Tekuće neuronske mreže

Žnidarić, Domagoj

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:011211

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

VARAŽDIN

Domagoj Žnidarić

TEKUĆE NEURONSKE MREŽE

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Domagoj Žnidarić

Matični broj: 0016130157

Studij: Informacijsko i programsko inženjerstvo

TEKUĆE NEURONSKE MREŽE

DIPLOMSKI RAD

Mentor:

izv. prof. dr. sc. Nikola Ivković

Varaždin, rujan 2023.

Domagoj Žnidarić

Izjava o izvornosti

Izjavljujem da je ovaj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI Radovi

Sažetak

Tema ovog rada su tekuće neuronske mreže (engl. *liquid neural networks* - LNN) i njezina primjena u autonomnoj vožnji. Rad istražuje kako LNN-ovi unaprjeđuju sposobnost autonomnih vozila da razumiju i reagiraju na dinamične prometne uvjete.
Teorijsko-metodološka polazišta uključuju detaljno objašnjenje koncepta LNN-a i njegovu sposobnost modeliranja sekvencijalnih podataka. Također, opisuje se razvoj i nastanak LNN-a, od formule običnih diferencijalnih jednadžbi (engl. ordinary differential equations - ODE), do ideje alternativne formulacije, formule LNN-a.
GPraktični dio opisuje implementaciju LNN-a u simulatoru Carla pomoću informacija iz senzora camere.
Zaključci ističu važnost LNN-ova u autonomnoj vožnji i predviđaju daljnji razvoj ove tehnologije. Iako su LNN-ovi već postigli značajne rezultate, postoji prostor za daljnje istraživanje i poboljšanja kako bi se osigurala sigurnija i učinkovitija autonomna vožnja u budućnosti.

Ključne riječi: umjetna inteligencija; duboko učenje; tekuće neuronske mreže; računalni vid; autonomna vožnja; tensorflow;

Sadržaj

1. Uvod

Područje umjetne inteligencije (AI) (engl. *artificial intelligence - AI*) više nije nikome strani pojam. Pojavom ChatGPT-a, agenta temeljenog na velikom jezičnom modelu, zauvijek se promjenio svijet. Već godinama umjetna inteligencija utječe na društvo, donoseći koristi, ali i etičke zabrinutosti. Različite industrije, uključujući zdravstvo, fintech, transport i ostale, doživjele su transformacijski učinak jer AI značajno povećava učinkovitost, točnost i procese donošenja odluka. Ovo je samo još jedna revolucija u povijesti i mi svjedočimo tome.

Razlog odabira teme tekućih neuronskih mreža (TNN) (engl. *liquid neural network - LNN*) je zbog više razloga. Do nedavno u svijetu AI postojali su ograničeni algoritmi u smislu onoga što i na koji način mogu napraviti. Najveću intrigu dobile su neuronske mreže (NN) i područje dubokog učenja (engl. *deep learning*), jer je ideja bazirana na neuronima koji se nalaze u ljudskom mozgu i omogućava nam da iz sekunde u sekundu donosimo kompleksne odluke. Trenutno govoreći TNN su po svojoj jedinstvenoj strukturi najbliži biološkim mozgovima. Svatko tko se bavi sa područjem AI zna da su TNN nešto revolucionarno, što će zasigurno promjeniti cjelokupno područje dubokog učenja. Konačno, s obzirom na to da je područje TNN relativno novo, ovaj diplomski rad omogućava mi biti u pratnji s inovacijama, te mojom implementacijom doprinositi napretku u ovom obećavajućem području.

TNN su posebno istaknute u zadacima koji obuhvaćaju detekciju obrazaca, prognoziranje i učenje iz neprestanih podataka. To ih čini atraktivnim izborom za različite primjene, uključujući predviđanje vremenskih serija i analizu senzorskih podataka. Njihova korist i primjena prepoznate su u industrijama kao što su automobilska i medicinska, pa je izazov za diplomski rad razviti inteligentni sustav za autonomnu vožnju. Prethodno proučavanje teorije i implementacije tekućih neuronskih mreža ključno je kako bi se steklo jasno razumijevanje ovih mreža i temelj za izradu pravilnog inteligentnog sustava.

2. Metode i tehnike rada

S obzirom na to da je područje TNN relativno novo, ne postoje knjige koje opisuju tematiku, već je sve bazirano na znanstvenim člancima i popratnim kodom na github platformi. Za proučavanje korišteni su spomenuti znanstveni članci i video zapisi sa platforme Youtube.

Plan izrade diplomskog rada je prvotna implementacija praktičnog dijela, a onda opisivanje teorije na kojoj se bazira praktični dio. Teorijski dio rada temama pristupa postupno, počevši s općim pojmovima koji čine osnovu za kasnije teme. Cilj ovog pristup je olakšati razumijevanje kako se određene tehnologije ili koncepti uklapaju unutar šireg konteksta, kao što su inteligentni sustavi ili umjetna inteligencija. Što se tiče praktičnog dijela rada, on kombinira proces izrade aplikacije s grupiranjem funkcionalnosti različitih dijelova aplikacije, kao i komponenti inteligentnog sustava.

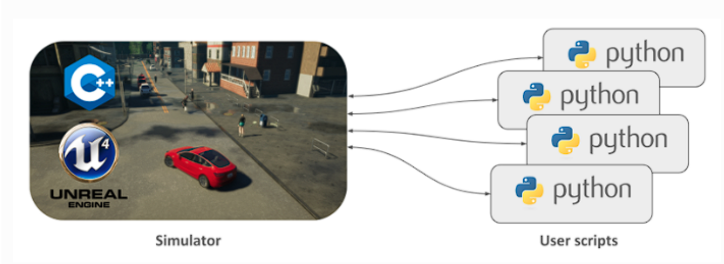
2.1. Korišteni alati

2.1.1. Visual studio code

Visual Studio Code lagani je, ali moćan uređivač koda razvijen od strane Microsofta dostupan za Windows, macOS i Linux. Dolazi s ugrađenom podrškom za JavaScript, TypeScript i Node.js te ima bogati proširenja (ekstenzija) za druge jezike, kao što su C ++, C#, Java, Python, PHP). Ne samo da sadrži proširenja za mnoštvo jezika, već za svaki jezik postoje službena ili korisnička proširenja za značajke koje uključuju podršku za ispravljanje pogrešaka, isticanje sintakse, inteligentno dovršavanje koda, isječke, preradu koda, ugrađeni sustav za verzioniranje koda (Git), itd.. Ono što ga čini odličnim alatom su proširujuće mogućnosti (engl. *extensions*), programeri ga mogu prilagoditi svojim specifičnim potrebama, čime ga čine osobnijim i učinkovitijim alatom.

2.1.2. Carla simulator

Carla simulator je moćan alat koji se koristi u području istraživanja i razvoja autonomne vožnje. Pruža realno virtualno okruženje za testiranje i obuku autonomnih vozila. Razvijen od strane Computer Vision Centra (CVC) i Universitat Autònoma de Barcelona (UAB), simulator Carla nudi širok spektar značajki i funkcionalnosti koje ga čine vrijednim resursom kako za istraživače, inženjere, tako i za studente [1]. S visoko detaljnim i prilagodljivim urbani okolinama, simulator Carla omogućava korisnicima simuliranje različitih scenarija vožnje i testiranje performansi različitih algoritama i sustava upravljanja. Nudi poveliku kolekciju unaprijed izrađenih karta gradova, scenarija prometa i modela senzora koji se lako mogu modificirati i prilagoditi specifičnim potrebama istraživanja. Nadalje, simulator Carla podržava različite senzore, uključujući kamere, lidare i radare, omogućavajući korisnicima precizno simuliranje sposobnosti percepcije autonomnih vozila.



Slika 1: Carla python api; preuzeto iz [1]

Simulator također pruža Python API koji korisnicima omogućava interakciju s simuliranim okruženjem i upravljanje ponašanjem virtualnih vozila. To omogućava istraživačima razvoj i testiranje vlastitih algoritama i strategija upravljanja. Dodatno, simulator Carla podržava integraciju vanjskog softvera i knjižnica, olakšavajući upotrebu najsuvremenijih algoritama i okvira za strojno učenje. Sveukupno, simulator Carla neprocjenjiv je alat za one koji se bave istraživanjem autonomne vožnje, pružajući realnu i fleksibilnu platformu za testiranje i usavršavanje tehnologija autonomnih vozila.

Tablica 1: Sažetak Gradova [1]

Grad	**Sažetak**
Grad01	Mali, jednostavan grad s rijekom i nekoliko mostova.
Grad02	Mali jednostavan grad s mješavinom stambenih i komercijalnih zgrada.
Grad03	Veći urbanistički plan s kružnim tokom i velikim raskrižjima.
Grad04	Mali grad smješten u planinama s posebnim beskrajnim autoputom u obliku broja "8".
Grad05	Grad s kvadratnim mrežnim planom, križnim raskrižjima i mostom. Ima više traka u svakom smjeru. Koristan za izvođenje promjena traka.
Grad06	Duge autoceste s mnogo ulaza i izlaza s autoceste. Također ima P zaokret.
Grad07	Seosko okruženje s uskim cestama, kukuruzom, štalama i rijetkim semaforima.
Grad08	Tajni "neviđeni" grad koji se koristi za izazov na ljestvici.
Grad09	Tajni "neviđeni" grad koji se koristi za izazov na ljestvici.
Grad10	Središnje urbanističko okruženje s neboderima, stambenim zgradama i šetnicom uz ocean.
Grad11	Velika karta koja nije uređena. Služi kao dokaz koncepta za izgradnju velikih karata.
Grad12	Velika karta s brojnim različitim regijama, uključujući visokoprizemnice, stambene i seoske okoline.

2.1.3. Anaconda

Anaconda je popularna i open-source distribucija programskih jezika Python i R koja se koristi za znanost o podacima (DS), strojno učenje i znanstveno računanje. Pruža sveobuhvatan ekosustav alata, biblioteka i paketa koji olakšavaju upravljanje i implementaciju projekata iz područja AI.

Ključne značajke i komponente Anaconde:

- Upravljanje paketima: Anaconda uključuje upravitelja paketima nazvanog "conda" koji pojednostavljuje instalaciju i upravljanje Python i R paketima. Conda može stvarati i upravljati izolirane okoline, omogućavajući rad s različitim verzijama paketa bez konflikata.
- Biblioteke za DS: Anaconda dolazi s širokim rasponom knjižnica i okvira za DS, uključujući NumPy, SciPy, pandas, scikit-learn, TensorFlow, PyTorch i druge.
- Jupyter bilježnice: Anaconda uključuje Jupyter Notebook, interaktivno web bazirano okruženje za izradu i dijeljenje dokumenata koji sadrže "živi" kod, jednadžbe, vizualizacije i narativni tekst. To je popularan izbor za istraživanje i prototipiranje podataka.
- Integrirana razvojna okruženja (IDE): može se integrirati s popularnim Python IDE-ovima poput JupyterLaba, Spydera i Visual Studio Code-a, pružajući praktično razvojno okruženje za data znanstvenike i programere.
- Vizualizacija podataka: uključuje knjižnice poput Matplotliba, Seaborna, Plotlyja i Bokeha za stvaranje vizualizacija i grafikona.
- Paralelno računanje: podržava paralelno i distribuirano računanje s alatima poput Daska i mpi4py, što ga čini pogodnim za rukovanje velikim skupovima podataka i računalno zahtjevnim zadacima.
- Zajednica i podrška: ima aktivnu i podržavajuću zajednicu, s resursima poput dokumentacije, foruma i tutorijala koji su dostupni kako bi korisnicima pomogli da započnu i rješavaju probleme.
- Višekratna platforma: Anaconda je dostupna za Windows, macOS i Linux, što je čini dostupnom širokom spektru korisnika i okruženjima.

Anaconda se najviše koristi zbog izoliranih okolina, jer u firmama se često radi na više projekata, a svaki projekt zahtjeva različite biblioteke i okoline, pa kako se verzije ne bi preklapale i okoline držale organizirane koristi se ova funkcionalnost.

2.2. Teorijska podloga

2.2.1. Umjetna inteligencija

Umjetna inteligencija (UI) postala je ključna snaga u današnjem svijetu vođenom tehnologijom, revolucionirajući različite aspekte naših života. S brzim rastom i razvojem, AI je dala značajan doprinos u mnogim područjima.

Prema Copelandu [2], AI je sposobnost digitalnog računala ili računalno kontroliranog robota da obavlja zadatke koji su obično povezani s inteligentnim bićima. Autor spominje da iako je računalna obrada postigla znatan napredak u brzini i kapacitetu memorije, još uvijek nema programa koji mogu parirati potpunoj ljudskoj fleksibilnosti u širem spektru područja ili u zadacima koji zahtijevaju opće znanje. S druge strane, neki programi su postigli razinu izvedbe ljudskih stručnjaka i profesionalaca u obavljanju specifičnih zadataka, što znači da se umjetna inteligencija koristi u ograničenim područjima kao što su medicinska dijagnoza, računalne tražilice, prepoznavanje glasa ili rukopisa i chatbotovi. Moglo bi se reći da pojavom velikih modela za obradu prirodnog jezika (LLM) (engl. *large language models* i tekst u sliku model (engl. *text to image model*) AI nije više ograničena za određena područja, već je dostupna, a i korisna svima.

Prema IBM-u [3], AI je polje koje kombinira računalne znanosti i robusne skupove podataka kako bi se omogućilo rješavanje problema. Također obuhvaća područja strojnog učenja i dubokog učenja, koja su pod područje umjetne inteligencije. Znači AI predstavlja interdisciplinarno područje koje koristi koncepte i tehnike iz računalnih znanosti, kao što su algoritmi, programiranje i računalni sustavi, te robusni skupovi podataka koji se odnose na velike i raznovrsne skupove informacija koje sustavi AI koriste za učenje i donošenje odluka.

AI duboko je utjecala na različite aspekte naših svakodnevnih života, od zdravstva i prijevoza do zabave i komunikacije. Međutim, brzi napredak u tehnologiji UI-a izazvao je značajne etičke zabrinutosti i potaknuo rasprave o njegovim budućim implikacijama. Prema Burtonu i suradnicima [4], te zabrinutosti uglavnom se odnose na pitanja kao što su privatnost, pristranost, odgovornost i potencijalni gubitak ljudskih poslova. S sposobnošću UI-a za prikupljanje i analizu velikih količina osobnih podataka, raste zabrinutost u vezi s privatnošću i sigurnošću pojedinaca. Zloupotreba ili nepropisno postupanje s tim podacima može dovesti do ozbiljnih posljedica, uključujući krađu identiteta i nadzor.

2.2.1.1. Povijest

Povijest umjetne inteligencije: ključni datumi i imena prema IBM-u [3]:

1950.: Alan Turing objavljuje rad "Computing Machinery and Intelligence". U tom radu, Turing, poznat po dešifriranju nacističkog ENIGMA koda tijekom Drugog svjetskog rata, predlaže odgovor na pitanje "mogu li strojevi razmišljati?" i uvodi Turingov test kako bi se utvrdilo može li računalo pokazati istu inteligenciju (ili rezultate iste inteligencije) kao ljudsko biće. Vrijednost Turingovog testa raspravlja se od tada.

1956.: John McCarthy prvi puta koristi izraz "umjetna inteligencija" na prvoj ikad održanoj AI konferenciji na Dartmouth fakultetu. (McCarthy će kasnije izmisliti programski jezik Lisp.) Kasnije te godine, Allen Newell, J.C. Shaw i Herbert Simon stvaraju Logic Theorist, prvi ikad pokrenuti AI softverski program.

1967.: Frank Rosenblatt gradi $Mark1Perceptron$, prvo računalo temeljeno na neuronskoj mreži koje "uči" putem ispitivanja i greške. Samo godinu dana kasnije, Marvin Minsky i Seymour Papert objavljuju knjigu pod nazivom "Perceptrons", koja postaje ključno djelo o neuronskim mrežama i, barem neko vrijeme, argument protiv budućih istraživačkih projekata u području neuronskih mreža.

1980-ih: Neuronske mreže koje koriste algoritam povratnog širenja (engl. *backpropagation*) postaju široko korištene u AI aplikacijama.

1997.: IBM-ov Deep Blue pobjeđuje tadašnjeg svjetskog prvaka u šahu Garryja Kasparova u meču (i revanšu).
2011.: IBM-ov Watson pobjeđuje prvake Kena Jenningsa i Brada Ruttera u kvizu "Jeopardy!".

