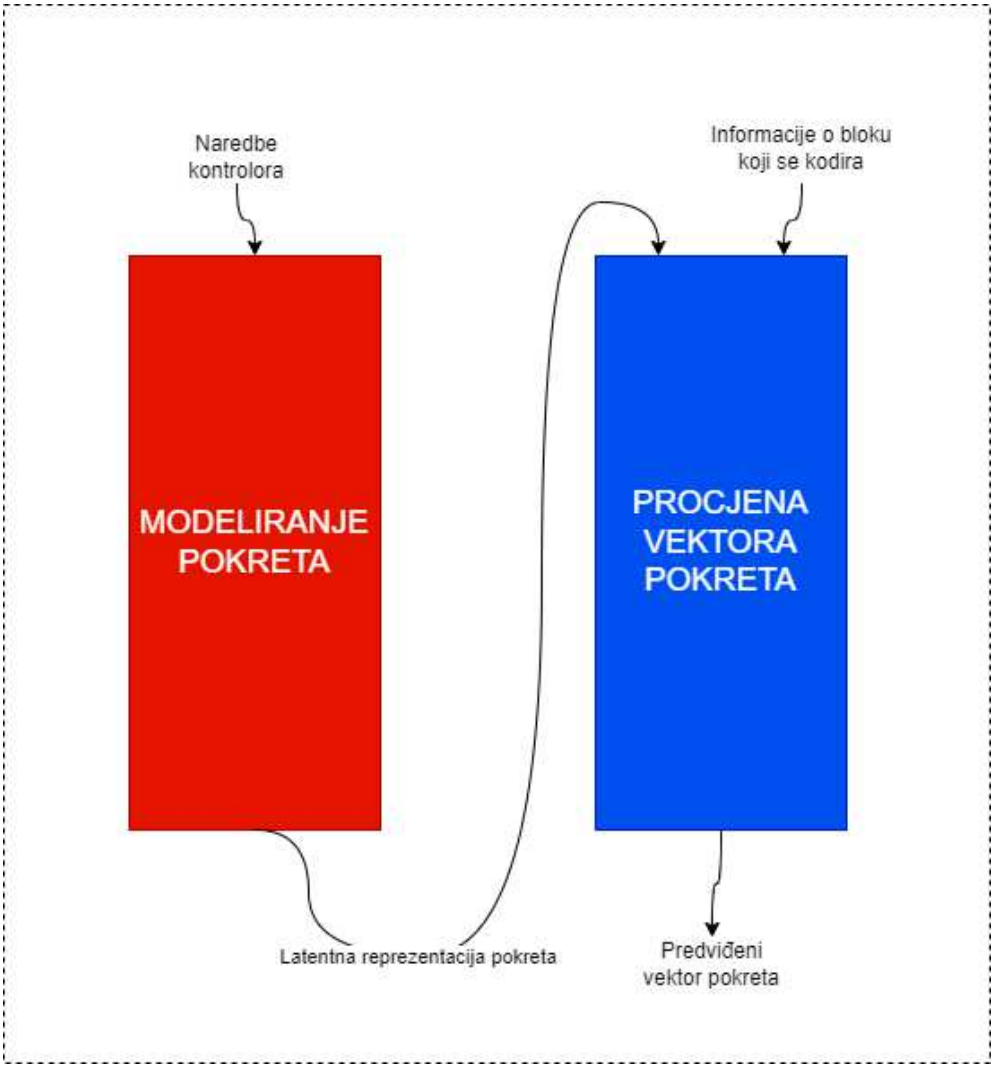
4. Implementacija modela strojnog učenja za procjenu pokreta

CSV skup podataka o procjeni pokreta možemo iskoristiti i za razvoj/treniranje modela strojno učenja za procjenu pokreta. Ima više pristupa na koji bi se mogao implementirati takav model, kako se proces estimacije pokreta u HEVC sastoji od 3 koraka: AMVP, cjelobrojna procjena pokreta i frakcionalna(podpikselska) procjena pokreta, možemo namijeniti model da zamijeni bilo koji korak ili više koraka odjednom.
Kako skup podataka sadrži informacije o cjelobrojnoj procjeni pokreta, u ovom radu se fokusiram na taj dio postupka, iako je opcija promijeniti kolekciju podataka da se najbolji vektor sprema poslije frakcionalne, umjesto cjelobrojne procjene pokreta. Prednost toga pristupa je također što omogućuje korištenje kompliciranijeg modela, kako frakcionalna procjena pokreta traje čak i dvostruko duže od cjelobrojne, te je u tom slučaju ne bi morali izvršavati.

4.1. Prijedlog modela za ujedinjeno predviđanje vektora pokreta

Kad smo se odlučili za kreiranje modela za cjelobrojnu procjenu pokreta, pitanje je na koji način iskoristiti informacije o kretanju vozila. Nastaviti koristiti jednostavnu detekciju smjera kretanja kao u [4], gdje se pokret klasificira u jedan od 9 kategorija je sigurno dobra opcija, kako te kategorije imaju veliku korelaciju sa smjerom MV-a. Međutim, ako bismo mogli implicitno modelirati kompleksnije kretanje vozila, takav pristup bi trebao moći predvidjeti i količinu pokreta te bi bio fleksibilniji na razne načine kretanja.
Stoga predlažem ujedinjeni model koji se koristi odvojeno – dio za modeliranje pokreta poziva se jednom za svaku novu sliku te modelira kretanje drona s novim uputama s upravljača, a dio modela za predviđanje samog MV poziva se više puta u svakoj slici,

jednom za svaki PU, a trenira ujedinjeno – propagacijom gradijenta gubitka iz modela za predviđanje vektora pokreta sve do ulaza mreže za modeliranje pokreta.



