Razvoj algoritma autonomne vožnje zasnovanog na procjeni kuta zakreta upravljača vozila i testiranje performansi algoritma u različitim simulatorima

Benja, Borna

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

Sveučilišni studij računarstva

RAZVOJ ALGORITMA AUTONOMNE VOŽNJE ZASNOVANOG NA PROCJENI KUTA ZAKRETA UPRAVLJAČA VOZILA I TESTIRANJE PERFORMANSI ALGORITMA U RAZLIČITIM SIMULATORIMA

Diplomski rad

Borna Benja

Osijek, 2022.

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

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 08.09.2022.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Borna Benja
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1103R, 13.10.2020.
OIB studenta:	15324730086
Mentor:	Izv. prof. dr. sc. Mario Vranješ
Sumentor:	,
Sumentor iz tvrtke:	David Mijić
Predsjednik Povjerenstva:	Izv.prof.dr.sc. Ratko Grbić
Član Povjerenstva 1:	Izv. prof. dr. sc. Mario Vranješ
Član Povjerenstva 2:	Prof. dr. sc. Marijan Herceg
Naslov diplomskog rada:	Razvoj algoritma autonomne vožnje zasnovanog na procjeni kuta zakreta upravljača vozila i testiranje performansi algoritma u različitim simulatorima
Znanstvena grana diplomskog rada:	**Umjetna inteligencija (zn. polje računarstvo)**
Zadatak diplomskog rada:	U ovom radu, potrebno je razviti vlastiti algoritam autonomne vožnje zasnovan na procjeni kuta zakreta upravljača vozila (engl. steering angle prediction). Zatim, potrebno je odabrati dva različita simulatora za razvoj autonomne vožnje, prikupiti trening podatke iz tih simulatora i istrenirati unaprijed osmišljeni algoritam autonomne vožnje zasnovan na procjeni kuta zakreta upravljača vozila nad prikupljenim podacima iz oba simulatora. Oba modela potrebno je testirati u oba simulatora i potrebno je međusobno usporediti rezultate. Nakon toga, dva novonastala trening skupa potrebno je objediniti u jedan i ponovno istrenirati algoritam i testirati ga u oba simulatora. Glavni cilj ovog rada je razviti novi algoritam autonomne vožnje zasnovan na procjeni kuta
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	08.09.2022.

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

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 16.09.2022.

Ime i prezime studenta:	Borna Benja
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1103R, 13.10.2020.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Razvoj algoritma autonomne vožnje zasnovanog na procjeni kuta zakreta upravljača vozila i testiranje performansi algoritma u različitim simulatorima**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Mario Vranješ

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

Sadržaj

1. UVOD

Svakodnevno se događa velik broj prometnih nesreća koje rezultiraju brojnim žrtvama. Jedan od najčešćih uzroka prometnih nesreća je nesmotrenost vozača [1]. Nepažnja pri upravljanju motornim vozilom može dovesti do nenamjernog napuštanja prometne trake, što je bio uzrok 51% prometnih nesreća sa smrtno stradalima u Sjedinjenim Američkim Državama između 2016. i 2018. godine [2]. U svrhu smanjenja broja prometnih nesreća i povećanja sigurnosti u prometu, automobilska industrija svakodnevno razvija nove sustave za pomoć vozaču (engl. *Advanced Driver Assistance Systems - ADAS*) i autonomnu vožnju. Jedan od takvih sustava za pomoć vozaču je sustav potpore zadržavanja vozila u prometnoj traci (engl. *Lane Keeping Assist Systems - LKAS*), kojemu je cilj održavati kut zakreta upravljača vozila takvim da se vozilo zadrži unutar odgovarajuće prometne trake.

Za uspješnu procjenu kuta zakreta upravljača vozila (engl. *steering angle prediction*) potrebno je prepoznati i razumjeti okolinu vozila, što uključuje cestu, znakove i oznake na cesti, ostale sudionike u prometu i moguće opasnosti na cesti. Veliki problem predstavlja raznolikost uvjeta prilikom vožnje: različite oznake prometnih traka kao što su isprekidane, pune i dvostruke kolničke crte koje mogu biti bijele ili žute, različiti vremenski uvjeti i doba dana, uvjeti na cesti i drugi čimbenici koji čine procjenu kuta zakreta upravljača vozila složenim i teškim zadatkom. Za prikupljanje informacija o okolini, u vozila se ugrađuju senzori poput kamera, lidara, sonara, globalnog sustava za pozicioniranje (engl. *Global Positioning System - GPS*) i sl. Dobivene informacije tada se obrađuju različitim tehnikama i algoritmima koji kao rezultat daju procijenjeni kut zakreta upravljača vozila. Algoritmi za autonomnu vožnju zasnovani na procjeni kuta zakreta upravljača vozila mogu se podijeliti u dvije kategorije: algoritme zasnovane na računalnom vidu (engl. *Computer Vision - CV*) i algoritme zasnovane na neuronskim mrežama [3]. Algoritmi zasnovani na računalnom vidu dijele zadatak procjene kuta zakreta upravljača vozila na manje zadatke koji se rješavaju pojedinačno, a kombinacijom rješenja dobiva se željeni konačni rezultat. Algoritmi zasnovani na neuronskim mrežama predstavljaju tzv. *end-to-end* pristup, što znači da je cijeli proces procjene kuta zakreta upravljača vozila sadržan unutar neuronske mreže bez potrebe za više algoritama koji obavljaju različite zadatke.

U ovom radu predložen je novi vlastiti *end-to-end* algoritam autonomne vožnje zasnovan na procjeni kuta zakreta upravljača vozila. Predloženi algoritam procjenjuje kut zakreta

upravljača vozila pomoću duboke neuronske mreže na osnovu slike dobivene s prednje kamere vozila. Predloženi algoritam treba uspješno procijeniti kut zakreta upravljača vozila neovisno o vremenskim uvjetima i obilježjima ceste. Predloženi algoritam izrađen je u programskom jeziku Python korištenjem TensorFlow i Keras biblioteka za strojno učenje, a testiran je u simulatorima za autonomnu vožnju CARLA (engl. *Car Learning to Act*) [4] i MetaDrive [5].

U drugom poglavlju rada objašnjen je problem procjene kuta zakreta upravljača vozila i dan je pregled postojećih rješenja za autonomnu vožnju zasnovanih na procjeni kuta zakreta upravljača vozila. U trećem poglavlju opisan je rad predloženog algoritma, proces stvaranja vlastite baze podatka korištenjem odabranih simulatora i proces treniranja vlastite neuronske mreže na prikupljenim podacima. Vlastita baza podataka sastoji se od parova slika s prednje kamere vozila i odgovarajućih vrijednosti zakreta upravljača dobivenih iz oba simulatora. U četvrtom poglavlju opisano je testno okruženje, dani su rezultati provedene evaluacije predloženog algoritma za autonomnu vožnju na podacima iz vlastite baze podataka te evaluacija u virtualnom okruženju odabranih simulatora. Dobiveni rezultati uspoređeni su s rezultatima postojećih algoritama za autonomnu vožnju zasnovanih na procjeni kuta zakreta upravljača vozila, dan je kritički osvrt na dobivene rezultate te su dane smjernice za budući rad i moguća unaprjeđenja predloženog algoritma. U petom poglavlju nalazi se zaključak rada.

2. PREGLED POSTOJEĆIH RJEŠENJA ZA AUTONOMNU VOŽNJU ZASNOVANIH NA PROCJENI KUTA ZAKRETA UPRAVLJAČA VOZILA

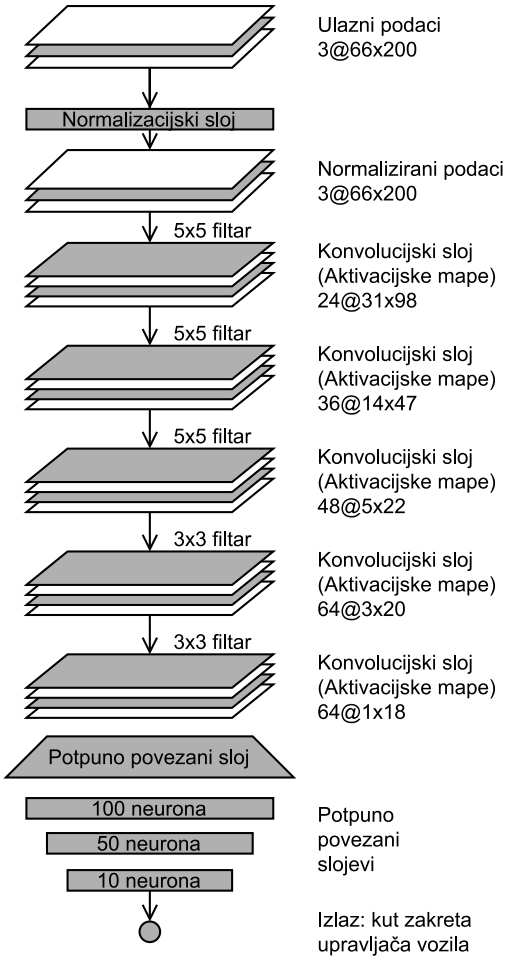
2.1. Problem procjene kuta zakreta upravljača vozila

Procjena kuta zakreta upravljača vozila predstavlja problem lateralne kontrole vozila. To je postupak određivanja kuta pod kojim je potrebno zakrenuti upravljač vozila kako bi se ono kretalo unutar odgovarajuće prometne trake. Algoritmi za procjenu kuta zakreta upravljača vozila dijele se na algoritme zasnovane na računalnom vidu i *end-to-end* algoritme. Algoritmi zasnovani na računalnom vidu dijele problem procjene kuta zakreta upravljača vozila na manje podzadatke kao što su detekcija kolničkih crta, detekcija prometnih traka, detekcija objekata, planiranje i slijeđenje puta i sl. Podzadaci se rješavaju pojedinačno, a procijenjeni kut zakreta upravljača dobiva se kombiniranjem rješenja podzadataka. Takvi algoritmi, zbog unaprijed definiranih vrijednosti i pravila, osjetljivi su na promjene u okolini vozila kao što su svjetlina i vremenski uvjeti, što utječe na njihove performanse. U zadnje vrijeme sve se više istražuje *end-to-end* pristup koji podsjeća na ljudsko rasuđivanje i upravljanje vozilom u smislu da se kut zakreta upravljača vozila procjenjuje na temelju vizualnih značajki, bez izričite podjele problema na podzadatke detekcije prometne trake, planiranja putanje i koordinacije ruku i smjera u kojem je potrebno usmjeriti vozilo. Takav pristup pokazao se otpornijim na razlike u boji i teksturi ceste, osvjetljenje i vremenske uvjete te vrstu i boju oznaka na kolniku. *End-to-end* pristup može se zasnivati na podržanom učenju (engl. *Reinforcement Learning - RL*) ili na oponašanju stručnjaka (engl. *imitation-based-learning*). U slučaju podržanog učenja, agent (neuronska mreža) istražuje okolinu te dobiva nagrade i kazne ovisno o poduzetoj radnji. Maksimiziranjem postignutog rezultata (nagrade) agent zaključuje koje radnje poduzeti i tako rješava dani problem. U slučaju procjene kuta zakreta upravljača vozila oponašanjem stručnjaka, rješenje je sadržano unutar duboke neuronske mreže. Takav pristup koristi se u ovom radu.

2.2. Pregled postojećih algoritama za autonomnu vožnju zasnovanih na procjeni kuta zakreta upravljača vozila

Jedni od prvih algoritama za autonomnu vožnju zasnovani na procjeni kuta zakreta upravljača vozila koji su koristili neuronske mreže su ALVINN (engl. *Autonomous Land Vehicle in a Neural Network*) [6] i DAVE (engl. *DARPA Autonomous Vehicle*) [7]. ALVINN sustav, kojeg je razvio D. A. Pomerleau, zasniva se na potpuno povezanoj mreži i jedan je od prvih pokazao da neuronska mreža može upravljati vozilom. DAVE projekt, u kojem model auta na daljinsko upravljanje treniran na podacima prikupljenim iz vožnji stručnjaka uspješno prolazi nove staze u sličnim uvjetima, demonstrirao je *end-to-end* princip, ali nije bio dovoljno robustan za korištenje izvan testnog okruženja. Glavni problemi navedenih radova bili su nepreciznost senzora i suočavanje s nepoznatom okolinom. Razvojem tehnologije povećana je preciznost senzora te su razvijene nove strukture neuronskih mreža. Najčešće korištene strukture neuronskih mreža za dani problem su konvolucijske neuronske mreže (engl. *Convolutional Neural Network - CNN*) i LSTM (engl. *Long-Short Term Memory*) neuronske mreže te njihove varijante [8]. Kako se za problem procjene kuta zakreta upravljača vozila kao ulazni podatak najčešće koristi slika s prednje kamere vozila, konvolucijske neuronske mreže predstavljaju logičan izbor za taj zadatak zbog svoje sposobnosti analize slikovnih podataka i prepoznavanja uzoraka. Konvolucijski slojevi sadrže filtar kojim skalarno množe danu sliku, što omogućava detekciju bridova, oblika, tekstura i drugih obilježja slike.

Jedan od značajnijih modela algoritama za autonomnu vožnju zasnovan na procjeni kuta zakreta upravljača vozila je NVIDIA PilotNet [9], koji je inspiriran prethodno navedenim projektima. PilotNet se zasniva na konvolucijskoj neuronskoj mreži koja procjenjuje naredbu zakreta upravljača na temelju slike s prednje kamere vozila. Naredba zakreta upravljača zapisana je kao $1/r$, gdje je r radijus kruga kojeg bi vozilo napravilo za dani kut zakreta upravljača. Takav zapis odabran je kako predloženo rješenje ne bi ovisilo o geometriji korištenog vozila. PilotNet neuronska mreža sastoji se od devet slojeva: normalizacijski sloj, 5 konvolucijskih slojeva i 3 potpuno povezana sloja te ima približno 250000 parametara. Na slici 2.1 dan je pregled slojeva PilotNet neuronske mreže. Ulazni podaci su slike s tri kanala (YUV) rezolucije 66×200 elemenata slike, što je na slici 2.1 zapisano kao 3@66x200.



Slika 2.1: Arhitektura PilotNet neuronske mreže [9]

Za uspješno treniranje neuronske mreže potrebno je ulazne podatke normalizirati, što znači pretvoriti svaki element slike iz cjelobrojnog zapisa raspona [0-255] u realni broj iz intervala [0-1]. Normalizacijski sloj je predefiniran i ne optimizira se prilikom treniranja, no omogućava bržu obradu ulaznih podataka pomoću grafičke kartice. Konvolucijski slojevi izlučuju značajke slike pomoću 5×5 i 3×3 filtara, čime nastaju tzv. aktivacijske mape čije se dimenzije mogu zapisati kao dimenzije slike: *dubina* @ *visina* x *širina*, a mogu se izračunati iz dimenzije podataka koji ulaze u odgovarajući sloj te veličine filtra i koraka pomaka filtra (engl. *stride*). Prvi, drugi i treći konvolucijski sloj koriste korak pomaka filtra veličine dva, dok četvrti i peti konvolucijski sloj koriste korak pomaka filtra veličine jedan. Izlaz zadnjeg konvolucijskog sloja pretvara se u jednodimenzionalni tenzor (engl. *flatten*) te se prosljeđuje potpuno povezanim slojevima, no zbog treniranja sustava *end-to-end* pristupom, ne može se točno odijeliti koji dijelovi mreže prepoznaju značajke slike, a koji na temelju značajki procjenjuju naredbu zakreta upravljača. Podaci za PilotNet prikupljeni su tako da tri kamere s prednje strane vozila snimaju cestu, a dekodiranjem poruka s CAN (engl.

Controller Area Network) sučelja automobila dohvaćaju se naredbe zakreta upravljača vozila za svaki trenutak tj. za svaku snimljenu sličicu. Skup podataka sastoji se od slika snimljenih prilikom vožnje različitim cestama u različitim vremenskim uvjetima i odgovarajućih naredbi zakreta upravljača. Vremenski uvjeti obuhvaćeni skupom podataka su: vedro, oblačno, maglovito, sniježno i kišovito vrijeme, a podaci su prikupljani i noću. Najzastupljenije su autoceste i brze ceste, a ukupno je snimljeno približno 72 sata vožnje. Za treniranje modela neuronske mreže odabrane su slike i pripadajuće naredbe zakreta upravljača vozila gdje je vozač slijedio odabranu prometnu traku, a ostatak podataka je odbačen, npr. vožnja kroz raskrižja, prestrojavanje i obilaženje. Nadalje, osiguran je veći broj podataka koji sadrže zavoje na cesti, kako trenirani model neuronske mreže ne bi imao sklonost prema vožnji ravno, a konačni skup podataka je augmentiran. Augmentacija podataka izvedena je tako da su slike transformirane da predstavljaju pogled iz drugačije perspektive (engl. *viewpoint transformation*), a naredbama zakreta upravljača promijenjene su vrijednosti tako da simuliraju povratak vozila u odgovarajuću prometnu traku, kako bi istu radnju nakon pogreške mogao izvesti i model neuronske mreže. Rezultat treniranog PilotNet modela predstavljen je pomoću mjere autonomije (engl. *autonomy*) koja je definirana kao:

$$\text{autonomija} = (1 - \frac{(\text{broj intervencija}) \times 6 \text{ sekundi}}{\text{proteklo vrijeme [s]}}) \times 100 \tag{2-1}$$

Mjera autonomije dobivena je brojanjem ljudskih intervencija prilikom testiranja modela neuronske mreže. Intervencija je potrebna ako je sredina simuliranog vozila udaljena od sredine željene prometne trake za više od jednog metra. Šest sekundi je određeno vrijeme za prosječnu ljudsku intervenciju kao vrijeme potrebno za preuzimanje kontrole nad vozilom, postavljanje vozila na sredinu prometne trake i ponovno uključivanje autonomne vožnje. Tijekom testiranja u simulatoru i na stvarnim cestama postignuta je mjera autonomije od 98%, a neki od testova odrađeni su bez intervencija.

Drugi tip neuronskih mreža koje se često koriste pri procjeni kuta zakreta upravljača vozila su LSTM mreže. LSTM je vrsta povratne neuronske mreže (engl. *Recurrent Neural Network - RNN*) koja kao takva ima mogućnost učenja na podacima koji ovise o kontekstu ili redoslijedu tj. sekvenci promatranih podataka. Ono čime se LSTM razlikuje od prethodno osmišljenih povratnih neuronskih mreža je rješavanje problema nestajućeg gradijenta, koji se pojavljuje prilikom treniranja neuronskih mreža. Rješenje problema nestajućeg gradijenta

omogućava učenje na kontekstno udaljenijim podacima, npr. riječi koje nisu napisane jedna pored druge ili u istoj rečenici, ali zajedno tvore kontekst paragrafa ili vremenski niz podataka gdje između dva bitna događaja postoji neodređeno dugo vremensko odstojanje. Problem nestajućeg gradijenta riješen je tako što izračunati gradijenti mogu prolaziti nepromijenjeni kroz memorijsku ćeliju, što nije slučaj kod ostalih tipova povratnih neuronskih mreža.

LSTM neuronske mreže često se koriste u obradi prirodnog jezika (engl. *Natural Language Processing - NLP*), prepoznavanju govora i prevođenju jer u takvim problemima postoji jasno definiran i ljudima intuitivan redoslijed znakova ili zvukova koji utječe na značenje ulaznih podataka, a isto tako utječe i na točnost rješenja. Zbog takvih svojstava prepoznavanja redoslijeda, LSTM neuronske mreže predstavljaju logičan korak u razvoju algoritama za autonomnu vožnju jer se upravljanje vozilom može promatrati kao niz vremenski zavisnih akcija: procijenjeni kut zakreta upravljača vozila za prethodnu sličicu utječe na trenutni procijenjeni kut zakreta upravljača. Neuronske mreže za dani problem koje se zasnivaju na LSTM slojevima mogu se razlikovati po svojoj strukturi, ali zajedničko im je korištenje LSTM slojeva za učenje vremenskih ovisnosti u podacima.

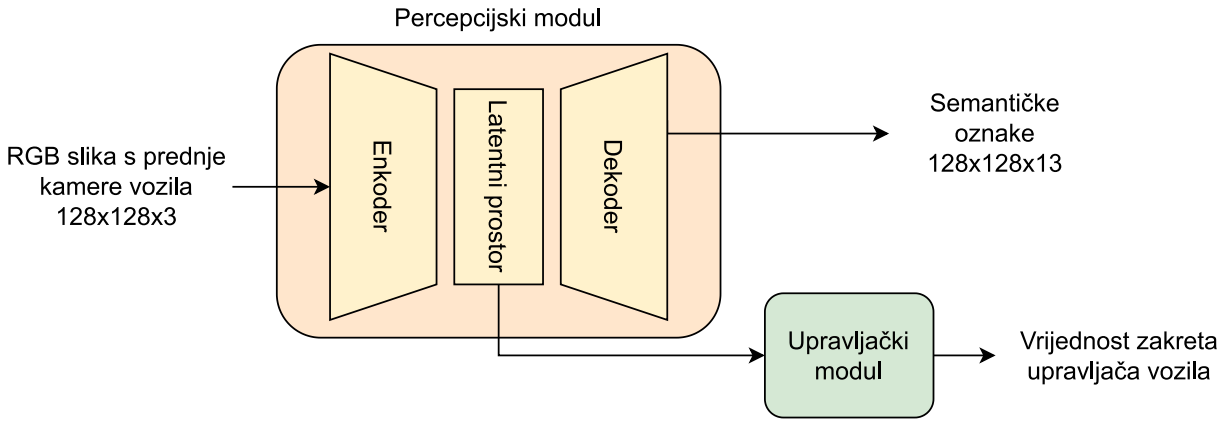
U [10], implementirana je trodimenzionalna konvolucija koja uz uobičajene visinu i širinu posjeduje dubinu, što je zapravo vrijeme između uzastopnih slika. To znači da trodimenzionalna konvolucija radi s volumenima podataka umjesto s plohama te se ulazni podaci moraju predati u obliku tzv. stogova slika (engl. *image stack*) pa tako npr. niz koji se sastoji od pet slika s tri kanala širine 320 i visine 120 elemenata slike ima oblik ($1\times5\times120\times320\times3$). Nakon slojeva za izlučivanje značajki trodimenzionalnom konvolucijom nalaze se LSTM slojevi koji uče vremenske ovisnosti u podacima. Na kraju neuronske mreže nalaze se potpuno povezani slojevi, koji kao i u PilotNet mreži, služe za procjenu kuta zakreta upravljača vozila iz izlučenih značajki. Predložena neuronska mreža trenirana je na Udacity skupu podataka [11], uz korištenje augmentacija koje predstavljaju različite uvjete na cesti. Primjeri slika iz navedenog skupa prikazani su na slici 2.2. Korištene augmentacije uključuju nasumično dodavanje sjena na sliku, povećanje i smanjenje svjetline slike te vertikalni i horizontalni pomak. Umjereno augmentiranje trening skupa pomoglo je predstavljenom modelu neuronske mreže u generalizaciji, ali pretjerano augmentiranje podataka pogoršalo je performanse modela. Predložena neuronska mreža ostvarila je sličan rezultat kao PilotNet model neuronske mreže na istom skupu podataka, no ograničenje su predstavljali računalni resursi.

Slika 2.2: Primjeri slika s prednje kamere vozila iz Udacity skupa podataka [11]

U [12] predložena je implementacija neuronske mreže slična implementaciji u [10], no korištena je dvodimenzionalna konvolucijska mreža, prethodno trenirana na Imagenet skupu podataka [13], koja obrađuje svaku sliku dobivenu s prednje kamere vozila te prosljeđuje rezultantni vektor značajki LSTM slojevima koji prepoznaju vremenske ovisnosti u podacima. U LSTM slojevima korištena je tehnika pomičnog prozora, što znači da se dobiveni vektor značajki određene sličice koristi prilikom dviju uzastopnih predikcija, ali na drugom mjestu u nizu, ovisno o vremenu pojavljivanja. Implementirana neuronska mreža trenirana je i testirana na Comma.ai skupu podataka [14]. Primjeri slika iz navedenog skupa prikazani su na slici 2.3. Postignut je 35% bolji rezultat u odnosu na rezultat PilotNet modela konvolucijske mreže na istom skupu podataka, no problem procjene kuta zakreta upravljača vozila sveden je s regresijskog problema na klasifikacijski radi poboljšanja performansi. Rješavanjem problema procjene kuta zakreta upravljača vozila kao klasifikacijskog problema gubi se rezolucija rješenja tj. nije moguće opisati kontinuirane vrijednosti kuta zakreta upravljača koje se pojavljuju u stvarnome svijetu.

Slika 2.3: Primjeri slika s prednje kamere vozila iz Comma.ai skupa podataka [14]

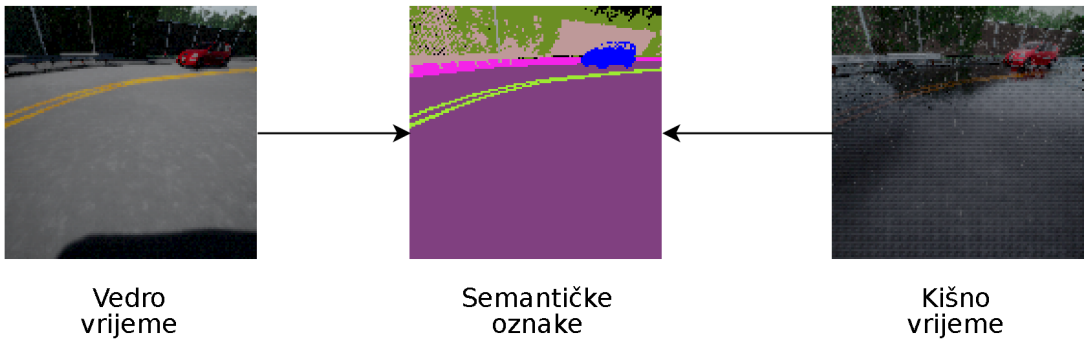
Rješenje dano u [15] zasnovano je na podržanom učenju, a cilj mu je odvojiti percepciju okoline od upravljanja vozilom. Stoga se sastoji od percepcijskog i upravljačkog modula (Slika 2.4).