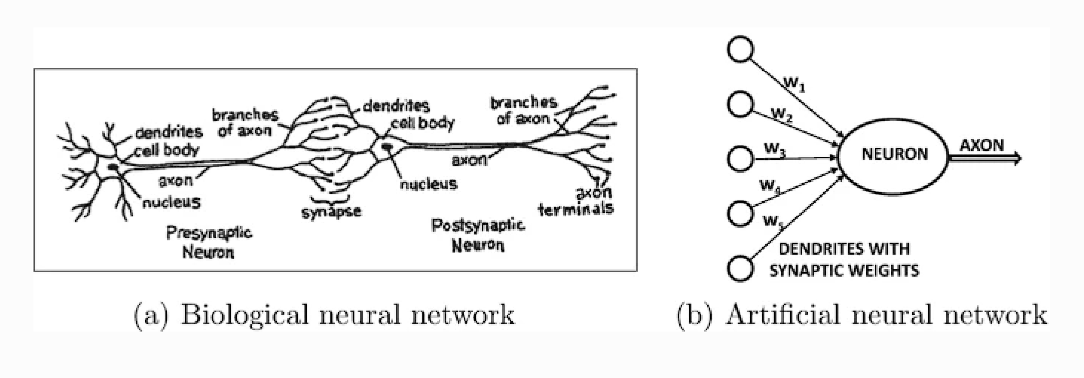
2015.: Baiduov superkompjuter Minwa koristi poseban tip duboke neuronske mreže nazvan konvolucijska neuronska mreža za identifikaciju i kategorizaciju slika s većom točnošću od prosječnog čovjeka.

2016.: DeepMindov AlphaGo program, pogonjen dubokom neuronskom mrežom, pobjeđuje Leeja Sedola, svjetskog prvaka u igri Go, u meču od pet igara. Pobjeda je značajna s obzirom na ogroman broj mogućih poteza kako igra napreduje (više od 14,5 bilijuna nakon samo četiri poteza). Kasnije je Google kupio DeepMind za navodnih 400 milijuna dolara.

2023.: Porast LLM, poput ChatGPT, donosi ogromne promjene u performansama AI i njegovom potencijalu za stvaranje vrijednosti za tvrtke. Pomoću ovih novih praksi generativne umjetne inteligencije, duboko učenje može biti prethodno trenirano na ogromnim količinama sirovih, neoznačenih podataka.

2.2.2. Neuronske mreže i duboko učenje

Neuronske mreže i duboko učenje revolucionirali su područje umjetne inteligencije, otvarajući put za prijelomne napretke u različitim domenama. Razumijevanje ključnih koncepata i tehnika dubokog učenja za neuronske mreže omogućava nam da cijenimo kompleksnost i snagu ovih modela. Nadalje, proučavanjem primjena i budućih implikacija neuronskih mreža i dubokog učenja možemo zamisliti svijet u kojem inteligentni sustavi mogu rješavati složene probleme i unaprijediti ljudske živote na dosad neviđene načine.

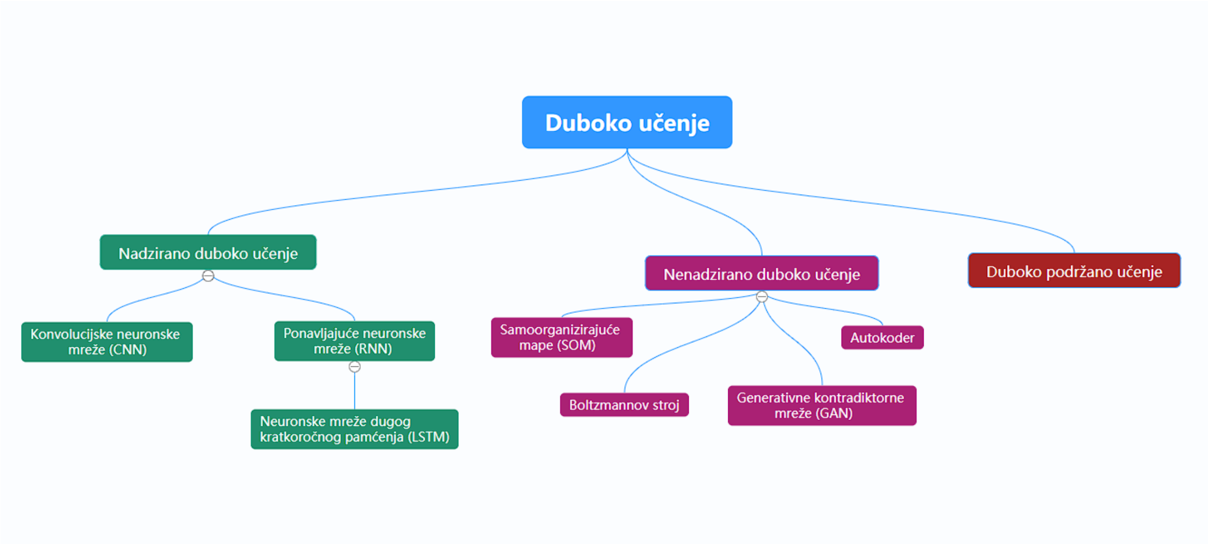


Slika 2: (a) biološka neuronska mreža; (b)umjetna neuronska mreža; preuzeto iz [5]

Umjetne neuronske mreže popularna su tehnika strojnog učenja koja oponaša proces učenja koji se događa u biološkim organizmima. U ljudskom živčanom sustavu postoje specijalizirane stanice poznate kao neuroni, a ovi neuroni međusobno komuniciraju putem aksona i dendrita, stvarajući sinaptičke veze (engl. *synapse*) između njih. Važno je napomenuti da jačina sinaptičkih veza može varirati kao odgovor na vanjske podražaje, što predstavlja mehanizam učenja u živim organizmima.

Prema Aggarwalu [5], neuronske mreže predstavljaju složene strukture sastavljene od različito povezanih slojeva umjetnih neurona, čije je funkcioniranje inspirirano organizacijom ljudskog mozga. Ovi slojevi mogu se svrstati u tri glavne kategorije: ulazne, skrivene i izlazne slojeve. Početni ulazni sloj prima sirove podatke, a zatim se ti podaci obrađuju putem niza skrivenih slojeva koji transformiraju ulaz u apstraktne reprezentacije. Svaki neuron u skrivenim slojevima primjenjuje ponderirani zbroj ulaznih podataka te prolazi kroz aktivacijsku funkciju radi uvođenja nelinearnosti. Konačna predviđanja ili klasifikacije generira izlazni sloj, temeljene na informacijama naučenim iz prethodnih slojeva.

Algoritmi dubokog učenja rade s gotovo svim vrstama podataka i zahtijevaju velike količine računalne snage i informacija za rješavanje kompliciranih problema, a na slici 3. prikazana je podjela algoritama dubokog učenja na nadzirano (engl. *supervised*, nenadzirano (engl. *unsupervised*) i duboko podržano učenje (engl. *deep reinforcement learning*). Za potrebe ovog rada objasnit će se samo algoritmi nadziranog učenja.

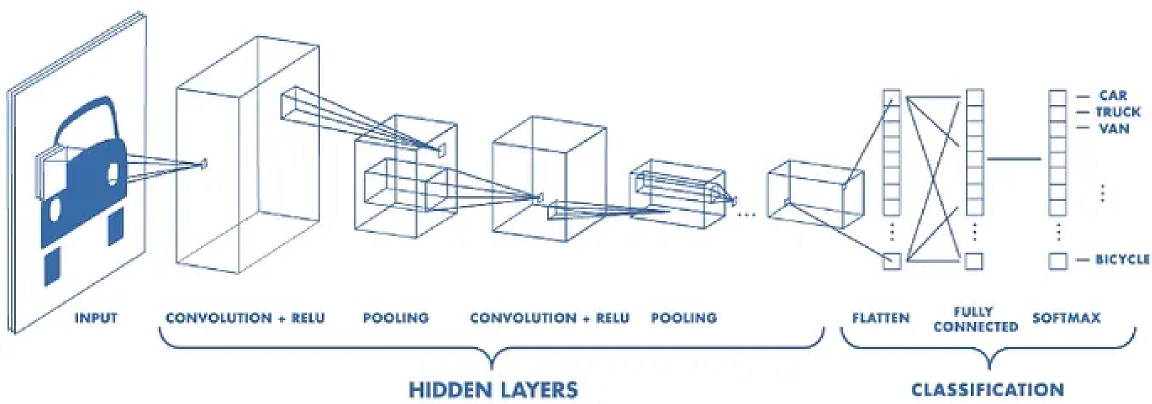


Slika 3: Algoritmi dubokog učenja; vlastita slika

2.2.2.1. Konvolucijske neuronske mreže (CNN)

Jedne od najkorištenijih neuronskih mreža u području računalnog vida su upravo konvolucijske neuronske mreže (CNN). CNN su nastale zbog potrebe na tržištu za izdvajanje značajki iz slika, koje su se prije postojanja CNN radile ručno, a to je mučan i dugotrajni proces. CNN sada pružaju skalabilniji pristup klasifikaciji slika i zadacima prepoznavanja objekata, koristeći principe linearne algebre, posebnog matričnog množenja, za prepoznavanje uzoraka unutar slike[3]. CNN imaju tri glavne vrste slojeva:

- Konvolucijski sloj
- Sloj udruživanja (engl. *pooling*)
- Potpuno povezani sloj (engl. *fully connected layer*)



Slika 4: CNN; preuzeto sa [6]

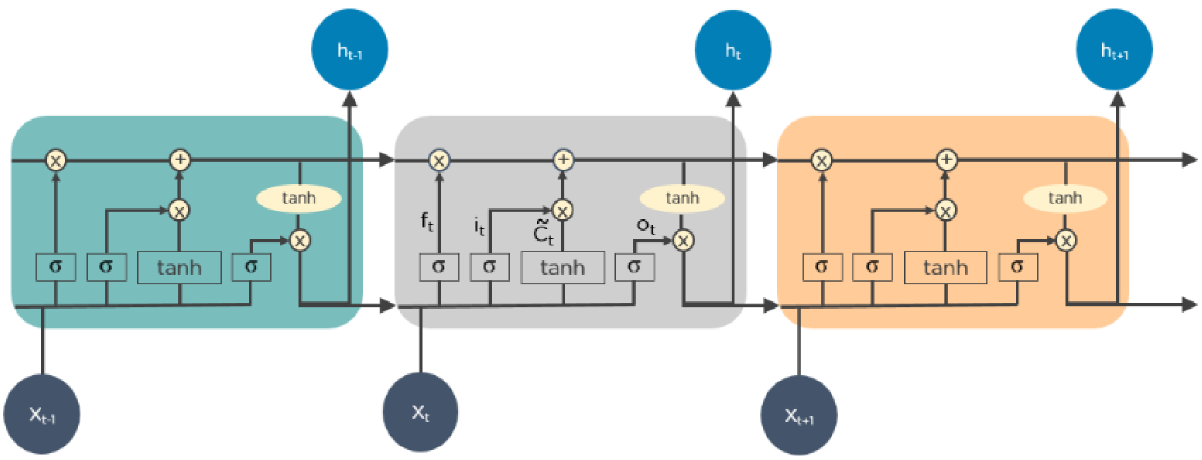
Konvolucijski sloj je prvi sloj kada se grade konvolucijske mreže, konvolucijski slojevi mogu biti praćeni dodatnim konvolucijskim slojevima ili slojevima udruživanja, potpuno povezani sloj je završni sloj. Sa svakim slojem CNN postaje sve složeniji, identificirajući veće dijelove slike. Raniji slojevi fokusirani su na jednostavne značajke, kao što su boje i rubovi. Kako slikovni podaci napreduju kroz slojeve CNN, počinju prepoznavati veće elemente ili oblike objekta dok konačno ne identificira željeni objekt.

2.2.2.2. Ponavljajuće neuronske mreže (RNN)

ponavljajuće neuronske mreže (RNN) su vrste umjetnih neuronskih mreža koje rade na sekvencijalan način i treniraju se pomoću algoritma povratnog širenja na označenim sekvencama obuke konačne duljine [7]. RNN su se pokazale problematične zbog njihovog gradijenta koji nestaje (engl. *vanishing gradient problem*) tijekom faze učenja, pa je ovaj problem riješen uvođenjem neuronske mreže dugog kratkoročnog pamćenja (LSTM), koji osigurava konstantan tok pogreške i uklanjanje nelinearnosti [8].

2.2.2.3. Neuronske mreže dugog kratkoročnog pamćenja (LSTM)

LSTM su vrsta RNN koja može učiti i dulje pamtiti prijašnje informacije. LSTM, s lančanom strukturom i četiri međusobno povezana sloja, zadržavaju informacije tijekom vremena i koriste se za predviđanje vremenskih serija pamteći prethodne ulaze [9]. Zbog mogućnosti pamćenja informacija ovi algoritmi odlični su za predviđanja podatke vremenskih serija, kao što su video zapisi, prepoznavanje govora, skladanje glazbe i autonomna vožnja.



Slika 5: LSTM; preuzeto sa [9]

Slika 6 prikazuje kako LSTM funkcioniraju. Prvo, eliminiraju nepotrebne aspekte prethodnog stanja, nakon toga, pažljivo ažuriraju vrijednosti stanja ćelije te na kraju, izvlače izlaz iz određenih dijelova stanja stanice. Na taj način osiguravaju pamćenje samo potrebnih informacija iz prijašnjih stanja.

2.2.3. Računalni vid i autonomna vozila

Računalni vid i autonomna vozila posljednjih su godina postali sve istaknutije polje istraživanja i razvoja. S pojavom naprednih tehnologija, kao što su strojno učenje i umjetna inteligencija, računalni vid igra ključnu ulogu u razvoju i radu autonomnih vozila.

Slika 6: Računalni vid za autonomna vozila; preuzeto sa [10]

Računalni vid igra vitalnu ulogu u razvoju i implementaciji autonomnih vozila, omogućujući im da percipiraju i tumače svoju okolinu. Prema Ranftu i Stilleru [11], računalni vid ključna je tehnologija koja autonomnim vozilima omogućuje razumijevanje okoline, otkrivanje i praćenje objekata te donošenje informiranih odluka na temelju prikupljenih vizualnih informacija. Upotrebom kamera i sofisticiranih algoritama, sustavi računalnog vida mogu analizirati slike i video zapise u stvarnom vremenu, izvlačeći relevantne informacije kao što su oznake prometnih traka, prometni znakovi i pješaci. Te informacije su ključne za sigurnu i učinkovitu navigaciju autonomnih vozila na cesti. Nadalje, tehnike računalnog vida kao što su prepoznavanje i klasifikacija objekata omogućuju autonomnim vozilima da otkriju i identificiraju različite objekte i prepreke na svom putu, omogućujući im da reagiraju u skladu s tim. Integracija računalnog vida s drugim senzorskim tehnologijama, poput LiDAR-a i radara, dodatno poboljšava sposobnosti percepcije autonomnih vozila, pružajući sveobuhvatno razumijevanje okolnog okoliša.

Računalni vid ključan je za autonomna vozila, no postoje izazovi u vezi s raznolikim okruženjima i brzom obradom podataka. Autonomna vozila moraju navigirati kroz različite scenarije, uključujući gradske ulice, autoceste i različite uvjete. Osim toga, trebaju brzu obradu podataka s kamera, lidara i radara kako bi donijela trenutačne odluke. Unatoč tim izazovima, algoritmima strojnog i dubokog učenja ostvaren je značajan napredak u računalnom vidu za autonomna vozila. Janai i sur. [12] raspravljaju o različitim pristupima i tehnikama koje su razvijene, kao što su detekcija objekata, praćenje i semantička segmentacija. Ovaj napredak otvorio je put razvoju sofisticiranijih autonomnih vozila koja mogu sigurno i učinkovito upravljati u složenim stvarnim okruženjima.

Prije nego što izađu na prometnice, samovozeći automobili prvo moraju proći kroz 6 razina. Društvo automobilskih inženjera (SAE) definira tih 6 razina automatizacije vožnje u rasponu od 0 (potpuno ručno) do 5 (potpuno autonomno):

Tablica 2: Nivoi automatizacije vožnje po SAE-u; Preuzeto sa [13]

Nivo	**Opis**
0 - Nema automatizacije	Vozač upravlja svojim vozilom i ne koristi nikakvu pomoć od strane vozila.
1 - Pomoć vozaču	Vozilo pruža pomoć u obliku upozorenja ili brzog reagiranja na opasnost, ali vozač mora biti u mogućnosti da preuzme kontrolu nad vozilom u bilo kojem trenutku.
2 - Djelomična automatizacija	Vozilo samostalno upravlja dijelovima vožnje, kao što su brzina i smjer, ali vozač mora biti uvijek spreman da preuzme kontrolu nad vozilom.
3 - Uvjetna automatizacija	Vozilo može samostalno upravljati većinom vožnje, ali vozač mora biti spreman da preuzme kontrolu nad vozilom u određenim situacijama.
4 - Visoka automatizacija	Vozilo može samostalno upravljati gotovo cijelom vožnjom, ali vozač mora biti uvijek spreman da preuzme kontrolu nad vozilom u određenim situacijama.
5 - Potpuna automatizacija	Vozilo može samostalno upravljati cijelom vožnjom bez potrebe za vozačevom intervencijom.