Slika 4.1: Predloženi model.

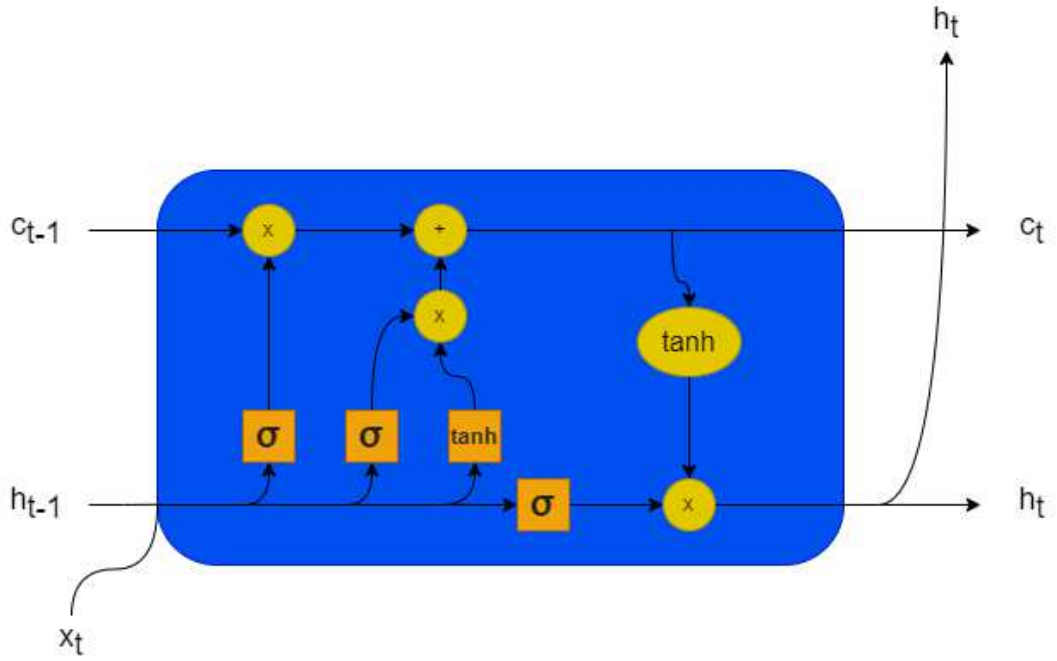
4.1.1. LSTM - modeliranje pokreta

Rekurentne neuronske mreže (eng. Recurrent Neural Network - RNN) vrsta su neuronskih mreža koje modeliraju sekvence na način da uzimaju na ulaz jedan po jedan element sekvence, s tim da imaju i ulaz "skrivenog stanja" u kojeg proslijeđuju izlaz prošlog elementa sekvence (0 za prvi element).

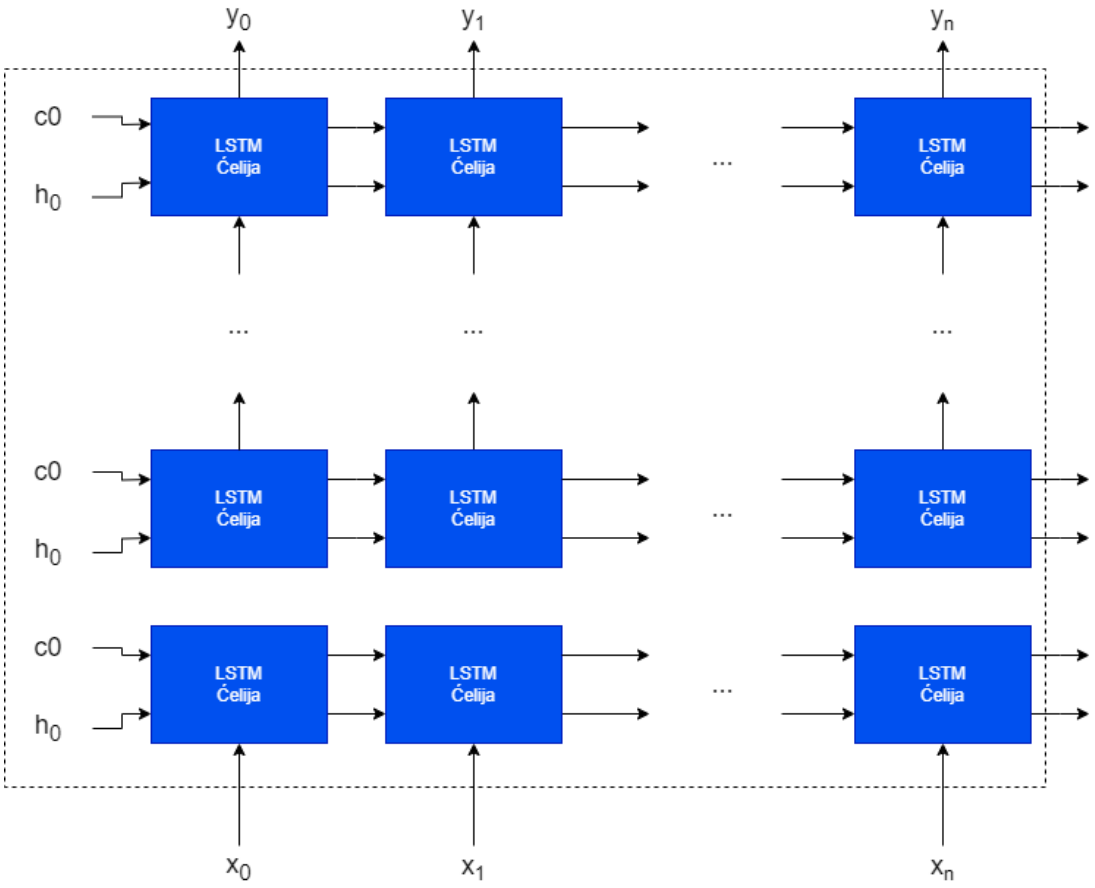
Obični RNN-novi često u praksi imaju problema s učenjem zbog problema nestajućih gradijenata, te iako bi u teoriji trebali biti sposobni modeliranjem ovisnosti velikih udaljenosti u praksi se to nije dešavalo. Long short-term memory (LSTM) [11] verzija je

ovih modela uvedena da popravi te probleme, te se u praksi pokazao veoma uspješnim modelom.