Slika 2.4: Arhitektura rješenja danog u [15]

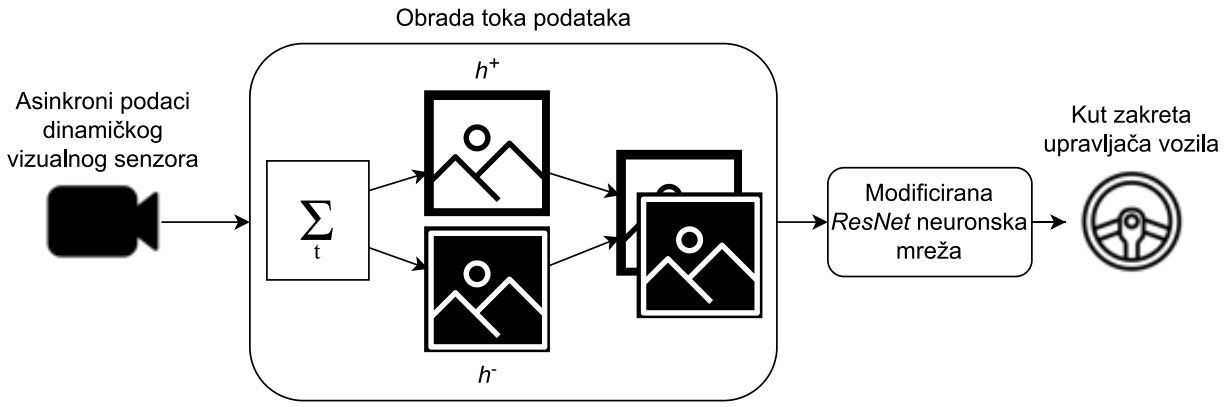
Percepcijski modul prima sliku RGB formata s prednje kamere vozila, obrađuje ju i generira značajke koje predstavljaju semantičku reprezentaciju svih elemenata slike. Percepcijski modul treniran je kao enkoder-dekoderska arhitektura, a zasnovan je na konvolucijskim neuronskim mrežama. Enkoder prima slike RGB formata i pretvara ih u vektor u latentnom prostoru. Dekoder prima navedeni vektor i iz njega pokušava rekonstruirati semantičku segmentaciju slike s ulaza modula. Izlaz dekodera jednake je rezolucije kao ulazna slika, ali ima 13 kanala od kojih svaki predstavlja jednu klasu objekata koji se mogu pojaviti u prometu. Tako svaki element slike sadrži vjerojatnost pojavljivanja pojedine klase. Važno je napomenuti da se neke klase pojavljuju češće nego ostale, npr. cesta se pojavljuje češće nego prometni znakovi. Zbog toga je svakoj klasi dodijeljena težina koja je obrnuto proporcionalna učestalosti pojavljivanja te klase. Drugi dio rješenja je upravljački modul koji se sastoji od potpuno povezanih slojeva s *ReLU* (engl. *Rectified Linear Unit*) aktivacijskim funkcijama, a treniran je pomoću *Q-learning* algoritma podržanog učenja unutar CARLA simulatora. Upravljački modul prima značajke iz enkodera te vraća niskodimenzionalni vektor tj. određuje koju akciju je potrebno poduzeti s obzirom na dobivene značajke. Jedna od prednosti semantičke segmentacije slike s prednje kamere je mogućnost smanjivanja dimenzionalnosti ulaznih podataka zbog njihove redundantnosti. Naime, većina elemenata slike nalazi se u susjedstvu elemenata koji imaju istu semantičku oznaku, što omogućava prikaz ulaznih podataka pomoću nižedimenzionalnih vektora u latentnom prostoru. Još jedna prednost je semantička jednakost pojedinih elemenata

slike bez obzira na vremenske uvjete i doba dana, što omogućava brže treniranje modela zasnovanih na podržanom učenju (Slika 2.5). Opisano rješenje relativno uspješno upravlja vozilom unutar CARLA simulatora, no prilikom određenih scenarija javljaju se problemi zbog diskontinuiteta u zadanoj kriterijskoj funkciji. Na primjer, prilikom sudara sa zaštitnom ogradom, ostvarena nagrada naglo pada te se simulacija prekida iako je vozilo do tog trenutka uspješno slijedilo zadanu prometnu traku.



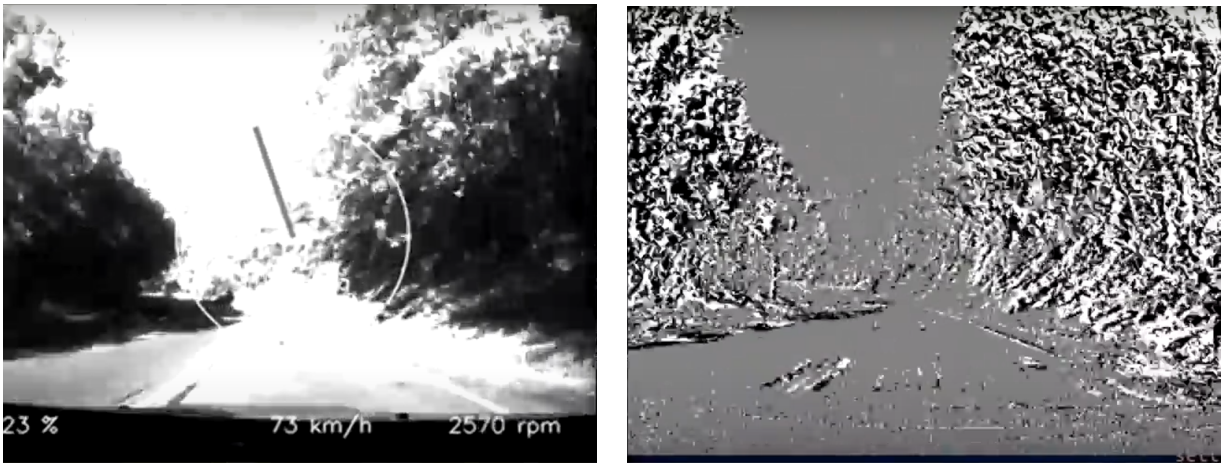
Slika 2.5: Prikaz jednakih semantičkih oznaka u različitim vremenskim uvjetima [15]

Rješenja poput [16] i [17] predlažu korištenje dinamičkog vizualnog senzora (engl. *Dynamic Vision Sensors - DVS*) uz uobičajenu prednju kameru vozila. Uobičajene kamere bilježe intenzitet svjetlosti svakog elementa slike (engl. *Active Pixel Sensor - APS*), a dinamički vizualni senzori bilježe lokalne promjene intenziteta svjetlosti na razini elementa slike u obliku toka podataka. Stoga se dinamički vizualni senzori ponašaju kao detektori pokreta i filtriraju sve vremenski redundantne informacije. U radu [16], postojeće *ResNet* arhitekture neuronskih mreža [18] trenirane su na podacima dobivenim pomoću dinamičkog vizualnog senzora. Tok podataka dobiven pomoću dinamičkog vizualnog senzora obrađen je tako da su svi dobiveni podaci unutar određenog vremenskog intervala (engl. *integration time*) zapisani u obliku dvokanalne slike, čiji prvi kanal određuje pozitivne promjene intenziteta svjetlosti, a drugi kanal određuje negativne promjene intenziteta svjetlosti, što je na slici 2.6 označeno s h^+ i h^-. Postojeće *ResNet* arhitekture korištene su kao ekstraktori značajki tj. korišteni su samo konvolucijski slojevi. Nakon konvolucijskih slojeva dodan je sloj sažimanja *Global Average Pooling* i potpuno povezani slojevi čiji izlaz predstavlja kut zakreta upravljača vozila. Za treniranje i testiranje korišten je DDD17 skup podataka [19]. Skup podataka za treniranje obrađen je tako da se ne koriste podaci gdje se vozilo kreće brzinom manjom od 20 km/h te je korišteno samo 30% podataka gdje je kut zakreta



Slika 2.6: Blok dijagram rješenja danog u [16]

upravljača u intervalu [-5°, 5°]. U suprotnom neuronska mreža za svaki ulazni podatak predviđa da je potreban kut zakreta upravljača 0°. Na testnom skupu podataka, koji je neobrađen, postignuti su bolji rezultati u odnosu na neuronske mreže jednake arhitekture trenirane na monokromatskim slikama iz istog skupa podatka. Neuronska mreža trenirana na podacima dobivenim pomoću dinamičkog vizualnog senzora otpornija je na značajne promjene osvjetljenja i brze pokrete. Na slici 2.7(b), dobivenoj obradom toka podataka dinamičkog vizualnog senzora, mogu se vidjeti kolničke crte, iako se one ne vide na monokromatskoj slici 2.7(a).



(a) (b)

Slika 2.7: Primjer podataka iz DDD17 skupa podataka: (a) monokromatska slika s prednje kamere vozila, (b) slika dobivena obradom toka podataka dinamičkog vizualnog senzora [19]

U radu [17] predstavljen je novi skup podataka DDD20 koji sadrži monokromatske slike, tok podataka dinamičkog vizualnog senzora i podatke o vozilu kao što su brzina, pritisak

papučice gasa i kočnice te kut zakreta upravljača za svaku snimljenu sliku. *ResNet-32* neuronska mreža modificirana je kao što je prethodno opisano u [16], a na jednak način obrađen je i skup podataka za treniranje. Na DDD20 testnom skupu podataka najbolji rezultati dobiveni su kada su za treniranje neuronske mreže korištene slike dobivene obradom toka podataka dinamičkog vizualnog senzora korištene u paru s odgovarajućom monokromatskom slikom. Nedostatak dinamičkih vizualnih senzora je vrijeme akumulacije podataka tj. vrijeme potrebno za dobivanje dovoljne količine podataka pomoću dinamičkog senzora. Najbolji rezultati iz [16] i [17] dobiveni su kada je vrijeme akumulacije iznosilo 50 ms, što ograničava maksimalni broj obrađenih okvira u sekundi, a time i primjenu algoritama zasnovanih na dinamičkim vizualnim senzorima u stvarnom svijetu.

3. PRIJEDLOG VLASTITOG ALGORITMA ZA AUTONOMNU VOŽNJU ZASNOVANOG NA PROCJENI KUTA ZAKRETA UPRAVLJAČA VOZILA

Glavni cilj ovog rada je razviti novi vlastiti algoritam autonomne vožnje zasnovan na procjeni kuta upravljača vozila. Predloženi algoritam potrebno je istrenirati na podacima prikupljenima iz dvaju različitih simulatora. Trenirane modele potrebno je testirati u obama simulatorima, a dobivene rezultate međusobno usporediti.

Predloženi algoritam zasnovan je na *Swin Transformer* [20] arhitekturi neuronske mreže jer su zadnjih godina transformerske arhitekture predmet istraživanja, pogotovo u vizualnim zadacima te su pokazale dobru moć generalizacije i vrlo dobre rezultate u području obrade prirodnog jezika. Ostvareni rezultati i dobre generalizacijske sposobnosti povezani su s gotovo neograničenim receptivnim poljem transformerskih arhitektura. Mogućnost zaključivanja iz prostorno udaljenih, a značenjski povezanih podataka, ključna je pri korištenju transformerskih neuronskih mreža za obradu slike. Predloženi algoritam kao ulazni podatak prima sliku s prednje kamere vozila, a kao izlaz daje vrijednost koja odgovara procijenjenom kutu zakreta upravljača vozila.

Za treniranje vlastitog algoritma potrebno je kreirati dva različita skupa podataka iz dvaju odabranih simulatora. Trening podaci se u ovom slučaju sastoje od slika s prednje kamere i odgovarajuće vrijednosti koja predstavlja kut zakreta upravljača vozila. Predloženi algoritam potrebno je trenirati i testirati na oba trening skupa zasebno te dobivene modele neuronske mreže testirati u obama simulatorima. Nakon toga potrebno je predtrenirane modele dotrenirati na podacima iz drugoga simulatora, što se naziva prijenosno učenje (engl. *transfer learning*). Dobivene dotrenirane modele potrebno je ponovno testirati u obama simulatorima. Svi dobiveni rezultati uspoređeni su i komentirani u 4. poglavlju.

3.1. Alati i tehnologije korišteni za razvoj vlastitog algoritma za autonomnu vožnju

3.1.1. TensorFlow i Keras biblioteke

Za implementaciju predloženog algoritma potrebne su biblioteke za strojno učenje za programski jezik Python. TensorFlow je besplatna biblioteka otvorenog koda za strojno

učenje i umjetnu inteligenciju koja pruža mogućnosti poput definiranja strukture neuronskih mreža, definiranje vlastitih čvorova, definiranje kriterijskih funkcija, optimizatora i metrika za mjerenje performansi modela za strojno učenje [21]. Uz to, TensorFlow uključuje proširenja za rad sa često korištenim skupovima podataka, za vizualizaciju, praćenje procesa treniranja i optimizaciju modela za strojno učenje i dr. Zasniva se na radu s višedimenzionalnim poljima (tenzorima) za koje su implementirane standardne matematičke operacije i operacije specifične za strojno učenje.

Keras je besplatna biblioteka otvorenog koda koja služi kao sučelje za rad s TensorFlow bibliotekom [22]. Keras je razvijen kako bi se pojednostavio i time ubrzao proces implementacije i razvoja modela za strojno učenje. Kao takav, sadrži definicije često korištenih slojeva neuronskih mreža, aktivacijskih funkcija, optimizatora i operacija za rad s tekstualnim, slikovnim i drugim vrstama podataka, dostupne kroz jednostavno sučelje (engl. *Application Programming Interface - API*).

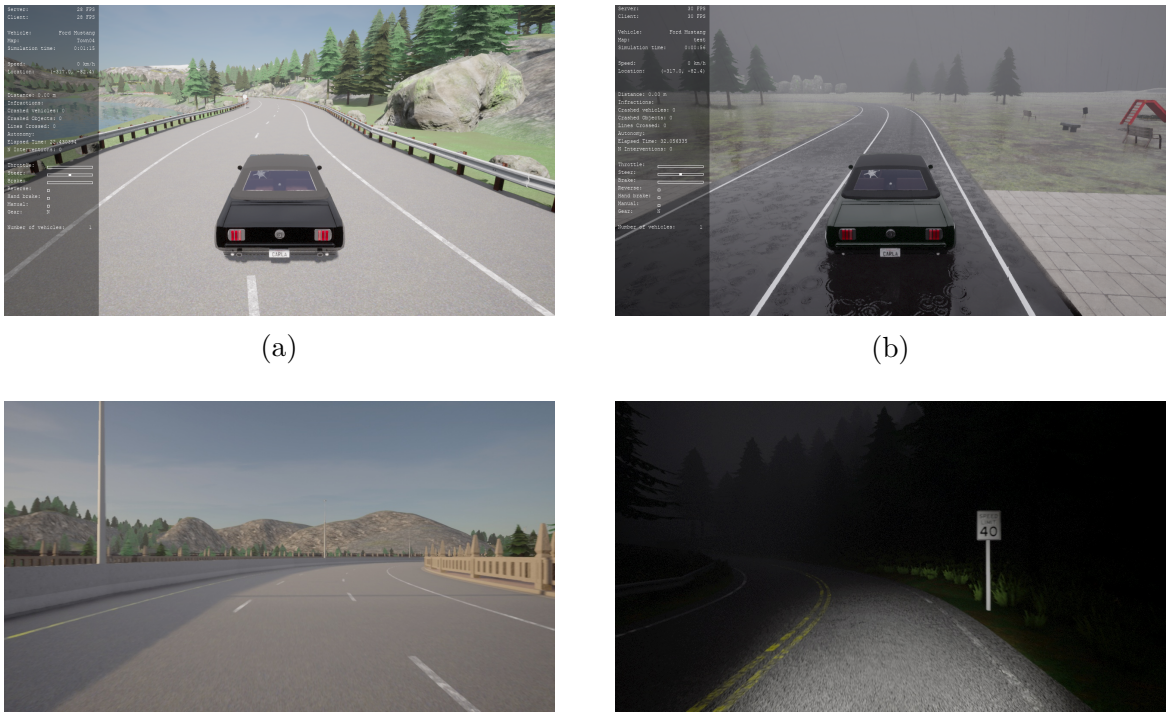
Tensorflow i Keras mogu se instalirati pomoću *Pip* alata za upravljanje paketima pozivanjem sljedećih naredbi u terminalu:

```
pip install tensorflow
pip install keras
```

3.1.2. CARLA simulator

Uz biblioteke za strojno učenje, potrebno je osigurati okolinu i odgovarajuće podatke za treniranje i testiranje predloženih rješenja. U tu svrhu koriste se simulatori za autonomnu vožnju. Oni omogućavaju sigurnu okolinu za treniranje i testiranje predloženih rješenja uz ponovljive scenarije i uvjete u virtualnom okruženju vozila. Nadalje, mogućnost promjene scenarija i vremenskih uvjeta može se koristiti za stvaranje baze podataka koja je ključna za treniranje i testiranje algoritama autonomne vožnje. CARLA (engl. *Car Learning to Act*) [4] je besplatni simulator otvorenog koda namijenjen istraživanju i razvoju autonomne vožnje. Omogućava razvoj, treniranje i testiranje algoritama za autonomnu vožnju. Veliki broj implementiranih senzora kao što su dinamički vizualni senzori, RGB i dubinske kamere, kamere koje računaju optički tok ili obavljaju semantičku segmentaciju, senzori kolizije i napuštanja prometne trake, lidar, radar, navigacijski sustav, akcelerometar, žiroskop i kompas te mogućnosti određivanja vremenskih uvjeta i upravljanja statičkim i dinamičkim elementima simulacije čine ovaj simulator svestranom platformom za razvoj algoritama

autonomne vožnje. Virtualna okolina simulatora u različitim vremenskim uvjetima i s različitim perspektivama prikazana je na slici 3.1. Ono što CARLA simulator čini posebnim je korištenje klijent-poslužitelj arhitekture. Virtualna okolina simulatora pokreće se zasebno pomoću izvršne datoteke i ima ulogu poslužitelja, dok klijentske skripte šalju zahtjeve za npr. konfiguraciju okruženja, stvaranje vozila i drugih objekata te upravljanje objektima. Na slici 3.2 dan je isječak programskog koda za postavljanje klijentske skripte, dohvaćanje virtualnog okruženja te stvaranje vlastitog vozila (engl. *ego vehicle*) i prednje kamere vozila.



(a) (b)

(c) (d)

Slika 3.1: Virtualno okruženje CARLA simulatora, pogled iz trećeg lica: (a) vedro vrijeme i (b) kišno vrijeme, pogled s prednje kemere vozila: (c) zalazak sunca i (d) noć

```python
import carla
import random

# Stvaranje klijentskog objekta, dohvaćanje virtualne okoline i učitavanje novog grada
client = carla.Client('localhost', 2000)
world = client.get_world()
world.load_world('Town05')

# Dohvaćanje nacrta vozila i lokacija gdje ih je moguće stvoriti
vehicle_blueprints = world.get_blueprint_library().filter('*vehicle*')
spawn_points = world.get_map().get_spawn_points()
```

```
# Stvaranje vozila kojim je moguće upravljati
ego_vehicle = world.spawn_actor(random.choice(vehicle_blueprints),
↪  random.choice(spawn_points))

# Stvaranje prednje kamere vozila
camera_init_trans = carla.Transform(carla.Location(z=1.5))
camera_bp = world.get_blueprint_library().find('sensor.camera.rgb')
camera = world.spawn_actor(camera_bp, camera_init_trans,
↪  attach_to=ego_vehicle)
```

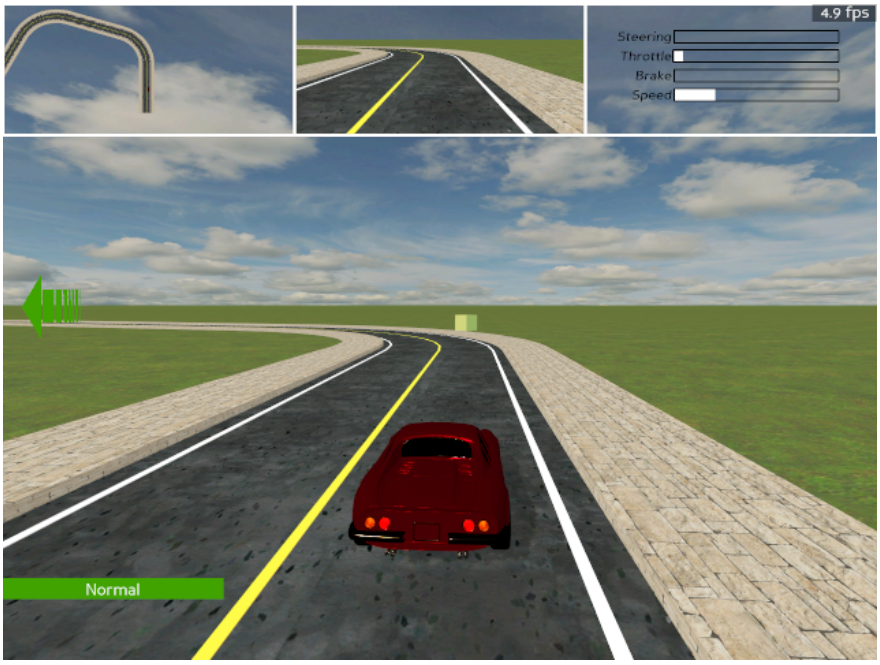
Slika 3.2: Programski kod za stvaranje virtualnog okruženja i vlastitog vozila u CARLA simulatoru

3.1.3. MetaDrive simulator

MetaDrive [5] je besplatni simulator otvorenog koda namijenjen razvijanju algoritama za autonomnu vožnju s naglaskom na podržano učenje. Omogućava rad s jednim ili više agenata te se može izvršavati brzinom do 300 sličica u sekundi (engl. *Frames Per Second - FPS*). Posebnost MetaDrive simulatora je proceduralno generiranje cesta, što algoritmima za autonomnu vožnju zasnovanim na podržanom učenju omogućava učenje u različitim okruženjima i razvoj generalizacijskih svojstava. Cesta se proceduralno generira koristeći predefinirane građevne blokove kao što su ravna cesta (S), zavoj (C), kružni tok (O) itd. Proceduralno generiranje okruženja moguće je definirati na dva načina. Prvi način je specificiranje broja korištenih blokova. Tada algoritam nasumično odabire koje građevne blokove koristiti, mijenja im parametre kao što je duljina te spaja blokove u jednu cestu dok ne dostigne željeni broj korištenih blokova. Drugi način je zadavanje niza građevnih blokova pomoću oznaka, npr. predavanjem oznaka SCS, algoritam stvara cestu koja sadrži ravnu cestu, zavoj te ponovno ravnu cestu.

Rad u MetaDrive simulatoru zasnovan je na *OpenAI gym* okruženjima [23]. Moguće je odabrati neko od predefiniranih okruženja, npr. okruženje za jednog ili više agenata, okruženje s opasnostima na cesti i sl. Željeno virtualno okruženje dodatno se definira pomoću rječnika tj. po principu ključ-vrijednost. Moguće je postaviti gustoću prometa, broj prometnih traka, nagrade i kazne u slučaju korištenja podržanog učenja, rezoluciju slike s prednje kamere vozila i slične parametre. Predefinirana okruženja sadrže vlastito vozilo kojim je moguće upravljati ili upravljanje prepustiti algoritmu uključenom u okruženje koji slijedi predefinirana pravila vožnje (engl. *driving policy*). Virtualno okruženje MetaDrive simulatora prikazano je na slici 3.3, a programski kod za definiranje i pokretanje jednog

takvog okruženja s deset uzastopnih zavoja, uključenom prednjom kamerom i vlastitim vozilom kojim je moguće upravljati prikazan je na slici 3.4.



Slika 3.3: Virtualno okruženje MetaDrive simulatora

```
import metadrive
import gym

# Definiranje konfiguracijskog riječnika
config = metadrive.MetaDriveEnv.default_config()
config.update(dict(environment_num=1,
                   manual_control=True,
                   traffic_density=0.05,
                   offscreen_render=True,
                   vehicle_config={"rgb_camera": (1280, 720),
                                   "image_source": "rgb_camera"},
                   map_config={BaseMap.LANE_WIDTH: LANE_WIDTH,
                               BaseMap.LANE_NUM: LANE_NUM,
                               BaseMap.GENERATE_TYPE:
                                ↪  MapGenerateMethod.BIG_BLOCK_SEQUENCE,
                               BaseMap.GENERATE_CONFIG: "CCCCCCCCCC"},
                   need_inverse_traffic=True,
                   random_traffic=True))

# Stvaranje virtualnog okruženja
env = metadrive.MetaDriveEnv(config=dict(config))
obs = env.reset()

# Pokretanje simulacije
for i in range(1000):
```

```
    obs, reward, done, info = env.step()
    env.render()
    if done:
        env.close()
```

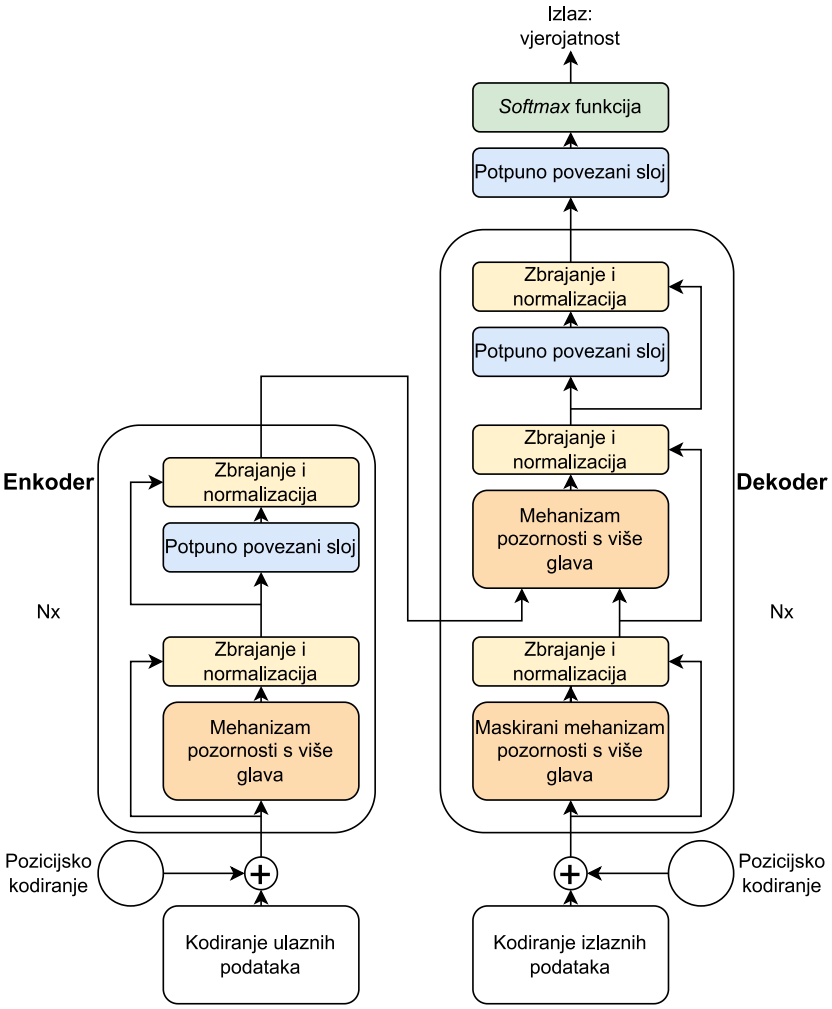
Slika 3.4: Programski kod za stvaranje virtualnog okruženja i vlastitog vozila u MetaDrive simulatoru

3.2. Opis vlastitog algoritma za autonomnu vožnju zasnovanog na procjeni kuta zakreta upravljača vozila

3.2.1. Opis transformerskih arhitektura

Transformerske arhitekture prvotno su razvijene za obradu prirodnog jezika, a glavni cilj bio je ostvariti bolje performanse nego tadašnji modeli povratnih neuronskih mreža. Nedostatak povratnih neuronskih mreža je relativno kratko pamćenje memorijskih ćelija, što može predstavljati problem ako su rečenice koje model treba obraditi dugačke jer to dovodi do gubitka konteksta ili zaboravljanja riječi koje su značenjski povezane, a nalaze se na udaljenim mjestima u rečenici. Transformerske neuronske mreže koriste pozornost (engl. *attention mechanism*) kako bi zaključile koje riječi u rečenici su povezane tj. kako bi dobile informacije o kontekstu. Struktura osnovne transformerske neuronske mreže prikazana je na slici 3.5, a zasnovana je na radu [24].

