Svima koji su me podržavali kroz izradu ovog rada, bez vas ne bih mogao.

SADRŽAJ

1. Uvod

U posljednje vrijeme, naročito u posljednjih par godina, umjetna inteligencija doživljava procvat kakav nikad dosad nije viđen. Postaje sveprisutna u sve više područja, kao što su trgovina, administracija, edukacija, zdravstvo, promet, navigacija, robotika, računalna sigurnost, zabava, pa čak i audiovizualna umjetnost.

Jedno od najranijih ciljnih područja umjetne inteligencije jest automatsko upravljanje vozilima, pogotovo letećim vozilima, odnosno zrakoplovima i drugim sličnim vozilima. Pojmovi kao što su *autopilot*, *bespilotna letjelica*, to jest, *dron* i *samovozeće vozilo* vrlo su poznati i prisutni u svakodnevnim razgovorima, i reprezentiraju uspješne posljedice spajanja umjetne inteligencije s vozilima, kao što je automatsko upravljanje zrakoplovima u mirnim segmentima komercijalnih letova, ali i projekte koji se još usavršavaju i pokazuju veliki potencijal za budućnost, kao što su samovozeći automobili za privatne ili komercijalne svrhe.

U sklopu ovog diplomskog rada praćen je i dokumentiran razvoj pojednostavljenog modela simulatora leta s mogućnosti automatskog upravljanja, od početne ideje do realizacije, s naglaskom na razvoj umjetne inteligencije za automatsko upravljanje. U drugom poglavlju predstavljena je kratka povijest simulatora leta, uključujući simulatore s automatskim upravljanjem. U trećem poglavlju nalazi se uvod u osnove strojnog učenja koje su potrebne za razumijevanje metode razvoja sustava za automatsko upravljanje. Četvrto poglavlje sadrži detaljno pojašnjenje kako su te osnove iskorištene za izradu samog modela strojnog učenja koji se koristi za automatsko upravljanje zrakoplovom. U petom poglavlju prikazano je kako su točno koncepti i strategije iz četvrtog poglavlja implementirane u programskom rješenju, to jest, u razvijenoj aplikaciji. Šesto poglavlje je zaključak u kojemu je sažeto ponovljeno sve što je ostvareno u sklopu ovog rada, uz završni osvrt na uspješnost u rješavanju danog zadatka.

2. Pregled područja

Povijest simulacije leta počinje još u vrlo ranim tridesetom godinama prošlog stoljeća, kada su simulatori leta bili većinom mehaničke naprave namijenjene treniranju budućih, kako civilnih, tako i vojnih pilota. Najraniji takav dokumentirani simulator leta zvao se Link Trainer (JEON, 2015).

Kako je vrijeme odmicalo i tehnologija je postajala sve naprednija, simulatori leta postali su kompleksni računalni programi s potpuno digitalnim virtualnim okruženjima. I dalje se za treniranje budućih pilota koriste simulatori leta koji su fizički izgrađeni od uvjerljivih replika pravih zrakoplova kojima će ti piloti upravljati, ali se simulacije na tim simulatorima u modernim vremenima odvijaju potpuno programski, iako s vrlo uvjerljivim ulazno-izlaznim jedinicama. No, postoje i simulatori leta koji su komercijalno dostupni široj javnosti u obliku grafičkih aplikacija i videoigara namijenjenim pokretanju na osobnim računalima. Jedan od prvih takvih komercijalno dostupnih simulatora leta, Microsoft Flight Simulator (Tolliver, 1996), bio je jednostavan simulator leta u pseudo-3D okruženju. Iako daleko impresivniji od najranijih simulatora leta, i dalje je bio vrlo ograničen snagom osobnih računala svojeg perioda.

Iz godine u godinu tehnologija je sve više napredovala, i nove verzije Microsoftovog Flight Simulatora izlazile su svakih nekoliko godina. U 2017. godini objavljen je Microsoft AirSim (Shah et al., 2017), projekt koji spaja prijašnje iskustvo u Microsoftu sa simulacijama leta i umjetnu inteligenciju, to jest, automatsko upravljanje. To je simulator leta i vožnje razvijen u pogonskom sustavu Unreal Engine, sustavu ne toliko drukčijem od sustava Unity, u kojem je izrađen ovaj rad, u svrhu podržavanja istraživanja automatskog upravljanja vozila, s primarnim ciljem letjelica. Projekt je otvorenog pristupa i svatko tko želi razvijati sustave automatskog upravljanja može preuzeti projekt i razviti svoj model autonomnog automobila ili letjelice.