Zbog sekvencijske prirode modeliranja pokreta (primamo sekvencu ulaznih kontrola s kojima procjenjujemo pokret), za taj problem odabrao sam koristiti LSTM model. Kao ulaz primamo vektor uputa vozilu (pitch, roll, throttle i yaw – u kodu `x1`) te skriveno stanje (na početku inicijalizirano na 0, kasnije se uzima izlaz prethodnog poziva – u kodu `(h_0, c_0)`), izlaz modela je to skriveno stanje koje se proslijeđuje (`(h_1, c_1)`) i "procijenjeno kretanje" drona – zapravo latentna varijabla koja služi za ulaz u model predviđanja vektora pokreta (`y1`).



Slika 4.2: Jedna LSTM ćelija.



Slika 4.3: Korištenje višeslojnog LSTM modela na sekvenci.

```
lstm_input = 4
lstm_hidden = 12
lstm_layers = 2

lstm = nn.LSTM(lstm_input, lstm_hidden, lstm_layers).to
   (device)
```

Programski kod 4.1: Kreiranje modela estimacije pokreta,

4.1.2. FCNN - predviđanje vektora pokreta

Kako se ovaj dio modela izvršava često, čak 40000 puta za svaku sliku, bitno je da nije prekompleksan. Stoga sam odabrao jednostavnu potpuno povezano neuronsku mrežu s jednim ulaznim, jednim skrivenim i jednim izlaznim slojem.

Kao ulaz prima izlaz LSTM-a `y1` uz svoje dodatne značajke (`x2`), specifične za PU koji predviđa. To su koordinate i dimenzije PU-a, te početni vektor pokreta dobiven AMVP-om.

Izlaz modela je predviđeni rezultat cjelobrojne procjene pokreta, vektor pokreta, u realnom obliku tako da se treba zaokružiti na najbliži cijeli broj kad se koristi u enkoderu.

```
class MotionFCNN(nn.Module):
    def __init__(self, input_size, hidden_size,
       output_size):
        super(MotionFCNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

lstm_hidden = 12
fcnn_feature_len = 6
fcnn_hidden = 64
fcnn_output = 2

fcnn = MotionFCNN(lstm_hidden + fcnn_feature_len,
   fcnn_hidden, fcnn_output).to(device)
```

Programski kod 4.2: Definicija i kreiranje modela za predviđanje MV.

4.1.3. Ujedinjeno treniranje

Skup podataka za potrebe učenja podijeljen je na 3 dijela: LSTM značajke, FCNN značajke i istinite vrijednosti najboljih vektora pokreta (definirano u programskom odsječku 4.3)

```
lstm_feature_columns = ['pitch', 'roll', 'throttle', '
   yaw']
fcnn_feature_columns = ['origin_x', 'origin_y', 'width'
   , 'height', 'start_mv_x', 'start_mv_y']
expected_output_columns = ['best_mv_x', 'best_mv_y']

class FrameStatisticsDataset(Dataset):
  // ...

class InterSearchStatisticsDataset(Dataset):

  // ...

  def __getitem__(self, idx):
    // ...
    if self.part == 'lstm':
      try:
```

```
        return torch.as_tensor(self.df[self.df.cur_idx
           == self.min_frame + idx][
           lstm_feature_columns].iloc[0].values).to(
           self.device)
      except IndexError:
        return None
    if self.part == 'fcnn':
      return FrameStatisticsDataset(self.df[self.df.
         cur_idx == self.min_frame + idx][
         fcnn_feature_columns], self.device)
    if self.part == 'target':
      return torch.as_tensor(self.df[self.df.cur_idx ==
          self.min_frame + idx][expected_output_columns
         ].values).to(self.device)
    // ...
```

Programski kod 4.3: Definiranje učitavanja podataka za učenje

Petlja treniranja modela prikazana je u 4.4. Kako je za LSTM bitna sekvenca, ne možemo provesti treniranje u minigrupama po slikama u videozapisu, ali je to moguće napraviti za svaku posebnu predikciju unutar slike, tako da se isti output LSTM-a za trenutnu sliku konkatenira s `batch_size` značajki za FCNN, provuče kroz mrežu te izračuna gubitak (korištena je srednja kvadratna greška). Nakon toga metodom `backward()` gubitka računa se gradijent ovisno o parametrima mreža te pomoću optimizatora (korišten Adam optimizator[8]) ažuriraju parametri oba modela.

```
criterion = nn.MSELoss().to(device)
optimizer = torch.optim.Adam(list(lstm.parameters()) +
   list(fcnn.parameters()), lr=0.001)

lstm.train()
fcnn.train()

for epoch in range(epochs):
  h_0 = torch.zeros(lstm_layers, 1, lstm_hidden).to(
     device)
  c_0 = torch.zeros(lstm_layers, 1, lstm_hidden).to(
     device)

  for frame in range(len(train_lstm)):
    x1, x2s, ys = train_lstm[frame], DataLoader(
       train_fcnn[frame], batch_size=batch_size),
       train_target[frame]
    for x2, y in zip(x2s, ys):
      optimizer.zero_grad()
      cur_batch_size = x2.shape[0]

      y = y.expand(cur_batch_size, -1)
```

```
        motion_param, (h_1, c_1) = lstm(x1.expand(1, 1,
            -1), (h_0, c_0))
        motion_param = torch.squeeze(motion_param, 0).
            expand(cur_batch_size, -1)
        fcnn_input = torch.cat((motion_param, x2), 1)

        fcnn_output = fcnn(fcnn_input)
        loss = criterion(fcnn_output, y)

        loss.backward()
        optimizer.step()
    h_0, c_0 = h_1.detach(), c_0.detach()
```

Programski kod 4.4: Zajedničko učenje oba modela.