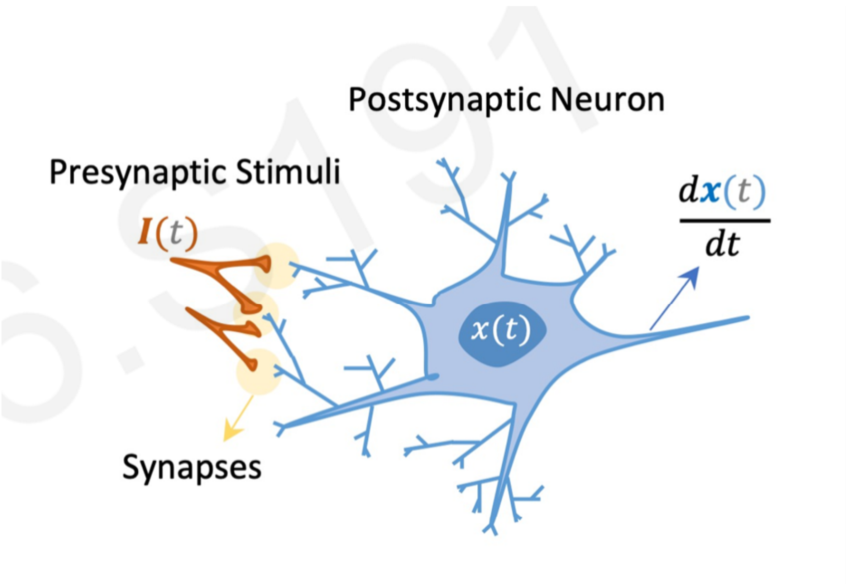
Implementacija računalnog vida u autonomnim vozilima također pokreće različita etička pitanja koja treba pažljivo razmotriti. Tehnologija računalnog vida omogućuje vozilima da opažaju i tumače svoju okolinu pomoću kamera i senzora, što im omogućuje donošenje odluka i navigaciju bez ljudske intervencije. Međutim, etički problemi javljaju se kada se razmatraju implikacije oslanjanja isključivo na računalne algoritme za kritično donošenje odluka u situacijama potencijalno opasnim po život. Jednood njavažnijih etičkog pitanja je zasigurno pitanje odgovornosti u slučaju nezgoda ili pogrešaka. Budući da autonomna vozila uvelike ovise o računalnom vidu, svaki kvar ili pogrešno tumačenje vizualnih podataka moglo bi imati teške posljedice. U takvim slučajevima postaje ključno utvrditi tko ili za što treba biti odgovoran.

2.2.4. Tekuće neuronske mreže

Tekuće vremenski konstantne neuronske mreže (LTC) (eng. *Liquid Time-Constant Networks* - LTCs) pojavile su se kao fascinantan koncept u računalnoj znanosti, nudeći novu perspektivu obrade informacija. Spadaju pod vrstu RNN koja se ističu u obradi vremenskih informacija. Prema Hasaniju i dr. [14], za razliku od tradicionalnih metoda definiranja dinamike učenja putem implicitnih nelinearnosti, ovaj pristup uvodi mreže s tekućim vremenskim konstantama. Ove mreže sastoje se od linearnih dinamičkih sistema prvog reda koji se upravljaju nelinearnim međusobno povezanim čvorovima. Rezultirajući modeli predstavljaju dinamičke sustave s promjenjivim vremenskim konstantama, povezane s njihovim skrivenim stanjem, i koriste numeričke metode za rješavanje diferencijalnih jednadžbi za izračun izlaza. Ove neuronske mreže pokazuju stabilno i ograničeno ponašanje i nude poboljšanu izražajnost unutar područja neuronskih običnih diferencijalnih jednadžbi (ODE) (engl. *ordinary differential equations*) [14]. Štoviše, demonstriraju poboljšane performanse u zadacima predviđanja vremenskih serija.

U tradicionalnoj neuronskoj mreži s diskretnim slojevima, unaprijed je definiran broj slojeva, a svaki sloj sastoji se od fiksnog broja neurona. Ti su slojevi naslagani jedan na drugi, a informacije kroz njih teku u diskretnim koracima ili slojevima tijekom treniranja. Kod LTC ti neuroni se prilagođavaju tijekom procesa učenja, a prilagođavaju se tako što njima upravljaju vremenski konstantni parametri, koji određuju stope opadanja unutarnjih stanja mreže. To omogućuje LTC da hvataju i obrađuju vremenske ovisnosti u podacima, što ih čini posebno učinkovitima u zadacima koji uključuju sekvencijalne ili vremenski promjenjive informacije.

Učinkovitost LTC leži u njihovoj sposobnosti modeliranja dinamičkih sustava, omogućujući im hvatanje dugoročnih vremenskih ovisnosti i pravljenje točnih predviđanja. Razumijevanjem i iskorištavanjem snage LTC, računalni znanstvenici mogu unaprijediti polja obrade autonomne vožnje, prepoznavanja govora i drugih domena koje se oslanjaju na analizu vremenski promjenjivih podataka.



Slika 7: Spoj sinapsa i neurona; preuzeto iz [15]

Slika 7 prikazuje spoj sinapse i neurona, te taj dio razlikuje tekuće neuronske mreže (LNN) od umjetnih neuronskih mreža. $I(t)$ je ulazna vrijednost, a $x(t)$ skriveno stanje, te oboje ovise o vremenu (t)

2.2.4.1. Formule

ODE

Za olakšanje razumijevanja LNN kao potklasu neuralnih ODE, nužno je zadubiti se u sveobuhvatno razumijevanje neuralnih ODE. Za postignuće ovog cilja, prvo je potrebno razmotriti koncept neuralnih ODE pomoću dane formule:

$$dx(t)/dt = f(x(t), t, \theta)$$

- $\frac{dx(t)}{dt}$: predstavlja brzinu promjene stanja varijable x u odnosu na vrijeme t. Drugim riječima, pokazuje kako se stanje x razvija tijekom vremena.
- $f(x(t), t, \theta)$: funkcija neuronske mreže koja definira kako se stanje x mijenja tijekom vremena. Prima kao ulaz trenutačno stanje $x(t)$, trenutačno vrijeme t i skup parametara θ povezanih s neuronskom mrežom. Ova funkcija zapravo obuhvaća ponašanje kontinuiranog sustava.
- $x(t)$: Ovo predstavlja stanje sustava u trenutku t. Može biti vektor ili skup varijabli koji opisuju stanje.
- t: neovisna varijabla, što je vrijeme u ovom kontekstu. To označava trenutačnu točku u vremenu u kojoj se procjenjuje stanje x.
- θ: parametri funkcije neuronske mreže f. Ovi se parametri uče tijekom procesa treniranja kako bi se uhvatila osnovna dinamika kontinuiranog sustava.

CT RNN

Umjesto definiranja izvedenica skrivenog stanja izravno pomoću neuronske mreže f, može se odrediti stabilnija vremenski kontinuirana ponavljajuća neuronska mreža (engl. *Continuous-time recurrent neural network* - CT RNN), sljedećom funkcijom:

$$\frac{dx(t)}{dt} = -\frac{x(t)}{\tau} + f(x(t), I(t), t, \theta)$$

U gornjoj funkciji izraz $(-x(t)/\tau)$ pomaže autonomnom sustavu da postigne ravnotežno stanje s vremenskom konstantom τ [14].

- $x(t)$ je skriveno stanje
- $I(t)$ je ulaz
- t predstavlja vrijeme
- f je parametrizirano sa θ.

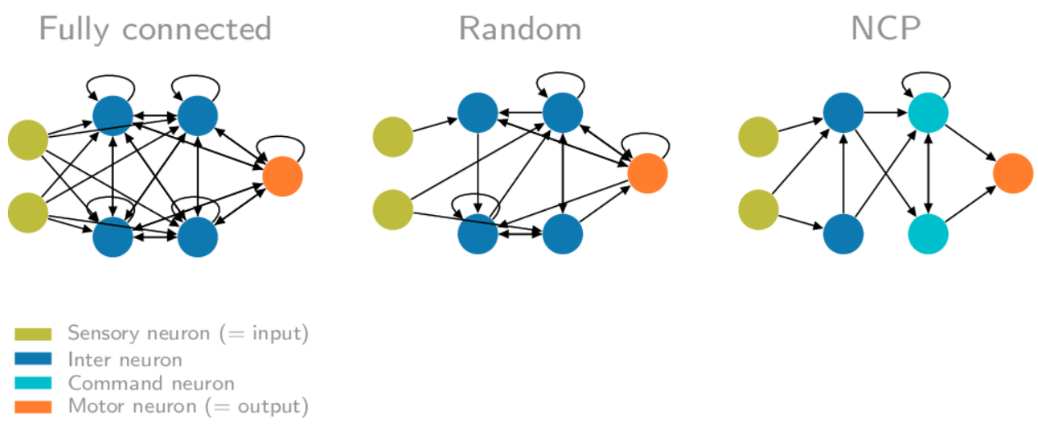
LNN

Hasani i dr. predlažu alternativnu formulaciju, a to je da se pusti da tok skrivenog stanja mreže bude deklariran sustavom linearnih ODE [14]: $\frac{dx(t)}{dt} = (-x(t)/\tau) + S_t$, i neka $S(t)$ predstavlja sljedeću nelinearnost određenu prema $S(t) = f(x(t), I(t), t, \theta)(A - x(t)$. Kada se S postavi unutar skrivenog sloja dobije se sljedeće:

$$\frac{dx(t)}{dt} = -\frac{x(t)}{\tau} + f(x(t), I(t), t, \theta)x(t) + f(x(t), I(t), t, \theta)A$$

U ovoj formulaciji promjenjiva vremenska konstanta τ_{sys} za sustav učenja ovisi o neuronskoj mreži `f` koja služi kao promjenjivi vremenski sustav ovisan o ulazu (vremenska konstanta je parametar koji karakterizira brzinu i osjetljivost povezivanja ODE) [14]. Ovo svojstvo omogućuje pojedinačnim elementima skrivenog stanja da za svaku vremensku točku iidentificiraju specijalizirane dinamičke sustave za ulazne značajke.

Dakle, ukratko, formula opisuje kako se derivacija $x(t)$ mijenja tijekom vremena, uzimajući u obzir učinke vremenske konstante τ i nelinearne funkcije f koja ovisi o $x(t)$, $I(t)$, t i θ. Funkcija neuronske mreže f modelira interakciju između varijabli, a parametar A skalira snagu te interakcije.



Slika 8: Različiti načini umrežavanja u RNN; preuzeto iz [16]

Slika 8 prikazuje kako izgledaju različito umrežavanje neurona u RNN. U praktičnom dijelu biti će detaljnije objašnjeno umrežavanje, sa različitim primjerima u praksi.

2.2.4.2. C. elegans

LNN su biološki nadahnute prema strukturi mozga $Caenorhabditiselegans$, mikroskopske, slobodoživuće, prozirne nematode (valjkasti crvi, obli crvi, oble gliste) ne duže od 1 mm. Istraživali su temeljno izračunavanje njegovog mozga, koji ima samo 302 neurona u živčanom sustavu, a ipak stvara složenu i adaptivnu dinamiku. Njegovi neuroni raspoređeni su u module koji sadrže neuronske krugove koji igraju ključnu ulogu u izvođenju različitih funkcija: kemosenzacija, termotaksija, mehanosenzacija, hranjenje, itd. [17]. Od svojih 302 neurona u crvu, samo se 4-6 neurona može smatrati čvorištima.

Postoji dovoljno dokaza koji sugeriraju da se arhitektura živčanih sustava životinja, od najjednostavnijih crva do složenih ljudi, temelji na istim osnovnim principima organizacije. Ovi principi su usklađeni s univerzalnim potrebama organizama za istovremenom obradom različitih vrsta senzorskih informacija. To je zbog činjenice da senzorski neuroni imaju ograničenje u kodiranju samo jednog tipa senzorskih podataka, pa organizmi moraju kombinirati ove različite vrste informacija kako bi donijeli odluke. [18]

2.2.4.3. Senzorna matrica susjedstva

Senzorna matrica susjedstva (engl. *sensory adjacency matrix*) je alat koji se koristi u neuroznanosti i umjetnoj inteligenciji za modeliranje veza između različitih osjetilnih sustava ili modaliteta u mozgu. To je matrica koja predstavlja prostorne odnose između različitih osjetilnih receptora ili neurona, poput onih odgovornih za vid, sluh, dodir, okus i miris. Matrica se može koristiti za simuliranje kako senzorske informacije iz jednog modaliteta mogu utjecati ili "prelijevati se" u drugi modalitet, stvarajući integriraniju i potpuniju percepciju okoline [19].

U kontekstu neuronskih mreža, matricu osjetilne susjednosti može se koristiti za dizajniranje modela koji oponašaju način na koji mozak procesira senzorske informacije. Na primjer, istraživači su koristili matrice osjetilne susjednosti kako bi razvili neuronske mreže koje mogu istovremeno obraditi vizualne i auditivne informacije, što dovodi do poboljšane izvedbe na zadacima poput prepoznavanja govora i prepoznavanja objekata [19].

Senzorna matrica susjedstva pruža okvir za mapiranje načina na koji se senzorne informacije obrađuju i povezuju unutar LTC.

Slika 9: Aktivacija neurona; preuzeto iz [20]

Senzorna matrica susjedstva omogućava dinamičnu aktivaciju neurona u LNN, za razliku od standarnih neuronskih mreža kod kojih je aktivnost statična 9.

Prednosti korištenja matrica osjetilne susjednosti u neuronskim mrežama uključuju:

1. Poboljšana izvedba na multisenzorskim zadacima: Simuliranjem načina na koji mozak integrira senzorske informacije iz više modaliteta, ovi modeli mogu bolje obavljati zadatke koji zahtijevaju obradu više vrsta senzorskog ulaza.

2. Povećana otpornost na buku i pogreške: Uzimajući u obzir prostorne odnose između senzorskih receptora, ovi modeli mogu biti manje osjetljivi na smetnje ili pogreške u pojedinim modalitetima.

3. Poboljšana generalizacija na nove situacije: Obukom na raznolikom skupu senzorskih ulaza, ovi modeli mogu naučiti prepoznavati obrasce i odnose između različitih modaliteta, poboljšavajući njihovu sposobnost generalizacije na nove situacije.

4. Bolje razumijevanje ljudske kognicije i percepcije: Istražujući svojstva matrica osjetilne susjednosti i njihovu ulogu u modelima neuronskih mreža, istraživači mogu dobiti uvid u to kako mozak procesira i integrira senzorske informacije, što može obogatiti naše razumijevanje ljudske kognicije i percepcije.

5. Potencijalne primjene u različitim područjima: Matrice osjetilne susjednosti mogu se primijeniti u širokom spektru domena, uključujući robotiku, autonomna vozila, zdravstvo, obrazovanje i zabavu. Na primjer, mogu se koristiti kako bi se poboljšala izvedba sustava virtualne stvarnosti ili poboljšala dostupnost multimedijskog sadržaja osobama s invaliditetom.

Model	Conv layers Param	RNN neurons	RNN synapses	RNN trainable param
CNN	5,068,900	-	-	-
CT-RNN	79,420	64	6112	6273
LSTM	79,420	64	24640	24897
NCP	79,420	**19**	**253**	**1065**

Slika 10: Performanse različitih modela; preuzeto iz [14]

Uspoređivanje različitih modela donosi zaključak, da do jednakog ili čak i boljeg rezultata moguće je i doći sa puno manje neurona i veza između njih, što dovodi do manje potrebe korištenja resursa i brže treniranje modela.

Izazovi

U području istraživanja LTC identificirano je nekoliko izazova koji koče razvoj ovog obećavajućeg područja. Bidollahkhani, Atasoy i Abdellatef [21] ističu poteškoće povezane s implementacijom LTC. Jedan od glavnih izazova je točno modeliranje i simulacija takvih mreža, budući da uključuju složene fizičke pojave i interakcije. Dodatno, autori ističu poteškoće u postizanju stabilnosti u tekućim vremenski konstantnim mrežama zbog inherentne nelinearnosti i vremenski promjenjive prirode sustava. Drugi izazov leži u dizajnu i izradi odgovarajućih hardverskih platformi koje mogu učinkovito podržati tekuće vremenske konstantne mreže. Integracija različitih komponenti, kao što su senzori, aktuatori i jedinice za obradu podataka, predstavlja značajan izazov u smislu kompatibilnosti i skalabilnosti. Štoviše, autori naglašavaju potrebu za učinkovitim algoritmima i kontrolnim strategijama za optimizaciju performansi LTC.
Ovi izazovi, iako strašni, također predstavljaju prilike za potencijalni napredak na tom području.