U akademskom svijetu također postoje brojni projekti koji se bave izradom pojednostavljenih modela bespilotnih letjelica i letjelica s automatskim upravljanjem s ciljem istraživanja koncepta. Pristup razvoju pojednostavljenog modela inteligentnog agenta koji upravlja vozilom navođen stanjem okoline zadatak je kojem se može pris-

tupiti na više načina. Na primjer, možemo razviti model letjelice koja svijet oko sebe vidi kroz kameru, to jest, video signal uživo. Alternativno letjelica može percipirati svoju okolinu pomoću infracrvenih senzora, sonara ili na brojne druge načine. Primjer jednog takvog akademskog rada je Nielsen (2021), simulator zrakoplova s vertikalnim uzlijetanjem i slijetanjem s mogućnosti automatskog upravljanja, razvijen u pogonskom sustavu Unreal Engine. Još jedan primjer takvog akademskog rada u kojemu letjelica svoju okolinu percipira pomoću senzora daljine i koristi umjetnu neuronsku mrežu za automatsko upravljanje je Bijelić (2018). Rad je napravljen u pogonskom sustavu Unreal Engine i predstavlja kako razviti sustav za automatsko upravljanje bespilotne letjelice, to jest, drona.

Ovaj diplomski rad u određenoj je mjeri inspiriran zadnjim navedenim radom iako se vrste i okoline letjelica te načini na koji one interpretiraju svoju okolinu uvelike razlikuju između dva rada.

3. Strojno učenje

3.1. Uvod

Strojno učenje je, kao podskup umjetne inteligencije, grana računarske znanosti koja se bavi razvijanjem tehnologije koja omogućava da računala obavljaju proizvoljne poslove bez da ih se nužno programira za te poslove, najčešće iz razloga što "ručno" ostvarivanje željenih ciljeva nije realistično izvedivo, učinkovito ili priuštivo. Primjer zadatka za koji bi bilo bolje koristiti strojno učenje nego "ručno" programiranje jest prepoznavanje ljudskih lica na fotografijama.

Ako je naš cilj ostvariti rješenje tog problema bez korištenja strojnog učenja, moguće je pojednostaviti problem na način da reduciramo objekt koji želimo prepoznati na njegove karakteristične manje dijelove. Uz puno posla i par matematičkih trikova, možemo napisati program koji na slici uspoređivanjem relativne količine svjetline u bojama traži vertikalnu liniju ispod koje je horizontalna linija i iznad koje su dvije ovalne regije tamnijih i kompleksnijih boja od ostatka lica. Iz načina na koji je to opisano, ljudsko lice moglo bi zadovoljiti sve kriterije, no takav je sustav previše krut u svojoj definiciji i mogao bi, na primjer, prepoznati voće razasuto po stolu kao jedno ili više lica, ili čak ne prepoznati ljudska lica koja su, na primjer, zaklonjena nečime kao što je kosa, ili jednostavno nisu uspravno orijentirana.

Postoji previše različitih načina da se ljudsko lice prikaže u fotografiji da bi ikakav programer pouzdano mogao napisati program koji bi ispravno prepoznao barem većinu prikazanih ljudskih lica, a kamoli izbjegao krivu klasifikaciju objekata koji nisu ljudsko lice. Također postoji previše različitih načina za uspješno upravljati simulatorom leta, i previše situacija i stanja u kojima simulacija hipotetski može biti.

3.1.1. Model, funkcija gubitka i optimizacijski postupak

Tri glavna dijela svakog algoritma strojnog učenja su model, funkcija gubitka i optimizacijski postupak. Najkraće rečeno, model je nešto što iz ulaznih podataka koje

mu priskrbimo daje izlazne podatke, nekako kao funkcija. Razlika je da model nije jedna funkcija, već se može opisati kao skup parametriziranih funkcija. Jedan primjer takvog konstrukta je linearna, to jest afina funkcija s promjenjivim parametrima. Opća jednadžba tog modela glasila bi:

$$f(x, y) = Ax + By + C$$

Možemo prepoznati varijable x i y kao ulaze u funkciju, A i B kao vezane parametre i C kao slobodni parametar. Modificiranjem vrijednosti A, B i C dobivamo drukčije funkcije, i upravo na taj način rade modeli strojnog učenja. Cilj algoritma strojnog učenja je podesiti parametre modela kako bismo dobili funkciju koja rješava željeni problem optimalno, ili barem najbolje moguće. Taj proces zovemo učenjem ili treniranjem modela.