Transformerska neuronska mreža sastoji se od enkodera koji generira kodiranja za ulazne podatke koja govore kako su ulazni podaci povezani i dekodera koji koristi obrađene ulazne podatke i prethodno generirane izlazne podatke kako bi generirao sljedeći izlazni podatak. Na primjeru prevođenja teksta s hrvatskog na engleski jezik, ulazni podaci bile bi riječi na hrvatskom, a izlazni podaci riječi na engleskom jeziku. Enkoder prvo pretvara svaku ulaznu riječ u latentni prostor u kojem se riječi sličnog značenja nalaze bliže te svakoj riječi pridružuje pozicijski vektor koji govori gdje se riječ nalazi u rečenici. Takvi vektori riječi predaju se mehanizmu pozornosti s više glava (engl. *multi-head self-attention*) koji govori na koji dio ulaznih podataka treba obratiti pažnju tj. koliko su riječi povezane. Računanje vektora pozornosti može se opisati kao mapiranje upita (engl. *query* - Q) i parova ključ-vrijednost (engl. *key-value pairs,* K-V) na izlaz gdje su sve navedene varijable vektori. Za svaki od ulaznih vektora računa se više vektora pozornosti čiji ponderirani prosjek daje uvid u kontekstualne odnose riječi. Zbog više vektora pozornosti koji se računaju paralelno,



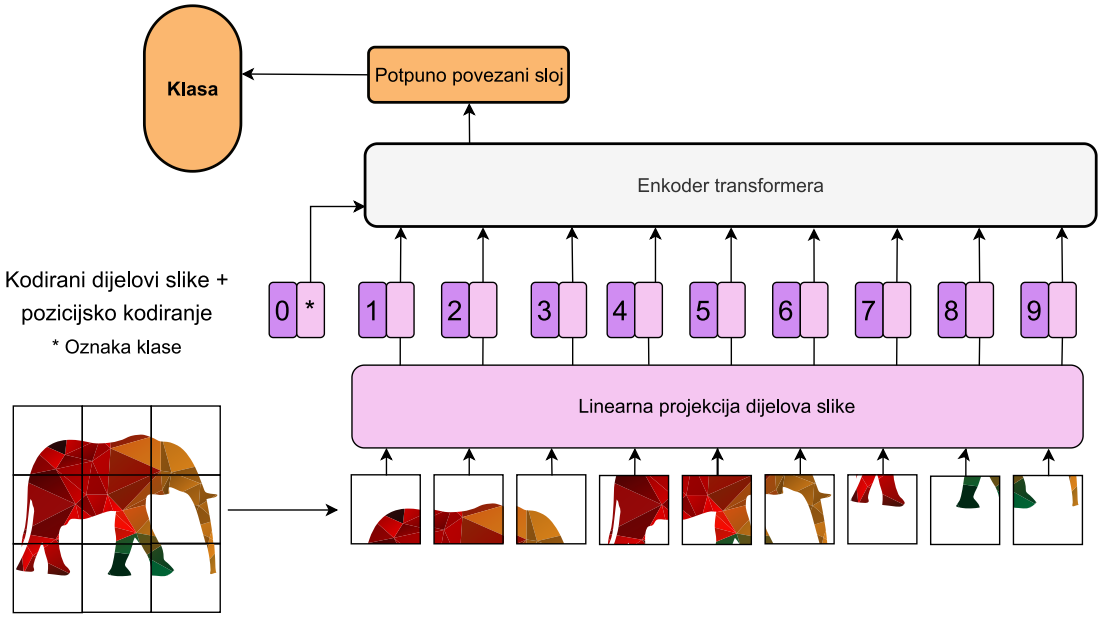
Slika 3.5: Struktura transformerske neuronske mreže [24]

taj sloj se naziva *multi-head*. Nakon toga slijedi potpuno povezani sloj koji obrađuje svaki dani vektor pozornosti, kako bi sljedeći transformerski slojevi mogli jednostavnije obraditi podatke te učiti iz posrednih reprezentacija ulaznih podataka. Zbog neovisnosti vektora pozornosti i dodavanja pozicijskih vektora kodiranim ulaznim riječima, svaki opisani dio obrade riječi može se odvijati paralelno, što značajno poboljšava performanse, a nije moguće kod povratnih neuronskih mreža. Također, uvedene su rezidualne veze i normalizacijski slojevi u svrhu poboljšanja performansi.

Dekoder je autoregresivan, što znači da generira izlazne vrijednosti na temelju ulaznih i prethodno generiranih izlaznih vrijednosti. Dekoder kodira prethodne izlazne vrijednosti i dodaje im pozicijske vektore te ih prosljeđuje sloju za određivanje pozornosti s više glava, kao i enkoder. Prvi sloj za određivanje pozornosti prilikom nadziranog učenja zanemaruje izlazne riječi koje tek trebaju biti generirane, što je postignuto maskiranjem vektora pozornosti. Drugi sloj za određivanje pozornosti kao ulaz prima podatke iz enkodera i prethodnog

dekoderskog sloja za određivanje pozornosti. U navedenom sloju značenjski se povezuju riječi iz enkodera i dekodera tj. određuje se koja ulazna riječ iz enkodera je relevantna za određenu riječ iz dekodera, što omogućava prijevod teksta. Nakon navedenog, nalazi se potpuno povezani sloj te linearni klasifikator čiji broj neurona odgovara broju riječi u jeziku na koji se tekst prevodi. *Softmax* aktivacijska funkcija klasifikatora pridružuje vjerojatnost svakoj riječi iz jezika na koji se prevodi da je baš ona sljedeća riječ u nizu. Riječ s najvećom pridruženom vjerojatnošću je izlaz neuronske mreže. Postupak se ponavlja dok najveća vjerojatnost ne bude pridružena znaku za kraj rečenice.

Iako je postala standard za obradu prirodnog jezika, osnovna transformerska arhitektura teško se može primijeniti na zadatke obrade slike. Vektori pozornosti računaju se tako da svaki token (riječ, ako je zadatak obrada teksta) utječe na svaki od preostalih tokena tj. računanje vektora pozornosti je operacija s kvadratnom složenosti. Takva arhitektura ne bi bila skalabilna pri izravnoj primjeni na elemente slike zbog iznimno velikog broja vektora pozornosti koje bi bilo potrebno izračunati. Na primjer, na MNIST skupu podataka [25], koji sadrži slike rukom napisanih znamenki rezolucije 28×28 elemenata slike, bilo bi potrebno izračunati $(28 \times 28)^2$ tj. 614656 vektora pozornosti za svaki od slojeva pozornosti i za svaku glavu (*self-attention head*) jer svaka glava računa vektore pozornosti između svakog para elemenata slike. Zbog toga vizualni transformeri (engl. *Vision Transformer - ViT*) [26] implementiraju tzv. lokaliziranu pozornost dijeljenjem ulazne slike na dijelove rezolucije 16×16 elemenata slike, što je vidljivo na slici 3.6.

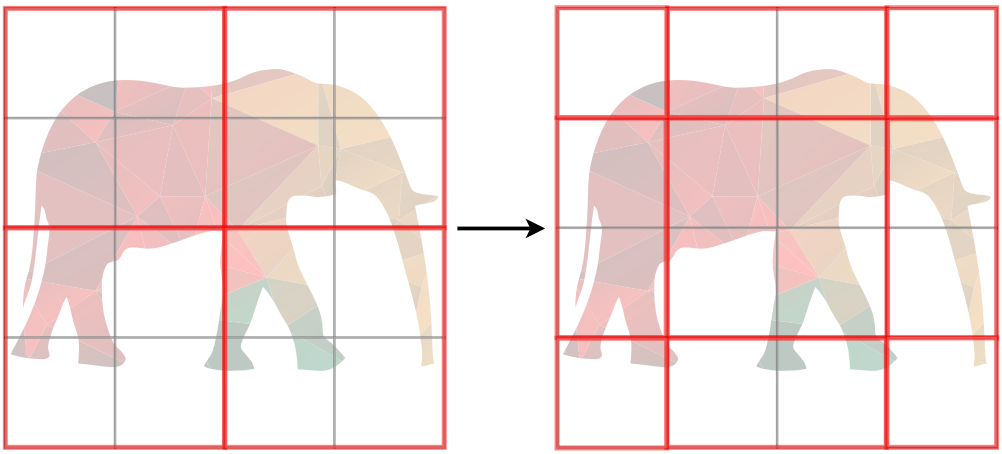


Slika 3.6: Obrada slike u vizualnim transformerima [26]

Dijelovi slike pretvoreni su u jednodimenzionalne vektore i dodane su im pozicijske oznake. Prilikom treniranja se uz navedene vektore osnovnoj arhitekturi enkoderskog sloja vizualnog transformera prosljeđuje i oznaka klase kojoj slika pripada. Prvi izlaz enkodera prosljeđuje se potpuno povezanom sloju sa *softmax* aktivacijskom funkcijom u svrhe klasifikacije. Vizualni transformeri ostvaruju bolje rezultate nego konvolucijske neuronske mreže ako su predtrenirani na dovoljno velikom skupu podataka. U radu [26], vizualni transformer nadmašio je *ResNet* arhitekturu kada se skup podataka za predtreniranje sastojao od 14 milijuna do 300 milijuna slika. Uz odgovarajuće predtreniranje, vizualni transformeri pokazali su se kao brža i računalno manje zahtjevna arhitektura za zadatke obrade slike. Razlog velikom skupu podataka za predtreniranje je dinamičko računanje težina za svaki token tj. nedostatak induktivnog pomaka (engl. *bias*) koji postoji kod konvolucijskih neuronskih mreža zbog početnog postavljanja i mijenjanja težina u konvolucijskim slojevima prilikom učenja. S druge strane, specifičnost nad ulaznim podacima omogućava vizualnim transformerima izravnavanje plohe kriterijske funkcije, što znači bolje generalizacijske sposobnosti u odnosu na *ResNet* arhitekture [27].

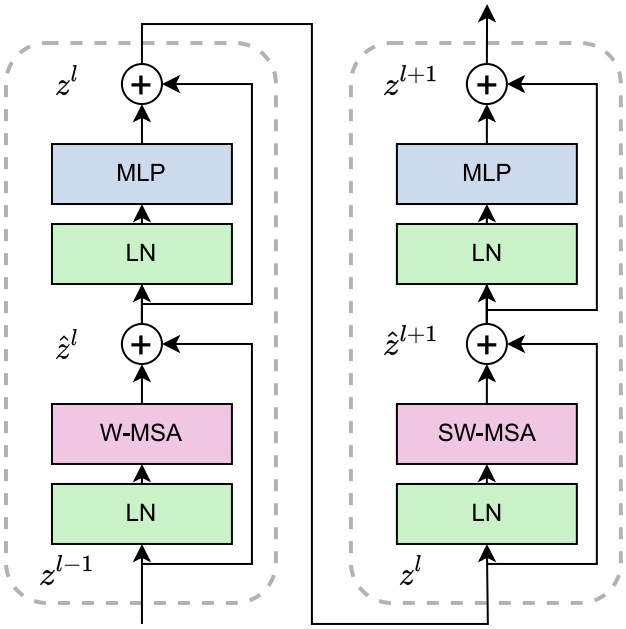
Swin transformeri (engl. *Shifted Windows*) vrsta su vizualnog transformera, ali obrađuju sliku hijerarhijski [20]. *Swin* arhitektura rješava problem kvadratne složenosti prijašnjih vizualnih transformera. Vizualni transformeri dijele sliku na dijelove rezolucije 16×16 elemenata slike, što je računalno zahtjevno ako je ulazna slika visoke ili ultravisoke rezolucije i ne daje dovoljno dobre rezultate na zadacima koji zahtijevaju preciznost na razini elementa slike, poput semantičke segmentacije. Za ulaznu sliku rezolucije 256×256, koja je vrlo mala po današnjim standardima, bilo bi potrebno izdvojiti 256 dijelova veličine 16×16, a za izdvojene dijelove izračunati 65536 vektora pozornosti za svaki *multi-head self-attention* sloj te za svaku glavu unutar sloja. Stoga je predloženo da se ulazna slika dijeli na manje dijelove koji se u dubljim slojevima mreže mogu spojiti. *Swin* transformer tako dijeli sliku na dijelove rezolucije 4×4 elementa slike koji se linearno pretvaraju u jednodimenzionalne vektore. Transformerski slojevi analiziraju dane vektore pomoću nepreklapajućih pomičnih prozora. Vektori pozornosti računaju se odvojeno unutar svakog prozora koji sadrži $M \times M$ dijelova slike umjesto za sve dijelove slike istovremeno. Na slici 3.7 crveno su označene pozicije prozora koji sadrže 2×2 sivo označena dijela slike u dva uzastopna transformerska sloja. Prozore koji ne sadrže $M \times M$ dijelova slike moguće je nadopuniti (engl. *padding*) pa maskirati prilikom računanja vektora pozornosti ili kružno pomicati prozore tako da uvijek

sadrže $M \times M$ dijelova slike.



Slika 3.7: Koncept pomicanja prozora između dva uzastopna transformerska sloja

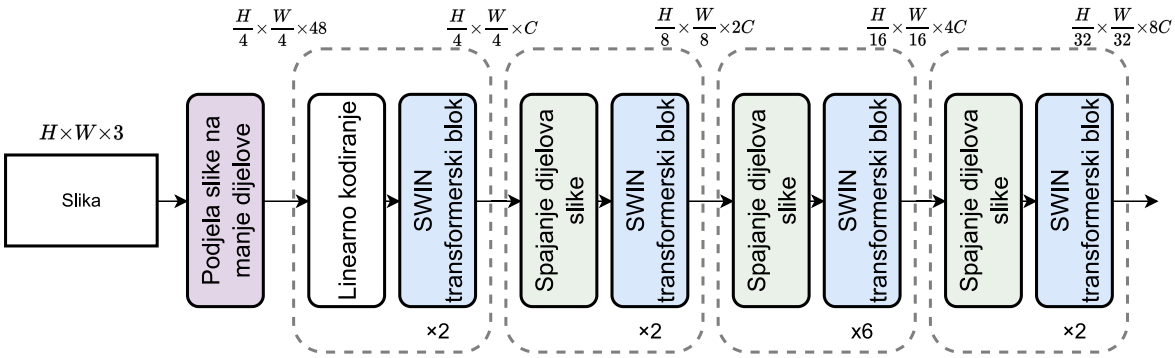
Takav pristup omogućava linearnu složenost s obzirom na rezoluciju slike. Nakon transformerskog sloja nalazi se sloj za spajanje (engl. *merging layer*) koji spaja dijelove slike u 2×2 susjedstvu, čime se smanjuje ukupan broj dijelova slike, a istovremeno povećava receptivno polje sljedećeg sloja kao u konvolucijskim neuronskim mrežama. Svaki sljedeći transformerski sloj pomiče prozore za dva dijela slike dolje i desno kako bi se pozornost mogla izračunati između dijelova slike koji nisu bili unutar istog prozora u prethodnom sloju (Slika 3.7). Na slici 3.8 prikazana su dva uzastopna transformerska sloja. LN označava normalizacijski sloj, a MLP potpuno povezani sloj. Strelicama i zbrajanjem prikazane su rezidualne veze kao i u standardnoj transformerskoj arhitekturi. Razlika je u slojevima za računanje vektora pozornosti koji koriste prozore (engl. *Windows Multi-Head Self-Attention - W-MSA*) i pomaknute prozore (engl. *Shifted Windows Multi-Head Self-Attention - SW-MSA*). Simboli $\hat{z}^l$ i z^l prikazuju izlazne značajke slojeva za računanje pozornosti odnosno potpuno povezanih slojeva gdje l označava redni broj sloja. Pomicanjem prozora također se povećava receptivno polje neuronske mreže. Prema [20], transformeri s pomičnim prozorom nadmašili su prethodne transformerske i konvolucijske arhitekture u klasifikaciji slika, detekciji objekata i semantičkoj segmentaciji te predstavljaju posljednje dostignuće i arhitekturu opće namjene u području računalnog vida.



Slika 3.8: Dva uzastopna transformerska sloja [20]

3.2.2. Implementacija vlastitog algoritma za autonomnu vožnju zasnovanog na procjeni kuta zakreta upravljača

Zbog prethodno navedenih prednosti, predloženi algoritam za autonomnu vožnju zasniva se na transformerskoj arhitekturi s pomičnim prozorom. Na slici 3.9 prikazana je struktura *Swin*-T modela iz [20], gdje se T (engl. *tiny*) odnosi na veličinu modela. U istom radu implementirane su dublje neuronske mreže *Swin*-S, B i L s više transformerskih blokova i većim kapacitetom C. Složenost *Swin*-T i *Swin*-S neuronskih mreža približno je jednaka složenosti *ResNet-50* i *ResNet-101* neuronskih mreža. Prikazane varijable H i W predstavljaju visinu i širinu slike u broju elemenata slike na ulazu u neuronsku mrežu.



Slika 3.9: Struktura *Swin*-T modela [20]

Implementacija dijeljenja slike na manje dijelove, kodiranja dijelova slike i slojeva za računanje pozornosti na temelju pomičnih prozora zasnovana je na implementaciji danoj u [28]. Dijeljenje ulazne slike, dodavanje pozicijskog kodiranja i spajanje dijelova slike implementirano je u *PatchExtract*, *PatchEmbedding* i *PatchMerge* klasama koje predstavljaju slojeve neuronske mreže i zbog toga nasljeđuju Keras klasu *Layer*. U prilogu P.3.1 dan je kod za implementaciju opisanih slojeva neuronske mreže. Nasljeđivanje klase *Layer* zahtjeva prepisivanje funkcija *init* i *call*, no tako napisani slojevi mogu se jednostavno koristiti s ugrađenim funkcijama za slaganje slojeva, treniranje i testiranje neuronske mreže. *PatchExtract* koristi ugrađenu TensorFlow funkciju za dijeljenje svake slike iz predanog skupa slika na dijelove zadane veličine uz zadani korak (engl. *stride*) i bez nadopune. Navedena klasa vraća tenzor koji sadrži dijelove slika. Potpuno povezani sloj i *Embedding* sloj, koji su također ugrađeni u TensorFlow, koriste se za kodiranje predanog dijela slike i dodavanje pozicijskog vektora u *PatchEmbedding*. Klasa *PatchMerging* spaja prethodno podijeljenu sliku povezivanjem susjednih dijelova slike i mijenjanjem oblika tenzora, kao što je prikazano na slici 3.9.

Klasa *WindowAttention* računa vektore pozornosti unutar nepreklapajućih prozora kao ponderirani zbroj vrijednosti, pri čemu se težina dodijeljena svakoj vrijednosti računa pomoću funkcije kompatibilnosti upita Q s odgovarajućim ključem K i vrijednošću V [24]. U ovom slučaju, Q, K i V su vektori dobiveni pomoću potpuno povezanog sloja koji opisuju ulazne podatke, a vektor pozornosti koji govori koliko je jedan djelić slike povezan s drugima unutar jednog prozora dobiva se pomoću formule:

$$\text{Attention}(Q, K, V) = \text{SoftMax}(QK^T/\sqrt{d} + B)V, \tag{3-1}$$

gdje je B relativni pozicijski pomak, a d dimenzija Q odnosno K vektora. Implementacija klase *WindowAttention* dana je u prilogu P.3.2.

Implementirana klasa *SwinBlock* obuhvaća jedan transformerski sloj opisan u prethodnom dijelu, a vidljiv na slici 3.8. Sloj za računanje vektora pozornosti koristi implementaciju iz priloga P.3.2, a potpuno povezani i normalizacijski slojevi dodani su kao *Dense* i *LayerNormalization* slojevi iz Keras biblioteke. Implementacija *SwinBlock* klase dana je u prilogu P.3.3. Izlazne značajke mogu se prosljeđivati sljedećim transformerskim slojevima ili opisanom *PatchMerging* sloju. U svim opisanim klasama bilo je potrebno prepisati *get_config()* funkciju kako bi se mogao spremiti i učitati model neuronske mreže.

Nakon što su implementirani svi potrebni slojevi, moguće je iskoristiti Keras sekvencijalni model (engl. *Sequential*) za dodavanje slojeva u listu redoslijedom kojim se trebaju pojavljivati u neuronskoj mreži. Sekvencijalni model ima jedan ulazni i jedan izlazni tenzor, u ovom slučaju, sliku s prednje kamere rezolucije 32×32 elementa slike i procijenjenu vrijednost zakreta upravljača vozila, no za kompleksnije modele s više ulaza ili izlaza (engl. *Multiple Input Single Output - MISO, Multiple Input Multiple Output - MIMO*) moguće je koristiti Keras funkcionalni API. Implementirano je nekoliko modela, no svi imaju sljedeći redoslijed slojeva: *PatchExtract*, *PatchEmbedding*, dva *SwinBlock* sloja i *PatchMerging* sloj. Nakon toga slijedi *GlobalAveragePooling1D* koji oblikuje podatke za sljedeće potpuno povezane slojeve s eksponencijalnim aktivacijskim funkcijama (engl. *Exponential Linear Unit - ELU*). Implementirani modeli razlikuju se u broju potpuno povezanih slojeva (engl. *Fully Connected - FC*) i broju neurona u pojedinom sloju, što određuje i konačni broj parametara modela (Tablica 3.1). Izlaz iz mreže predstavlja jedan neuron, a decimalna vrijednost u rasponu od -1 do 1 koju daje predstavlja procijenjeni kut zakreta upravljača vozila. U prilogu P.3.4 prikazan je kod za implementaciju opisane neuronske mreže.