Istraživači se mogu usredotočiti na razvoj robusnih tehnika modeliranja koje točno hvataju dinamiku LTC. Osim toga, dizajn novih materijala i metoda izrade može pomoći u prevladavanju ograničenja postojećih hardverskih platformi. Nadalje, razvoj naprednih kontrolnih algoritama, poput strojnog učenja i pristupa koji se temelje na umjetnoj inteligenciji, može poboljšati sposobnosti i prilagodljivost LTC. Sve u svemu, rješavanje ovih izazova i istraživanje potencijalnih napretka doprinijet će napretku i praktičnoj primjeni LTC u različitim domenama.

3. Autonomna vožnja pomoću tekućih neuronskih mreža

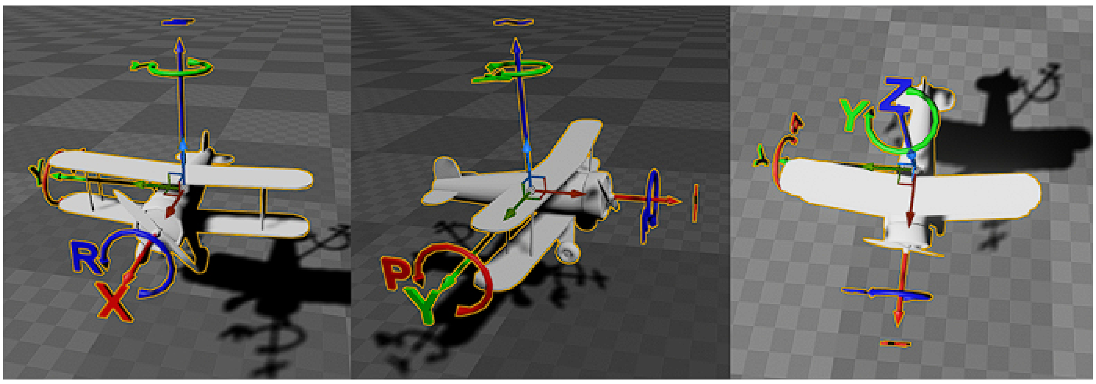
Za izradu autonomnog modela auta korištena je skupina alata i okruženja koja su detaljno opisana pod temom korišteni alati. Simulator Carla je korišten za treniranje, validaciju i testiranje tekućih neuronskih mreža zbog njegovog realističnog i sveobuhvatnog okruženja. Simulator sadrži bogati skup senzorskih podataka, kao što su lidar, radar, sonar i raznorazne vrste kamera, koje služe za učinkovito percipiranje i navigiranje u dinamičnim okolinama. Dodatno, simulator Carla nudi fleksibilan i prilagodljiv okvir koji omogućava jednostavno integriranje modela tekućih neuronskih mreža i algoritama u okruženje simulacije.

3.1. Dohvat podataka

Kod dohvata podataka postojalo je više ideja o tome kako i na koji način preuzeti veliku količinu podataka, a te ideje su najviše bile potaknute budućim modelom. Problematika "hranjenja" (engl. *feeding*) podataka modelom uključivala je više stvari, a one će biti detaljnije opisane u poglavlju izrade modela. Prvotna ideja je bila izraditi skriptu koja će kroz više iteracija, pomoću autopilota prolaziti scenarije te zabilježavati podatke iz senzora, koji bi u modelu bili ulazi (engl. *inputs*), dok bi izlazi (engl. *outputs*) bili podaci o trenutnom voznom stanju auta (stupanj zakrenutosti volana (engl.*steer*), pritisak papučice gasa (engl. *throttle*) i pritisak kočnice (engl. *brake*)).

3.1.1. Carla skup podataka

Iako na internetu postoji više skupa podataka kreiranih unutar Carla simulatora, za naš model i predikciju modela potrebna je različita vrsta podataka. U skripti $generate_{d}ataset.py$ nalazi se kod za prikupljanje podataka, koji dohvaća slike iz senzora kamere postavljene na vozilo i povezane informacije o promjenom kuta u odnosu na početnu orijentaciju.



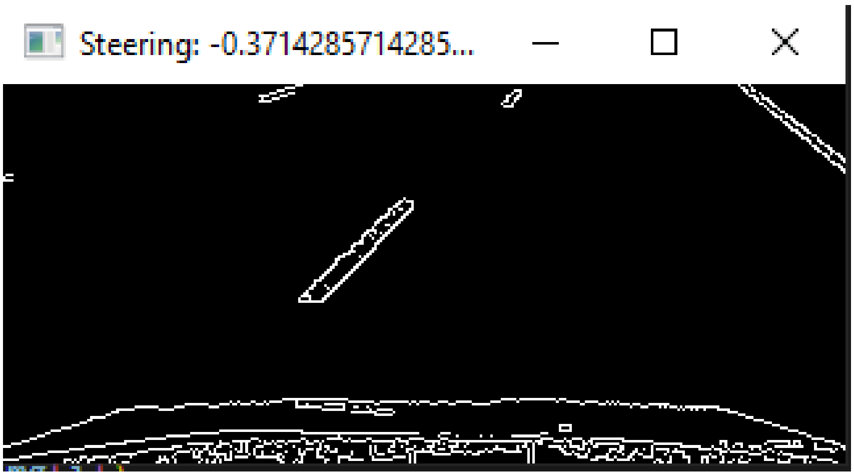
Slika 11: Rotacije unutar Carla simulatora; preuzeto iz [1]

```
for lane in good_lanes:
```

```
#loop within a lane
if quit:
    break
    for wp in lane[0].next_until_lane_end(20):
        start_point = wp.transform
        ego_vehicle.set_transform(start_point)
        time.sleep(2)
        initial_yaw = start_point.rotation.yaw

        for i in range(5):
            world.tick()

            trans = start_point
            angle_adj = random.randrange(-YAW_ADJ_DEGREES,
             YAW_ADJ_DEGREES, 1)
            trans.rotation.yaw = initial_yaw +angle_adj
            ego_vehicle.set_transform(trans)
            time.sleep(1)
            if cv2.waitKey(1) == ord('q'):
                quit = True
                break

            s_frame = sensor_queue.get(True, 1.0)
            image = s_frame[0]

            actual_angle = ego_vehicle.get_transform().rotation.
             yaw - initial_yaw
            if actual_angle <-180:
                actual_angle +=360
            elif actual_angle >180:
                actual_angle -=360
            actual_angle = str(int(actual_angle))
            angle = float(actual_angle)/YAW_ADJ_DEGREES
            angles.append(angle)

            img = np.float32(img)
            img_gry = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
            image = img_gry[height_from:,width_from:width_to]
            canny = cv2.Canny(np.uint8(image),50,150)
            images.append(canny[:, :, None] / 255)

            if(len(images) == 64):
                try:
```

```
            np.save(f'{path}/images/64_images_{num_batches}', np.array(images))
            np.save(f'{path}/labels/64_labels_{num_batches}', np.array(angles))
        except FileNotFoundError:
            os.makedirs(f"{path}")
            np.save(f'{path}/images/64_images_{num_batches}', np.array(images))
            np.save(f'{path}/labels/64_labels_{num_batches}', np.array(angles))
        num_batches += 1
        images = []
        angles = []
```

Glavni dio koda unutar skripte, koji zapravo obavlja spremanje podataka nalazi se u petlji. Za svaku traku unutar traka na kojima se može učiti model i za svaku putnu točku unutar te trake zabilježuju se informacije o transformaciji (položaj i orijentacija) iz trenutne međutočke, postavlja se transformacija ego vozila na tu transformaciju. Nakon toga uvodi se odgodu mirovanja od 2 sekunde, kako bih simulator stigao odraditi transformacije i zabilježuje se početni kut skretanja (orijentacija) ego vozila. Radi boljeg izračuna kuta skretanja naknadno je dodana petlja koja pet puta okreće auto za određeni stupanj koji je određen pseudoslučajno. Za svaki od tih koraka zapisujemo sliku koju pretvaramo u sivu boju (engl. *grayscale*), režemo sliku tako da je vidljiva samo cesta, te koristimo popularnu $Canny$ funkciju za prepoznavanje rubova. Zajedno sa slikom spremamo i promjenu u orijentaciji ego vozila, te se normaliziraju podaci u razmaku od -1 do 1.



Slika 12: ekstrakcija rubova sa slike i prateći kut upravljanja; vlastita slika

Način spremanja podataka je $.npy$ format radi brzog spremanja i kasnije lakog dohvata podataka u traženom formatu. Podaci se spremaju u serije (engl. *batch*) od 64 para jer model može obrađivati više serija od jednom, a veličina serije se računa formulom n^2 i standardni veličina je 8,32,64,128 ili čak 256, ovisno o jačini grafičke kartice.

3.2. Obrada podataka

Priprema podataka je ključan korak u osiguravanju kvalitete i pouzdanosti podataka prije nego što se dalje koriste, a uključuje pretvaranje sirovih podataka u format koji je prikladan za analizu i modeliranje. Proces pripreme podataka uključuje nekoliko koraka poput čišćenja, normalizacije, transformacije i redukcije podataka.

Čišćenje podataka uključuje uklanjanje ili ispravljanje pogrešaka ili nekonzistentnosti u podacima, kao što su nedostajuće vrijednosti, izdvojeni podaci ili duplicirani unosi. Pošto je ovaj skup podataka rađen u kontroliranom okruženju i kontroliranim uvjetima nije potrebno čiščenje podataka.

Transformacija podataka uključuje pretvaranje podataka u standardni format ili skaliranje u zajednički raspon. Ovaj korak je ključan za osiguravanje da različite varijable imaju jednaku važnost tijekom analize.

Normalizacija podataka (za duboko učenje), je proces normaliziranja podataka na fiksni raspon kako bi bili prikladni za treniranje neuronskih mreža. Postoji više vrsta normalizacija, najkorišteniji način uključuje skaliranje podataka u rasponu [0,1] ili [-1,1].

Redukcija podataka koristi se za smanjenje dimenzija skupa podataka, uklanjanje redundantnih ili irelevantnih značajki kako bi se poboljšala računalna učinkovitost i izvedba modela.

Sveukupno, priprema podataka igra ključnu ulogu u osiguravanju kvalitete i pouzdanosti podataka prije nego što se koriste za analizu, omogućujući istraživačima i analitičarima da izvuku smislene uvide i donesu točne predikcije.

3.2.1. Carla obrada podataka

Pošto su podaci bili obrađivani pa direktno spremljeni u $.npy$ datoteke, njihova dodatna obrada nije potrebna. Jedina obrada koja se događa kod dohvata podataka labela. Unutar klase $BatchDataGenerator$ prije dohvata stupnja skretanja vršimo takozvanu "stvarnu" normalizaciju podataka, u kojoj najveća vrijednost u listi postaje 1 (absolutni maksimum), a ostale vrijednosti se skaliraju s obzirom na određenu maksimalnu vrijednost.

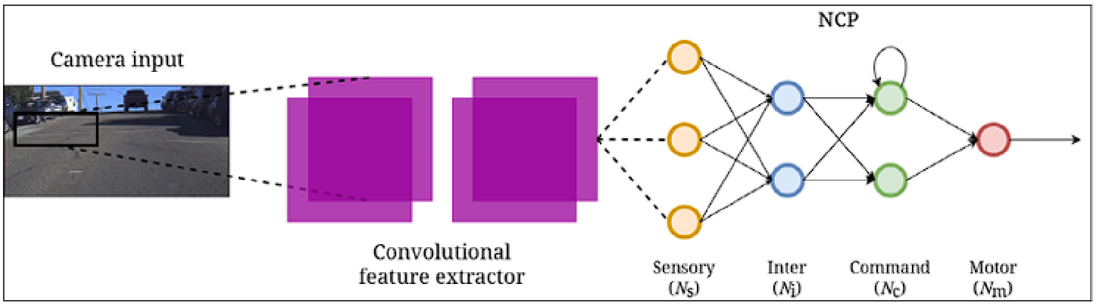
```
    true_max_train = max(abs(min(batch_y)), max(batch_y))
    batch_y *= (1.0 / true_max_train)
```

Za ulazne slike opisana je obrada koja uključuje konverziju slike u crno-bijelu, izrezivanje kako bi se zadržao samo dio koji prikazuje cestu, te nakon toga korištenje funkcije $Canny$ iz $cv2$ biblioteke za prepoznavanje rubova. Naravno, kako bi se slika mogla spremiti u $.npy$ format, mora se pretvoriti u listu koja ima određeni oblik (visina, širina, broj kanala), a u ovom trenutnom slučaju oblik je (144, 320, 1).

3.3. Izrada modela

Izgradnja modela neuronskih mreža nekada je bila složen i pedantan proces koji je zahtijevao da znanstvenici pomno definiraju svaki aspekt svojih neuronskih mreža, od arhitekture i inicijalizacije težina do podešavanja hiperparametara (engl. *hyperparameters*) i optimizacije gradijenta. Međutim, dolaskom moćnih okvira za duboko učenje poput TensorFlowa i PyTorcha, ovaj nekada težak zadatak značajno je pojednostavljen. Ovi alati nude apstrakcije visokog nivoa i predefinirane module koji omogućuju istraživačima i razvijateljima da se više usmjere na problem koji žele riješiti umjesto na detalje implementacije neuronskih mreža. Zbog te fasade jednostavnosti odabran je Tensorflow i njegov API Keras.

TensorFlow-ov Keras API je sučelje visokog nivoa koja pojednostavljuje proces oblikovanja, treniranja i implementacije modela neuronskih mreža za učenje. Pruža intuitivno i korisnički prijateljsko sučelje koje omogućuje razvijateljima da definiraju složene arhitekture neuronskih mreža s minimalnim kodom. Keras nudi širok spektar unaprijed izgrađenih slojeva, aktivacijskih funkcija i optimizatora, što olakšava eksperimentiranje s različitim konfiguracijama modela. Besprijekorno se integrira s TensorFlow-om, omogućavajući korisnicima da iskoriste punu snagu TensorFlow-ove računalne i optimizacijske sposobnosti dok istovremeno koriste prednosti Keras-ove jednostavnosti korištenja[22]. Ukratko, TensorFlow-ov Keras API je vrijedan alat kako za početnike tako i za iskusne praktičare, omogućujući brzo prototipiranje i eksperimentiranje u području umjetne inteligencije.



Slika 13: CNN-NCP struktura(inter=12,command=8); preuzeto iz [23]

Na ovoj slici nalazi se jednostavni prikaz kako funkcionira izrađeni model. Modelu se šalje slika na obradu kroz nekoliko konvolucijskih slojeva za izvlačenje značajki, te se te značajke sa slike šalju sloju ponavljajućoj neuronskoj mreži, koja sadrži ćeliju nadahnutu funkcioniranjem mozga. S ovom hibridnom arhitekturom cilj je poboljšati izdvajanje sekvencijalnih informacija od strane konvolucijskih slojeva i postići unaprijeđenu kontrolu dinamike vozila.

3.3.1. Umrežavanje

Za izradu modela od velike pomoći je bila dokumentacija politike neuronskih krugova [24], koja proširuje $keras$ sa bibliotekom $kerasncp$. Kao što je već objašnjeno ranije, TNN baziraju se na politike neuronskih krugova (NCP), te za izradu modela TNN, $kerasncp$ pruža detaljno specificiranje NCP umrežavanja i kreiranje $LTCCell$ ćelije. Kreirana NCP ćelija bazirana je na LTC neuronu te samim time predstavlja TNN.

Biblioteka $kerasncp$ korisnicima pruža dva načina modeliranja NCP umrežavanja, a za odabir kojeg važno je razumjeti razliku između NCP i $AutoNCP$ povezivanja.

Modeliranje NCP umrežavanja moguće je automatski način sa $AutoNCP$:

```
wiring = AutoNCP(21,1)
```

ili na manualni način:

```
wiring = NCP(
    inter_neurons=12,  # Number of inter neurons
    command_neurons=8,  # Number of command neurons
    motor_neurons=1,  # Number of motor neurons
    sensory_fanout=4,  # How many outgoing synapses has each sensory neuron
    inter_fanout=4,  # How many outgoing synapses has each inter neuron
    recurrent_command_synapses=4,  # Now many recurrent synapses are in the
    # command neuron layer
    motor_fanin=6,  # How many incoming synapses has each motor neuron
)

# Create the the NCP cell based on the LTC neuron.
ncp_cell = LTCCell(wiring)
```

NCP se odnosi na tradicionalni način ručnog određivanja veza između slojeva neuronske mreže. U ovom pristupu korisnik ima potpunu kontrolu nad arhitekturom i može eksplicitno definirati koji su slojevi povezani međusobno.