Skaliranje značajki

Zbog numeričke stabilnosti prilikom treniranja modela, idealno je da sve značajke su u sličnim mjerilima. Često se to postiže skaliranjem svih značajki tako da se postigne raspodjela s srednjom vrijednošću 0, a varijancom 1. Takvo skaliranje je moguće u ovakvom korištenju modela, gdje se on trenira i izvršava u odvojenim okolinama (treinra se u Pytorch razvojnom okviru u Pythonu, a izvršava se u C enkoderu Kvazaar), međutim da bi se iskoristili pravilni parametri skaliranja, kako bi se oni promijenili pri svakoj promjeni skupa podataka za treniranje bilo bi potrebno pri svakoj takvoj promjeni ažurirati i C izvorni kod s novim parametrima. Zbog te komplikacije odlučio sam umjesto toga skalirati na konstantan način kako je prikazano u sljedećem programskom isječku:

```
WIDTH, HEIGHT = 3840, 2160
LCU = 64

df['origin_x'] /= WIDTH
df['origin_y'] /= HEIGHT
df['width'] /= LCU
df['height'] /= LCU
df['start_mv_x'] /= LCU
df['start_mv_y'] /= LCU
df['best_mv_x'] /= LCU
df['best_mv_y'] /= LCU
```

Programski kod 4.5: Skaliranje skupa podataka.

Pozicija bloka koji se kodira podijeljena je s brojem piksela u slici na toj osi, tako da je skalirana na raspon od 0 do 1, te zapravo predstavlja postotak cijelog ekrana.

Ostali podaci podijeljeni su s veličinom najvećeg mogućeg bloka piksela u enkoderu `LCU` koja iznosi 64 piksela. Za širinu i visinu bloka to znači skaliranje između 0 (zapravo 0.0625) i 1, ali vektori pokreta nisu ograničeni veličinom bloka, te i nakon skaliranja mogu biti veći od 1.

4.2. Inferencija modela u Kvazaar enkoderu

Kada imamo model spreman u Pythonu, potreban je način kako iskoristiti taj model u Kvazaar encoderu. Pytorch nema C API za svoje modele, ali postoji otvoreni standard za reprezentaciju različitih neuronskih mreža u jednome formatu – ONNX [9] (Open Neural Network Exchange), kojeg podržava veliki broj razvojnih okruženja (Pytorch, Tensorflow, scikit-learn, ...).

4.2.1. Spremanje modela u ONNX formatu

Spremanje naših modela možemo napraviti jednostavno, koristeći funkciju `torch.onn.export` (4.6 i 4.7) (također postoji noviji način prevođenja modela u Pytorchu, funkcijom `torch.onnx.dynamo_export`, međutim imao sam problema s greškama pri pokušaju spremanja LSTM modela te ju nisam uspio iskoristiti).
Modele je potrebno spremiti odvojeno, jer će se tako i koristiti, svaki posebno, a ne poput jednog ujedinjenog modela, kao što je bio slučaj pri treniranju.

```
lstm.eval()

x1 = torch.randn(1, 1, lstm_input).to(device)

h0 = torch.zeros(lstm_layers, 1, lstm_hidden).to(device
   )
c0 = torch.zeros(lstm_layers, 1, lstm_hidden).to(device
   )

inputs = (x1, (h0, c0))

lstm_onnx = torch.onnx.export(
    lstm,
    inputs,
    "lstm.onnx",
    verbose=True,
    input_names=["x1", "h0", "c0"],
    output_names=["y1", "h1", "c1"]
)
```

Programski kod 4.6: Spremanje LSTM modela u ONNX formatu.

```
fcnn.eval()

x2 = torch.randn(1, lstm_hidden + fcnn_feature_len)

fcnn_onnx = torch.onnx.export(
    fcnn,
    x2,
    "fcnn.onnx",
    verbose=True,
    input_names=["y1_x2"],
    output_names=["y"]
)
```

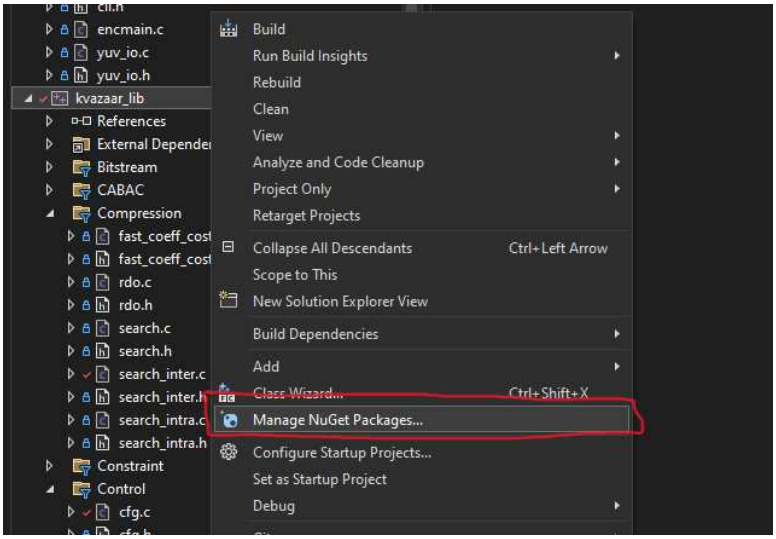
Programski kod 4.7: Spremanje FCNN modela u ONNX formatu.