Tablica 3.1: Arhitektura implementiranih neuronskih mreža

Ime modela	**Broj *Swin* slojeva**	**Broj neurona u *FC* slojevima**				**Broj parametara**
		1. sloj	**2. sloj**	**3. sloj**	**4. sloj**	
Swin1	2	64	32	128	256	200017
Swin2	2	512	256	128	/	382577
Swin3	2	256	/	/	/	185457
Swin4	2	128	128	/	/	185329

3.3. Stvaranje skupova podataka iz simulatora

Za treniranje predloženih neuronskih mreža bilo je potrebno stvoriti vlastite skupove podataka iz odabranih simulatora. Skup podataka sastoji se od slika s prednje kamere vozila i odgovarajućih vrijednosti koje predstavljaju kutove zakreta upravljača vozila. U oba simulatora implementirano je dohvaćanje slike s prednje kamere i dohvaćanje kontrola vlastitog vozila, što uključuje i vrijednost zakreta upravljača te spremanje u liste. U slučaju CARLA simulatora, modificirana je dostupna skripta *manual_control.py* koja koristi objekt *CameraManager* za dohvaćanje slike s odabrane kamere funkcijom *_parse_image()*. Unutar navedene funkcije dodan je uvjet gdje se provjerava je li uključeno snimanje. Ako je snimanje uključeno, trenutno dohvaćeni okvir tj. slika pretvara se iz *Carla.Image* objekta u standardni

RGB zapis funkcijom *to_rgb_array()* i sprema u listu slika (Slika 3.10). Uz to, u drugu listu spremaju se kontrole vozila (vrijednost zakreta upravljača i vrijednost pritiska papučice gasa) u odgovarajućem trenutku i redni broj slike kojoj odgovaraju. Spremanje kontrola vozila i slika u liste odvija se prilikom svake promjene okvira unutar *game_loop()* funkcije koja upravlja cijelom simulacijom (Slika 3.11). To omogućava rad u realnom vremenu tj. 30 FPS. Pri uključivanju snimanja, ako ne postoji direktorij za spremanje skupa podataka, dohvaća se ime trenutno učitanog CARLA grada i vremenski uvjet. Skupovi podataka tada imaju oznake formata *ime_grada_vremenski_uvjet_redni_broj* te je jednostavno pronaći željene podatke npr. *town04_clear_night_0*. Važno je napomenuti da se dohvaćena slika i kontrole prilikom spremanja u liste moraju duboko kopirati, što znači da se za svaki novi objekt alocira novi dio memorije. U suprotnom će sve spremljene slike i kontrole biti identične. Uz to, ugrađena funkcija za spremanje slika ne može raditi na 30 FPS na zadanom očvrsju. Stoga je implementirano spremanje slika u listu te spremanje svih slika iz liste na disk u obliku *numpy* volumena (Slika 3.12). Kontrole vozila iz liste spremaju se u *.csv* datoteku.

```
def to_rgb_array(image):
    array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
    array = np.reshape(array, (image.height, image.width, 4))
    array = array[:, :, :3]
    return array
```

Slika 3.10: Programski kod funkcije za pretvaranje slike u RGB zapis

```
if recording:
    if not (os.path.exists(f"{os.path.dirname(os.path.abspath(__file__))}
    /{DATASET_NAME}")):
        params = sim_world.get_weather()
        DATASET_NAME = sim_world.get_map().name.split("/")[2].lower() + "_"
        for preset in find_weather_presets():
            if params == preset[0]:
                DATASET_NAME += preset[1].lower().replace(" ", "_")
        os.makedirs(f"{os.path.dirname(os.path.abspath(__file__))}
        /{DATASET_NAME}")
    if frame is not None:
        image_list.append(np.copy(frame))
        control_list.append(np.copy(control))
    if len(image_list) == 100: np_save();
```

Slika 3.11: Programski kod za stvaranje direktorija i spremanje dohvaćenih slika i kontrola vozila u CARLA simulatoru

```
def np_save():
    global image_list, DATASET_NUMBER, control_list
    cwd = os.path.dirname(os.path.abspath(__file__))
    np.save(f"{cwd}/{DATASET_NAME}/dataset{DATASET_NUMBER}",
    ↪  np.asarray(image_list))
    with open(f"{cwd}/{DATASET_NAME}/data.txt", "a") as f:
        for c in control_list:
            f.write(f"{c[0]},{c[1].steer},{c[1].throttle},{c[1].brake}\n")
    image_list, control_list = [], []
    DATASET_NUMBER += 1
```

Slika 3.12: Programski kod funkcije za spremanje skupa podataka iz CARLA simulatora

Za MetaDrive simulator potrebno je uključiti prednju kameru i postaviti rezoluciju dohvaćene slike kao na slici 3.4. Problem je predstavljalo stvaranje vlastitog vozila u zraku i padanje na cestu, zbog čega se u glavnoj petlji programa snimanje započinje tek nakon 20 sličica (Slika 3.13). Simulacija se završava kada vlastito vozilo dosegne kraj generirane ceste, zbog čega je iz liste slika uklonjeno zadnjih 50 unosa gdje cesta nije vidljiva. Svi podaci o vlastitom vozilu dobiveni su unutar *info* rječnika kojeg simulator generira prilikom svakog koraka izvršavanja glavne petlje. Vlastita funkcija *save_data()* sprema slike i kontrole vozila iz lista, a zasnovana je na ugrađenoj funkciji za spremanje slike (Slika 3.14). Zbog kraćih cesta u MetaDrive simulatoru, sve slike i kontrole vozila spremaju se na kraju vožnje.

```
# Unutar glavne petlje simulatora
if recording and i > 20:
    image_list.append(cam.get_image(env.vehicle))
    control_list.append((info["raw_action"][0], info["raw_action"][1],
    ↪  info["velocity"]))
# Nakon izlaza iz glavne petlje simulatora
if recording and (info['arrive_dest'] or info['max_step']):
    del image_list[-50:], control_list[-50:]
    save_data()
```

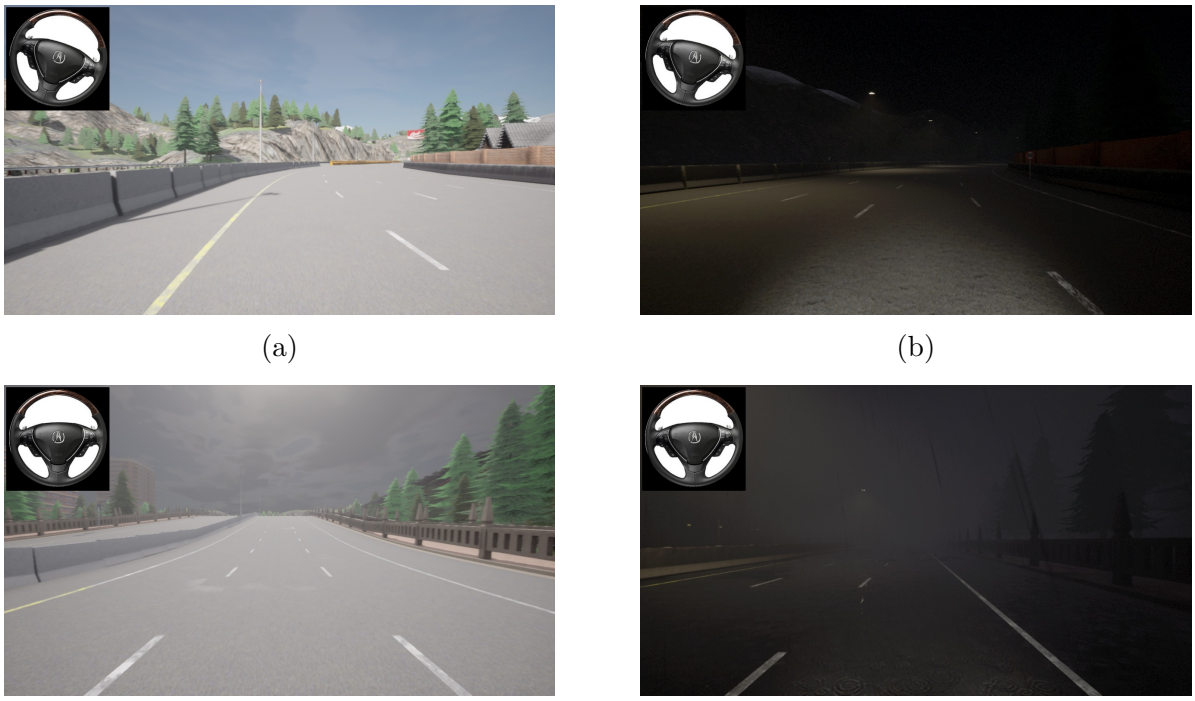
Slika 3.13: Programski kod za dohvaćanje i spremanje kontrola vozila i slike s prednje kamere u MetaDrive simulatoru

```
def save_data():
    global image_list, image_counter, control_list
    with open(f"{folder}/data.txt", "a") as file:
        for img, c in zip(image_list, control_list):
            img.write(f"{folder}/{image_counter}.jpg")
            file.write(f"{image_counter},{c[0]},{c[1]},{c[2]}\n")
```

```
            image_counter += 1
    image_list, control_list = [], []
```

Slika 3.14: Programski kod funkcije za spremanje skupa podataka iz MetaDrive simulatora

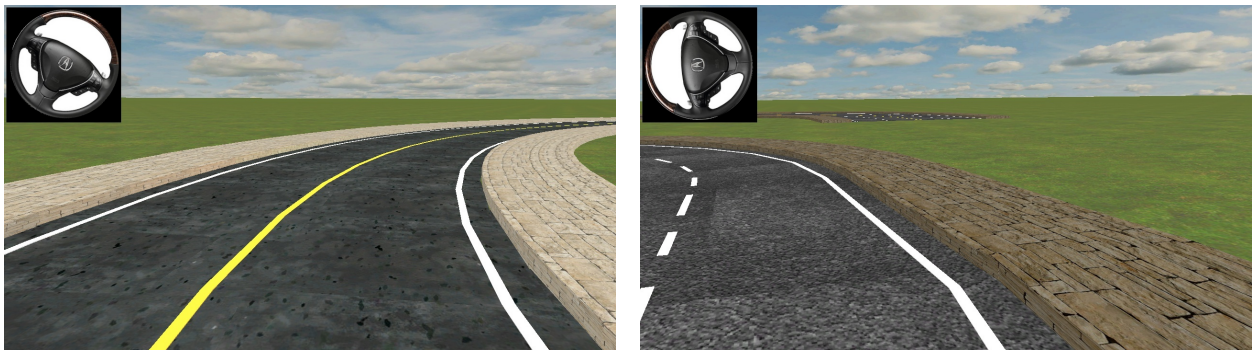
Ukupno je snimljeno 83600 slika u CARLA simulatoru raspoređenih u 27 vožnji u pet virtualnih gradova. Korišteni su dijelovi gradova broj 4, 5, 6, 7 i 10 gdje se ne pojavljuju raskrižja, po uzoru na [9], a podaci su snimljeni u različitim vremenskim uvjetima po danu, noći i u zalazak sunca. U MetaDrive simulatoru snimljeno je 77396 slika u 30 vožnji na nasumično generiranim cestama. Različite ceste generirane su postavljanjem inicijalizacijske vrijednosti proceduralnog generatora (engl. *seed*). MetaDrive simulator u trenutku pisanja ovog rada nema mogućnost promjene vremenskih uvjeta ili doba dana, stoga su korištene dodatne teksture ceste. Sve snimljene slike imaju rezoluciju 1280×720 elementa slike, a vrijednosti koje ukazuju na kutove zakreta upravljača vozila zapisane su kao decimalni brojevi s pomičnim zarezom u intervalu [-1, 1], što odgovara maksimalnom zakretu upravljača vozila unutar odabranih simulatora u obje strane (lijevo i desno). Neki od prikupljenih podataka mogu se vidjeti na slikama 3.15 i 3.16.



(a) (b)

(c) (d)

Slika 3.15: Podaci iz CARLA simulatora: (a) vedro podne, vrijednost zakreta 0.03653492033481598, (b) vedra noć, vrijednost zakreta 0.028466051444411278, (c) oblačni zalazak, vrijednost zakreta -0.00047324676415883005, (d) kišna noć, vrijednost zakreta -0.002709166146814823



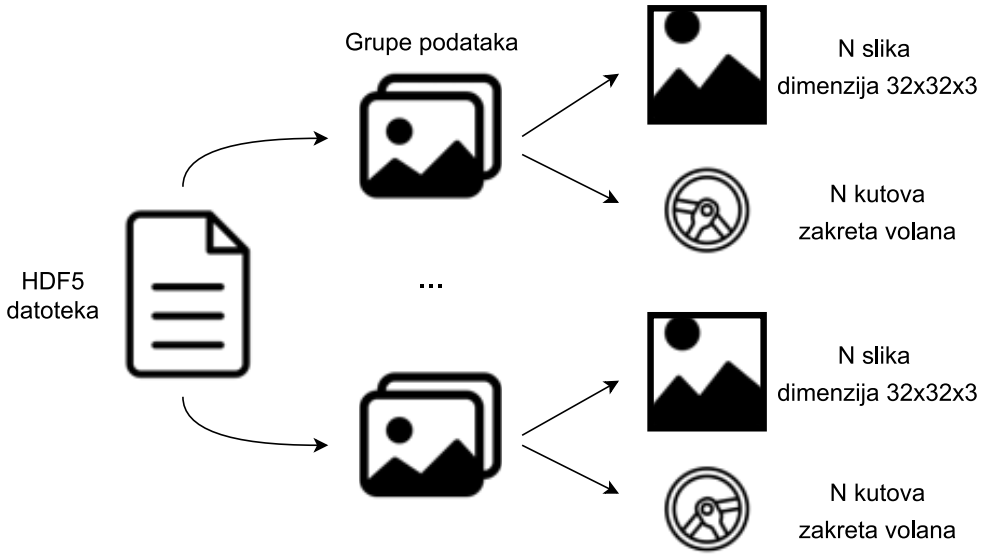
(a) (b)

Slika 3.16: Podaci iz MetaDrive simulatora: (a) originalna tekstura ceste, vrijednost zakreta 0.06897040218732774, (b) promijenjena tekstura ceste, vrijednost zakreta -0.17987288070762983

3.3.1. Oblikovanje vlastite baze podataka pomoću HDF5 datoteka za skupove podataka

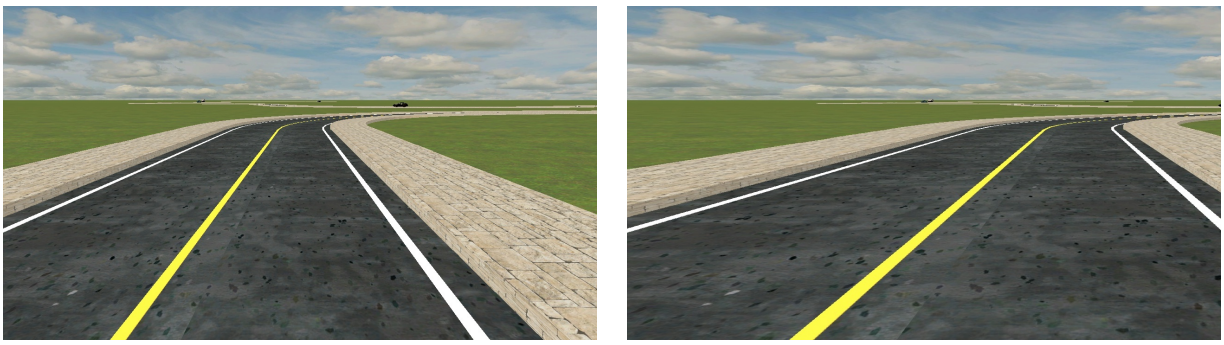
Hijerarhijski format datoteka (engl. *Hierarchical Data Format - HDF*) skup je standarda za pohranu i organizaciju velikih skupova podataka. Podržava višedimenzionalne skupove podataka, gdje svaki element u skupu može biti kompleksan objekt. Također, podržava podjelu skupova podataka u grupe te dodavanje opisa radi lakšeg snalaženja. Vlastita baza podataka sastoji se od više datoteka HDF5 formata koje sadrže sve skupove podataka iz pojedinog simulatora u formatu koji odgovara ulazu implementirane neuronske mreže. Time je smanjeno vrijeme predobrade podataka za treniranje neuronske mreže. Podaci su organizirani u grupe koje predstavljaju pojedine snimke tj. vožnje. Unutar svake grupe nalaze se parovi (X, y) gdje X predstavlja obrađenu sliku s prednje kamere vozila, a y vrijednost koja predstavlja kut zakreta upravljača (Slika 3.17).

Funkcija *make_h5_file()* služi za kreiranje *.h5* datoteke na način da prolazi kroz sve poddirektorije unutar predanog direktorija, izdvaja slike iz *numpy* volumena, skalira ih i pridružuje im vrijednosti zakreta upravljača te ih smješta u grupe odgovarajućim redoslijedom. Implementacija *make_h5_file()* koja radi s podacima iz CARLA simulatora dana je u prilogu P.3.5. Implementacija funkcije za kreiranje *.h5* datoteke slična je za podatke iz MetaDrive simulatora, ali se slike s prednje kamere vozila ne nalaze u *numpy* volumenu, nego su pojedinačno spremljene u *.png* formatu.



Slika 3.17: Struktura HDF5 datoteke

Po uzoru na [9, 10], filtriranjem je smanjen broj podataka gdje je vrijednost zakreta upravljača vozila blizu nule, kako neuronska mreža ne bi imala sklonost prema vožnji ravno te je odbačen gornji dio slike na kojemu se ne nalazi cesta jer ne nosi informacije korisne za rješavanje danog zadatka. Implementirane su augmentacije translacija i transformacija kuta gledanja (engl. *viewpoint transformation*) koje simuliraju odstupanja vozila od sredine prometne trake jer izvorni skup nije sadržavao takve situacije (Slika 3.18).

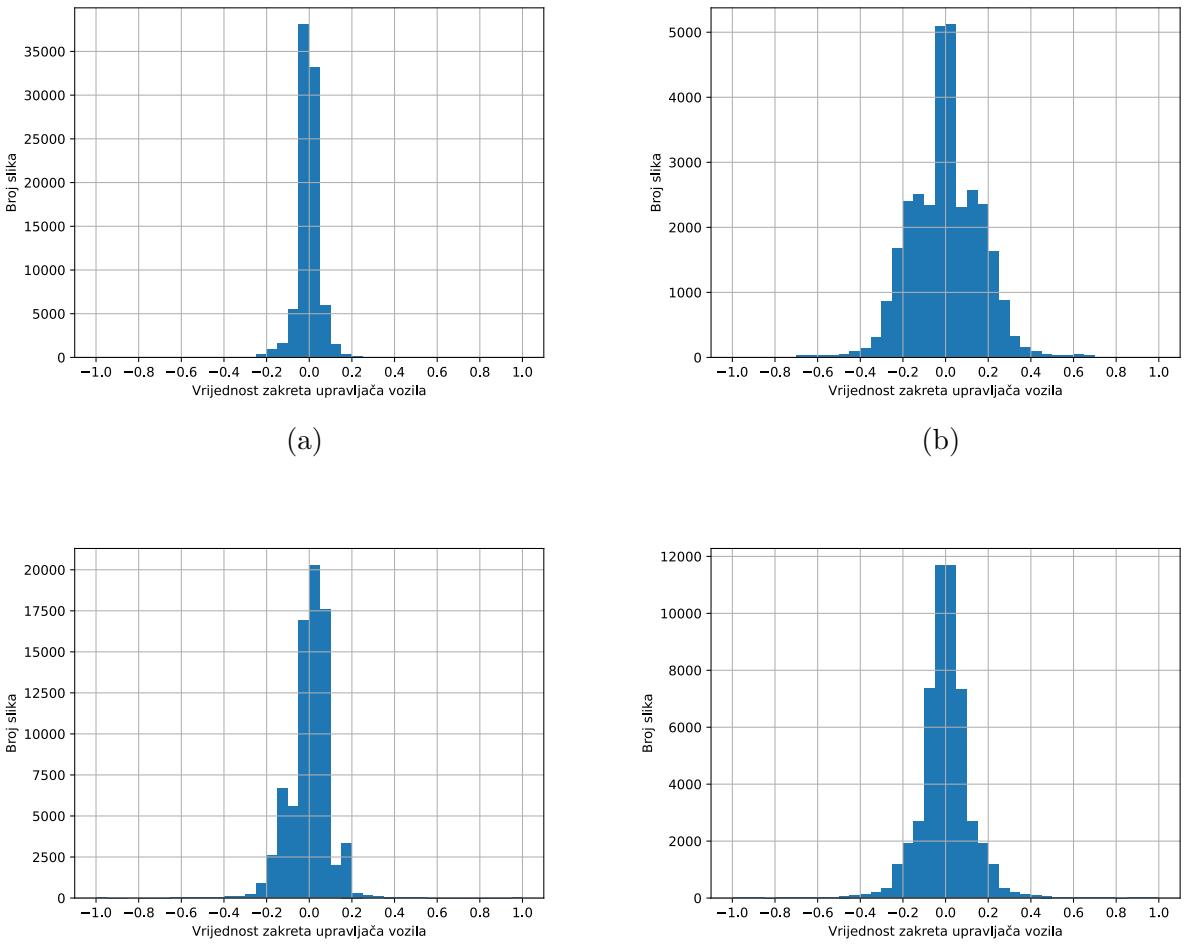


(a) (b)

Slika 3.18: Transformacija kuta gledanja na podacima iz MetaDrive simulatora: (a) originalni okvir, (b) okvir s transformacijom kuta gledanja

Augmentiranim podacima korigirana je vrijednost zakreta upravljača kako bi neuronska mreža naučila vratiti vlastito vozilo u odgovarajuću prometnu traku. Prilikom treniranja i testiranja neuronske mreže nije zabilježena razlika između treniranja s translacijom i treniranja s transformacijom kuta gledanja, no zabilježeni su lošiji rezultati u slučaju kada

je augmentacija podataka izostavljena. Histogrami raspodjele vrijednosti zakreta upravljača prije i nakon filtriranja i augmentacija prikazani su na slici 3.19. Filtrirani i augmentirani skupovi podataka iz CARLA i MetaDrive simulatora sadrže 35516, odnosno 51480 obrađenih podataka. Skup trening podataka iz CARLA simulatora sadrži podatke iz gradova 4, 5, 6 i 10, dok se nefiltrirani i neaugmentirani podaci iz grada broj 7 koriste za validaciju i testiranje. Skup trening podataka iz MetaDrive simulatora sastoji se od podataka iz vožnji gdje je početna vrijednost proceduralnog generatora postavljena na vrijednosti iz intervala [1, 29], dok nefiltrirani i neaugmentirani podaci iz vožnje gdje je početna vrijednost proceduralnog generatora jednaka 0 služe za validaciju i testiranje treniranih modela neuronskih mreža.



Slika 3.19: Histogrami vrijednosti zakreta upravljača vozila za: (a) originalno prikupljeni skup podataka iz CARLA simulatora, (b) skup podataka iz CARLA simulatora nakon filtriranja i augmentacije, (c) originalno prikupljeni skup podataka iz MetaDrive simulatora, (d) skup podataka iz MetaDrive simulatora nakon filtriranja i augmentacije

3.4. Treniranje predloženog algoritma za autonomnu vožnju na prikupljenim podacima iz simulatora

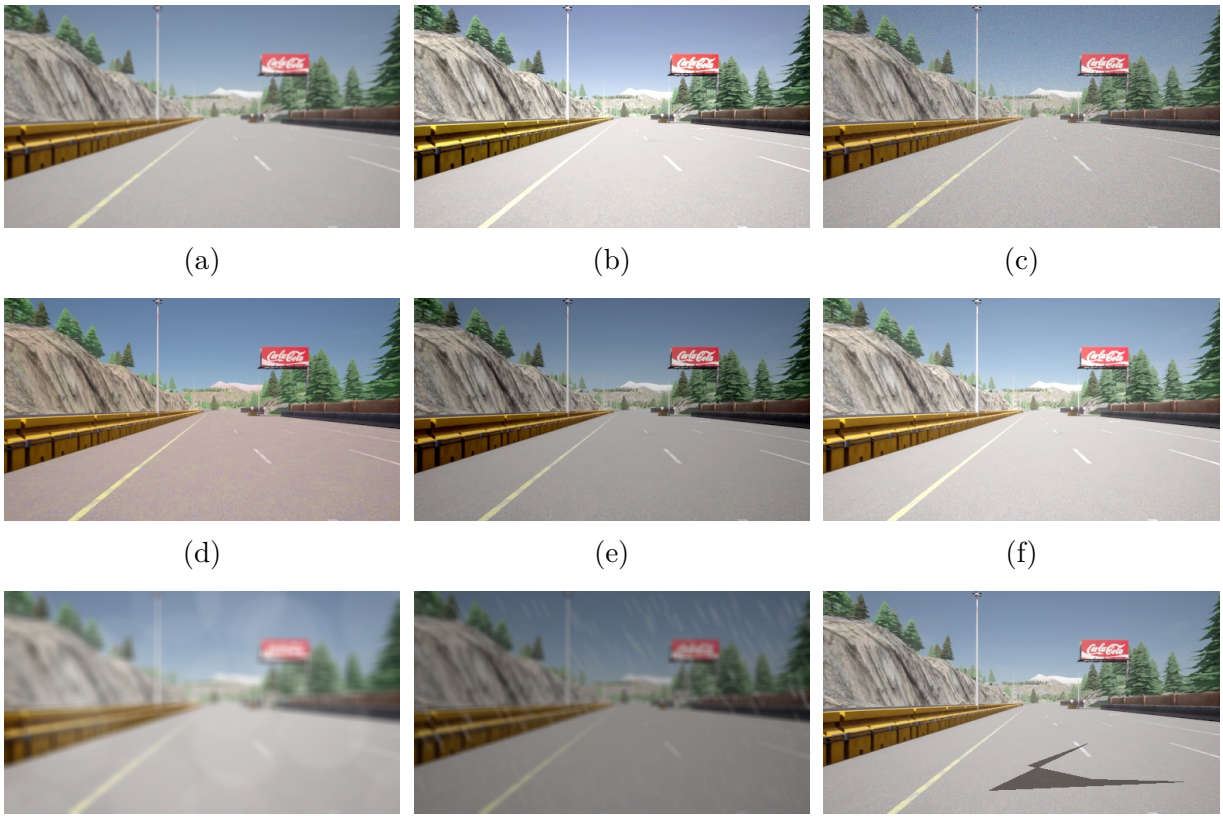
Kako bi rezultati predloženog algoritma mogli biti na odgovarajući način vrednovani, uz predloženu neuronsku mrežu implementirane su PilotNet neuronska mreža iz [9] i vlastita LSTM neuronska mreža realizirana po uzoru na [10, 12], s razlikom u korištenju vremenski raspoređenog sloja (engl. *Time Distributed*) iz Keras biblioteke koji omogućava primjenu predanog sloja na svaki vremenski odsječak iz ulazne sekvence. Implementacija opisane LSTM neuronske mreže dana je u prilogu P.3.6. Za treniranje predložene neuronske mreže na podacima iz *.h5* datoteka bilo je potrebno realizirati objekte za učitavanje podataka, tzv. generatore. Implementirana su dva generatora. Prvi generator učitava pojedinačne slike i koristi se za treniranje vlastitog algoritma i PilotNet neuronske mreže, a drugi učitava nizove slika i koristi se za treniranje implementirane LSTM neuronske mreže. Implementacija prvog opisanog generatora nalazi se u prilogu P.3.7. Klasa generatora mora sadržavati funkcije za inicijalizaciju, dohvaćanje duljine generatora i dohvaćanje podataka. Funkcija za inicijalizaciju služi za postavljanje putanje do *.h5* datoteke, duljine niza podataka (engl. *batch size*), željenih augmentacija, odabir miješanja podataka na kraju epohe i postavljanje praga. Prag se koristi za odabir ulaznih podataka gdje je vrijednost zakreta upravljača vozila veća od zadanog praga. Treniranjem modela prvo na podacima gdje je vrijednost zakreta upravljača veća, ostvareni su bolji rezultati u zavojitim dijelovima ceste. Duljina generatora količnik je ukupnog broja podataka i duljine niza podataka. Podaci iz *.h5* datoteke dohvaćaju se pomoću indeksa dobivenih iz funkcije *get_indices()* koja prima nazive grupa iz *.h5* datoteke za testni i validacijski skup i vraća indekse svih podataka iz odabranih vožnji (Slika 3.20). Trening skup iz CARLA simulatora sadrži 35516, a trening skup iz MetaDrive simulatora sadrži 51480 podataka. Oba validacijska skupa sadrže po 3400 podataka, a oba testna skupa sadrže po 3002 podatka.

```
def get_indices(f, train_ds, val_ds):
    train_idx, val_idx = [], []
    for ds in train_ds:
        indices = range(f[ds]["X"].shape[0])
        train_idx.append([[ds, i] for i in indices])
    for ds in val_ds:
        indices = range(f[ds]["X"].shape[0])
        val_idx.append([[ds, i] for i in indices])
    train_idx = np.concatenate(train_idx)
```

```
    val_idx = np.concatenate(val_idx)
    return train_idx, val_idx
```