S druge strane, $AutoNCP$ povezivanje je automatizirani pristup gdje se povezivanje između slojeva automatski određuje od strane same biblioteke. To znači da korisnik ne mora ručno određivati veze, već se knjižnica brine za to na temelju određenih pravila ili algoritama.

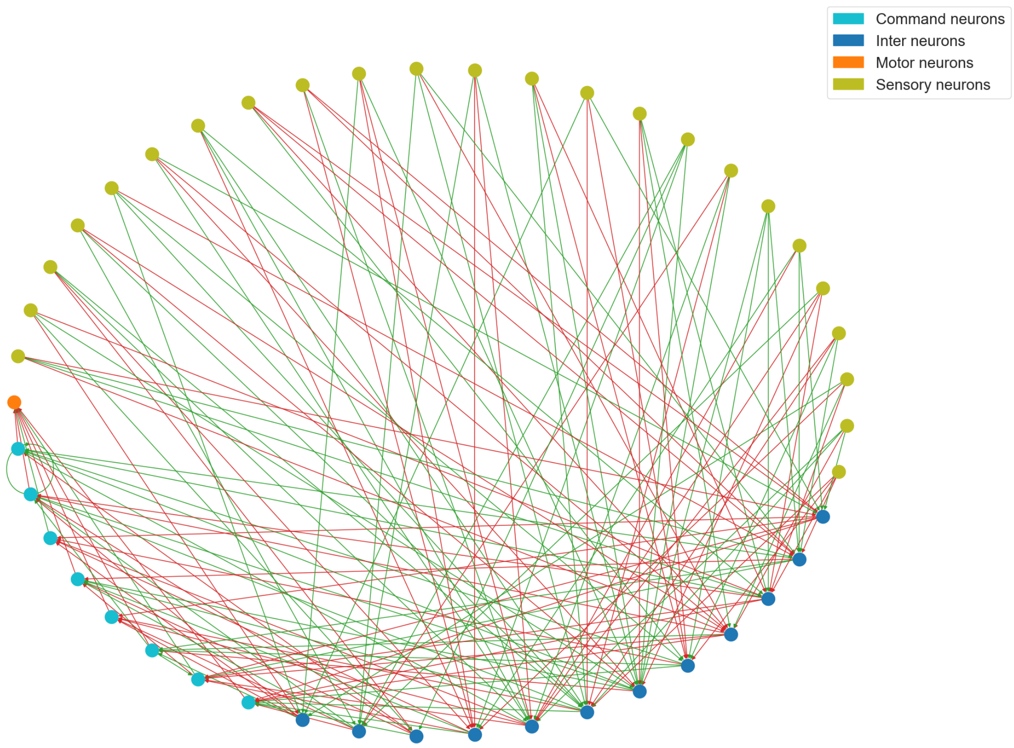
$AutoNCP$ povezivanje može biti korisno u scenarijima gdje je arhitektura mreže složena ili kada korisnik želi eksperimentirati s različitim konfiguracijama veza bez ručnog mijenjanja koda. Međutim, NCP povezivanje pruža veću fleksibilnost i kontrolu nad arhitekturom, omogućavajući korisniku dizajniranje mreže prema svojim specifičnim potrebama. Konačan izbor između NCP i $AutoNCP$ povezivanja ovisi o zahtjevima projekta i razini kontrole koju korisnik želi imati nad arhitekturom mreže.

Za ovaj model odabran je tradicionalni način, a parametri koje on prima su:

- **interni neuroni** – broj interneurona (sloj 2)

- **naredbeni neuroni** – broj naredbenih neurona (sloj 3)
- **motorički neuroni** – broj motoričkih neurona (sloj 4 = broj izlaza)
- **senzorni broj** – Prosječan broj izlaznih sinapsi od senzornih do interneurona
- **inter broj** – Prosječan broj izlaznih sinapsi od interneurona do naredbenih neurona
- **ponavljajuće naredbene sinapse** – Prosječan broj povratnih veza u sloju naredbenih neurona
- **motorički broj** – Prosječan broj ulaznih sinapsi motoričkih neurona od naredbenih neurona
- **sjeme** – Slučajna sjemenka korištena za generiranje povezivanja

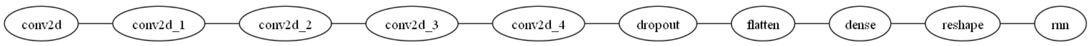
Najbitniji parametri su broj ulaza (interni) i broj izlaza (motorički), a svi ostali imaju mogućnost mijenjanja radi postizanja boljeg rezultata.



Slika 14: Umrežavanje NCP neurona; Vlastita slika

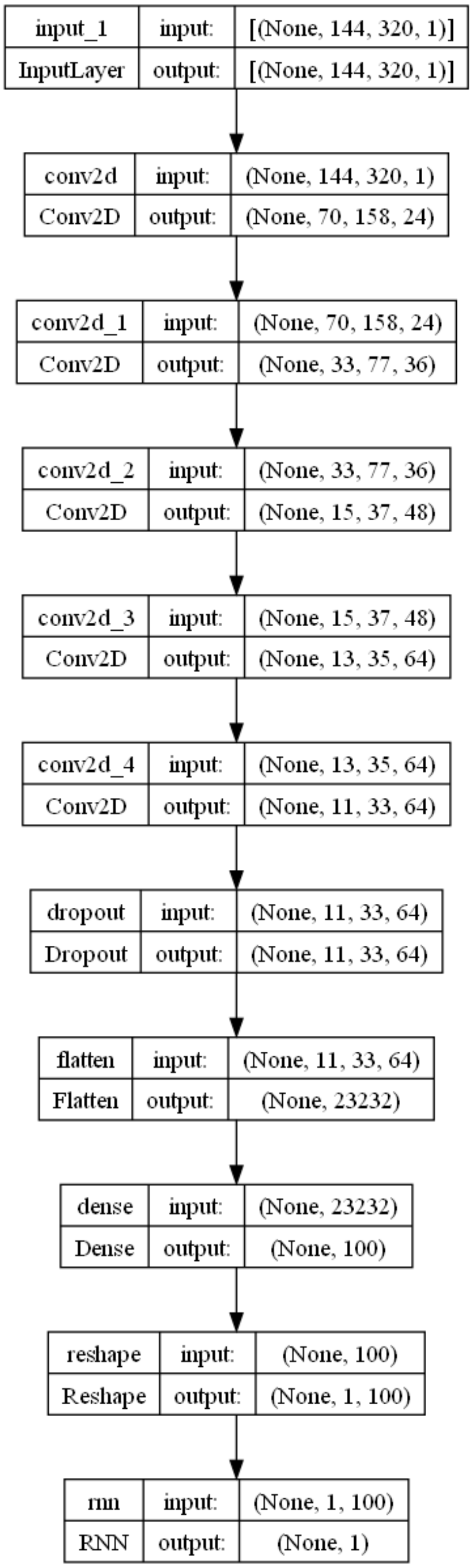
3.3.2. Model

Nakon dobro postavljenog umrežavanja, kreira se model, koji mora sadržavati karakteristike ranije opisane ideje.



Slika 15: Jednostavni prikaz modela; vlastita slika

```
model = Sequential()
model.add(InputLayer(input_shape=PROCESSED_IMG_SHAPE))
model.add(Conv2D(24, 5, 2, activation='relu'))
model.add(Conv2D(36, 5, 2, activation='relu'))
model.add(Conv2D(48, 5, 2, activation='relu'))
model.add(Conv2D(64, 3, activation='relu'))
model.add(Conv2D(64, 3, activation='relu'))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Reshape((1, -1)))
model.add(RNN(ncp_cell, unroll=True))
return model
```



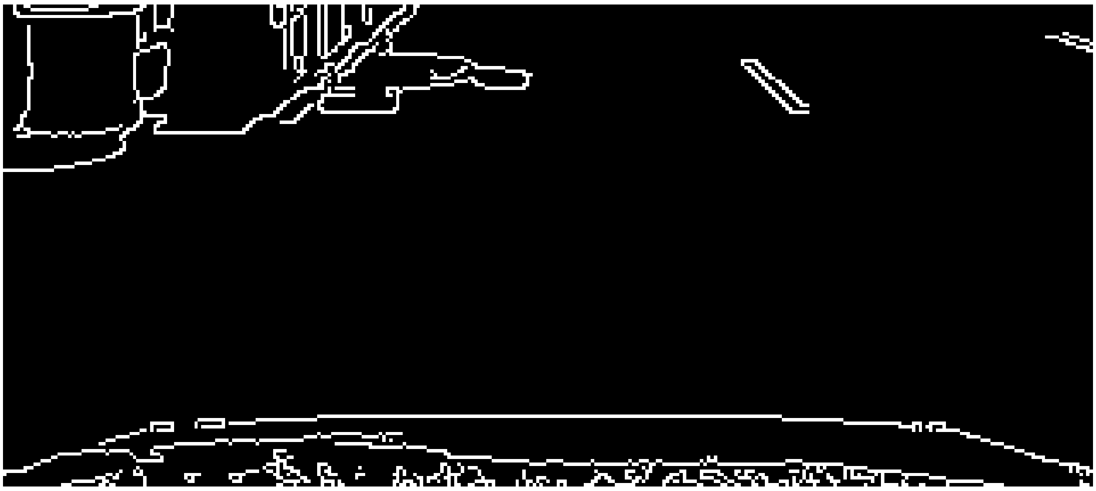
Slika 16: Detaljni prikaz modela; vlastita slika

Komponente i funkcionalnosti ovog modela implementirane su pomoću Kerasa:

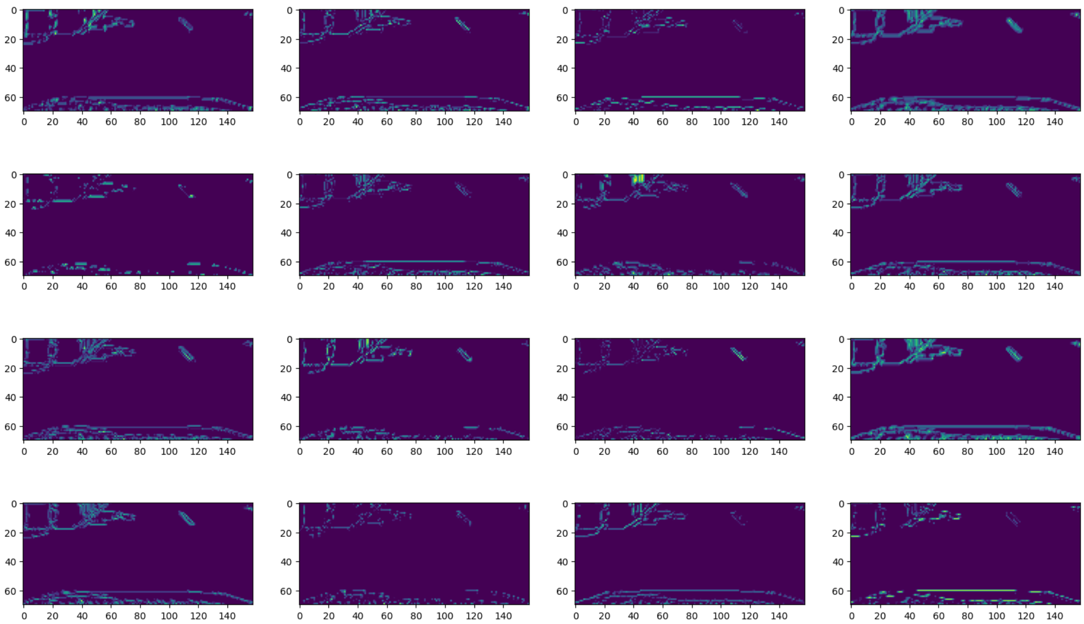
1. **Sequential**: Ova linija inicijalizira sekvencijalni model, što je linearni niz slojeva. To znači da će se slojevi dodavati jedan za drugim u nizu.
2. **InputLayer**: Model počinje s ulaznim slojem koji opisuje oblik ulaznih vrijednosti kao $PROCESSED\_IMG\_SHAPE$. Ovdje model očekuje ulazne podatke. Oblik ulaznih podataka trebao bi odgovarati rezoluciji slike, a to je (144, 320, 1).
3. **Conv2D**(24, 5, 2, activation='relu'): prvi konvolucijski sloj s 24 filtra veličine 5x5 i korakom pomaka od 2. Koristi ReLU (Rectified Linear Unit) aktivacijsku funkciju.
4. **Conv2D**(36, 5, 2, activation='relu'): Drugi konvolucijski sloj s 36 filtra, također veličine 5x5 i korakom pomaka od 2, koristi ReLU aktivaciju.
5. **Conv2D**(48, 5, 2, activation='relu'): Treći konvolucijski sloj s 48 filtra, veličine 5x5 i korakom pomaka od 2, s ReLU aktivacijom.
6. **Conv2D**(64, 3, activation='relu'): Četvrti konvolucijski sloj s 64 filtra veličine 3x3 i ReLU aktivacijom.
7. **Conv2D**(64, 3, activation='relu'): Peti konvolucijski sloj s 64 filtra veličine 3x3 i ReLU aktivacijom.
8. **Dropout**(0.5): Ovo je sloj za isključivanje s vjerojatnošću isključivanja od 0,5, što znači da će tijekom obuke otprilike 50% neurona u prethodnom sloju biti nasumično isključeni ili postavljeni na nulu. Isključivanje je tehnika regularizacije koja sprječava prenaučenost.
9. **Flatten**(): Ovaj sloj pretvara izlaz iz prethodnih slojeva u jednodimenzijski vektor. Priprema podatke za ulaz u potpuno povezane slojeve.
10. **Dense**(100, activation='relu'): Ovo je potpuno povezan (engl. *dense*) sloj s 100 neurona i ReLU aktivacijom.
11. **Reshape**((1, -1)): ovaj sloj preoblikuje izlaz iz potpuno povezanog sloja u tenzor s oblikom (1, -1), gdje se druga dimenzija automatski izračunava na temelju ulazne veličine.
12. **RNN**(ncp_cell, unroll=True): sloj ponavljajuće neuronske mreže RNN je ključan sloj s ranije definiranom ncp_cell ćelijom. Argument unroll=True označava da RNN treba biti razmotan, što znači da procesira cijeli niz u jednom prolazu prema naprijed.

Ukratko ovaj model počinje s konvolucijskim slojevima za izdvajanje bitnih značajki iz slika, zatim koristi isključivanje neurona i potpuno povezane slojeve za učenje značajki više razine. Na kraju koristi RNN za sekvencijalnu obradu, što je najbitniji dio jer umjesto "običnog" umrežavanja, koristi se umrežavanje TNN.

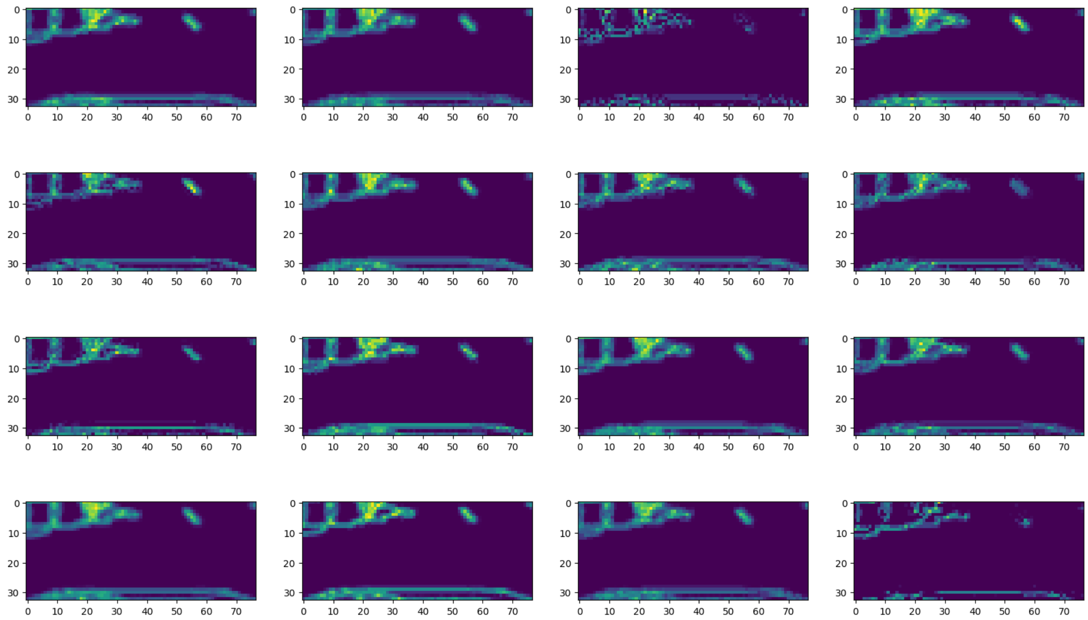
Kako konvolucijske mreže funckioniraju objašnjeno je u teorijskom dijelu rada, a za bolje razumijevanje teorije za ovaj model kreirani su izlazi za neke od slojeva iz modela.