4.2.2. Instaliranje ONNX Runtime biblioteke

Da bi iskoristili modele koje smo spremili u ONNX formatu, potrebno je iskoristiti neko od okruženja za njihovo izvršavanje (lista okruženja može se vidjeti na poveznici `https://onnx.ai/supported-tools.html#deployModel`) koje podržava programski jezik C. Jedno od takvih je ONNX Runtime [7], on osim API-a za C jezik podržava i razne druge jezike (Python, C++, C#, Java, JavaScript, Objective C i Windows Runtime).

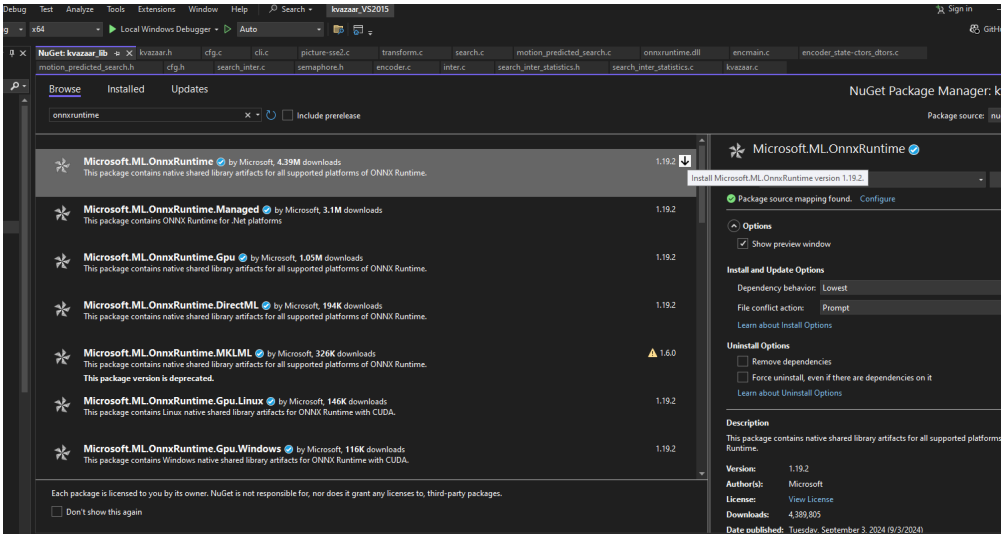
Najjednostavniji način za dodati ONNX Runtime u naš projekt je korištenjem Nuget package managera.

U Visual Studiu potrebno je napraviti desni klik na modul "`kvazaar_lib`" te odabrati opciju “Manage NuGet Packages...”.



Slika 4.4: Otvaranje NuGet package managera.

Zatim je potrebno pronaći i instalirati paket “Microsoft.ML.OnnxRuntime”.



Slika 4.5: Instalacija ONNX Runtime modula.

Kako je ONNX Runtime često korištena biblioteka uobičajeno već instalirana na računalu, te u mom slučaju Visual Studio nije ispravno kopirao `onnxruntime.dll` datoteku u izvršni direktorij preporučujem to napraviti ručno. U suprotnom se može učitati kriva verzija biblioteke te baciti grešku nekompatibilne verzije API-a.

4.2.3. Postavljanje okvira za korištenje modela

Svu implementaciju korištenja ONNX modela odlučio sam napraviti u posebnoj datoteci *motion_predicted_search.c*, tako da ostatak koda ne mora brinuti o implementacijskim detaljima, već samo može koristiti API prikazan u 4.8.

```
int init_models();

int lstm_step(float pitch, float roll, float throttle,
   float yaw, float **y1);

int fcnn_step(float* y1, int origin_x, int origin_y,
   int width, int height, int start_mv_x, int
   start_mv_y, int* mv_x, int* mv_y);

int free_models();
```

Programski kod 4.8: API za korištenje definiranih modela u Kvazaaru.

Funkcija `init_models` učitava oba modela te inicijalizira podatkovne strukture potrebne za njihovo korištenje. Prvo se dohvati ONNX runtime API, pa se pomoću njega kreira okruženje (`env`). Pomoću njih kreiraju se 2 sesije za korištenje svakod od modela funckijom `CreateSession` koja među ostalim prima i naziv datoteke u koju je model spremljen. Također kreiraju se tenzori sa stanje LSTM-a koji se prenose iz koraka u korak te se ovdje inicijaliziraju na početnu vrijednost 0 kao globalne varijable. I tenzor za ulaz u FCNN ("y1_x2") se ovdje kreira te se alocira memorija za njega, da se može koristiti pri svakom korištenju FCNN-a bez ponovne alokacije. Ova inicijalizacija poziva se u `kvazaar_open` funkciji *kvazaar.c* datoteke.

```
g_ort = OrtGetApiBase()->GetApi(ORT_API_VERSION);
g_ort->CreateEnv(ORT_LOGGING_LEVEL_WARNING, "test", &
   env);

g_ort->CreateSessionOptions(&session_options_lstm);
ORT_ABORT_ON_ERROR(g_ort->CreateSession(env, L"lstm.
   onnx", session_options_lstm, &session_lstm));


g_ort->CreateSessionOptions(&session_options_fcnn);
g_ort->CreateSession(env, L"fcnn.onnx",
   session_options_fcnn, &session_fcnn);

g_ort->CreateCpuMemoryInfo(OrtArenaAllocator,
   OrtMemTypeDefault, &memory_info);
```