Funkcija gubitka je funkcija koja govori koliko model griješi, to jest koliki je "gubitak" točnosti, na jednom primjeru. Riječ primjer koristimo za jedan ulazni podatak (ili najčešće par podataka, ovisi o veličini ulaza u model) prikazan modelu. Treniranje modela zapravo je "prikazivanje primjera" modelu, to jest izračunavanje izlaza koji model dodijeli tim primjerima. Funkcija gubitka - na jednom primjeru koji je prikazan modelu, to jest funkcija pogreške - na cijelom skupu podataka koji je prikazan modelu, vraća brojčanu vrijednost koja označava koliko model "griješi" pri rješavanju danog zadatka - veće vrijednosti označavaju veću pogrešku i obrnuto. Funkciju gubitka podešavamo svakom različitom zadatku, na primjer, ako je željeno rješenje locirati neku točku u prostoru, funkcija gubitka mogla bi biti kvadratna udaljenost modelom predviđene lokacije od stvarne lokacije točke. Tada bi se "kažnjavalo" netočne pokušaje lociranja točke, i kazna bi kvadratno rasla ovisno o tome koliko je pokušaj predviđanja daleko od cilja.

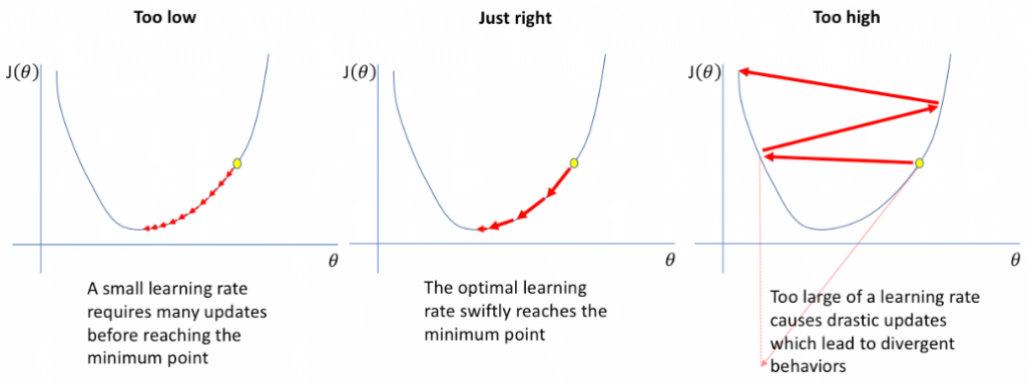
Optimizacijski postupak je postupak čijom primjenom mijenjamo model, to jest, parametre modela, sukladno vrijednosti funkcije gubitka, odnosno funkcije pogreške, s ciljem smanjivanja funkcije gubitka i bolje učinkovitosti u zadanom zadatku. Mijenjanje parametra modela može se odvijati nakon svakog prikazanog primjera, što zovemo *on-line* učenje, nakon određenog broja primjera, što zovemo učenje s mini-grupama (engl. *mini-batch learning*), ili nakon prikazivanja svih primjera namijenjenih za učenje (engl. *batch learning*)

Jedan od najosnovnijih i najpoznatijih optimizacijskih algoritama u strojnom učenju je stohastički gradijentni spust (engl. *stochastic gradient descent* - SGD). Ideja iza stohastičkog gradijentnog spusta je ta da težine modela kojeg treniramo postepeno mijenjamo u onom smjeru koji bi teoretski najbrže i najviše smanjio iznos funkcije

gubitka. To se postiže na način da napravimo derivaciju, to jest gradijent funkcije gubitka po parametrima koje želimo optimizirati, čime dobijemo smjer rasta funkcije u ovisnosti o tim parametrima, i zatim postepeno mijenjamo te parametre u suprotnom smjeru, ovisno o iznosu derivacije i konstanti koju nazivamo stopom učenja (engl. *learning rate* - LR - ili η). Stopa učenja je tu da odredi koliko naglo želimo mijenjati težine modela prema percipiranom optimumu. Uz θ_n kao oznaku vrijednosti nekog parametra θ u n-toj iteraciji optimizacijskog postupka, jedan korak, to jest jedna iteracija postupka stohastičkog gradijentnog spusta izražava se kao:

$$\theta_{n+1} = \theta_n - \eta \cdot \frac{\partial L}{\partial \theta_n} \tag{3.1}$$

Važno je odrediti prikladnu vrijednost stope učenja jer uz nedovoljno veliku stopu učenja model uči presporo, a uz preveliku stopu učenja postoji vjerojatnost da model sa svojim prevelikim "korakom" prema optimumu divergira i udaljuje se sve više od optimuma umjesto da se približava i konvergira.