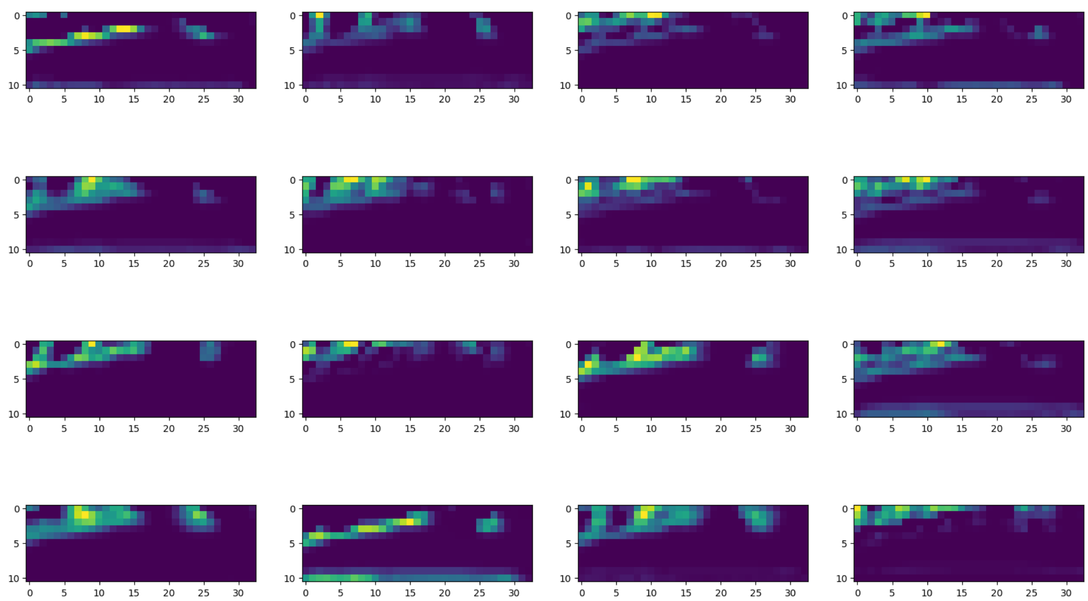
Slika 17: Slika koja se provlači kroz konvolucije; vlastita slika



Slika 18: Prikaz prvog konvolucijskog sloja; vlastita slika



Slika 19: Prikaz drugog konvolucijskog sloja; vlastita slika



Slika 20: Prikaz zadnjeg konvolucijskog sloja; vlastita slika

Nakon prve konvolucije nije se dogodilo ništa značajno, ali ona je početak izvlačenja obilježja. Već nakon prolaska slike kroz drugu konvolucijsku mrežu stvaraju se pikseli rubova. Na zadnjoj slici je slika koja je prošla sve konvolucije i jasno pokazuje piksele koji su daljnjem modelu bitni. Isto tako rezolucija se smanjila, što je vidljivo povećanim pikselima, a i mjerilima pored slike.

3.4. Testiranje modela i rezultati

Kako bi se saznalo da li je model dobro postavljen sve što je potrebno je učitati model, postaviti određene postavke, kao što su optimizacije i metrike i pozvati funkciju za treniranje napravljenog modela. Za lakšu pratnju izvedbe modela dodaje se i atribut `callbacks` pomoću kojeg je moguće dobiti uvid u interna stanja i statistiku modela tijekom treniranja.

```
checkpoint = ModelCheckpoint(
cps_path,
monitor="val_mean_squared_error",
verbose=VERBOSITY,
save_best_only=True,
mode="auto",
)
NAME = f"{model_name}_model_{int(time.time())}"
tensorboard = TensorBoard(
    log_dir="logs/{}".format(NAME),
    write_graph=True,
    update_freq="epoch",
    histogram_freq=1,
)

# Start training the model
history = model.fit(
    train_gen,
    epochs=NB_EPOCHS,
    verbose=VERBOSITY,
    validation_data=val_gen,
    callbacks=[checkpoint, tensorboard]
)
```

Lista `callbacks` posjeduje dva parametra. Parametar `checkpoint` služi za spremanje najboljeg modela ovisno o varijabli koja se prati, dok `tensorboard` daje uvid u izvođenje modela preko zapisa (engl. *logs*) koji se učitavaju preko stranice i kreiraju zatražene grafove.

Jedan od načina kako provjeriti da li model radi ispravno je praćenje rezultata nad validacijskim skupom podataka. Validacija u modelu je postupak procjene točnosti modela putem evaluacije njegovih predviđanja na nekorištenom skupu podataka. To je ključno za razumijevanje sposobnosti modela za generalizaciju i otkrivanje problema kao što su pretjerana (engl. *overfitting*) ili premala prilagodba (engl. *underfitting*) modela. Postoje različite metode validacije, poput validacije zadržavanjem (engl. *holdout*), unakrsne validacije (engl. *k-fold cross-validation*) i ostavi jednog izvan (engl. *leave-one-out*) [25].

Ovaj proces pomaže u ocjeni performansi modela i omogućuje poboljšanja prije implementacije, a postoji nekoliko metrika koje se mogu pratiti:

- Gubitak (engl. *Loss*)
- Preciznost (engl. *Precision*)
- Recall (engl. *Osjetljivost*)
- F1-uspjeh (engl. *F1-Score*)
- ROC Krivulja (engl. *ROC Curve*)
- AUC (engl. *Površina ispod ROC Krivulje*)
- Srednja Apsolutna Greška (engl. *Mean Absolute Error - MAE*)
- Srednja Kvadratna Greška (engl. *Mean Squared Error - MSE*)
- R-kvadrat (engl. *R2*)
- Matrica Konfuzije (engl. *Confusion Matrix*)
- Logaritamski Gubitak (engl. *Log-Loss*)

Odabir prave metrike za treninga modela dubokog učenja ovisi o konkretnom problemu koji se pokušava riješiti i ciljevima prilikom treniranja modela. Metrike pružaju korisne informacije, ali usredotočuju se na različite aspekte performansi modela.

Za predviđanje kuta skretanja odabrana je metrika srednja kvadratna greška (MSE):

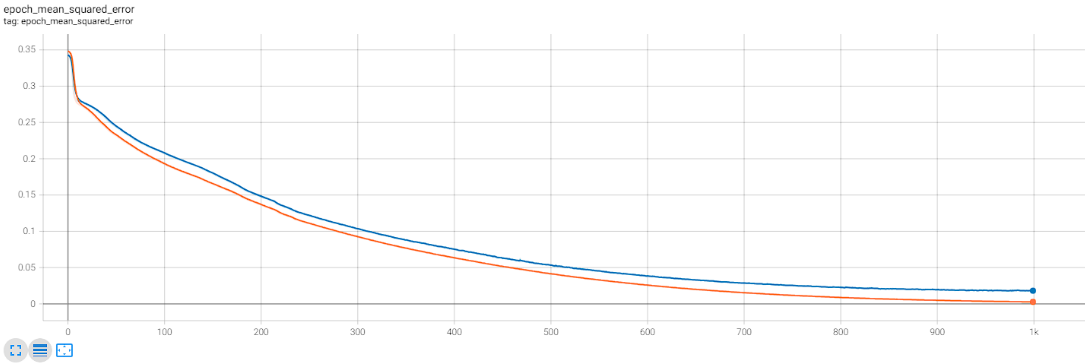
$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

gdje je:

$$\begin{aligned} \text{MSE} &\text{ - srednja kvadratna greška} \\ n &\text{ - broj podataka} \\ y_i &\text{ - stvarna ciljna vrijednost za podatak } i \\ \hat{y}_i &\text{ - predviđena ciljna vrijednost za podatak } i \\ \sum_{i=1}^{n} &\text{ - zbroj svih podataka od 1 do } n \end{aligned}$$

MSE izračunava prosječnu kvadratnu razliku između stvarnih i predviđenih vrijednosti, pri čemu veće greške dobivaju veću težinu zbog kvadriranja. Razlog odabira ove metrike je zato što se ona odabire kada je cilj modela smanjiti kvadratnu grešku između predviđenih i stvarnih vrijednosti, odnosno želimo što bolje pogađanje kuta skretanja.

Model se trenirao na 1000 epoha i vidljiv je konstantni pad, u početku naglo, a onda polagano prema nuli. Već kod 900 epoha vidljivo je izravnanje srednje kvadratne greške sa apscisom, što znači da je model naučen koliko on može naučiti, te se tu često zaustavlja model kako se ne bi došlo do prekomjernog treniranja.

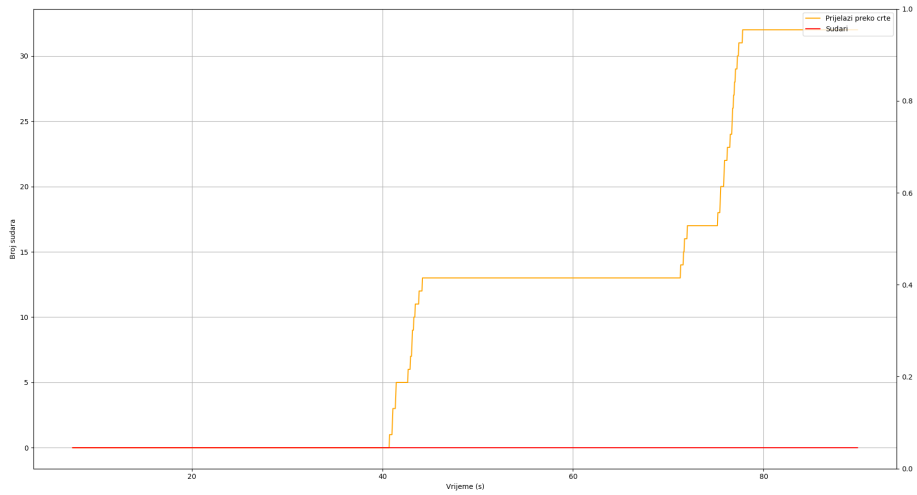


Slika 21: Srednja kvadratna greška (MSE); vlastita slika

Prilikom pokretanja simulacije praćene su dvije varijable pomoću dodatnih senzora koje pruža Carla, a to su $lane_invasion$ i $collision$. Kako bi model bio ocjenjen i vidjeli kako se TNN prilagođavaju, skripta za automatsku vožnju pokrenuta je u tri vremenska uvjeta: $Sunny$, $WetNoon$ i $HardRainSunset$.

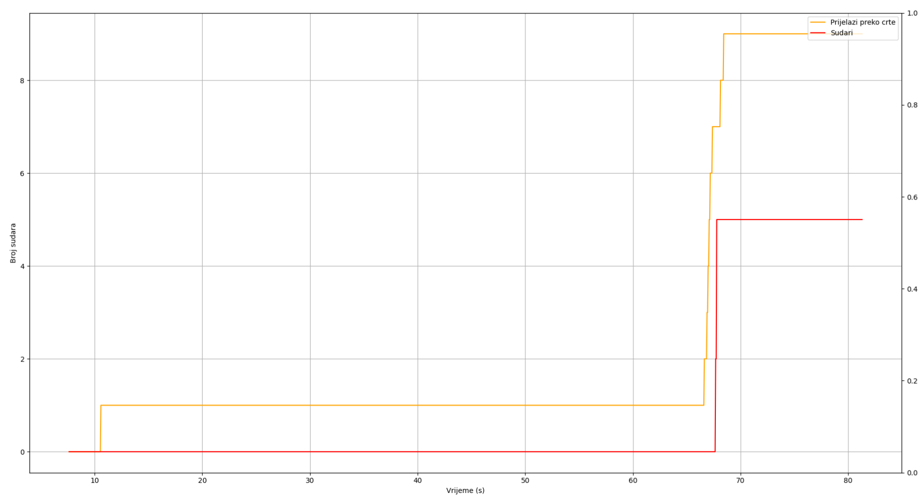
Slika 22: Vožnje pod suncem; vlastita slika

Iako je model treniran na sunčanoj mapi unutar 2 minute ima najviše prekršaja napravljenih, i to čak preko 30. Jedna od pretpostavki je da se model prekomjerno istrenirao u tom okruženju i zbog toga radi krive predikcije. Pošto rezultat ovisi o više stvari, na kojoj poziciji na mapi se auto stvorio, da li se vozi samo autoputom ili i gradom, koliko je raskrižja prošao, da li je napravio skretanje na raskrižju i još puno ostalih parametara, ne bi se trebalo zaključivati iz jedne epohe.

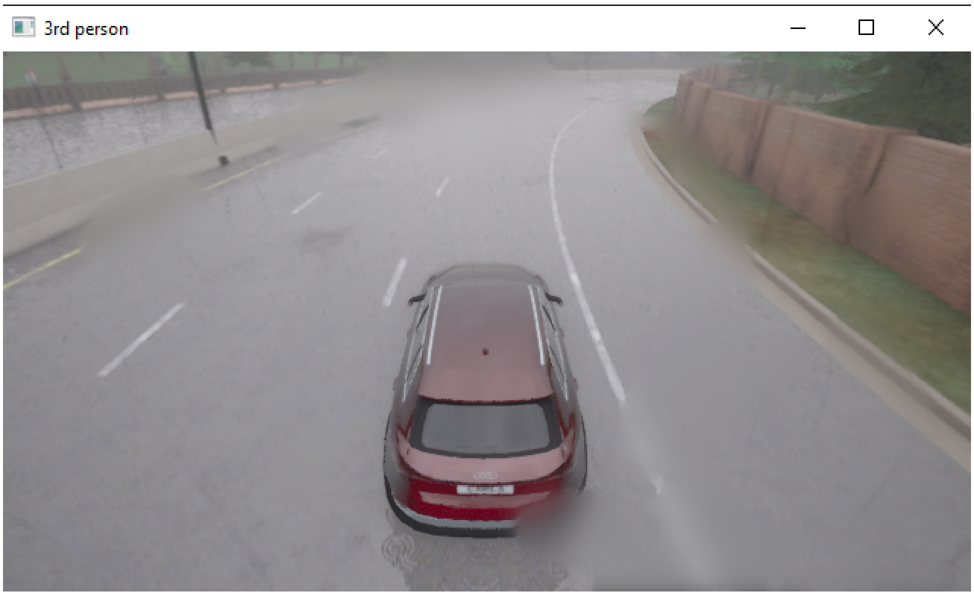


Slika 23: Rezultat vožnje pod suncem; vlastita slika

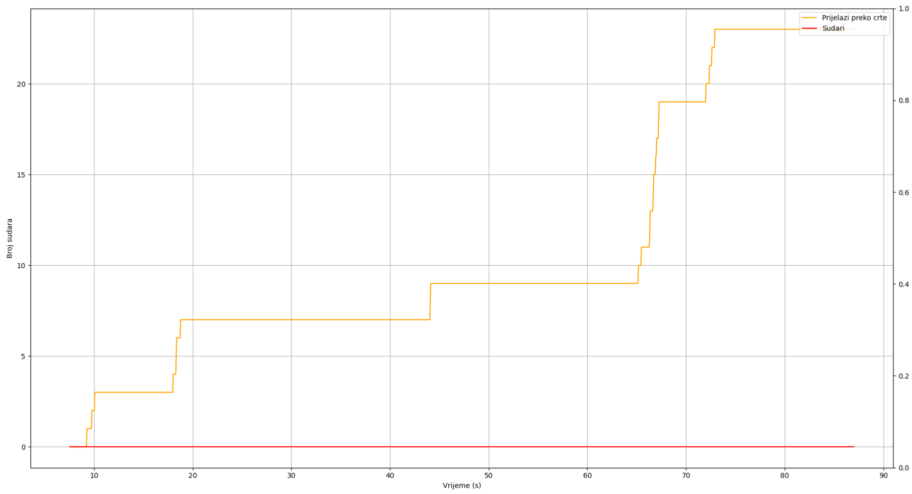
Slika 24: Vožnja na mokroj cesti; vlastita slika



Slika 25: Rezultat vožnje na mokroj cesti; vlastita slika



Slika 26: Vožnja dok pada jaka kiša; vlastita slika



Slika 27: Rezultat vožnje dok pada jaka kiša; vlastita slika

4. Zaključak

Zaključno, razvoj i implementacija TNN u autonomnim automobilima pokazali su ogroman potencijal u području samoupravljajućih vozila. Korištenjem različitih metoda, tehnika i alata, uključujući Carla simulator, VS Code, okruženje Anaconda i programski jezik Python, dobiveni su vrijedni rezultati. Metoda rada je bila prvobitan fokus na izradu uspješne implementacije TNN, koje su poznate po svojoj sposobnosti prilagodbe i učenja u dinamičnim okruženjima. Obučavanjem mreže koristeći podatke dobivene iz simulatora Carla, model je pokazao izvanredne performanse u navigaciji kroz složene scenarije vožnje. Integracija VS Code-a i Anaconde omogućila je besprijekorno razvojno okruženje, olakšavajući proces implementacije i eksperimentiranja.