Programski kod 4.9: Dohvaćanje ONNX runtime API-a i postavljanje okoline i sesija modela.

```
h0_data = calloc(LSTM_LAYERS * LSTM_HIDDEN, sizeof(
    float));
int64_t h0_dims[] = { LSTM_LAYERS, 1, LSTM_HIDDEN };
size_t h0_size_in_bytes = LSTM_LAYERS * 1 * LSTM_HIDDEN
    * sizeof(float);
g_ort->CreateTensorWithDataAsOrtValue(memory_info,
    h0_data, h0_size_in_bytes, h0_dims, 3,
    ONNX_TENSOR_ELEMENT_DATA_TYPE_FLOAT, &h0_tensor);

c0_data = calloc(LSTM_LAYERS * LSTM_HIDDEN, sizeof(
    float));
int64_t c0_dims[] = { LSTM_LAYERS, 1, LSTM_HIDDEN };
size_t c0_size_in_bytes = LSTM_LAYERS * 1 * LSTM_HIDDEN
    * sizeof(float);
g_ort->CreateTensorWithDataAsOrtValue(memory_info,
    c0_data, c0_size_in_bytes, c0_dims, 3,
    ONNX_TENSOR_ELEMENT_DATA_TYPE_FLOAT, &c0_tensor);

size_t y1_x2_size_in_bytes = (LSTM_HIDDEN +
    FCNN_FEATURE_LEN) * sizeof(float);
y1_x2_data = malloc(y1_x2_size_in_bytes);
int64_t y1_x2_dims[] = { 1, LSTM_HIDDEN +
    FCNN_FEATURE_LEN };
g_ort->CreateTensorWithDataAsOrtValue(memory_info,
    y1_x2_data, y1_x2_size_in_bytes, y1_x2_dims, 2,
    ONNX_TENSOR_ELEMENT_DATA_TYPE_FLOAT, &y1_x2_tensor);
```

Programski kod 4.10: Definicija i inicijalizaicija tenzora s njihovim podacima, koji će se stalno ponovno koristiti tijekom izvedbe programa.

`lstm_step` provodi jedan korak LSTM modela, spremajući njegov izlaz u varijablu stanja trenutne slike, nazivom `y1` (`state->frame->y1`). To se dešava jedanput za svaku sliku u `kvazaar_encode` funkciji u fazi pripreme stanja prije pokretanja `kvz_encode_one_frame(state, frame)` za kodiranje sljedeće slike.

Prvo se kreira ulazni tenzor `x1_tensor` pomoću ulaznih podatak primljenih kao argumente funkcije, koji se zajedno s `h0_tensor` i `h0_tensor` šalje kao ulaz u `g_ort->Run` te se time dobivaju 3 izlazna tenzora koji predstavljaju izlaz *y1* i nova stanja *h1* i *c1*. Zatim se pristupaju podaci iz tih tenzora koristeći `GetTensorMutableData`, kako bi se podatci iz *y1* kopirali u izlazni argument, te podatci iz *h1* i *c1* kopirali nazad u *h0* i *c0* za sljedeću iteraciju.

```
float x1_data[1 * BATCH_SIZE_LSTM * LSTM_INPUT] = {
    pitch, roll, throttle, yaw };
OrtValue* x1_tensor;
int64_t x1_dims[] = { 1, BATCH_SIZE_LSTM, LSTM_INPUT };
g_ort->CreateTensorWithDataAsOrtValue(memory_info,
```

```
    x1_data, sizeof(x1_data), x1_dims, 3,
    ONNX_TENSOR_ELEMENT_DATA_TYPE_FLOAT, &x1_tensor);

const char* input_names[] = { "x1", "h0", "c0" };
const char* output_names[] = { "y1", "h1", "c1" };

OrtValue* output_tensors[3] = { NULL, NULL, NULL };

g_ort->Run(session_lstm, NULL, input_names, (const
    OrtValue * []) { x1_tensor, h0_tensor, c0_tensor },
    3, output_names, 3, output_tensors);
```

Programski kod 4.11: Pripremanje ulaza za LSTM, te pokretanje izvođenja modela.

```
float* y1_output_data = NULL;
g_ort->GetTensorMutableData(output_tensors[0], (void**)
    &y1_output_data);
*y1 = malloc(sizeof(float) * LSTM_HIDDEN);
memcpy(*y1, y1_output_data, sizeof(float) * LSTM_HIDDEN
    );

float* h1_output_data = NULL;
g_ort->GetTensorMutableData(output_tensors[1], (void**)
    &h1_output_data);
memcpy(h0_data, h1_output_data, sizeof(float) *
    LSTM_HIDDEN_TENSOR_SIZE);

float* c1_output_data = NULL;
g_ort->GetTensorMutableData(output_tensors[2], (void**)
    &c1_output_data);
memcpy(c0_data, c1_output_data, sizeof(float)*
    LSTM_HIDDEN_TENSOR_SIZE);
```

Programski kod 4.12: Korištenje izlaznih podataka modela: kopiranje y1 u izlazni argument, a h1 i c1 u h0 i c0.

```
g_ort->ReleaseValue(x1_tensor);
g_ort->ReleaseValue(output_tensors[0]);
g_ort->ReleaseValue(output_tensors[1]);
g_ort->ReleaseValue(output_tensors[2]);
```

Programski kod 4.13: Oslobađanje korištenih tenzora kreiranih u `lstm_step`.

`fcnn_step` koristi izlaz LSTM-a za trenutnu sliku i informacije o bloku koji se trenutno kodira kako bi predvidio vektor pokreta za trenutni blok. To predstavlja korak cjelobrojne procjene pokreta te se poziva u novokreiranoj funkciji algoritma za cjelobrojnu procjenu `nn_predicted_search_mv` u *search_inter.c*.