Slika 3.1: Učinci različitih vrijednosti stope učenja (Konar et al., 2020)

Primjer jednog potpuno drukčijeg optimizacijskog algoritma je genetski algoritam, koji pripada evolucijskim algoritmima, čija inspiracija također leži u prirodnim procesima, ali ne procesima učenja pojedinačnog živog bića, već u procesima razvoja i adaptacije cijele populacije živih bića kroz brojne generacije.

Način na koji se u genetskom algoritmu optimiziraju parametri bilo kojeg modela je ostvaren kroz par koraka - mjerenje dobrote, selekcija, križanje i mutacija. Kao prvo, genetski algoritam sadrži "populaciju" - skup od određeno mnogo modela, koje u ovom kontekstu zovemo jedinke. Najčešće su to na početku procesa modeli s nasumično generiranim parametrima. Svaka konfiguracija parametara modela naziva se kromosomom. Dakle svaku jedinku, to jest, model u populaciji, možemo gledati kao strukturu koja sadrži kromosom određen specifičnim iznosima parametara te jedinke.

Mjerenje dobrote je ocjenjivanje koliko je svaka jedinka učinkovita u obavljanju postavljenog zadatka, i dodjeljivanje svakoj jedinki vrijednosti koja odražava tu razinu učinkovitosti - na neki način suprotno funkciji gubitka, to jest kazne, ali s istom svrhom. Postupkom selekcije se na neki način, najčešće u ovisnosti o dobrotama jedinki, iz trenutačne populacije izdvaja određen broj jedinki u pripremi za sljedeći korak - križanje.

Križanje je stvaranje nove jedinke od odabranih jedinki na način da se određenim postupkom njihovi kromosomi pomiješaju, to jest, da se kromosom novostvorene jedinke sastoji od dijelova kromosoma njenih "roditelja". Drugim riječima, parametri nove jedinke su kombinacija parametara jedinki od kojih je ona stvorena, jedinki koje su najvjerojatnije odabrane kao najkvalitetnije jedinke u populaciji, čija svojstva želimo barem imitirati, ali na kraju i usavršiti.

Za kraj, mutacija je događaj s određenom nasumičnom vjerojatnosti prilikom kojeg se vrijednost dijela kromosoma novonastale jedinke mijenja po odabranom algoritmu, uvodeći potencijalne novosti u skup kromosoma u populaciji. Ovaj korak genetskog algoritma pomaže u izbjegavanju problema gdje sustav zapinje u stanju gdje nijedna od jedinki nema parametre povoljne za optimalno rješavanje zadatka.

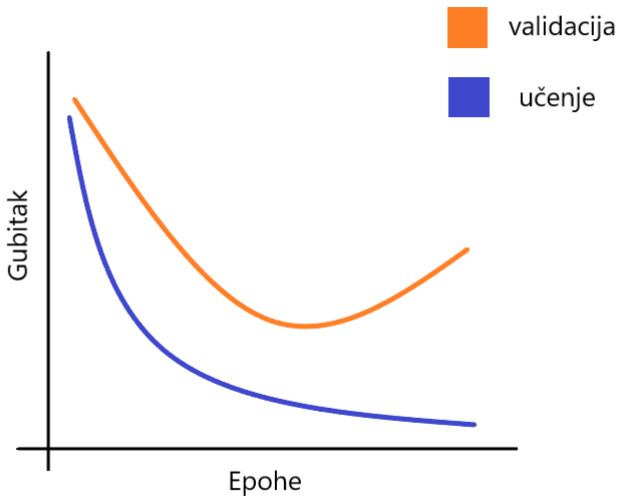
Navedeni koraci ponavljaju se određeni broj puta, ili dok ne nastane jedinka s iznosom funkcije dobrote većim od nekog zadanog praga. Jedno ponavljanje svih koraka naziva se jednom generacijom. U jednoj generaciji se stara populacija ili potpuno zamijeni novom, nastalom križanjem, ili djelomično, izbacivanjem jedinki s najmanjom izmjerenom dobrotom. Općenito, svaki od navedenih koraka ima mnogo varijacija i iz tog su razloga opisani podosta neodređeno, a na programeru koji dizajnira sustav je da odluči koje od tih varijacija bi bilo najpogodnije koristiti za specifični problem čije se rješenje traži.

Postupak optimizacije korišten u ovom diplomskom radu je ranije spomenuti stohastički gradijentni spust, u nadi najjednostavnijeg i najbržeg dostizanja krajnjeg cilja. Iako se ovaj problem može riješiti pomoću više različitih optimizacijskih algoritama, uključujući i gore spomenuti genetski algoritam, algoritam stohastičkog gradijentnog spusta relativno je jednostavan za implementaciju i pritom optimiziranje vrši usmjereno, prema smjeru najnižeg iznosa funkcije pogreške.