Slika 3.20: Programski kod funkcije za dohvaćanje indeksa iz *.h5* datoteke

Slike učitane iz *.h5* datoteka moguće je augmentirati predavanjem liste augmentacijskih objekata iz biblioteke *Albumentations*. Korištene augmentacije simuliraju različite uvjete u kojima se vozilo može naći, a uključuju nasumično dodavanje sjena, kiše ili magle, dodavanje šuma, promjene svjetline i kontrasta te zamućivanje i izoštravanje slike (Slika 3.21). Funkcija za dohvaćanje podataka vraća niz parova obrađenih slika i odgovarajućih vrijednosti zakreta upravljača. Miješanje skupa podataka na kraju epohe izvodi se pomoću indeksa kako bi slike uvijek ostale povezane s odgovarajućom vrijednosti zakreta upravljača vozila. Objekt za učitavanje podataka za treniranje LSTM neuronske mreže slijedi iste principe, no obrađuju se nizovi podataka određene duljine umjesto pojedinačnih parova slika i vrijednosti zakreta upravljača.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Slika 3.21: Implementirane augmentacije podataka: (a) zamućivanje, (b) promjena boje, (c) dodavanje šuma, (d) promjena zasićenja, (e) promjena svjetline, (f) promjena kontrasta, (g) dodavanje magle, (h) dodavanje kiše, (i) dodavanje sjena

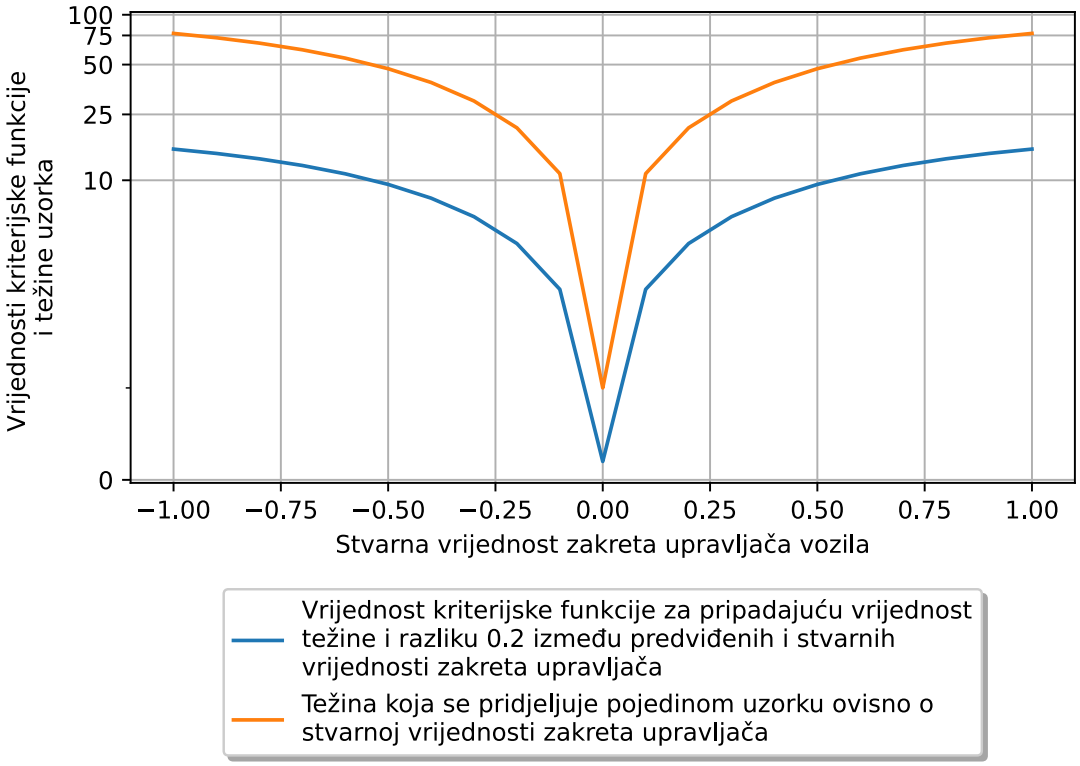
Treniranje predloženog algoritma autonomne vožnje izvedeno je pomoću ugrađene Keras funkcije *fit()* koja prima ulazne podatke, broj epoha, validacijske podatke i povratne (engl. *callback*) funkcije. Ulazni i validacijski podaci predaju se u obliku prethodno opisanih generatorskih objekata koji *fit()* funkciji predaju nizove podataka zadane duljine. Implementirane su povratne funkcije za smanjivanje stope učenja, zaustavljanje treniranja, ispis grafova vrijednosti kriterijske funkcije i spremanje modela nakon svake epohe (Slika 3.22). Stopa učenja smanjuje se za 10 puta ako model neuronske mreže prestane učiti tj. ako se vrijednost kriterijske funkcije na validacijskim podacima ne promijeni za više od 0.0001 unutar 15 epoha. Treniranje modela zaustavlja se ako se vrijednost kriterijske funkcije nije promijenila u zadnjih 20 epoha. *Tensorboard* objekt služi za prikazivanje grafova vrijednosti kriterijske funkcije na trening i validacijskim podacima.

```
stop_early = EarlyStopping(monitor='val_loss', patience=20)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=15,
↪  min_lr=1e-7, verbose=1, mode='min')
tb = TensorBoard(log_dir="./logs/{model.name}")
cp = ModelCheckpoint('{epoch:02d}_{val_loss:.2f}.h5', save_freq='epoch')
```

Slika 3.22: Programski kod za postavljanje povratnih funkcija

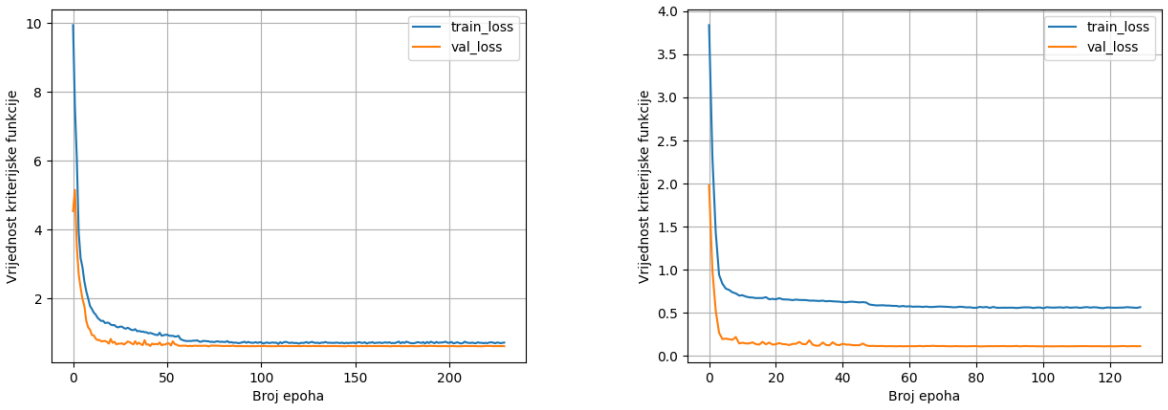
Modeli neuronskih mreža trenirani su s inkrementalno nižim pragom prvih 10 epoha, počevši s pragom 0.5. To znači da se svakom epohom povećava skup trening podataka dodavanjem podataka čija je vrijednost zakreta upravljača iznad zadanog praga. Uz to, implementirana je vlastita kriterijska funkcija koja većim vrijednostima zakreta upravljača pridodaje veći značaj pri računanju odstupanja tj. pogreške. Zasnovana je na otežanom srednjem apsolutnom odstupanju (engl. *Weighted Mean Absolute Error - WMAE*). Težine za svaku predikciju računaju se po formuli (3-2), gdje je y_{true} stvarna vrijednost zakreta upravljača. Na slici 3.23 vidljive su težine za stvarne vrijednosti zakreta upravljača u intervalu [-1, 1] te vrijednost kriterijske funkcije u slučaju da sve procijenjene vrijednosti odstupaju za 0.2 od stvarnih vrijednosti zakreta upravljača. Implementacijom praga u treniranju modela i težina u kriterijskoj funkciji smanjene su pogreške u procjeni vrijednosti zakreta upravljača vozila pri većim zavojima, što dovodi do boljih rezultata pri testiranju u simulatorima i na vlastitom skupu podataka.

$$w = tanh(|y_{true}|) * 100 + 1 \tag{3-2}$$



Slika 3.23: Graf težina dodijeljenih stvarnim vrijednostima zakreta upravljača i vrijednosti kriterijske funkcije uz razliku 0.2 između stvarnih i predviđenih vrijednosti zakreta upravljača

Pri treniranju na podacima iz CARLA simulatora, najbolji rezultati ostvareni su nakon 200 epoha, dok su pri treniranju na podacima iz MetaDrive simulatora najbolji rezultati ostvareni nakon 100 epoha. Pretpostavka je da zbog manjeg broja tekstura i nedostatka raznolikosti vremenskih uvjeta u MetaDrive simulatoru u usporedbi s CARLA simulatorom, neuronske mreže brže nauče izlučiti relevantne značajke iz podataka. Daljnjim treniranjem povećava se pogreška na validacijskom skupu tj. vrijednost kriterijske funkcije, dok se pogreška na trening skupu podataka smanjuje ili stagnira. Drugim riječima, tada se model neuronske mreže previše prilagođava trening podacima (engl. *overfitting*). Vrijednosti kriterijske funkcije na skupovima za treniranje i validaciju prikazane su na slici 3.24. *Train_loss* označava vrijednost kriterijske funkcije na trening skupu, a *val_loss* označava vrijednost kriterijske funkcije na validacijskom skupu podataka.

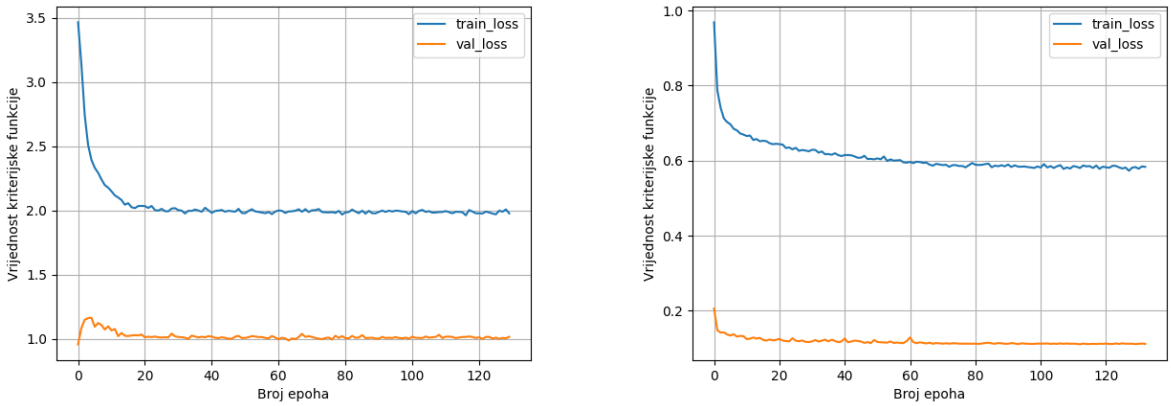


(a) (b)

Slika 3.24: Vrijednost kriterijske funkcije pri treniranju na podacima iz (a) CARLA simulatora i (b) MetaDrive simulatora

3.4.1. Treniranje modela zasnovano na prijenosnom učenju

Prijenosno učenje je metoda u strojnom učenju gdje se model neuronske mreže koji je razvijen za specifičan zadatak prenamjenjuje za drugi srodni zadatak. Modeli neuronskih mreža trenirani prijenosnim učenjem predtrenirani su na određenom skupu podataka te se specijaliziraju za rješavanje sličnih zadataka dotreniranjem na novom skupu podataka. Korištenjem predtreniranih modela smanjuje se vrijeme potrebno za razvoj rješenja za željeni problem koje će ostvariti dobre rezultate. U ovom radu, modeli trenirani prijenosnim učenjem predtrenirani su na skupu podataka iz jednoga simulatora, a nakon toga dotrenirani i testirani na podacima iz drugog simulatora. Inicijalizirani su modeli neuronskih mreža koji su ostvarili najbolje rezultate na skupu podataka iz jednog simulatora te je treniranje nastavljeno na skupu podataka iz drugog simulatora, dok nije došlo do stagnacije vrijednosti kriterijske funkcije uz prethodno smanjivanje stope učenja, kao i kod modela treniranih na svakom skupu podataka zasebno (detaljnije opisano ranije). Do prestanka učenja dolazi nakon 40 epoha u slučaju dotreniranja na CARLA skupu podataka te nekon 100 epoha u slučaju dotreniranja na MetaDrive skupu podataka. Modeli dobiveni nakon navedenih brojeva epoha korišteni su za testiranje. Vrijednosti kriterijske funkcije prilikom dotreniranja prikazane su na slici 3.25. *Train_loss* označava vrijednost kriterijske funkcije na trening skupu, a *val_loss* označava vrijednost kriterijske funkcije na validacijskom skupu podataka.



(a) (b)

Slika 3.25: Vrijednost kriterijske funkcije pri dotreniranju modela na podacima iz (a) CARLA simulatora i (b) MetaDrive simulatora

3.5. Upute za pokretanje programskog koda

Kako bi se izvršio opisani programski kod potrebno je preuzeti i instalirati verziju 3.9 programskog jezika Python sa službene stranice `https://www.python.org/downloads/`.

Korištenjem terminala i *Pip* alata za upravljanje bibliotekama potrebno je dohvatiti i instalirati sljedeće biblioteke:

- Tensorflow 2.7.0
- Keras 2.7.0
- Albumentations 1.1.0
- Numpy 1.22.3
- OpenCV 4.5.4.60
- Matplotlib 3.5.1
- H5py 3.6.0

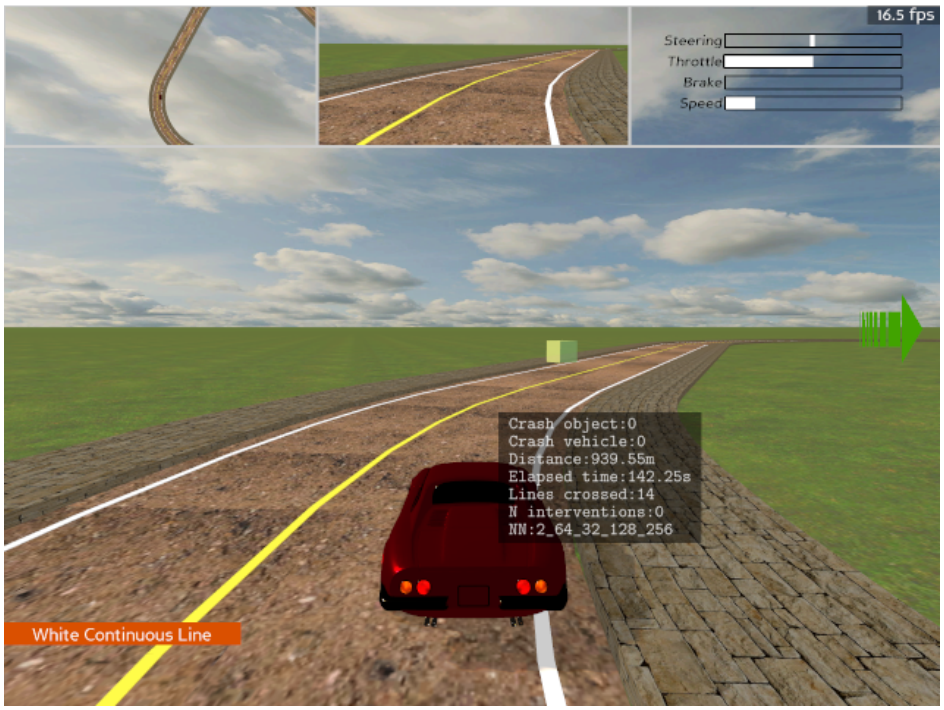
CARLA simulator moguće je preuzeti sa službene GitHub stranice `https://github.com/carla-simulator/carla/releases`, a u ovom radu korištena je verzija 0.9.13. MetaDrive simulator moguće je instalirati korištenjem GitHub servisa i *Pip* alata pozivanjem sljedećih naredbi u terminalu:

```
git clone https://github.com/metadriverse/metadrive.git
cd metadrive
pip install -e .
```

Za treniranje i testiranje implementiranih neuronskih mreža potrebno je preuzeti pripremljene skupove podataka koji su dostupni na upit. Python skripte koje definiraju slojeve i strukturu modela te treniranje i testiranje na pripremljenim skupovima podataka dane su kao elektronički prilozi P.3.8, P.3.9, P.3.10, P.3.11 i P.3.12. Potrebno je pokrenuti skriptu *train_test.py* koja će prema postavkama opisanim u ovom radu trenirati definirane modele neuronskih mreža te ih spremiti u odvojeni direktorij i prikazati rezultate evaluacije na validacijskim skupovima. Trenirane modele moguće je testirati u MetaDrive simulatoru pokretanjem *md_main.py* skripte uz postavljanje konstanti *MODEL_PATH*, koja pokazuje na putanju do željenog modela neuronske mreže i *SEED*, koja predstavlja početnu vrijednost proceduralnog generatora, upisivanjem sljedeće naredbe u terminal:

```
python md_main.py --modelpath putanja/do/modela/neuronske/mreže --seed broj
```

Nakon pokretanja simulatora potrebno je pritisnuti tipku *E* kako bi se učitao zadani model neuronske mreže i pokrenulo testiranje. Pri testiranju, rezultati se prikazuju na grafičkom sučelju simulatora, a po završetku testiranja, ostvareni rezultati ispisuju se u terminal (Slika 3.26).



Slika 3.26: Grafičko sučelje u MetaDrive simulatoru s ispisom broja prelazaka kolničkih crta, broja intervencija, prijeđene udaljenosti i proteklog vremena

Za testiranje modela neuronske mreže u CARLA simulatoru potrebno je pokrenuti poslužitelj *CarlaUE4.exe*, učitati željeno okruženje koristeći *config.py* skriptu preuzetu sa simulatorom i pokrenuti *control.py* skriptu s predanim argumentom koji sadrži putanju do modela. Nakon pokretanja CARLA poslužitelja i klijentske skripte, također je potrebno pritisnuti tipku *E* kako bi se učitao zadani model. Tijekom testiranja, rezultati se prikazuju na grafičkom sučelju simulatora te se ispisuju u terminal po završetku izvođenja klijentske skripte (Slika 3.27). Navedene skripte pokreću se iz terminala na sljedeći način:

```
python config.py -m ime_grada
python control.py --modelpath putanja/do/modela/neuronske/mreže
```

Slika 3.27: Grafičko sučelje u CARLA simulatoru s ispisom broja prelazaka kolničkih crta, broja intervencija, prijeđene udaljenosti i proteklog vremena

4. EVALUACIJA PERFORMANSI PREDLOŽENOG ALGORITMA ZA AUTONOMNU VOŽNJU ZASNOVANOG NA PROCJENI KUTA ZAKRETA UPRAVLJAČA

Treniranje i testiranje opisanih neuronskih mreža izvedeno je na osobnom računalu s Windows 10 Pro operacijskim sustavom, Intel Core i7-6700 procesorom, NVIDIA GeForce GTX 1070 Ti grafičkom karticom i 16GB radne memorije. Ukupno vrijeme treniranja konačnih implementacija neuronskih mreža iznosilo je 32 sata, od kojih je potrebno 7 sati i 6 minuta za treniranje svih *Swin* transformerskih neuronskih mreža na skupu podataka iz CARLA simulatora, a 8 sati za treniranje svih *Swin* transformerskih neuronskih mreža na skupu podataka iz MetaDrive simulatora. Ostatak vremena potreban je za treniranje PilotNet i LSTM neuronskih mreža na MetaDrive i CARLA skupovima podataka. Testiranje treniranih modela neuronskih mreža podijeljeno je na dvije faze: testiranje na podacima iz simulatora i testiranje unutar simulatora. Testiranjem na podacima iz simulatora dobivene su standardne metrike za usporedbu s ostalim algoritmima za autonomnu vožnju zasnovanim na procjeni kuta zakreta upravljača vozila. Testiranjem u simulatorima ispitane su performanse modela neuronskih mreža u virtualnom okruženju, što daje uvid u potencijalne nedostatke i moguća unaprjeđenja algoritama prije testiranja u stvarnom svijetu.

4.1. Testiranje predloženog algoritma na vlastitoj bazi podataka

Kako bi trenirani modeli neuronskih mreža mogli biti međusobno uspoređeni, unutar ugrađene *model.compile()* funkcije zadane su standardne metrike za regresijske probleme: srednje kvadratno odstupanje (engl. *mean squared error - MSE*) i srednje apsolutno odstupanje tj. pogreška (engl. *mean absolute error - MAE*). Navedene metrike računaju se na sljedeći način:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{4-1}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|, \tag{4-2}$$

gdje je n ukupan broj promatranih podataka, Y_i predstavlja stvarne vrijednosti, a $\hat{Y}_i$ predstavlja procijenjene vrijednosti tj. izlaz neuronske mreže. Srednje kvadratno odstupanje

osjetljivije je na veća odstupanja predviđenih od stvarnih podataka (engl. *outliers*), no nekoliko predviđenih vrijednosti koje znatno odstupaju od stvarnih vrijednosti mogu znatno povećati izračunatu pogrešku iako su ostale predviđene vrijednosti vrlo blizu očekivanih vrijednosti te je zbog toga korišteno i srednje apsolutno odstupanje. Potrebno je napomenuti da je rezultat srednjeg kvadratnog odstupanja u mjernoj jedinici na kvadrat. Prikaz u originalnoj mjernoj jedinici moguće je dobiti korjenovanjem srednjeg kvadratnog odstupanja (engl. *Root Mean Square Error - RMSE*). Korištenjem ugrađenih Keras funkcija *predict()* i *evaluate()* na testnim skupovima podataka iz obaju odabranih simulatora dobiveni su rezultati prikazani tablicama 4.1 i 4.2. Testni skup s podacima iz MetaDrive simulatora sadrži 3002 podatka iz vožnji gdje je početna vrijednost proceduralnog generatora postavljena na 0, a cesta se sastoji od 2 prometne trake. Testni skup s podacima iz CARLA simulatora sadrži 3002 podatka iz vožnje u zalazak sunca u virtualnom gradu broj 7. Podaci iz testnih skupova nisu augmentirani.

Tablica 4.1: Rezultati različitih modela neuronskih mreža na testnom skupu iz MetaDrive simulatora (treniranih samo na MetaDrive trening skupu)

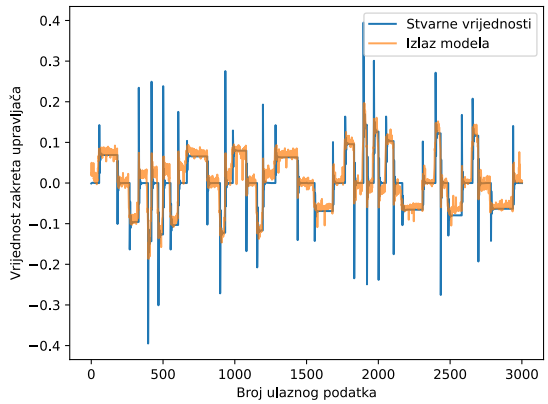
Model neuronske mreže	**MAE**	**MSE**
Swin1	0.01823	0.00092
Swin2	0.01707	0.00088
Swin3	0.01937	0.00097
Swin4	0.02125	0.00105
PilotNet [9]	**0.01128**	**0.00065**
LSTM	0.01393	0.00076

Tablica 4.2: Rezultati različitih modela neuronskih mreža na testnom skupu iz CARLA simulatora (treniranih samo na CARLA trening skupu)

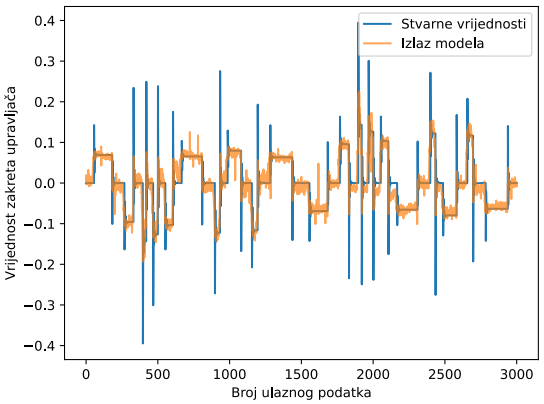
Model neuronske mreže	**MAE**	**MSE**
Swin1	0.06691	0.00820
Swin2	0.06373	0.00675
Swin3	0.06375	0.00728
Swin4	0.06453	0.00743
PilotNet [9]	**0.04883**	**0.00496**
LSTM	0.06793	0.00743

Na osnovu rezultata iz tablica 4.1 i 4.2 te usporedbom izlaznih vrijednosti modela sa stvarnim vrijednostima (Slike 4.1 i 4.2), vidljivo je da svi modeli vrlo uspješno rješavaju dani problem. U oba testiranja, na CARLA i MetaDrive skupovima podataka, najbolje

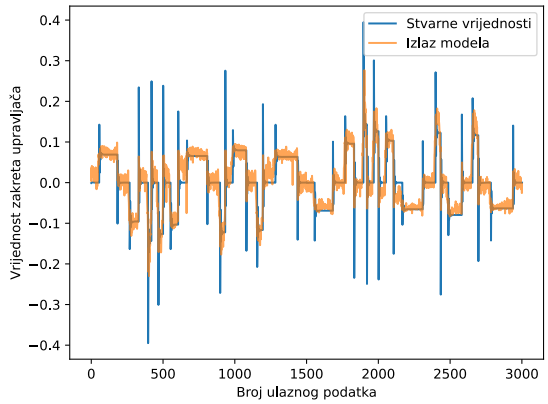
rezultate postigao je PilotNet model neuronske mreže, dok je drugi najbolji rezultat na MetaDrive skupu podataka dao model zasnovan na LSTM, a drugi najbolji rezultat na CARLA skupu podataka postigao je Swin2 model. *Swin* transformerski modeli bolje rezultate ostvarili su na podacima iz MetaDrive simulatora u usporedbi s istim modelima treniranim na podacima iz CARLA simulatora. Dobiveno srednje kvadratno odstupanje je 6 do 9 puta manje na podacima iz MetaDrive simulatora. To se može povezati s brojem podataka u skupu za treniranje te raznovsnošću danih podataka. Moguće je da bi proširenjem skupa podataka rezultati u obama simulatorima bili podjednaki, a modeli zasnovani na *Swin* transformerskim neuronskim mrežama dali bolje rezultate od modela zasnovanih na konvolucijskim i povratnim neuronskim mrežama, kao što je opisano u [26]. Također, izlazi modela neuronskih mreža s transformerskim slojevima zašumljeniji su od izlaza PilotNet modela, što se također može povezati s brojem dostupnih podataka za treniranje. Zašumljenost izlaza očituje se kao velik broj malih odstupanja izlaza modela od stvarne vrijednosti zakreta upravljača, a vidljiva je na slikama 4.2(a), 4.2(b), 4.2(c) i 4.2(d). Izlaz modela moguće je filtrirati niskopropusnim filtrom, čime se eliminira zašumljenost i poboljšava rezultat srednjeg kvadratnog odstupanja. Zašumljenost izlaza nije predstavljala problem pri izvođenju testova u simulatorima, stoga filtriranje izlaza nije implementirano u konačnoj verziji vlastitog algoritma za autonomnu vožnju zasnovanog na procjeni kuta zakreta upravljača vozila.