Implementacija koja koristi ONXX runtime API strukturirana je slično kao `lstm_step`, samo nakon izvršavanja modela izlaz se inverzno skalira tako da se

pomnoži najvećim mogućim dimenzijama bloka (LCU, 64 piksela) te se zaokruži na najbližu cjelobrojnu vrijednost prije nego što se kopira u izlazne argumente funckije `int *mv_x` i `int *mv_y`. Kako ONNX sesije modela nisu sigurne za paralelno korištenja funkcija je također označena kao kritični odsječak kako bi se osiguralo sigurno izvođenje (moguće je paralelizirati izvođenje na razne načine poput npr. pool od N različitih sesija ili organizirano pokretanje modela sa većim grupama podataka).

```
float x2[FCNN_FEATURE_LEN] = { origin_x / (float) WIDTH
   , origin_y / (float) HEIGHT, width / (float) LCU,
   height / (float) LCU, start_mv_x / (float) LCU,
   start_mv_y / (float)LCU };

memcpy(y1_x2_data, y1, LSTM_HIDDEN * sizeof(float));
memcpy(y1_x2_data + LSTM_HIDDEN, x2, FCNN_FEATURE_LEN *
    sizeof(float));

const char* input_names[] = { "y1_x2" };

const char* output_names[] = { "y" };
OrtValue* output_tensor = NULL;

g_ort->Run(session_fcnn, NULL, input_names, (const
   OrtValue* const*)&y1_x2_tensor, 1, output_names, 1,
   &output_tensor);
```

Programski kod 4.14: Pripremanje ulaza za FCNN, te pokretanje izvođenja modela.

```
float* output_data = NULL;
g_ort->GetTensorMutableData(output_tensor, (void**)&
   output_data);

*mv_x = (int) rintf(output_data[0] * LCU);
*mv_y = (int) rintf(output_data[1] * LCU);
```

Programski kod 4.15: Skaliranje, zaokruživanje i spremanje dobivenog izlaza modela.

Funkcija `free_models` oslobađa sve strukture podataka korištene u prethodnim funkcijama, te se poziva pri izlasku iz programa u `kvazaar_close` funkciji.

```
DeleteCriticalSection(&cs);

g_ort->ReleaseValue(h0_tensor);
free(h0_data);
g_ort->ReleaseValue(c0_tensor);
free(c0_data);
g_ort->ReleaseValue(y1_x2_tensor);
free(y1_x2_data);

g_ort->ReleaseMemoryInfo(memory_info);
```

```
g_ort->ReleaseSessionOptions(session_options_lstm);
g_ort->ReleaseSession(session_lstm);
g_ort->ReleaseSessionOptions(session_options_fcnn);
g_ort->ReleaseSession(session_fcnn);

g_ort->ReleaseEnv(env);
```

Programski kod 4.16: Oslobađanje svih korištenih struktura.

4.2.4. Kreiranje algoritma za procjenu pokreta koji koristi modele

Sa implementiranim korištenjem modela, možemo dodati novi algoritam procjene pokreta koji će iskorisiti taj API. Novi algoritam dodaje se na jednaki način kao i u 3.2.1, sa novom funkcijom algoritma:

```
static void nn_predicted_search_mv(inter_search_info_t*
    info,
  vector2d_t extra_mv,
  double* best_cost,
  double* best_bits,
  vector2d_t* best_mv) {
  vector2d_t mv_cand = { 0,0 };

  fcnn_step(info->state->frame->y1, info->origin.x,
     info->origin.y, info->width, info->height,
     extra_mv.x, extra_mv.y, &mv_cand.x, &mv_cand.y);

  check_mv_cost(info, mv_cand.x, mv_cand.y, best_cost,
     best_bits, best_mv);
}
}
```

Programski kod 4.17: Funckcija novog algoritma cjelobrojne procjene pokreta koja predviđa vektor pokreta pomoću

Sve što taj algoritam radi je izvođenje modela potpuno povezane neuronske mreže za predviđanje vektora pokreta, te provjera dobivenog vektora pokreta funkcijom `check_mv_cost`.

5. Rezultati i budući rad

Prilikom izvedbe novog algoritma za procjenu pokreta koji koristi metode strojnog učenja mjereno je vrijeme izvođenja pojedinih dijelova koda pomoću `Windows.h` API-a:

```
LARGE_INTEGER frequency;
LARGE_INTEGER start_t, end_t;
double elapsedMicroseconds;

QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&start_t);

...

QueryPerformanceCounter(&end_t);

elapsedMicroseconds = (double)(end_t.QuadPart - start_t
   .QuadPart) * 1000000.0 / frequency.QuadPart;

printf("Elapsed time: %.3f microseconds\n",
   elapsedMicroseconds);
```

Programski kod 5.1: Provjera trajanja izvođenja odsječka koda.

Zabilježeno je trajanje LSTM koraka oko 400μs, a FCNN koraka 10-40μs, dok je trajanje izvođenja klasičnog `dia_search` algoritma ispod 10μs. Čak i sa odabranim najjednostavnijim verzijama modela, njihovo izvođenje traje duže od već postojećeg algoritma procjene pokreta. Razlog tome su uspješnost AMVP-a, zbog koje sam algoritam cjelobrojne procjene pokreta mora napraviti mali broj iteracija, te hardverske optimizacije za računanje SAD-a u obliku SSE2 instrukcija. Izvođenje modela strojnog učenja moglo bi se poboljšati ako se model koristi u minigrupama umjesto izvođenja svake instance procjene pokreta posebno, međutim za tu modifikaciju potrebna je značajna promjena Kvazaar enkodera, ali je svakako vrijedno istraživanja. Također ovakav pristup naučene procjene pokreta može se koristiti kako bi zamijenio i frakcionalnu (a ne samo cjelobrojnu) procjenu pokreta te uštedio dodatno vrijeme izvođenja.