Korištenjem promjenjive vremenske konstante, ove su mreže uspješno riješile ograničenja tradicionalnih neuronskih mreža, kao što su njihova fiksna struktura i statičke težine. Prilagodljivost i priroda samoorganiziranja TNN omogućuje im kontinuirano učenje i razvoj, što ih čini prikladnima za dinamičnu i nepredvidivu prirodu scenarija vožnje u stvarnom svijetu. Nadalje, integracija TNN s naprednim tehnologijama senzora, kao što su LiDAR i radar, dodatno bi se moglo poboljšati percepciju i razumijevanje okoline vozila, što u konačnici rezultira sigurnijom navigacijom na cestama.

Osim toga, projekt je istaknuo važnost sveobuhvatnog procesa izrade modela, započevši prikupljanjem i obradom podataka, kreiranje, treniranje i naposljetku testiranje modela.

Kako se ova tehnologija nastavlja razvijati, ima potencijal otključati još veće mogućnosti u autonomnim automobilima, kao što su poboljšano predviđanje, prilagodba u stvarnom vremenu i besprijekorna integracija s drugim inteligentnim sustavima. Međutim, važno je priznati da još uvijek postoje izazovi i ograničenja kojima se treba pozabaviti, uključujući etička razmatranja, regulacijske okvire i javno prihvaćanje.

Sve u svemu, projekt o tekućim neuronskim mrežama u autonomnim automobilima pokazao je golemi potencijal ove tehnologije, utirući put budućnosti u kojoj se samoupravljajuća vozila mogu kretati našim cestama uz neviđenu sigurnost i učinkovitost.

Popis literature

[1] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez i V. Koltun, „CARLA: An Open Urban Driving Simulator," *Proceedings of the 1st Annual Conference on Robot Learning*, 2017., str. 1–16.

[2] B. Copeland. „Artificial Intelligence," Britannica. (2020.), adresa: https://www.britannica.com/technology/artificial-intelligence (pogledano 10. 9. 2023.).

[3] IBM. „What is artificial intelligence (AI)?" IBM Web Page, IBM. (), adresa: https://www.ibm.com/topics/artificial-intelligence (pogledano 10. 9. 2023.).

[4] E. Burton, J. Goldsmith, S. Koenig, B. Kuipers, N. Mattei i T. Walsh, „Ethical Considerations in Artificial Intelligence Courses," *AI Magazine*, sv. 38, br. 2, str. 22–34, 7. 2017. DOI: 10.1609/aimag.v38i2.2731.

[5] C. Aggarwal, „An Introduction to Neural Networks," *Neural Networks and Deep Learning: A Textbook*. Cham: Springer International Publishing, 2023., str. 1–27, ISBN: 978-3-031-29642-0. DOI: 10.1007/978-3-031-29642-0_1.

[6] „Convolutional neural networks, explained," Medium. (), adresa: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939 (pogledano 11. 9. 2023.).

[7] D. E. Rumelhart, G. E. Hinton i R. J. Williams, „Learning representations by back-propagating errors," *Nature*, sv. 323, str. 533–536, 1986.

[8] W. Razzaq i H. Mo, „Neural Circuit Policies Imposing Visual Perceptual Autonomy," *Neural Processing Letters*, 2. 2023. DOI: 10.1007/s11063-023-11194-4.

[9] „Deep learning algorithms tutorial," Simplilearn. (), adresa: https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-algorithm (pogledano 11. 9. 2023.).

[10] DeepLobe. „Computer vision-navigating the future of autonomous vehicles," DeepLobe. (), adresa: https://deeplobe.ai/howcomputer-vision-is-navigating-the-future-of-autonomous-vehicles/ (pogledano 12. 9. 2023.).

[11] B. Ranft i C. Stiller, „The Role of Machine Vision for Intelligent Vehicles," *IEEE Transactions on Intelligent Vehicles*, sv. 1, br. 1, str. 8–19, 2016. DOI: 10.1109/TIV.2016.2551553.

[12] J. Janai, F. Güney, A. Behl i A. Geiger, *Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art*, 2021. arXiv: 1704.05519 [cs.CV].

[13] Synopsys. „Autonomous Driving Levels." Online article. (2022.), adresa: https://www.synopsys.com/automotive/autonomous-driving-levels.html (pogledano 12. 9. 2023.).

[14] R. Hasani, M. Lechner, A. Amini, D. Rus i R. Grosu, *Liquid Time-constant Networks*, 2020. arXiv: 2006.04439 [cs.LG].

[15] R. Hasani, *Modern era of statistics*, 12. 1. 2023.

[16] „Google Colab Notebook Example." Online Resource. (), adresa: https://colab.research.google.com/drive/1IvVXVSC7zZPo5w-PfL3mk1MC3PIPw7Vs?usp=sharing (pogledano 12. 9. 2023.).

[17] R. K. Pan, N. Chatterjee i S. Sinha, „Mesoscopic organization reveals the constraints governing Caenorhabditis elegans nervous system," *PloS one*, sv. 5, br. 2, e9240, 2010.

[18] G. Zamora-López, C. Zhou i J. Kurths, „Exploring Brain Function from Anatomical Connectivity," *Frontiers in Neuroscience*, sv. 5, 2011., ISSN: 1662-453X. DOI: 10.3389/fnins.2011.00083.

[19] L. R. Varshney, B. L. Chen, E. Paniagua, D. H. Hall i D. B. Chklovskii, „Structural Properties of the Caenorhabditis elegans Neuronal Network," *PLOS Computational Biology*, sv. 7, br. 2, str. 1–21, 2. 2011. DOI: 10.1371/journal.pcbi.1001066.

[20] Massachusetts Institute of Technology (MIT). „MIT 6.S191: The Modern Era of Statistics." Online Video. (2023.), adresa: https://www.youtube.com/watch?v=p1NpGC8K-vs&t=2290s (pogledano 12. 9. 2023.).

[21] M. Bidollahkhani, F. Atasoy i H. Abdellatef, *LTC-SE: Expanding the Potential of Liquid Time-Constant Neural Networks for Scalable AI and Embedded Systems*, 2023. arXiv: 2304.08691 [cs.LG].

[22] Martín Abadi, Ashish Agarwal, Paul Barham i dr., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015.

[23] H. Ahmadov. „Brain-inspired-deep-imitation-learning-for-autonomous-driving-systems," archive.org. (2021.), adresa: https://github.com/Intenzo21/Brain-Inspired-Deep-Imitation-Learning-for-Autonomous-Driving-Systems (pogledano 6. 9. 2023.).

[24] M. Lechner. „Neural circuit policies's documentation." (2016.), adresa: https://ncps.readthedocs.io/ (pogledano 9. 9. 2023.).

[25] S. Raschka, „Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning," *CoRR*, sv. abs/1811.12808, 2018. arXiv: 1811.12808.

Popis slika

Popis tablica

Popis isječaka koda

Prilozi

1. Prilog 1

data_generator.py

```python
import os
import random
import time
from queue import Empty, Queue

import cv2
import numpy as np

import carla

YAW_ADJ_DEGREES = 35

PREFERRED_SPEED = 10

CAMERA_POS_Z = 1.6
CAMERA_POS_X = 0.9

HEIGHT = 360
WIDTH = 640

HEIGHT_REQUIRED_PORTION = 0.4
WIDTH_REQUIRED_PORTION = 0.5

YAW_ADJ_DEGREES = 35

height_from = int(HEIGHT * (1 -HEIGHT_REQUIRED_PORTION))
width_from = int((WIDTH - WIDTH * WIDTH_REQUIRED_PORTION) / 2)
width_to = width_from + int(WIDTH_REQUIRED_PORTION * WIDTH)

good_roads = [12, 34, 35, 36, 37, 38, 1201, 1236, 2034, 2035, 2343, 2344]


def sensor_callback(sensor_data, sensor_queue, sensor_name):

    image = np.reshape(np.copy(sensor_data.raw_data),(sensor_data.height,sensor_data.width,4))
    sensor_queue.put((image, sensor_name))

```

```python

def camera_rgb_install():
    camera_transform = carla.Transform(
            carla.Location(x=CAMERA_POS_X, z=CAMERA_POS_Z)
        )
    camera_blueprint = world.get_blueprint_library().find("sensor.camera.rgb")
    camera_blueprint.set_attribute("image_size_x", '640')
    camera_blueprint.set_attribute("image_size_y", '360')
    camera = world.spawn_actor(
        camera_blueprint, camera_transform, attach_to=ego_vehicle
    )

    camera.listen(lambda image: sensor_callback(image, sensor_queue, "camera"))
    actor_list.append(camera)
    sensor_list.append(camera)

actor_list = []

client = carla.Client('localhost', 2000)
client.set_timeout(10)

client.load_world('Town05')


world = client.get_world()

try:

    traffic_manager = client.get_trafficmanager(8000)
    settings = world.get_settings()
    traffic_manager.set_synchronous_mode(True)
    # option preferred speed
    # traffic_manager.set_desired_speed(vehicle,float(PREFERRED_SPEED))
    settings.synchronous_mode = True
    settings.fixed_delta_seconds = 0.05
    world.apply_settings(settings)

    sensor_queue = Queue()

    town_map = world.get_map()
```

```python

    spawn_points = town_map.get_spawn_points()
    good_spawn_points = []
    for point in spawn_points:
        this_waypoint = town_map.get_waypoint(point.location,
         project_to_road=True, lane_type=(carla.LaneType.Driving))
        if this_waypoint.road_id in good_roads:
            good_spawn_points.append(point)

    all_waypoint_pairs = town_map.get_topology()
    good_lanes = []
    for w in all_waypoint_pairs:
        if w[0].road_id in good_roads:
            good_lanes.append(w)

    blueprint_library = world.get_blueprint_library()
    vehicle_blueprints = blueprint_library.filter("*vehicle*")

    start_point = random.choice(good_spawn_points)
    ego_vehicle = world.spawn_actor(
        blueprint_library.filter("etron")[0], start_point
    )
    actor_list.append(ego_vehicle)

    sensor_list = []


    # kamera RGB
    camera_rgb_install()

    #main loop
    quit = False
    images = []
    angles = []
    path = f"64_batched_data_town05"
    num_batches = 0
    for lane in good_lanes:
        #loop within a lane
        if quit:
            break
        for wp in lane[0].next_until_lane_end(20):
            start_point = wp.transform
            ego_vehicle.set_transform(start_point)
```

```
        time.sleep(2)
        initial_yaw = start_point.rotation.yaw

        for i in range(5):
            world.tick()

            trans = start_point
            angle_adj = random.randrange(-YAW_ADJ_DEGREES, YAW_ADJ_DEGREES, 1)
            trans.rotation.yaw = initial_yaw +angle_adj
            ego_vehicle.set_transform(trans)
            time.sleep(1)
            if cv2.waitKey(1) == ord(’q’):
                quit = True
                break

            s_frame = sensor_queue.get(True, 1.0)
            image = s_frame[0]

            actual_angle = ego_vehicle.get_transform().rotation.yaw - initial_yaw
            if actual_angle <-180:
                actual_angle +=360
            elif actual_angle >180:
                actual_angle -=360
            actual_angle = str(int(actual_angle))
            angle = float(actual_angle)/YAW_ADJ_DEGREES
            angles.append(angle)

            img = np.float32(img)
            img_gry = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
            image = img_gry[height_from:,width_from:width_to]
            canny = cv2.Canny(np.uint8(image),50,150)
            images.append(canny[:, :, None] / 255)

            if(len(images) == 64):
                try:
                    np.save(f’{path}/images/64_images_{num_batches}’, np.array(images))
                    np.save(f’{path}/labels/64_labels_{num_batches}’, np.array(angles))
                except FileNotFoundError:
                    os.makedirs(f"{path}")
```

```
                        np.save(f'{path}/images/64_images_{num_batches}', np.array(images))
                        np.save(f'{path}/labels/64_labels_{num_batches}', np.array(angles))
                    num_batches += 1
                    images = []
                    angles = []


finally:
    #clean up
    cv2.destroyAllWindows()
    for actor in actor_list:
        actor.destroy()
    for sensor in world.get_actors().filter('*sensor*'):
        sensor.destroy()
```

2. Prilog 2

models.py

```python

from constants import PROCESSED_IMG_SHAPE
from keras.layers import RNN, Conv2D, Dense, Dropout, Flatten, InputLayer, Reshape
from keras.models import Sequential
from kerasncp.tf import LTCCell
from kerasncp.wirings import NCP


def ltc_model():

    # Set the NCP wiring
    wiring = NCP(
        inter_neurons=12,  # Number of inter neurons
        command_neurons=8,  # Number of command neurons
        motor_neurons=1,  # Number of motor neurons
        sensory_fanout=4,  # How many outgoing synapses has each sensory neuron
        inter_fanout=4,  # How many outgoing synapses has each inter neuron
        recurrent_command_synapses=4,  # Now many recurrent synapses are in the
        # command neuron layer
        motor_fanin=6,  # How many incoming synapses has each motor neuron
    )

    # Create the the NCP cell based on the LTC neuron.
    ncp_cell = LTCCell(wiring)

    # Build the sequential hybrid model
    model = Sequential()
    model.add(InputLayer(input_shape=PROCESSED_IMG_SHAPE))
    model.add(Conv2D(24, 5, 2, activation=’relu’))
    model.add(Conv2D(36, 5, 2, activation=’relu’))
    model.add(Conv2D(48, 5, 2, activation=’relu’))
    model.add(Conv2D(64, 3, activation=’relu’))
    model.add(Conv2D(64, 3, activation=’relu’))
    model.add(Dropout(0.5))
```

```
    model.add(Flatten())
    model.add(Dense(100, activation='relu'))
    model.add(Reshape((1, -1)))
    model.add(RNN(ncp_cell, unroll=True))
    model.summary()
    return model
```

3. Prilog 3

data_generator.py

```python
import matplotlib.pyplot as plt
import numpy as np
from keras.utils import Sequence

class DataGenerator(Sequence):
    """
    Class that instantiates the Keras data generators.
    """

    def __init__(self, x_files, y_files, dims=(80, 80), n_channels=3):

        self.dims = dims
        self.x_files = x_files
        self.y_files = y_files
        self.n_batches = len(self)  # Number of batches
        self.n_channels = n_channels

    def __len__(self):
        return len(self.x_files)

    def __getitem__(self, index):
        # Load the input image frames at the current index
        x = np.load(self.x_files[index])

        # Load the output steering angles at the current index
        y = np.load(self.y_files[index])

        return x, y

    def on_epoch_end(self):
        pass

    def __data_generation(self):
        pass
```

4. Prilog 4

batch_data_generator.py

```python
from data_generator import DataGenerator

class BatchDataGenerator(DataGenerator):
    def __init__(self, x_set, y_set, batch_size, shuffle=True, normalize_labels=True):
        super().__init__(x_set, y_set, batch_size, shuffle)
        self.normalize_labels = normalize_labels

    def __getitem__(self, index):
        # Generate one batch of data
        batch_x, batch_y = super().__getitem__(index)

        if self.normalize_labels:
            # Perform label normalization here
            true_max_train = max(abs(min(batch_y)), max(batch_y))
            batch_y *= (1.0 / true_max_train)

        return batch_x, batch_y
```

5. Prilog 5

constants.py

```
TRAIN_DATA_DIR = "./64_batched_data_town05/images/*.npy"
TRAIN_LABELS_DIR = "./64_batched_data_town05/labels/*.npy"

TEST_SIZE = 0.2
RANDOM_STATE = 42

BATCH_SIZE = 64
NB_EPOCHS = 1000
LR = 5e-06  # Since trains the model to the best val_loss in 10
 epochs (model starts overfitting)
VERBOSITY = 1

INPUT_SHAPE = (360, 640)  # Image input shape

PROCESSED_IMG_SHAPE = (144, 320, 1)  # Image shape after processing
```