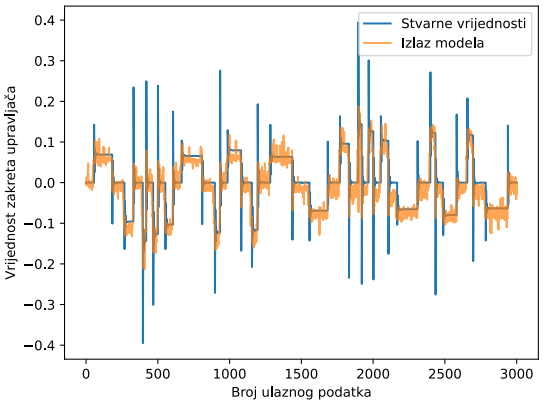
(a)



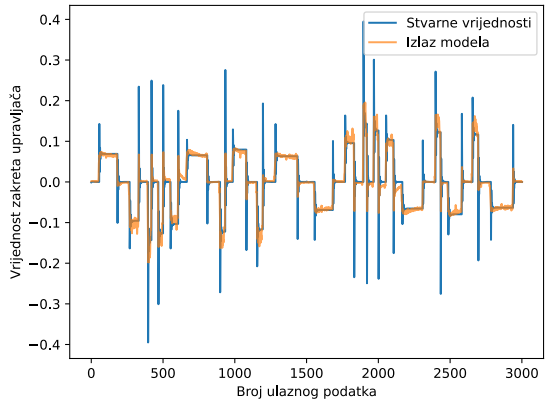
(b)



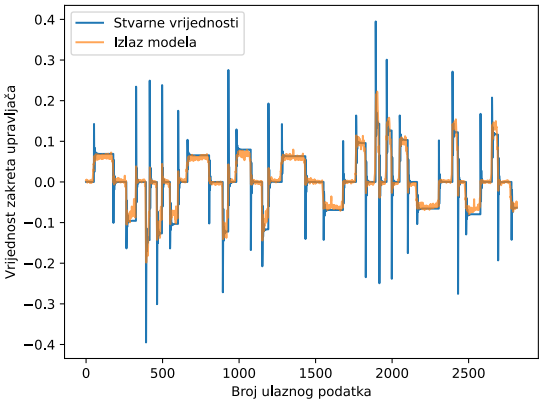
(c)



(d)

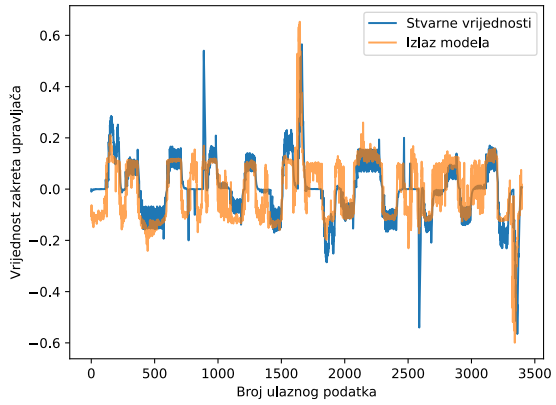


(e)

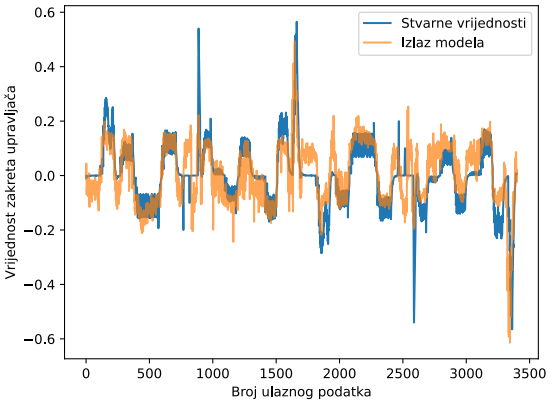


(f)

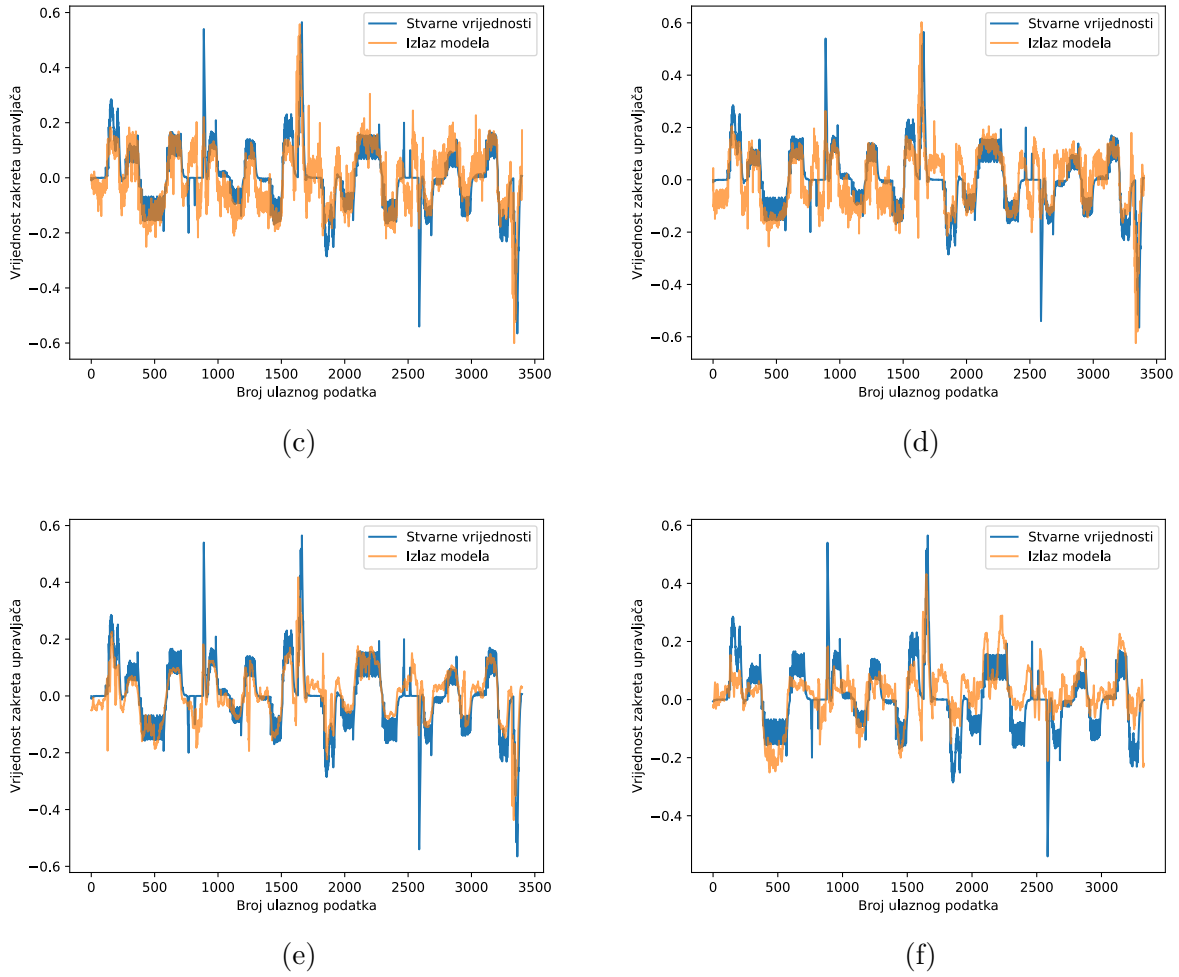
Slika 4.1: Usporedba izlaza različitih modela i stvarnih vrijednosti iz MetaDrive skupa podataka: (a) Swin1, (b) Swin2, (c) Swin3, (d) Swin4, (e) PilotNet, (f) LSTM



(a)



(b)



Slika 4.2: Usporedba izlaza različitih modela i stvarnih vrijednosti iz CARLA skupa podataka: (a) Swin1, (b) Swin2, (c) Swin3, (d) Swin4, (e) PilotNet, (f) LSTM

Uz standardne metrike, promatrano je i prosječno vrijeme inferencije pojedinog modela. Vrijeme inferencije označava vrijeme koje je neuronskoj mreži potrebno da obradi ulazne podatke i vrati odgovarajuće izlazne podatke te kao takvo izravno utječe na brzinu rada mreže i broj obrađenih podataka u jedinici vremena. U ovom radu vrijeme inferencije zabilježeno je korištenjem Python *time* biblioteke, točnije, *perf_counter()* funkcije. Važno je napomenuti da ovaj način mjerenja obuhvaća alokaciju memorije i premještanje podataka iz radne memorije u grafičku karticu, zagrijavanje grafičke kartice te sve procesorske pozive potrebne za inferenciju modela. Time je dobiveno prosječno vrijeme izvršavanja pojedinog modela na očvrsju opisanom na početku poglavlja (Tablica 4.3).

Tablica 4.3: Vremena inferencije različitih modela neuronskih mreža

Ime modela	**Prosječno vrijeme inferencije [s]**	**Brzina obrade podataka [FPS]**
Swin1	**0.0624**	**16.04**
Swin2	0.0691	14.48
Swin3	0.0632	15.83
Swin4	0.0670	14.92
PilotNet [9]	0.0693	14.43
LSTM	0.0919	10.88

Najveću brzinu obrade podatka postižu modeli Swin1 i Swin3 s približno 16 FPS, što je očekivano s obzirom na broj parametara navedenih modela. Neočekivan je rezultat Swin4 modela koji unatoč manjem broju parametara od Swin1 modela ima manju brzinu obrade podataka. Pretpostavka je da je struktura slojeva Swin1 modela, gdje slojevi s manje neurona prethode slojevima s više neurona, pogodnija za bržu obradu podataka. Svi *Swin* modeli brži su od PilotNet modela, što je moguće povezati s rezolucijom ulazne slike koju pojedini model obrađuje. Najmanju brzinu obrade podataka postiže LSTM model jer u isto vrijeme mora obraditi 5 uzastopnih okvira s prednje kamere vozila.

4.2. Testiranje predloženog algoritma u simulatorima

Za testiranje modela unutar odabranih simulatora implementirane su skripte *md_main.py* i *control.py* opisane u potpoglavlju 3.5. Obje skripte pri pokretanju iz terminala primaju putanju do odabranog modela neuronske mreže. Nakon pokretanja skripte postavlja se virtualno testno okruženje, a pritiskom na tipku E učitava se odabrani model neuronske mreže i započinje testiranje. U oba simulatora rezultati se tijekom testiranja prikazuju pomoću ugrađenog grafičkog sučelja simulatora, a nakon testiranja ispisuju se u terminal. Metrike koje se prate su broj prelazaka kolničkih crta, broj intervencija korisnika te prijeđeni put i proteklo vrijeme od početka testiranja. Mjerenje proteklog vremena implementirano je pomoću *time.perf_counter()* funkcije u obama simulatorima. Proteklo vrijeme dobiveno je kao suma svih razlika vrijednosti funkcije *time.perf_counter()* između dvaju uzastopnih okvira. Računanjem udaljenosti između lokacije vlastitog vozila u trenutnom i prošlom okviru dobivena je prijeđena udaljenost. Broj intervencija označava koliko puta je bilo potrebno preuzeti upravljanje vlastitim vozilom i vratiti ga u odgovarajuću prometnu traku, nakon čega upravljanje vozilom ponovno preuzima odabrani model neuronske mreže. Broj

intervencija povećava se kada se tijekom testiranja pritisne tipka E, tj. kada se upravljanje vozilom dodijeli korisniku. Broj prelazaka kolničkih crta povećava se kada vozilo dotakne rubnu ili razdjelnu crtu. To je realizirano provjerom doticanja kolničkih crta u prethodnom okviru te usporedbom sa stanjem u trenutnom okviru. Time je spriječena višestruka detekcija doticanja kolničkih crta, a moguće je implementirati i vremensku zadršku kako ne bi dolazilo do velikog broja detekcija ako vlastito vozilo nekoliko puta dotakne kolničku crtu unutar nekoliko uzastopnih okvira. Programski kod za brojanje prelazaka kolničkih crta u MetaDrive simulatoru prikazan je na slici 4.3. U CARLA simulatoru korišten je ugrađeni senzor za detekciju prelaska kolničkih crta sadržan u objektu *LaneInvasionSensor*.

```
on_line = env.vehicle.on_broken_line or
↪  env.vehicle.on_white_continuous_line or
↪  env.vehicle.on_yellow_continuous_line
        if last_step['on_line'] < on_line:
            infractions['line_crossed'] += 1
```

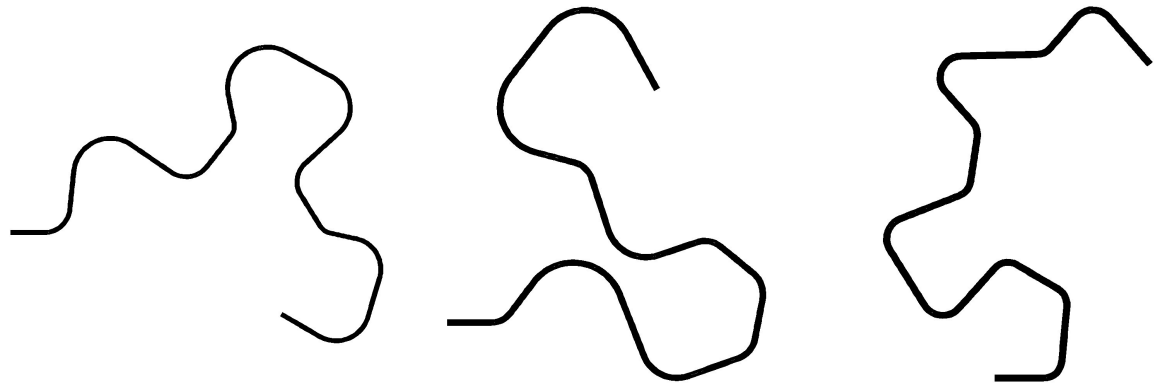
Slika 4.3: Programski kod za brojanje prelazaka kolničkih crta u MetaDrive simulatoru

Upravljanje brzinom vlastitog vozila prikazano je na slici 4.4, gdje varijabla *throttle* predstavlja trenutnu razinu pritiska papučice gasa i postavlja se nakon svake predikcije vrijednosti zakreta upravljača vozila, tj. za svaki novi okvir simulatora. Vrijednosti varijable *throttle* odabrane su empirijski, a matematička funkcija za određivanje vrijednosti varijable *speed* postavljena je tako da se vozilo u zavojima kreće manjom brzinom u odnosu na ravnu cestu. Uz to, maksimalna brzina kretanja vozila postavljena je na 50 km/h, što je 20 km/h više od maksimalne brzine ugrađenog MetaDrive autopilota. Povećanjem maksimalne brzine kretanja vozila ističu se pogreške modela neuronskih mreža koje bi ostale neprimijećene pri brzini kretanja autopilota. Slično upravljanje brzinom vozila implementirano je u CARLA simulatoru, a razlika je postavljanje vrijednosti pritiska papučice gasa isključivo na vrijednosti 0 ili 1 zbog drukčijeg ponašanja vozila u odnosu na MetaDrive simulator.

```
throttle = 0.0
if info["velocity"] < 50:
    speed = min(1 / (0.3 * abs(angle)), 50)
    if speed > info["velocity"]:
        throttle = 0.5
    else:
        throttle = -0.03
```

Slika 4.4: Programski kod za upravljanje brzinom vlastitog vozila u MetaDrive simulatoru

Testovi u MetaDrive simulatoru izvedeni su s trima različitim početnim vrijednostima proceduralnog generatora, što rezultira trima generiranim cestama (Slika 4.5). Uz navedeno, u dva testa promijenjena je zadana tekstura ceste (Slika 4.6), a u zadnjem testu smanjen je zadani minimalni radijus generiranih zavoja kako bi se ispitala generalizacijska svojstva modela. Za testiranje je korišten zadani model vozila *ferra* sa zadanom pozicijom prednje kamere. U opisanim testovima najbolje rezultate ostvarili su modeli Swin2, PilotNet i LSTM, koji testove s ugrađenom teksturom ceste mogu završiti bez prelaska kolničkih traka i intervencija (Slika 4.7). Svi modeli ostvaruju bolje rezultate na testovima s ugrađenom teksturom ceste nego na testovima s novim teksturama. Rezultati postaju lošiji većim vizualnim odstupanjem korištene teksture od ugrađene. Razlike u rezultatima pri korištenju drukčijih tekstura mogle bi se smanjiti proširenjem skupa podataka slikama s različitim teksturama ceste.

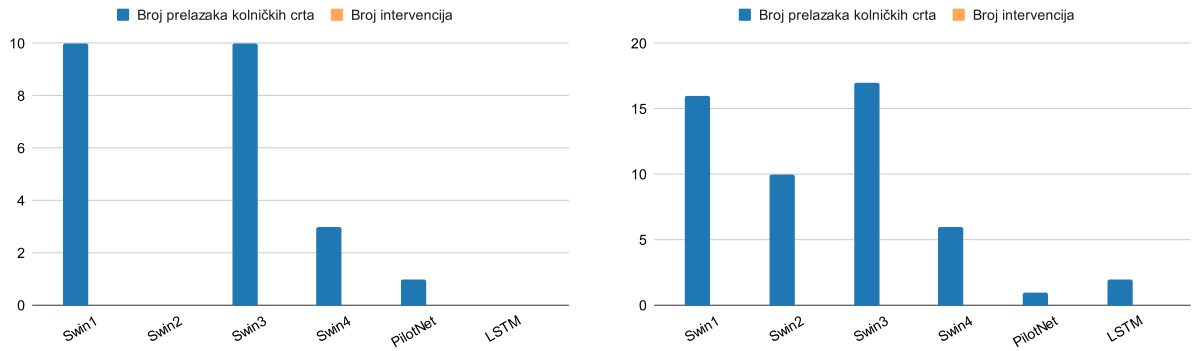


(a) (b) (c)

Slika 4.5: Testne ceste u MetaDrive simulatoru: (a) *seed*=100, (b) *seed*=200, (c) *seed*=300

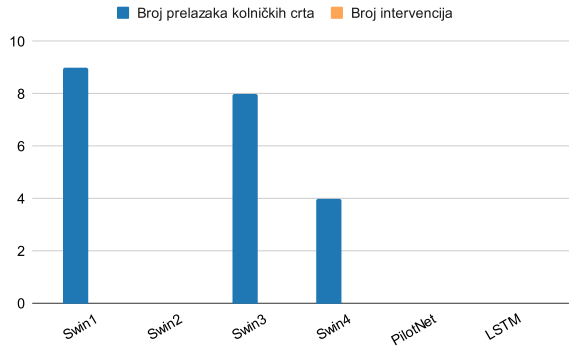
(a) (b) (c)

Slika 4.6: (a) Originalna tekstura ceste, (b) i (c) promijenjene teksture ceste u MetaDrive simulatoru

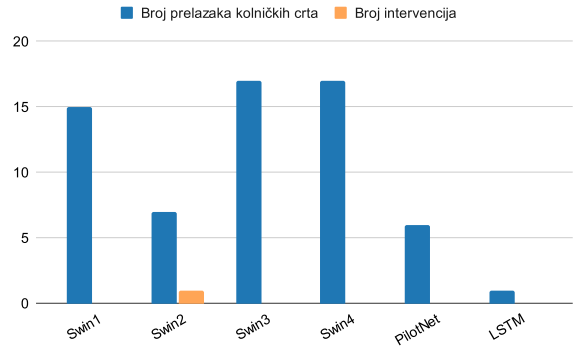


(a)

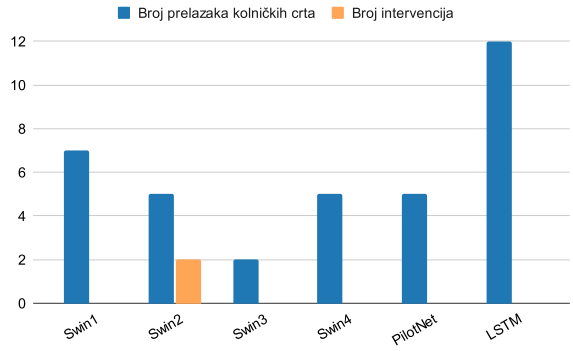
(b)



(c)



(d)



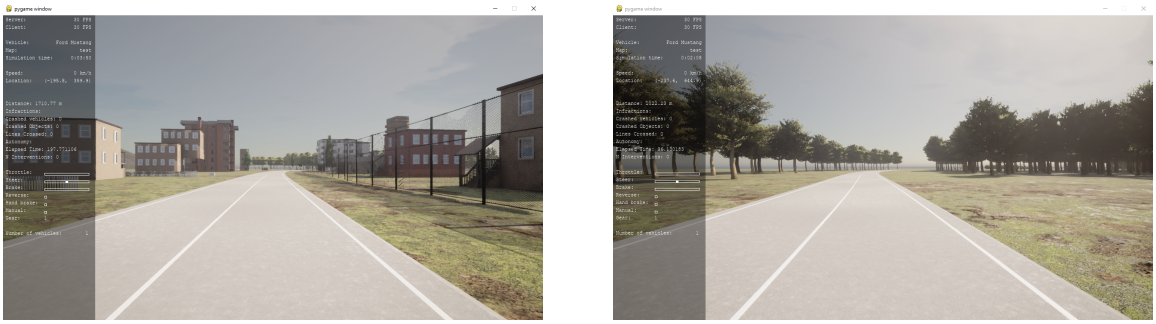
(e)

Slika 4.7: Rezultati testiranja različitih modela u MetaDrive simulatoru pri različitim postavkama: (a) *seed*=100, (b) *seed*=100 uz teksturu ceste sa slike 4.6(b), (c) *seed*=200, (d) *seed*=200 uz teksturu ceste sa slike 4.6(c), (e) *seed*=300 i smanjeni minimalni radijus zavoja

Testiranje u CARLA simulatoru izvedeno je u virtualnom okruženju koje je dostupno kao elektronički prilog P.3.13, a sastoji se od vožnje kroz naselje i izvan naselja po cesti s 2 prometne trake (Slika 4.8). Tri vremenska uvjeta u kojima je provedeno testiranje su vedro vrijeme po danu, vedro vrijeme po noći te jaka kiša u zalazak sunca. Test s ugrađenim postavkama za jaku kišu odabran je jer uz kišu sadrži maglu i lokve na cesti te kao takav

predstavlja najteži od provedenih testova. Za testiranje je korišten model Ford Mustang automobila, a lokacija prednje kamere vozila zadana je kao:

```
carla.Transform(carla.Location(x=+0.8 * bound_x, y=+0.0 * bound_y, z=1.3 *
↪  bound_z)), Attachment.Rigid)
```

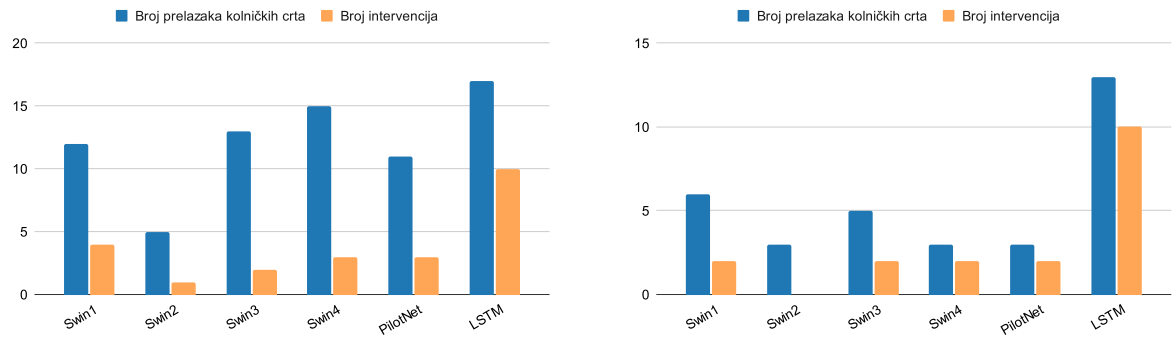


(a) (b)

Slika 4.8: (a) Vožnja kroz naselje i (b) vožnja izvan naselja u CARLA simulatoru

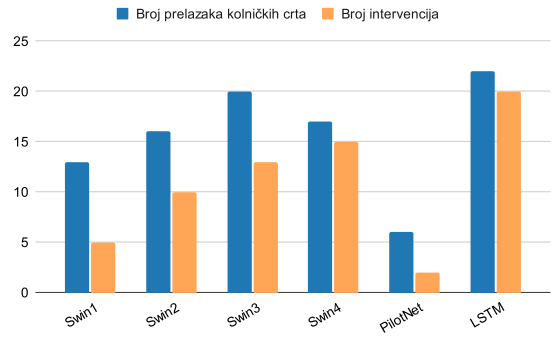
Najbolje rezultate u CARLA simulatoru ostvarili su Swin2 i PilotNet modeli završivši većinu testova s vrlo malim brojem intervencija (Slika 4.9). Svi testirani modeli neuronskih mreža najbolje rezultate ostvaruju na testu noćne vožnje. Pretpostavka je da zbog većeg kontrasta između osvijetljenih značajki ceste i okoline modeli lakše izlučuju značajke bitne za procjenu kuta zakreta upravljača. U skladu s time, najlošiji rezultati ostvareni su po kišnome vremenu. Pretpostavka je da zbog dodatnog zamućivanja i šuma na slici kojeg unose kiša i magla, modeli imaju problema pri izlučivanju značajki potrebnih za rješavanje danog problema.

Računanjem i usporedbom prosječnog broja prelazaka i prosječnog broja korisničkih intervencija po testu iz slika 4.7 i 4.9, vidljivo je da gotovo svi modeli ostvaruju značajno bolje rezultate u MetaDrive simulatoru. Manji prosječan broj prelazaka kolničkih crta i značajno manji prosječan broj intervencija moguće je pripisati nedostatku raznolikosti vremenskih uvjeta i velikoj sličnosti virtualnih okruženja u MetaDrive simulatoru unatoč promjeni teksture ceste. Jedina iznimka je model Swin1 koji ima manji prosječni broj prelazaka kolničkih crta u CARLA simulatoru u odnosu na MetaDrive simulator zbog boljeg rezultata ostvarenog pri kišnom vremenu u usporedbi s ostalim modelima. Moguće je da navedeni model zbog najmanjeg broja neurona u prvom potpuno povezanom sloju naspram ostalih Swin modela bolje izlučuje značajke korisne za predikciju vrijednosti zakreta upravljača filtriranjem šuma koji unosi kiša.