Još jedan problem je što ovako definirani modeli nisu uspjeli pravilno naučiti predviditi vektore pokreta zbog svoje jednostavnosti. Te bi u budućnosti trebalo istražiti kompleksnije opcije, te je korištenje konvolucijskih neuronskih mreža (CNN) u dijelu modela za predviđanje MV sigurno jedna od prvih opcija.

6. Zaključak

Kvazaar enkoder pokazao se laganim za modificiranje, te je uspješno implementirano izvođenje modela strojnog učenja razvijenog u PyTorch okruženju te spremljeno u ONNX formatu, u koraku cjelobrojne procjene pokreta koristeći ONNX Runtime API. Predloženi model strojnog učenja za ujedinjeno praćenje kretanja letjelice koristeći informacije od pilota i procjenu pokreta (predviđanje vektora pokreta) nažalost nije uspješno skratio vrijeme izvođenja kodiranja videa, međutim postoje razne opcije za daljnji rad na modelu poput modifikacije Kvazaar enkodera za omogućavanje grupiranja izvođenja modela te istraživanja korištenje kompleksnijih mreža za predviđanje s fokusom na korištenje konvolucijskih neuronskih mreža.

LITERATURA

[1] Jason Ansel et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. U *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Travanj 2024. doi: 10.1145/3620665.3640366. URL https://pytorch.org/assets/pytorch2-2.pdf.

[2] Jakov Benjak i Daniel Hofman. Exploring the influence of motion estimation algorithm selection and its parameters on the quality of hevc-encoded 4k drone footage. U *2023 8th International Conference on Smart and Sustainable Technologies (SpliTech)*, stranice 1–6. IEEE, 2023.

[3] Jakov Benjak, Daniel Hofman, Josip Knezović, i Martin Žagar. Performance comparison of h. 264 and h. 265 encoders in a 4k fpv drone piloting system. *Applied Sciences*, 12(13):6386, 2022.

[4] Jakov Benjak, Daniel Hofman, i Hrvoje Mlinarić. Efficient motion estimation for remotely controlled vehicles: A novel algorithm leveraging user interaction. *Applied Sciences*, 14(16):7294, 2024.

[5] Benjamin Bross, Ye-Kui Wang, Yan Ye, Shan Liu, Jianle Chen, Gary J Sullivan, i Jens-Rainer Ohm. Overview of the versatile video coding (vvc) standard and its applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(10):3736–3764, 2021.

[6] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, et al. An overview of core coding tools in the av1 video codec. U *2018 picture coding symposium (PCS)*, stranice 41–45. IEEE, 2018.

[7] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2021. Version: 1.19.2.

[8] P Kingma Diederik. Adam: A method for stochastic optimization. *(No Title)*, 2014.

[9] Linux Foundation. Onnx. https://onnx.ai/, 2017. Version: 1.16.2.

[10] Brian E Granger i Fernando Pérez. Jupyter: Thinking and storytelling with code and data. *Computing in Science & Engineering*, 23(2):7–14, 2021.

[11] S Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

[12] The pandas development team. pandas-dev/pandas: Pandas, Veljača 2020. URL https://doi.org/10.5281/zenodo.3509134.

[13] Iain E Richardson. *The H. 264 advanced video compression standard*. John Wiley & Sons, 2011.

[14] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, i Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[15] Marko Viitanen, Ari Koivula, Ari Lemmetti, Arttu Ylä-Outinen, Jarno Vanne, i Timo D. Hämäläinen. Kvazaar: Open-source hevc/h.265 encoder. U *Proceedings of the 24th ACM International Conference on Multimedia*, 2016. ISBN 978-1-4503-3603-1. URL http://doi.acm.org/10.1145/2964284.2973796.

[16] Gregory K Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

Algoritam za procjenu pokreta pri kodiranju videa proširen informacijama o kretanju kamere

Sažetak

Bespilotne letjelice s pogledom iz prvog lica (eng. First Person View, FPV) korisne su u raznim primjenama. Velika prepreka kod FPV letjelica je kvaliteta i brzina kašnjenja videa s letjelice do pilota kako bi bolje mogao s njom upravljati. Ovaj rad bavi se iskorištavanjem informacije upravljačih naredbi pilota kako bi se skratilo vrijeme trajanja jednog od zahtjevnijih djelova algoritma HEVC kompresije videa - procjene pokreta, rezultirajući manjim kašnjenjem. Istraženo je razvijanje modela strojnog učenja koje koristi takve informacije za predviđanje vektora pokreta slike s referentnom, te je prikazano njegovo izvođenje unutar već postojećeg HEVC enkodera otvorenog koda.

Ključne riječi: bespilotna letjelica, pogled iz prvog lica, HEVC, video kompresija, procjena pokreta, PyTorch, ONNX, ONNX Runtime

Video encoding motion estimation algorithm augmented with camera motion information

Abstract

Unmanned aerial vehicles with first-person view (FPV) are useful in various applications. A significant challenge with FPV drones is the quality and latency of the video from the vehicle to the pilot to facilitate better control. This work focuses on utilizing the information from the pilot's control commands to reduce the duration of one of the more demanding parts of the HEVC video compression algorithm - motion estimation, resulting in lower latency. Development of a machine learning model which utilizes such information to predict motion vectors from one frame in reference to another is explored, as well as its inference in an existing open source HEVC encoder.

Keywords: unmanned aerial vehicle (UAV), first-person view (FPV), HEVC, video compression, motion estimation, PyTorch, ONNX, ONNX Runtime