6. Prilog 6

model_build.py

```
import time
from glob import glob  # Finds all path names matching specified pattern

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from batch_data_generator import BatchDataGenerator
from constants import (
    BATCH_SIZE,
    LR,
    NB_EPOCHS,
    RANDOM_STATE,
    TEST_SIZE,
    TRAIN_DATA_DIR,
    TRAIN_LABELS_DIR,
    VERBOSITY,
)
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoard
from tensorflow.keras.optimizers import Adam

from models import ltc_model

# Get the HDF5 sunny camera and log files
cam_files = sorted(glob(TRAIN_DATA_DIR))
log_files = sorted(glob(TRAIN_LABELS_DIR))

X_train, X_val, y_train, y_val = train_test_split(
    cam_files, log_files, test_size=TEST_SIZE, random_state=RANDOM_STATE
)


# Instantiate the train, validation and test data generators
""" train_gen = DataGenerator(X_train, y_train)
val_gen = DataGenerator(X_val, y_val) """

```

```
train_gen = BatchDataGenerator(X_train, y_train, batch_size=BATCH_SIZE, normalize_labels=True)
val_gen = BatchDataGenerator(X_val, y_val, batch_size=BATCH_SIZE, normalize_labels=True)

# Store architecture function references and model names in a dictionary.
# This dictionary is utilised to get the necessary model from the parsed command line arguments.
arch_dict = {"ltc": [ltc_model, "ltc", "CNN_LTC"]}

model, model_name, model_name_plot = arch_dict["ltc"]

# Initialise the optimiser
optimizer = Adam(learning_rate=LR)

# Get the Keras sequential/functional model by using its reference
model = model()

model.compile(
    loss="mean_squared_error", optimizer=optimizer, metrics=["mean_squared_error"]
)


cps_path = f"models/{model_name}_model" + "-{val_loss:03f}.h5"

# Create a Keras 'ModelCheckpoint' callback to save the best model
checkpoint = ModelCheckpoint(
    cps_path,
    monitor="val_mean_squared_error",
    verbose=VERBOSITY,
    save_best_only=True,
    mode="auto",
)
NAME = f"{model_name}_model_{int(time.time())}"
tensorboard = TensorBoard(
    log_dir="logs/{}".format(NAME),
    write_graph=True,
    update_freq="epoch",
    histogram_freq=1,
)

```

```python
# Start training the model
history = model.fit(
    train_gen,
    epochs=NB_EPOCHS,
    verbose=VERBOSITY,
    validation_data=val_gen,
    callbacks=[checkpoint, tensorboard]
)

# Plot the training and validation losses
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.title(f"{model_name_plot} Model Loss (learning rate: {LR})")
plt.ylabel("loss")
plt.xlabel("epoch")
plt.legend(["train_loss", "val_loss"], loc="upper left")
plt.show()
```

7. Prilog 7

visualize_model.py

```
# %%
from tensorflow.keras.utils import load_img, img_to_array, plot_model
import numpy as np

from tensorflow.keras.models import load_model, Model
from kerasncp.tf import LTCCell
from kerasncp.wirings import NCP
import matplotlib.pyplot as plt
import cv2
import pydot
import seaborn as sns

from constants import PROCESSED_IMG_SHAPE
from keras.layers import (
    RNN,
    Conv2D,
    Dense,
    Dropout,
    Flatten,
    InputLayer,
    Reshape,
)
from keras.models import Model, Sequential

# %%

img_path = './images/Screenshot.png'
img = load_img(img_path, target_size=(144,320),color_mode="grayscale")
img_tensor = img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
print(img_tensor.shape)

# %%
model = load_model("./models/cnn_ncp_model-0.052588.h5", custom_objects={"LTCCell": LTCCell})
activation_model = Model(inputs=model.inputs, outputs=model.layers[0].output)
activation = activation_model(img_tensor)
```

```

# %%
plt.figure(figsize=(20,12))
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow(activation[0,:,:,i])
plt.show()

# %%
plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)

# %%
# Create a new graph with horizontal layout and colors
graph = pydot.Dot(graph_type='graph', rankdir='LR', bgcolor='white')

# Iterate through each layer in the model
for layer in model.layers:
    if isinstance(layer, Model):
        # For submodels (e.g., recurrent layers), add subgraph clusters
        subgraph = pydot.Cluster(str(id(layer)), label=layer.name)
        for sub_layer in layer.layers:
            subgraph.add_node(pydot.Node(sub_layer.name))
        graph.add_subgraph(subgraph)
    else:
        # For regular layers, add nodes with layer names
        graph.add_node(pydot.Node(layer.name))

# Add edges to connect layers
for i in range(1, len(model.layers)):
    graph.add_edge(pydot.Edge(model.layers[i-1].name, model.layers[i].name))

# Save the visualization to a file
graph.write_png('images/model_plot.png')


# %%
LTCCell = model.layers[9].get_config()['cell']
print(LTCCell)
print(LTCCell['config'])
```

```python
sensory_adjacency_matrix = np.array(LTCCell['config']['sensory_adjacency_matrix'])
print(sensory_adjacency_matrix.shape)


# %%
wiring = NCP(
        inter_neurons=12,  # Number of inter neurons
        command_neurons=8,  # Number of command neurons
        motor_neurons=1,  # Number of motor neurons
        sensory_fanout=4,  # How many outgoing synapses has each sensory neuron
        inter_fanout=4,  # How many outgoing synapses has each inter neuron
        recurrent_command_synapses=4,  # Now many recurrent synapses are in the
        # command neuron layer
        motor_fanin=6,  # How many incoming synapses has each motor neuron
    )

ncp_cell = LTCCell(wiring)
model = Sequential()
model.add(InputLayer(input_shape=PROCESSED_IMG_SHAPE))
model.add(Conv2D(24, 5, 2, activation='relu'))
model.add(Conv2D(36, 5, 2, activation='relu'))
model.add(Conv2D(48, 5, 2, activation='relu'))
model.add(Conv2D(64, 3, activation='relu'))
model.add(Conv2D(64, 3, activation='relu'))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(24, activation='relu'))
model.add(Reshape((1, -1)))
model.add(RNN(ncp_cell, unroll=True))
model.summary()

# %%
sns.set_style("white")
plt.figure(figsize=(20, 15))
legend_handles = ncp_cell.draw_graph(layout='shell',neuron_colors={"command": "tab:cyan"})
plt.legend(handles=legend_handles, loc="upper center", bbox_to_anchor=(1, 1), fontsize=20)
```

```
sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.show()
```

8. Prilog 8

scenario_runner.py

```
import glob
import math
import os
import random
import sys
from queue import Empty, Queue

import cv2
import matplotlib.pyplot as plt
import numpy as np
from kerasncp.tf import LTCCell
from kerasncp.wirings import NCP
from ncps.tf import LTC
from PIL import Image
from tensorflow.keras.models import load_model

try:
    sys.path.append(
        glob.glob(
            "../carla/dist/carla-*%d.%d-%s.egg"
            % (
                sys.version_info.major,
                sys.version_info.minor,
                "win-amd64" if os.name == "nt" else "linux-x86_64",
            )
        )[0]
    )
except IndexError:
    pass

import carla

PREFERRED_SPEED = 60
SPEED_THRESHOLD = 5

# max angle when tarining images were produced
YAW_ADJ_DEGREES = 35
MAX_STEER_ANGLE = 35

```

```
#mount point of camera on the car
CAMERA_POS_Z = 1.6
CAMERA_POS_X = 0.9

HEIGHT = 360
WIDTH = 640

HEIGHT_REQUIRED_PORTION = 0.4 #bottom share, e.g. 0.1 is take lowest
 10% of rows
WIDTH_REQUIRED_PORTION = 0.5

# image crop - same as in model input
height_from = int(HEIGHT * (1 -HEIGHT_REQUIRED_PORTION))
width_from = int((WIDTH - WIDTH * WIDTH_REQUIRED_PORTION) / 2)
width_to = width_from + int(WIDTH_REQUIRED_PORTION * WIDTH)

#adding params to display text to image
font = cv2.FONT_HERSHEY_SIMPLEX
# org
org = (30, 30)
org2 = (30, 50)
fontScale = 0.5
# white color
color = (255, 255, 255)
# Line thickness
thickness = 1


model = load_model("Lane_model/models/ltc_model-0.017563.h5",
 custom_objects={"LTCCell": LTCCell},compile=False)
model.compile()


def sensor_callback(sensor_data, sensor_queue, sensor_name):

    image = np.reshape(np.copy(sensor_data.raw_data),(sensor_data.
     height,sensor_data.width,4))
    sensor_queue.put((image, sensor_name))


def camera_rgb_install():
    camera_transform = carla.Transform(
            carla.Location(x=CAMERA_POS_X, z=CAMERA_POS_Z)
```

```python
        )
    camera_blueprint = world.get_blueprint_library().find("sensor.
     camera.rgb")
    camera_blueprint.set_attribute("image_size_x", '640')
    camera_blueprint.set_attribute("image_size_y", '360')
    camera = world.spawn_actor(
        camera_blueprint, camera_transform, attach_to=ego_vehicle
    )

    camera.listen(lambda image: sensor_callback(image, sensor_queue,
     "camera"))
    actor_list.append(camera)
    sensor_list.append(camera)

def maintain_speed(s):
    '''
    this is a very simple function
    too maintan desired speed
    s arg is actual current speed
    '''
    if s >= PREFERRED_SPEED:
        return 0
    elif s < PREFERRED_SPEED - SPEED_THRESHOLD:
        return 0.8
    else:
        return 0.3


def predict_angle(im):
    # tweaks for prediction
    img = np.float32(im)
    img_gry = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    img_gry = cv2.resize(img_gry, (WIDTH,HEIGHT))
    # this version adds taking lower side of the image
    img_gry = img_gry[height_from:,width_from:width_to]
    img_gry = img_gry.astype(np.uint8)
    canny = cv2.Canny(img_gry,50,150)

    #cv2.imshow('processed image',canny)
    canny = canny /255
    input_for_model = canny[ :, :, None]
    input_for_model = np.expand_dims(input_for_model, axis=0)
    #print('input shape: ',input_for_model.shape)
```

```
    angle = model(input_for_model,training=False)

    return  angle.numpy()[0][0] * YAW_ADJ_DEGREES / MAX_STEER_ANGLE


actor_list = []

client = carla.Client("localhost", 2000)
client.set_timeout(10.0)
client.load_world("Town05")


# Once we have a client we can retrieve the world that is currently
 running.
world = client.get_world()

#world.set_weather(carla.WeatherParameters.WetNoon)

try:
    # We need to save the settings to be able to recover them at the
     end
    # of the script to leave the server in the same state that we
     found it.
    original_settings = world.get_settings()
    settings = world.get_settings()

    traffic_manager = client.get_trafficmanager(8000)
    traffic_manager.set_synchronous_mode(True)

    # We set CARLA syncronous mode
    settings.fixed_delta_seconds = 0.05
    settings.synchronous_mode = True
    world.apply_settings(settings)

    sensor_queue = Queue()

    spectator = world.get_spectator()

    blueprint_library = world.get_blueprint_library()


    vehicle_blueprints = blueprint_library.filter("*vehicle*")

```

```
    town_map = world.get_map()
    good_roads = [37]
    spawn_points = town_map.get_spawn_points()
    good_spawn_points = []
    for point in spawn_points:
        this_waypoint = town_map.get_waypoint(point.location,
         project_to_road=True, lane_type=(carla.LaneType.Driving))
        if this_waypoint.road_id in good_roads:
            good_spawn_points.append(point)


    start_point = random.choice(good_spawn_points)
    ego_vehicle = world.spawn_actor(
        blueprint_library.filter("etron")[0], start_point
    )
    actor_list.append(ego_vehicle)


    """ for i in range(0, 50):
        npc_vehicle = world.try_spawn_actor(
            random.choice(vehicle_blueprints), random.choice(
             spawn_points)
        )
        if npc_vehicle is not None:
            npc_vehicle.set_autopilot(True)
            actor_list.append(npc_vehicle) """


    sensor_list = []
    # kamera RGB
    camera_rgb_install()

    camera_bp = world.get_blueprint_library().find("sensor.camera.rgb
     ")
    x_pos = 0
    y_pos = 0
    print(ego_vehicle.get_transform().rotation.yaw)
    if ego_vehicle.get_transform().rotation.yaw == 0.0:
        x_pos = -5
    else:
        y_pos = -5


```

```
    camera_transform = carla.Transform(
        carla.Location(x=x_pos,y=y_pos, z=4),
        carla.Rotation(pitch=-20, yaw=ego_vehicle.get_transform().rotation.yaw),
    )
    camera_bp.set_attribute("image_size_x", '640')
    camera_bp.set_attribute("image_size_y", '360')
    camera = world.spawn_actor(camera_bp, camera_transform, attach_to=ego_vehicle)
    camera.listen(lambda image: sensor_callback(image, sensor_queue, "3rd person camera"))
    sensor_list.append(camera)

    distance_traveled = 0.0
    lane_crossings = 0
    collisions = 0
    previous_lane_id = None

    time_steps = []
    distances = []
    crossings = []
    collision_count = []

    # Lane invasion sensor
    bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
    sensor = world.spawn_actor(bp, carla.Transform(), attach_to=ego_vehicle)
    sensor.listen(lambda event: on_invasion(event))

    def on_invasion(event):
        global lane_crossings
        # Handle lane invasion event here
        lane_crossings += 1
        print("Lane invasion detected:", event)

    # Collision sensor
    coll_bp = world.get_blueprint_library().find('sensor.other.collision')
    coll_sensor = world.spawn_actor(coll_bp, carla.Transform(), attach_to=ego_vehicle)
    coll_sensor.listen(lambda event: on_collision(event))

```

```python
def on_collision(event):
    global collisions
    # Handle collision event here
    collisions += 1
    cv2.imwrite('collision_%{collisions}.jpg',np.array(event.frame))
    print("Collision detected:", event)



while True:
# Carla Tick
    world.tick()
    if cv2.waitKey(1) == ord('q'):
        quit = True
        break

    try:
        s_frame = sensor_queue.get(True, 1.0)
        image = s_frame[0]

        predicted_angle = predict_angle(image)
        image = cv2.putText(image, 'Predicted angle in lane: '+str(int(predicted_angle * 90)), org, font, fontScale, color, thickness, cv2.LINE_AA)

        v = ego_vehicle.get_velocity()
        a = ego_vehicle.get_acceleration()

        speed = round(3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2),0)
        image = cv2.putText(image, 'Speed: '+str(int(speed)), org2, font, fontScale, color, thickness, cv2.LINE_AA)

        acceleration = round(math.sqrt(a.x**2 + a.y**2 + a.z**2),1)
        estimated_throttle = maintain_speed(speed)

        location = ego_vehicle.get_location()
        previous_location = location
        if previous_location:
            distance_traveled += location.distance(previous_location)
```

```


            time_steps.append(world.get_snapshot().timestamp.elapsed_seconds)
            distances.append(distance_traveled)
            crossings.append(lane_crossings)
            collision_count.append(collisions)

            ego_vehicle.apply_control(
                carla.VehicleControl(steer=-predicted_angle, throttle=estimated_throttle)
            )
            #cv2.imshow('RGB Camera',image)

            camera_3rd = sensor_queue.get(True, 1.0)

            if(camera_3rd[0] is not None):
                cv2.imshow('3rd person',np.array(camera_3rd[0]))


        except Empty:
            print("Some of the sensor information is missed")

finally:
    cv2.destroyAllWindows()
    world.apply_settings(original_settings)
    for actor in actor_list:
        actor.destroy()
    for sensor in world.get_actors().filter('*sensor*'):
        sensor.destroy()


    fig, (ax1) = plt.subplots(1, 1, figsize=(10, 12))

    # Plot lane crossings
    ax1.plot(time_steps, crossings, label='Prijelazi preko crte', color='orange')
    ax1.set_xlabel('Vrijeme (s)')
    ax1.set_ylabel('Broj prijelaza preko crte')
    ax1.legend(loc='upper left')

    # Add number of collisions to the plot
    ax1.twinx()  # Create a twin y-axis
```

```
ax1.plot(time_steps, collision_count, label='Sudari', color='red'
 )
ax1.set_ylabel('Broj sudara')
ax1.legend(loc='upper right')

ax1.grid(True)

# Show the plot
plt.tight_layout()
plt.show()
print("done.")
```