(a)

(b)



(c)

Slika 4.9: Rezultati testiranja različitih modela u CARLA simulatoru u različitim vremenskim uvjetima: (a) dan, vedro, (b) noć, vedro, (c) zalazak sunca, jaka kiša

4.3. Testiranje modela treniranih prijenosnim učenjem

Modeli neuronskih mreža koji su trenirani metodom prijenosnog učenja testirani su na jednak način kao i modeli koji su trenirani na svakom skupu podataka zasebno. Modeli koji su predtrenirani na podacima iz jednog simulatora dotrenirani su i testirani na podacima iz drugog simulatora, a zatim su testirani i u virtualnom okruženju drugog simulatora. Rezultati testiranja različitih modela treniranih prijenosnim učenjem prikazani su u tablicama 4.4 i 4.5.

Rezultati dobiveni testiranjem *Swin* modela treniranih metodom prijenosnog učenja na testnom skupu iz MetaDrive simulatora neznatno su bolji od rezultata modela treniranih na svakom skupu podataka zasebno. Rezultati dobiveni testiranjem *Swin* modela na testom skupu CARLA simulatora su u većini slučajeva lošiji od rezultata *Swin* modela treniranih bez prijenosnog učenja. Moguće je da se modeli predtrenirani na podacima iz MetaDrive simulatora, zbog jednolikosti virtualne okoline i nedostatka vremenskih uvjeta u MetaDrive simulatoru, teže prilagođavaju podacima iz CARLA simulatora nego modeli trenirani bez

Tablica 4.4: Rezultati različitih modela treniranih prijenosnim učenjem na testnom skupu iz MetaDrive simulatora

Model neuronske mreže	**MAE za model treniran prijenosnim učenjem**	**MSE za model treniran prijenosnim učenjem**	**MAE za model treniran bez prijenosnog učenja**	**MSE za model treniran bez prijenosnog učenja**
Swin1	0.01549	0.00087	0.01823	0.00092
Swin2	0.01490	0.00090	0.01707	0.00088
Swin3	0.01698	0.00088	0.01937	0.00097
Swin4	0.01550	0.00087	0.02125	0.00105
PilotNet [9]	**0.01298**	**0.00070**	**0.01128**	**0.00065**
LSTM	0.01379	0.00072	0.01393	0.00076

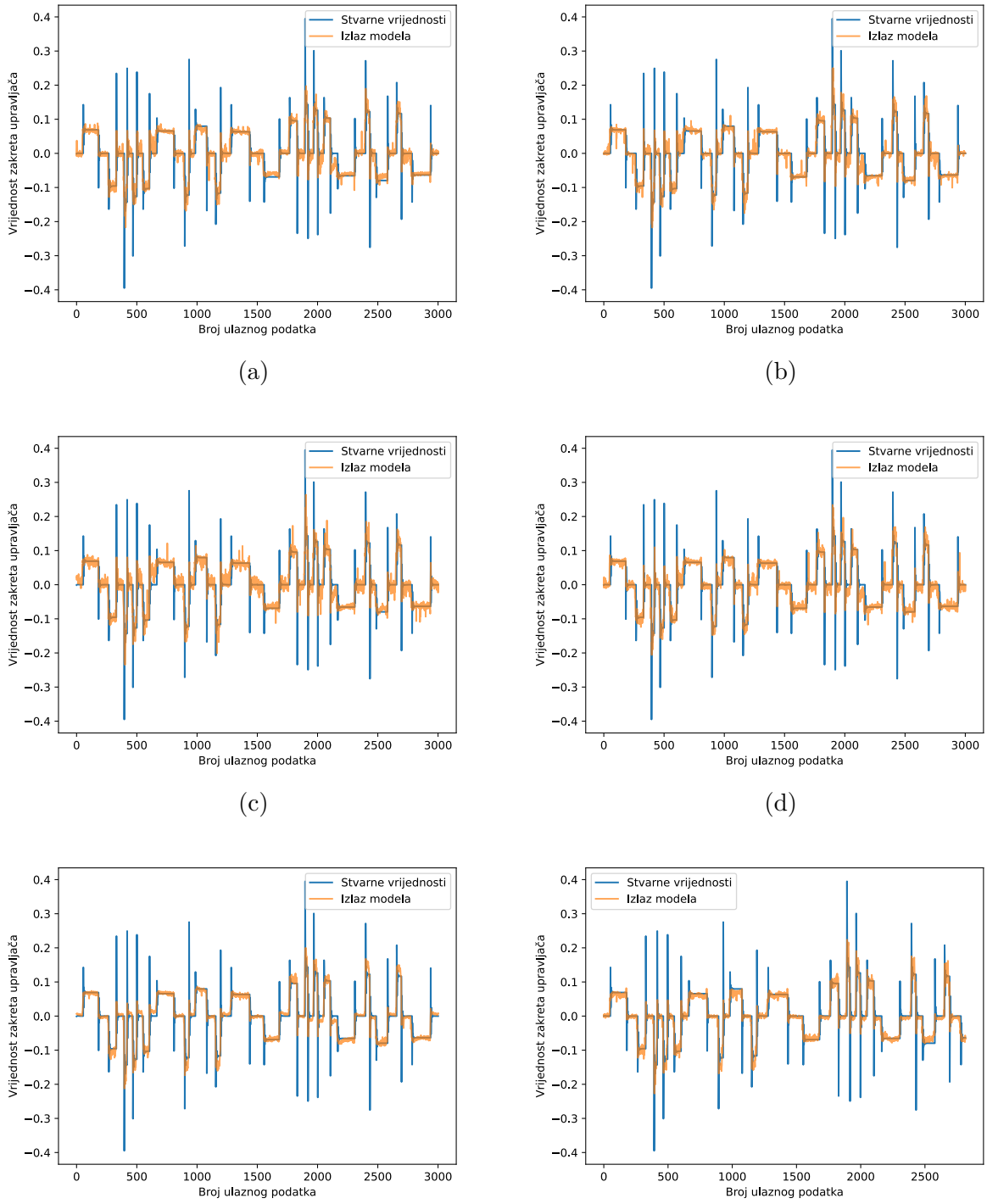
Tablica 4.5: Rezultati različitih modela treniranih prijenosnim učenjem na testnom skupu iz CARLA simulatora

Model neuronske mreže	**MAE za model treniran prijenosnim učenjem**	**MSE za model treniran prijenosnim učenjem**	**MAE za model treniran bez prijenosnog učenja**	**MSE za model treniran bez prijenosnog učenja**
Swin1	0.05812	**0.00599**	0.06691	0.00820
Swin2	0.06881	0.00762	0.06373	0.00675
Swin3	0.07078	0.00764	0.06375	0.00728
Swin4	0.06573	0.00710	0.06453	0.00743
PilotNet [9]	**0.04983**	0.00748	**0.04883**	**0.00496**
LSTM	0.06553	0.00674	0.06793	0.00743

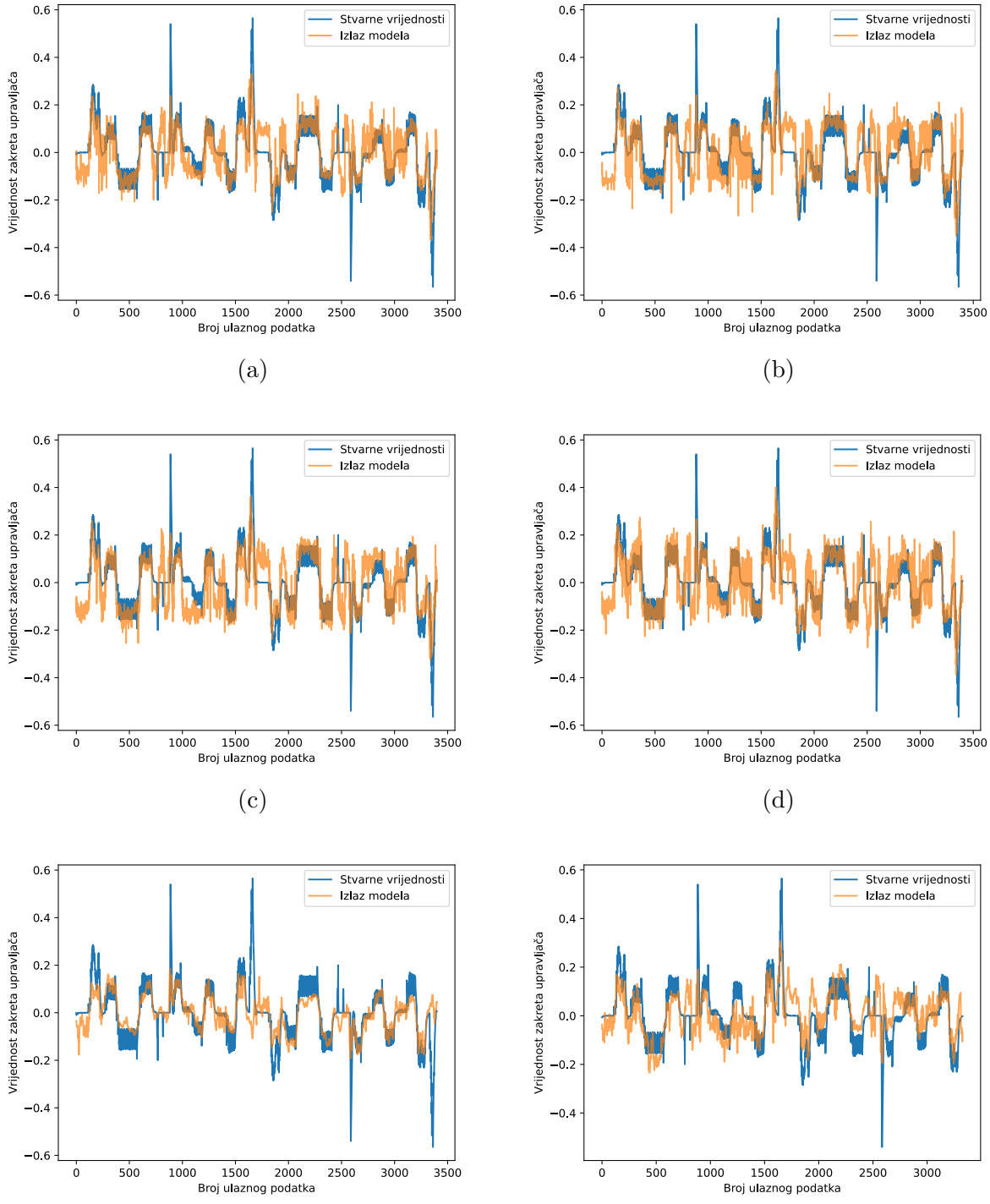
prijenosnog učenja. PilotNet model treniran metodom prijenosnog učenja na oba testna skupa postiže lošije rezultate od PilotNet modela treniranog na svakom skupu podataka zasebno. Pretpostavka je da se model vrlo dobro prilagođava jednom skupu podataka, no zbog različitog ponašanja vlastitog vozila unutar simulatora, a time i različitih kontrola vozila za naizgled jednaku akciju, manje se uspješno prilagođava novim podacima. U slučaju testiranja vlastitog LSTM modela treniranog prijenosnim učenjem, zabilježena su neznatna smanjenja srednjeg kvadratnog odstupanja i srednjeg apsolutnog odstupanja na oba skupa podataka u odnosu na istu neuronsku mrežu treniranu bez prijenosnog učenja.

Usporedbom predviđenih i stvarnih vrijednosti na testnim skupovima podataka, moguće je vidjeti da dotrenirani *Swin* modeli daju manje zašumljene izlazne veličine s manje stršećih vrijednosti s obzirom na modele jednake arhitekture trenirane na svakom skupu zasebno (Slike 4.10 i 4.11), što objašnjava bolje rezultate prilikom testiranja na podacima

iz MetaDrive simulatora. To je moguće povezati s činjenicom da transformerske neuronske mreže trebaju veliku količinu podataka za uspješno rješavanje problema [26]. Moguće je da bi daljnjim proširivanjem skupa podataka za treniranje, *Swin* modeli dali bolje rezultate.



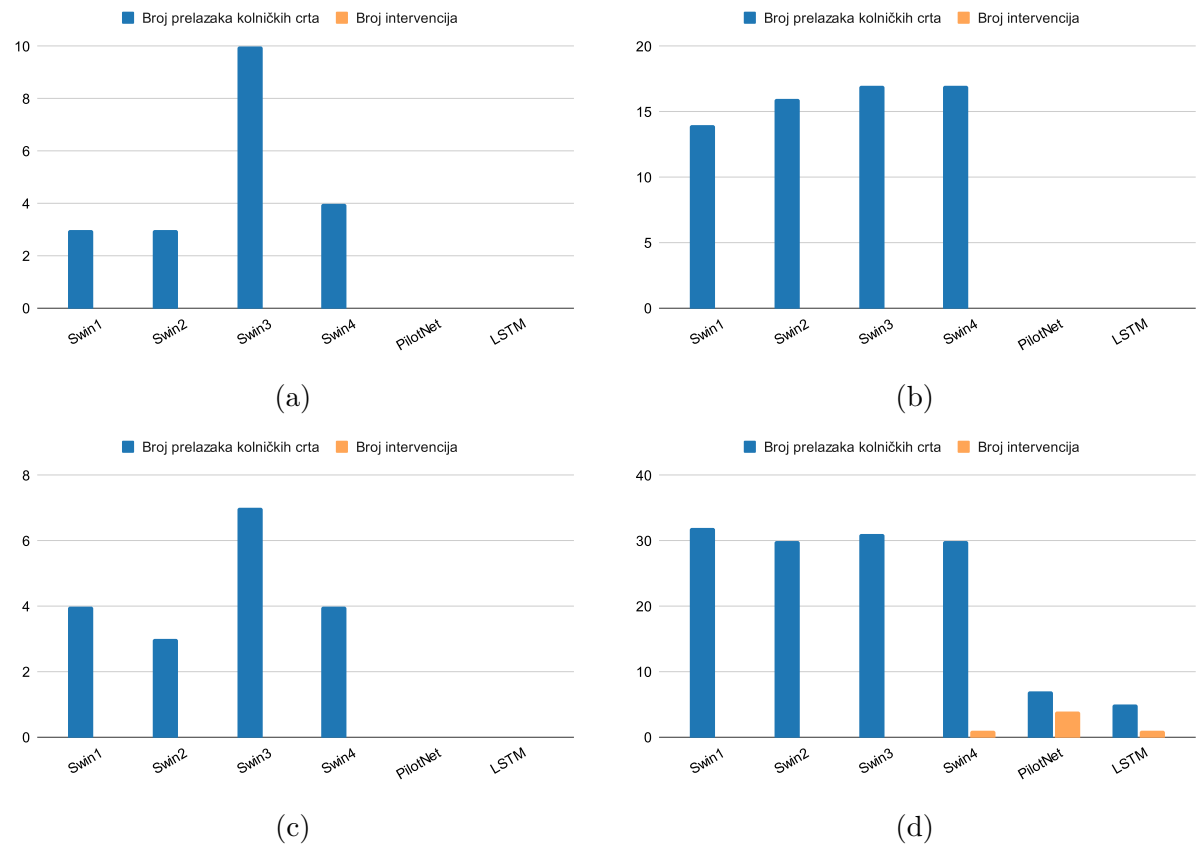
Slika 4.10: Usporedba izlaza različitih modela treniranih prijenosnim učenjem i stvarnih vrijednosti iz MetaDrive skupa podataka: (a) Swin1, (b) Swin2, (c) Swin3, (d) Swin4, (e) PilotNet, (f) LSTM

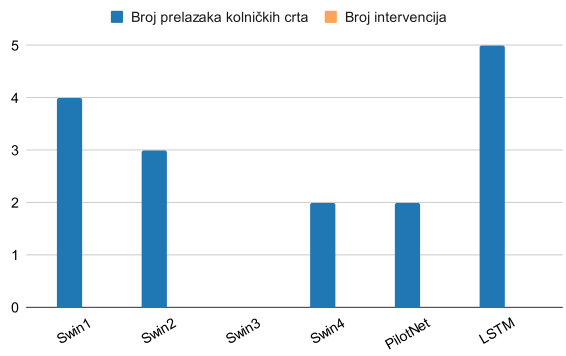


Slika 4.11: Usporedba izlaza različitih modela treniranih prijenosnim učenjem i stvarnih vrijednosti iz CARLA skupa podataka: (a) Swin1, (b) Swin2, (c) Swin3, (d) Swin4, (e) PilotNet, (f) LSTM

Modeli neuronskih mreža trenirani metodom prijenosnog učenja ostvarili su bolje rezultate u MetaDrive simulatoru na cestama gdje je korištena ugrađena tekstura ceste. Promjenom teksture ceste rezultati postaju značajno lošiji. Trening skup podataka

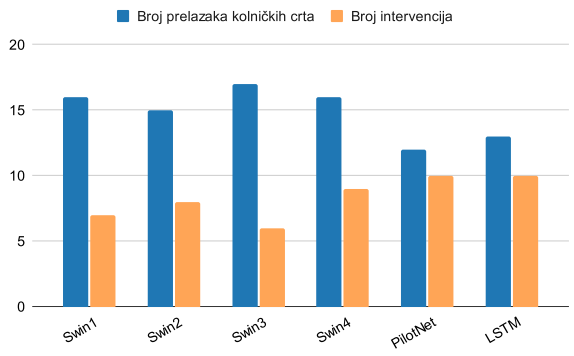
moguće je proširiti tako da uključuje više različitih tekstura ceste, čime bi se poboljšale generalizacijske sposobnosti modela. Rezultati testiranja modela treniranih prijenosnim učenjem u MetaDrive simulatoru prikazani su na slici 4.12. Pri testiranju u CARLA simulatoru, modeli trenirani prijenosnim učenjem generalno daju lošije rezultate nego modeli iste arhitekture koji su trenirani na svakom skupu zasebno. Značajno se povećao broj intervencija, osim za Swin3 i Swin4 modele. Pretpostavka je da se ti modeli, zbog manjeg broja parametara, bolje prilagođavaju novim podacima, ali zadržavaju generalizacijske sposobnosti stečene na skupu na kojem su predtrenirani. Rezultati testiranja modela treniranih prijenosnim učenjem u CARLA simulatoru prikazani su na slici 4.13. Kao i kod modela koji su trenirani na svakom skupu zasebno, najlošiji rezultati dobiveni su na kišovitom vremenu. Iz dobivenih rezultata može se zaključiti da metoda prijenosnog učenja u većini slučajeva nije isplativa ako se modeli neuronskih mreža treniraju na podacima iz jednog, a testiraju na podacima iz drugog simulatora i u virtualnom okruženju drugog simulatora. Lošije rezultate u virtualnim okruženjima moguće je pripisati različitom ponašanju vozila u različitim simulatorima, vizualnim razlikama i razlikama u načinu rada svakog od simulatora. Proširenje skupova podataka za treniranje vjerojatno bi dovelo do boljih rezultata nego prijenosno učenje na podacima iz različitih simulatora.



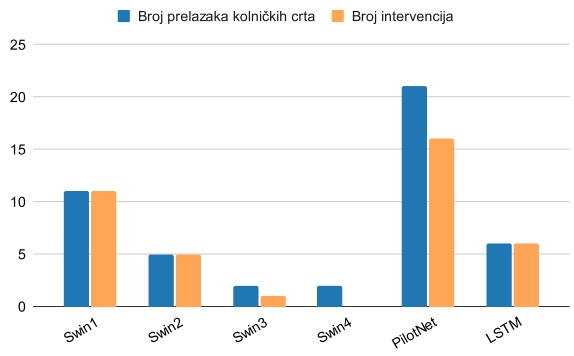


(e)

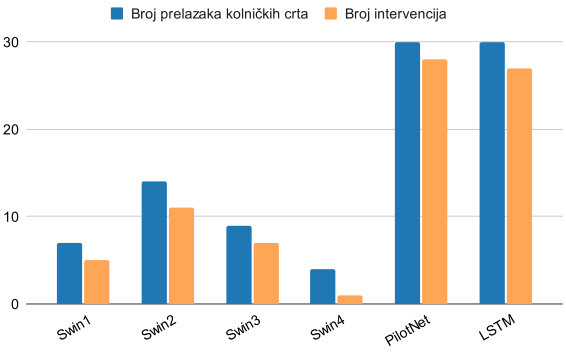
Slika 4.12: Rezultati testiranja različitih modela treniranih prijenosnim učenjem u MetaDrive simulatoru pri različitim postavkama: (a) *seed*=100, (b) *seed*=100 uz teksturu ceste sa slike 4.6(b), (c) *seed*=200, (d) *seed*=200 uz teksturu ceste sa slike 4.6(c), (e) *seed*=300 i smanjeni minimalni radijus zavoja



(a)



(b)



(c)

Slika 4.13: Rezultati testiranja različitih modela treniranih prijenosnim učenjem u CARLA simulatoru u različitim vremenskim uvjetima:
(a) dan, vedro, (b) noć, vedro, (c) zalazak sunca, jaka kiša

5. ZAKLJUČAK

Cilj ovog diplomskog rada bio je izraditi novi vlastiti algoritam za autonomnu vožnju zasnovan na procjeni kuta zakreta upravljača vozila te testirati performanse predloženog rješenja u dvama različitim simulatorima. Algoritam autonomne vožnje predložen u ovom radu zasniva se na *Swin* transformerskoj arhitekturi za detekciju značajki slike i potpuno povezanim slojevima, koji procjenjuju kut zakreta upravljača vozila na osnovu detektiranih značajki. Predloženi algoritam razvijen je u programskom jeziku Python pomoću biblioteka za strojno učenje Keras i TensorFlow, a treniran je i testiran na vlastitim skupovima podataka prikupljenim u simulatorima CARLA i MetaDrive. Implementirana su 4 modela neuronske mreže (Swin1, Swin2, Swin3 i Swin4) koji se razlikuju po broju potpuno povezanih slojeva i broju neurona u pojedinom sloju. Za testiranje na podacima iz simulatora korištene su standardne metrike za regresijske probleme: srednje kvadratno odstupanje i srednje apsolutno odstupanje. Rezultati dobiveni testiranjem na skupovima podataka iz simulatora približno su jednaki rezultatima koje postižu suvremeni algoritmi, kao što su PilotNet [9] i vlastita implementacija LSTM mreže po uzoru na [10, 12]. Također, promatrano je i vrijeme inferencije tj. brzina obrade podataka različitih modela neuronskih mreža. Najveću brzinu obrade postigao je Swin1 model s 16.04 FPS. Predloženi algoritmi testirani su i u virtualnim okruženjima simulatora, gdje je ispitivano upravljanje vlastitim vozilom u različitim vremenskim uvjetima na različitim konfiguracijama ceste. U CARLA simulatoru, Swin2 model ostvario je bolje rezultate nego algoritmi iz literature na većini testova. Svi algoritmi najbolje rezultate ostvaruju pri noćnoj vožnji, a pretpostavka je da zbog većeg kontrasta okoline i osvijetljenih značajki ceste modeli neuronskih mreža lakše raspoznaju značajke bitne za rješavanje danog problema. Sukladno tome, najlošiji rezultati ostvareni su na kišnom vremenu. Pretpostavka je da kiša unosi šum koji otežava prepoznavanje bitnih značajki ceste na slici. U MetaDrive simulatoru, Swin2 model ostvario je približno jednake rezultate kao i PilotNet neuronska mreža. Uz navedeno, testirani su i modeli trenirani metodom prijenosnog učenja, gdje je predložena neuronska mreža trenirana na skupu podataka iz jednog simulatora, a dotrenirana i testirana na podacima iz drugog simulatora. Takav pristup treniranju nije dao značajno bolje rezultate od treniranja neuronskih mreža na svakom skupu zasebno, osim u slučaju Swin3 i Swin4 modela predtreniranih na MetaDrive podacima, a testiranih u CARLA simulatoru, gdje je zabilježen manji broj prelazaka

kolničkih crta te manji broj korisničkih intervencija nego kod Swin3 i Swin4 modela treniranih na svakom skupu zasebno. Pretpostavka je da se navedeni modeli zbog svoje manje dubine, odnosno zbog manjeg broja parametara, bolje prilagođavaju novom skupu podataka i zadržavaju generalizacijska svojstva. Analizom dobivenih rezultata zaključeno je da algoritmi zasnovani na *Swin* transformerskim neuronskim mrežama mogu biti korišteni kao algoritmi za autonomnu vožnju zasnovani na procjeni kuta zakreta upravljača te mogu dati vrlo dobre rezultate ako je dostupan dovoljno velik i raznovrstan skup podataka za treniranje. Predobradom slike s prednje kamere vozila, proširivanjem skupa podataka za treniranje i modifikacijom implementirane neuronske mreže moguće je ostvariti bolje rezultate.

LITERATURA

[1] A. El Khatib, C. Ou, and F. Karray, "Driver inattention detection in the context of next-generation autonomous vehicles design: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 11, pp. 4483–4496, 2020.

[2] U.S. Department of Transportation, Federal Highway Administration, "Roadway departure safety." , s Interneta, https://safety.fhwa.dot.gov/roadway_dept/ [1.8.2022.].

[3] A. Oussama and M. Talea, *A Literature Review of Steering Angle Prediction Algorithms for Self-driving Cars*, 02 2020, pp. 30–38.

[4] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[5] Q. Li, Z. Peng, Z. Xue, Q. Zhang, and B. Zhou, "Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning," *arXiv preprint arXiv:2109.12674*, 2021.

[6] D. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network," in *Proceedings of (NeurIPS) Neural Information Processing Systems*, D. Touretzky, Ed. Morgan Kaufmann, December 1989, pp. 305 – 313.

[7] Net-Scale Technologies, Inc., "Autonomous off-road vehicle control using end-to-end learning - final technical report," 2004. , s Interneta, http://net-scale.com/doc/net-scale-dave-report.pdf [1.8.2022.].

[8] H. Saleem, F. Riaz, L. Mostarda, M. A. Niazi, A. Rafiq, and S. Saeed, "Steering angle prediction techniques for autonomous ground vehicles: A review," *IEEE Access*, vol. 9, pp. 78 567–78 585, 2021.

[9] M. Bojarski, D. D. Testa, D. Dworakowski *et al.*, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016. , s Interneta, http://arxiv.org/abs/1604.07316 [1.8.2022.].

[10] S. Du, H. Guo, and A. Simpson, "Self-driving car steering angle prediction based on image recognition," 12 2019.

[11] Udacity, "Udacity self-driving car dataset." , s Interneta, https://github.com/udacity/self-driving-car/tree/master/datasets [1.8.2022.].

[12] H. M. Eraqi, M. N. Moustafa, and J. Honer, "End-to-end deep learning for steering autonomous vehicles considering temporal dependencies," *CoRR*, vol. abs/1710.03804, 2017. , s Interneta, http://arxiv.org/abs/1710.03804 [1.8.2022.].

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[14] H. Schafer, E. Santana, A. Haden, and R. Biasini, "A commute in data: The comma2k19 dataset," 2018.

[15] Q. Khan, T. Schön, and P. Wenzel, "Latent space reinforcement learning for steering angle prediction," 2019. , s Interneta, https://arxiv.org/abs/1902.03765 [1.8.2022.].

[16] A. I. Maqueda, A. Loquercio, G. Gallego, N. Garcia, and D. Scaramuzza, "Event-based vision meets deep learning on steering prediction for self-driving cars," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[17] Y. Hu, J. Binas, D. Neil, S.-C. Liu, and T. Delbruck, "Ddd20 end-to-end event camera driving dataset: Fusing frames and events with deep learning for improved steering prediction," 2020. , s Interneta, https://arxiv.org/abs/2005.08605 [1.8.2022.].

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition." , s Interneta, https://arxiv.org/abs/1512.03385 [1.8.2022.].

[19] J. Binas, D. Neil, S.-C. Liu, and T. Delbruck, "Ddd17: End-to-end davis driving dataset," 2017. , s Interneta, https://arxiv.org/abs/1711.01458 [1.8.2022.].

[20] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," 2021. , s Interneta, https://arxiv.org/abs/2103.14030 [1.8.2022.].

[21] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. , s Interneta, https://www.tensorflow.org/ [1.8.2022.].

[22] F. Chollet *et al.*, "Keras," 2015. , s Interneta, https://keras.io/ [01.08.2022.].

[23] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. , s Interneta, https://arxiv.org/abs/1706.03762 [01.08.2022.].

[25] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[26] A. Dosovitskiy, L. Beyer, A. Kolesnikov *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," 2020. , s Interneta, https://arxiv.org/abs/2010.11929 [1.8.2022.].

[27] N. Park and S. Kim, "How do vision transformers work?" 2022. , s Interneta, https://arxiv.org/abs/2202.06709 [1.8.2022.].

[28] R. Dagli, "Image classification with swin transformers." , s Interneta, https://keras.io/examples/vision/swin_transformers/ [1.8.2022.].

SAŽETAK

U ovom radu dan je pregled postojećih algoritama za autonomnu vožnju zasnovanih na procjeni kuta zakreta upravljača vozila. Izrađeni su novi vlastiti algoritmi za autonomnu vožnju zasnovani na *Swin* transformerskoj neuronskoj mreži. Implementirane su 4 neuronske mreže (Swin1, Swin2, Swin3 i Swin4) koje se razlikuju po broju potpuno povezanih slojeva i broju neurona u pojedinom sloju. Implementirane neuronske mreže trenirane su i testirane na skupovima podataka prikupljenim u CARLA i MetaDrive simulatorima, a dobiveni rezultati približno su jednaki rezultatima PilotNet neuronske mreže i vlastite LSTM neuronske mreže. Također, promatrano je i vrijeme inferencije tj. brzina obrade podataka različitih neuronskih mreža. Najveću brzinu obrade postigla je Swin1 neuronska mreža s 16.04 FPS. Uz testiranje na prikupljenim podacima, modeli neuronskih mreža testirani su u virtualnim okruženjima obaju simulatora te su postigli visoke performanse. U CARLA simulatoru, Swin2 model ostvario je bolje rezultate nego algoritmi iz literature na većini testova. U MetaDrive simulatoru, Swin2 model ostvario je približno jednake rezultate kao i PilotNet neuronska mreža. Modeli neuronskih mreža koji su postigli najbolje rezultate dotrenirani su na trening skupu podataka iz drugog simulatora metodom prijenosnog učenja, no dobiveni rezultati nisu bili značajno bolji u odnosu na rezultate dobivene treniranjem modela na svakom skupu zasebno. Zaključeno je da neuronske mreže zasnovane na *Swin* transformerskoj neuronskoj mreži mogu biti osnova algoritma autonomne vožnje zasnovanog na procjeni kuta zakreta upravljača vozila uz dovoljno velik i raznovrstan skup podataka za treniranje.

Ključne riječi: transformer, konvolucija, neuronska mreža, simulator, autonomna vožnja

DEVELOPMENT OF THE AUTONOMOUS DRIVING ALGORITHM BASED ON VEHICLE STEERING ANGLE PREDICTION AND ALGORITHM PERFORMANCE TESTING IN DIFFERENT SIMULATORS

This paper provides an overview of the existing autonomous driving algorithms based on steering angle prediction. Proprietary autonomous driving algorithms based on the *Swin* transformer neural network were developed. Four neural network models were implemented (Swin1, Swin2, Swin3 and Swin4) differing in the number of fully connected layers and the number of neurons in each layer. The implemented neural networks were trained and tested on datasets collected in CARLA and MetaDrive simulators, and the obtained results are comparable to the results of the PilotNet neural network and the proprietary LSTM neural network. Also, the inference time i.e. the speed of data processing of different neural networks was observed. The highest processing speed was achieved by the Swin1 neural network with 16.04 FPS. In addition to testing on collected datasets, neural network models were tested in the virtual environments of both simulators and achieved high performance. In CARLA simulator, the Swin2 model achieved better results than contemporary algorithms on most tests. In MetaDrive simulator, the Swin2 model achieved results comparable to those of the PilotNet neural network. The models that achieved the best results were trained on the training dataset from another simulator using the transfer learning method but the results obtained were not significantly better than the results obtained by training neural network models on each dataset separately. It was concluded that neural networks based on the *Swin* transformer neural network can provide a platform for an autonomous driving algorithm based on steering angle prediction given a sufficiently large and diverse training dataset.

Key words: transformer, convolution, neural network, simulator, autonomous driving

ŽIVOTOPIS

Borna Benja rođen je 28.6.1998. u Osijeku. Završio je srednjoškolsko obrazovanje u III. gimnaziji u Osijeku s vrlo dobrim uspjehom. Nakon srednje škole upisuje preddiplomski sveučilišni studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Akademski naziv sveučilišni prvostupnik (baccalaureus) inženjer računarstva stječe 2020. godine. Iste je godine upisao diplomski sveučilišni studij računarstva, smjer robotika i umjetna inteligencija te nakon položenog testiranja postaje stipendist TTTech Auto CEE u Osijeku (tadašnji institut RT-RK).

PRILOZI

Prilog P.3.1: Programski kod za definiranje slojeva neuronske mreže za obradu dijelova slike

```
class PatchExtract(layers.Layer):
    def __init__(self, patch_size, **kwargs):
        super(PatchExtract, self).__init__(**kwargs)
        self.patch_size_x = patch_size[0]
        self.patch_size_y = patch_size[1]

    def get_config(self):
        config = super(PatchExtract, self).get_config()
        config.update({"patch_size": patch_size})
        return config

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(images=images, sizes=(1, self.patch_size_x, self.patch_size_y, 1),
        ↪  strides=(1, self.patch_size_x, self.patch_size_y, 1), rates=(1, 1, 1, 1), padding="VALID",)
        patch_dim = patches.shape[-1]
        patch_num = patches.shape[1]
        return tf.reshape(patches, (batch_size, patch_num * patch_num, patch_dim))

class PatchEmbedding(layers.Layer):
    def __init__(self, num_patch, embed_dim, **kwargs):
        super(PatchEmbedding, self).__init__(**kwargs)
        self.num_patch = num_patch
        self.proj = layers.Dense(embed_dim)
        self.pos_embed = layers.Embedding(input_dim=num_patch, output_dim=embed_dim)

    def call(self, patch):
        pos = tf.range(start=0, limit=self.num_patch, delta=1)
        return self.proj(patch) + self.pos_embed(pos)

    def get_config(self):
        config = super(PatchEmbedding, self).get_config().copy()
        config.update({"num_patch": self.num_patch,
                       "embed_dim": embed_dim})
        return config

class PatchMerging(tf.keras.layers.Layer):
    def __init__(self, num_patch, embed_dim, **kwargs):
        super(PatchMerging, self).__init__(**kwargs)
        self.num_patch = num_patch
        self.embed_dim = embed_dim
        self.linear_trans = layers.Dense(2 * embed_dim, use_bias=False)

    def call(self, x):
        height, width = self.num_patch
        _, _, C = x.get_shape().as_list()
        x = tf.reshape(x, shape=(-1, height, width, C))
        x0 = x[:, 0::2, 0::2, :]
        x1 = x[:, 1::2, 0::2, :]
        x2 = x[:, 0::2, 1::2, :]
        x3 = x[:, 1::2, 1::2, :]
        x = tf.concat((x0, x1, x2, x3), axis=-1)
        x = tf.reshape(x, shape=(-1, (height // 2) * (width // 2), 4 * C))
        return self.linear_trans(x)

    def get_config(self):
        config = super(PatchMerging, self).get_config().copy()
        config.update({"num_patch": self.num_patch,
                       "embed_dim": embed_dim})
        return config
```

Prilog P.3.2: Programski kod sloja za računanje vektora pozornosti

```
class WindowAttention(layers.Layer):
    def __init__(self, dim, window_size, num_heads, qkv_bias=True, dropout_rate=0.0, **kwargs):
        super(WindowAttention, self).__init__(**kwargs)
        self.dim = dim
        self.window_size = window_size
        self.num_heads = num_heads
        self.scale = (dim // num_heads) ** -0.5
        self.qkv = layers.Dense(dim * 3, use_bias=qkv_bias)
        self.dropout = layers.Dropout(dropout_rate)
        self.proj = layers.Dense(dim)

    def get_config(self):
        config = super(WindowAttention, self).get_config().copy()
        config.update({"dim": self.dim,
                       "window_size": self.window_size,
                       "num_heads": self.num_heads,
                       "qkv_bias": self.qkv,
                       "dropout_rate": self.dropout})
        return config

    def build(self, input_shape):
        num_window_elements = (2 * self.window_size[0] - 1) * (2 * self.window_size[1] - 1)
        self.relative_position_bias_table = self.add_weight(
            shape=(num_window_elements, self.num_heads),
            initializer=tf.initializers.Zeros(),
            trainable=True,
            name="rpbt")
        coords_h = np.arange(self.window_size[0])
        coords_w = np.arange(self.window_size[1])
        coords_matrix = np.meshgrid(coords_h, coords_w, indexing="ij")
        coords = np.stack(coords_matrix)
        coords_flatten = coords.reshape(2, -1)
        relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :]
        relative_coords = relative_coords.transpose([1, 2, 0])
        relative_coords[:, :, 0] += self.window_size[0] - 1
        relative_coords[:, :, 1] += self.window_size[1] - 1
        relative_coords[:, :, 0] *= 2 * self.window_size[1] - 1
        relative_position_index = relative_coords.sum(-1)

        self.relative_position_index =
        ↪  tf.Variable(initial_value=tf.convert_to_tensor(relative_position_index), trainable=False)

    def call(self, x, mask=None):
        _, size, channels = x.shape
        head_dim = channels // self.num_heads
        x_qkv = self.qkv(x)
        x_qkv = tf.reshape(x_qkv, shape=(-1, size, 3, self.num_heads, head_dim))
        x_qkv = tf.transpose(x_qkv, perm=(2, 0, 3, 1, 4))
        q, k, v = x_qkv[0], x_qkv[1], x_qkv[2]
        q = q * self.scale
        k = tf.transpose(k, perm=(0, 1, 3, 2))
        attn = q @ k

        num_window_elements = self.window_size[0] * self.window_size[1]
        relative_position_index_flat = tf.reshape(
            self.relative_position_index, shape=(-1,))
        relative_position_bias = tf.gather(
            self.relative_position_bias_table, relative_position_index_flat)
        relative_position_bias = tf.reshape(
            relative_position_bias, shape=(num_window_elements, num_window_elements, -1))
        relative_position_bias = tf.transpose(relative_position_bias, perm=(2, 0, 1))
        attn = attn + tf.expand_dims(relative_position_bias, axis=0)

        if mask is not None:
            nW = mask.get_shape()[0]
            mask_float = tf.cast(tf.expand_dims(tf.expand_dims(mask, axis=1), axis=0), tf.float32)
            attn = (tf.reshape(attn, shape=(-1, nW, self.num_heads, size, size)) + mask_float)
            attn = tf.reshape(attn, shape=(-1, self.num_heads, size, size))
            attn = keras.activations.softmax(attn, axis=-1)
        else:
```

```
    attn = keras.activations.softmax(attn, axis=-1)
attn = self.dropout(attn)

x_qkv = attn @ v
x_qkv = tf.transpose(x_qkv, perm=(0, 2, 1, 3))
x_qkv = tf.reshape(x_qkv, shape=(-1, size, channels))
x_qkv = self.proj(x_qkv)
x_qkv = self.dropout(x_qkv)
return x_qkv
```

Prilog P.3.3: Programski kod za *Swin* transformerski sloj

```
class SwinBlock(layers.Layer):
    def __init__(self, dim, num_patch, num_heads, window_size=7, shift_size=0, num_mlp=1024, qkv_bias=True,
    ↪  dropout_rate=0.0, **kwargs,):
        super(SwinBlock, self).__init__(**kwargs)
        self.dim = dim  # number of input dimensions
        self.num_patch = num_patch  # number of embedded patches
        self.num_heads = num_heads  # number of attention heads
        self.window_size = window_size  # size of window
        self.shift_size = shift_size  # size of window shift
        self.num_mlp = num_mlp  # number of MLP nodes
        self.norm1 = layers.LayerNormalization(epsilon=1e-5, name="norm1")
        self.attn = WindowAttention(dim, window_size=(self.window_size, self.window_size),
        ↪  num_heads=num_heads, qkv_bias=qkv_bias, dropout_rate=dropout_rate,)
        self.drop_path = DropPath(dropout_rate)
        self.norm2 = layers.LayerNormalization(epsilon=1e-5, name="norm2")
        self.mlp = keras.Sequential([
                layers.Dense(num_mlp),
                layers.Activation(keras.activations.gelu),
                layers.Dropout(dropout_rate),
                layers.Dense(dim),
                layers.Dropout(dropout_rate),],
            name="mlp")
        if min(self.num_patch) < self.window_size:
            self.shift_size = 0
            self.window_size = min(self.num_patch)

    def build(self, input_shape):
        if self.shift_size == 0:
            self.attn_mask = None
        else:
            height, width = self.num_patch
            h_slices = (slice(0, -self.window_size), slice(-self.window_size, -self.shift_size),
            ↪  slice(-self.shift_size, None),)
            w_slices = (slice(0, -self.window_size), slice(-self.window_size, -self.shift_size),
            ↪  slice(-self.shift_size, None),)
            mask_array = np.zeros((1, height, width, 1))
            count = 0
            for h in h_slices:
                for w in w_slices:
                    mask_array[:, h, w, :] = count
                    count += 1
            mask_array = tf.convert_to_tensor(mask_array)
            mask_windows = window_partition(mask_array, self.window_size)
            mask_windows = tf.reshape(mask_windows, shape=[-1, self.window_size * self.window_size])
            attn_mask = tf.expand_dims(mask_windows, axis=1) - tf.expand_dims(mask_windows, axis=2)
            attn_mask = tf.where(attn_mask != 0, -100.0, attn_mask)
            attn_mask = tf.where(attn_mask == 0, 0.0, attn_mask)
            self.attn_mask = tf.Variable(initial_value=attn_mask, trainable=False)

    def call(self, x):
        height, width = self.num_patch
        _, num_patches_before, channels = x.shape
        x_skip = x
        x = self.norm1(x)
        x = tf.reshape(x, shape=(-1, height, width, channels))
        if self.shift_size > 0:
            shifted_x = tf.roll(x, shift=[-self.shift_size, -self.shift_size], axis=[1, 2])
        else:
            shifted_x = x
        x_windows = window_partition(shifted_x, self.window_size)
        x_windows = tf.reshape(x_windows, shape=(-1, self.window_size * self.window_size, channels))
        attn_windows = self.attn(x_windows, mask=self.attn_mask)

        attn_windows = tf.reshape(attn_windows, shape=(-1, self.window_size, self.window_size, channels))
        shifted_x = window_reverse(attn_windows, self.window_size, height, width, channels)
        if self.shift_size > 0:
            x = tf.roll(shifted_x, shift=[self.shift_size, self.shift_size], axis=[1, 2])
        else:
            x = shifted_x
        x = tf.reshape(x, shape=(-1, height * width, channels))
```

```
        x = self.drop_path(x)
        x = x_skip + x
        x_skip = x
        x = self.norm2(x)
        x = self.mlp(x)
        x = self.drop_path(x)
        x = x_skip + x
        return x

    def get_config(self):
        config = super(SwinBlock, self).get_config().copy()
        config.update({
            "name": self.name,
            "dim": self.dim,
            "num_patch": self.num_patch,
            "num_heads": self.num_heads,
            "window_size": self.window_size,
            "shift_size": self.shift_size,
            "num_mlp": self.num_mlp,
            "qkv_bias": qkv_bias,
            "dropout_rate": dropout_rate})
        return config
```

Prilog P.3.4: Programski kod implementirane neuronske mreže zasnovane na *Swin* tansformeru

```
def build_swin_model_2_256():
    model = Sequential(name="2_256")
    model.add(PatchExtract(patch_size, input_shape=input_shape))
    model.add(PatchEmbedding(num_patch_x * num_patch_y, embed_dim))
    model.add(SwinBlock(
        dim=embed_dim,
        num_patch=(num_patch_x, num_patch_y),
        num_heads=num_heads,
        window_size=window_size,
        shift_size=shift_size,
        num_mlp=num_mlp,
        qkv_bias=qkv_bias,
        dropout_rate=dropout_rate,
    ))
    model.add(SwinBlock(
        dim=embed_dim,
        num_patch=(num_patch_x, num_patch_y),
        num_heads=num_heads,
        window_size=window_size,
        shift_size=shift_size,
        num_mlp=num_mlp,
        qkv_bias=qkv_bias,
        dropout_rate=dropout_rate,
    ))
    model.add(PatchMerging((num_patch_x, num_patch_y), embed_dim=embed_dim))
    model.add(GlobalAveragePooling1D())
    model.add(Dense(256, activation="elu", kernel_regularizer=keras.regularizers.l2(0.005)))
    model.add(Dense(1, dtype='float32'))
    model.compile(loss=weighted_loss,
                  optimizer=optimizer,
                  metrics=['mean_absolute_error', 'mean_squared_error'])
    return model
```

Prilog P.3.5: Programski kod funkcije za kreiranje *.h5* datoteke

```
def make_h5_file(path):
    folders = [folder for folder in glob(path) if not os.path.basename(folder).endswith(".npy")]
    numpys = natsorted([n for folder in folders for n in glob(f"{folder}/dataset*.npy")])
    with h5py.File("D:\\bbenja\\datasets\\carla_dataset_32x32.hdf5", "a") as f:
        for folder in folders[:]:
            grp = f.create_group(folder.split("\\")[-1])
            files = [n for n in numpys if n.startswith(f"{folder}\\")]
            grp.create_dataset("X", (len(files) * 100, 32, 32, 3), dtype="float32")
            for i, n in enumerate(files):
                X = np.load(n)
                X = np.divide(X, 255.0)
                X = X[:, -300:, :, :]
                X = [cv2.resize(x, (32, 32), interpolation=cv2.INTER_AREA) for x in X]
                f[folder.split("\\")[-1]]['X'][i * 100:i * 100 + 100, :, :, :] = X
            with open(f"{folder}/data.txt") as datafile:
                y = []
                for line in datafile:
                    y.append(float(line.split(",")[1]))
                for index in range(1, len(y) - 1):
                    if y[index] == 0.0:
                        y[index] = (y[index - 1] + y[index + 1]) / 2.0
            y = np.asarray(y, dtype="float32")
            grp.create_dataset("y", data=y, dtype="float32")
```

Prilog P.3.6: Programski kod implementirane LSTM neuronske mreže

```
def build_LSTM_multiple_input_model():
    model = Sequential(name="LSTM_multiple_input")
    model.add(TimeDistributed(Conv2D(24, (5, 5),
                                     kernel_initializer='he_normal',
                                     activation='relu',
                                     strides=(2, 2),
                                     padding='valid'), input_shape=(5, 66, 200, 3)))
    model.add(TimeDistributed(Conv2D(36, (5, 5),
                                     kernel_initializer='he_normal',
                                     activation='relu',
                                     strides=(2, 2),
                                     padding='valid')))
    model.add(TimeDistributed(Conv2D(48, (5, 5),
                                     kernel_initializer='he_normal',
                                     activation='relu',
                                     strides=(2, 2),
                                     padding='valid')))
    model.add(TimeDistributed(Conv2D(64, (3, 3),
                                     kernel_initializer='he_normal',
                                     activation='relu',
                                     padding='valid')))
    model.add(TimeDistributed(Conv2D(64, (3, 3),
                                     kernel_initializer='he_normal',
                                     activation='relu',
                                     padding='valid')))
    model.add(TimeDistributed(Flatten()))
    model.add(LSTM(64, return_sequences=True, implementation=2))
    model.add(LSTM(64, return_sequences=True, implementation=2))
    model.add(LSTM(64, implementation=2))
    model.add(Dropout(0.2))
    model.add(Dense(128, kernel_initializer='he_normal', activation='relu', kernel_regularizer=l2(1e-3)))
    model.add(Dropout(0.2))
    model.add(Dense(1, kernel_initializer='he_normal', kernel_regularizer=l2(1e-3)))
    model.compile(loss='mean_squared_error',
                  optimizer=optimizer,
                  metrics=['mean_absolute_error', 'mean_squared_error'])
    return model
```

Prilog P.3.7: Programski kod generatora za učitavanje podataka iz .h5 datoteka

```
class H5DataGenerator(Sequence):
    def __init__(self, file, indices, batch_size, augmentations, shuffle=False, threshold=0):
        self.file = file
        self.threshold = threshold
        self.batch_size = batch_size
        self.augmentations = augmentations
        self.shuffle = shuffle
        self.indices = [[ds, i] for ds, i in indices if abs(self.file[ds]["y"][int(i)]) > self.threshold]

    def __len__(self):
        return int(np.ceil(len(self.indices) / float(self.batch_size)))

    def __getitem__(self, idx):
        batch = self.indices[idx * self.batch_size: (idx + 1) * self.batch_size]
        batch_x = [self.file[d]['X'][int(i)] for d, i in batch]
        batch_y = [self.file[d]['y'][int(i)] for d, i in batch]

        if self.augmentations:
            aug = choice(self.augmentations)
            return np.stack([aug(image=x)["image"] for x in batch_x], axis=0), np.array(batch_y)
        else:
            return np.array(batch_x), np.array(batch_y)

    def get_angles(self):
        y = [self.file[ds]["y"][int(i)] for ds, i in self.indices]
        return np.array(y)

    def on_epoch_end(self):
        if self.shuffle:
            np.random.shuffle(self.indices)


class H5LSTMGenerator(Sequence):
    def __init__(self, file, indices, batch_size, n_steps=5, augmentations=None, shuffle=False, threshold=0):
        self.file = file
        self.batch_size = batch_size
        self.augmentations = augmentations
        self.n_steps = n_steps
        self.shuffle = shuffle
        self.threshold = threshold
        self.indices = [[ds, i] for ds, i in indices if abs(self.file[ds]["y"][int(i)]) > self.threshold]

    def __len__(self):
        length = int(np.floor((len(self.indices)) / float(self.batch_size)))
        if len(self.indices) % float(self.batch_size) < self.n_steps:
            length -= 1
        if length < 1:
            raise Exception("Invalid number of indices (too short)")
        return length

    def __getitem__(self, idx):
        batch = self.indices[idx * self.batch_size: (idx + 1) * self.batch_size + self.n_steps - 1]
        batch_x, batch_y = [], []
        for index in range(self.n_steps, self.batch_size + self.n_steps):
            seq = batch[index - self.n_steps:index]
            batch_x.append([self.file[d]['X'][int(i)] for d, i in seq])
            batch_y.append(self.file[batch[index - 1][0]]['y'][int(batch[index - 1][1])])
        if self.augmentations:
            aug = choice(self.augmentations)
            return np.stack([np.array([aug(image=x)["image"] for x in seq]) for seq in batch_x], axis=0),
            ↪  np.array(
                batch_y, dtype=np.float32)
        else:
            return np.array(batch_x, dtype=np.float32), np.array(batch_y, dtype=np.float32)

    def get_angles(self):
        y = [self.file[ds]["y"][int(i)] for ds, i in
             self.indices[self.n_steps - 1:len(self) * self.batch_size + self.n_steps - 1]]
        return y
```

```
    def on_epoch_end(self):
        if self.shuffle:
            n_indices = int(np.floor(len(self.indices) / self.n_steps) * self.n_steps)
            temp = np.array(self.indices[:n_indices])
            temp = np.reshape(temp, (-1, self.n_steps, 2))
            np.random.shuffle(temp)
            temp = np.reshape(temp, (-1, 2))
            self.indices = temp.tolist() + self.indices[n_indices:]
```

Prilog P.3.8: Python skripta za definiranje modela neuronskih mreža - *models.py* (elektronički prilog)

Prilog P.3.9: Python skripta za učitavanje podataka iz *.h5* datoteka - *generators.py* (elektronički prilog)

Prilog P.3.10: Python skripta za treniranje i testiranje modela neuronskih mreža - *train_test.py* (elektronički prilog)

Prilog P.3.11: Python skripta za pokretanje MetaDrive simulatora i upravljanje vozilom - *md_main.py* (elektronički prilog)

Prilog P.3.12: Klijentska Python skripta za upravljanje vozilom unutar CARLA simulatora - *control.py* (elektronički prilog)

Prilog P.3.13: Virtualno okruženje CARLA simulatora (elektronički prilog)

Prilog P.3.14: Klijentska Python skripta za postavljanje virtualnog okruženja CARLA simulatora - *config.py* (elektronički prilog)