3.1.2. Ispitivanje modela

Pripremanje modela strojnog učenja za rad uključuje učenje, to jest treniranje modela, kao što je i prije spomenuto, no kako možemo osigurati da je učenje djelotvorno i da

rezultira dobrim modelom? Odgovor je da, uz treniranje također i ispitujemo model, kao i ljude u školskom i akademskom sustavu. Nakon određenog vremena provedenog trenirajući model, ispitujemo njegovu točnost na primjerima kojima znamo točno rješenje kako bismo evaluirali koliko dobro model rješava željeni zadatak. Pritom je najvažniji dio da primjeri na kojima ispitujemo model moraju nužno biti prethodno neviđeni modelu, jer u suprotnom, ako je model bio treniran na istim primjerima pomoću kojih ga ispitujemo, riskiramo da model jednostavno savršeno nauči raditi zadatak na primjerima za treniranje, s lošim performansama na nikad viđenim primjerima, što bismo nazvali prenaučenost (engl. *overfitting*). Sprječavanje prenaučenosti je zapravo i jedna od glavnih motivacija za razdvajanje dostupnog skupa primjera na skup primjera za učenje, to jest treniranje i na skup za ispitivanje, kako se model ne bi treniralo da rješava zadatak "napamet", po uzoru na skup podataka za učenje, već da generalizira i nalazi uzorke koji su prisutni i u neviđenim podatcima. Kao optimalne parametre modela nećemo izabrati one koji dovode do najnižeg iznosa funkcije gubitka na skupu podataka za učenje, već one koji dovode do najnižeg iznosa funkcije gubitka na skupu podataka za ispitivanje, to jest, *validaciju* modela. Suprotnost prenaučenosti je podnaučenost (engl. *underfitting*), situacija kada model iz jednog ili drugog razloga nema dovoljnu sposobnost rješavanja danog zadatka. Do toga je, na primjer, moguće doći ako model treniramo nedovoljno dugo ili na nedovoljno mnogo primjera, ili ako za stopu učenja odaberemo premalu vrijednost. Odnos između gubitka modela na skupu podataka za učenje i na validacijskom skupu prikazan je slikom 3.2.



Slika 3.2: Gubitak na skupu za učenje i validacijskom skupu podataka kroz epohe

Treniranje modela na skupu modela za učenje uključuje pokazivanje modelu cijelog skupa podataka za učenje, i to često više nego jednom. Jedno iskorištavanje svih primjera iz skupa podataka za učenje u sklopu treniranja modela naziva se jedna epoha. Pri treniranju modela možemo odabrati kroz koliko epoha želimo trenirati model, i toliko će se puta svaki primjer prikazati modelu u svrhu učenja, što radi na sličan način kao ponavljanje već pročitanog gradiva za ispit. Malo više tehnički opis bio bi da optimizacijskim algoritmima jedan prolaz kroz sve podatke za učenje najčešće nije dovoljan da parametre modela dovedu u optimalne vrijednosti, budući da se većina njih bazira na postepenim inkrementima vrijednosti parametara u potrazi za optimumom.

3.2. Vrste strojnog učenja

Postoje tri velike vrste strojnog učenja: nadzirano, nenadzirano i podržano.

Nadzirano strojno učenje osnovano je na činjenici da od modela želimo da napravi određeni zadatak za koji znamo ocijeniti određene ili sve pokušaje rješavanja tog zadatka. Model treniramo na označenom skupu podataka za učenje, što znači da se svaki primjer korišten za treniranje modela sastoji od podataka za ulaz u model i oznake, to jest željenog izlaza iz modela. Model se tada podešava kako bi njegovi izlazi bili što bliži traženim izlazima, to jest oznakama iz primjera za učenje. Dva istaknuta zadatka pogodna za nadzirano strojno učenje su klasifikacija i regresija. Klasifikacija je svrstavanje primjera u jednu ili više klasa s obzirom na ulazne podatke, to jest pridjeljivanje diskretne oznake primjeru. Regresija je pridjeljivanje oznake primjeru iz kontinuiranog raspona, to jest, pridodavanje brojevne oznake na temelju ulaznih podataka.

Nenadzirano strojno učenje odvija se tako da se modelu daju podatci bez ispravne oznake, to jest željenog izlaza, najčešće jer željeni izlaz ne postoji ili je nepoznat. Iz tog razloga cilj ove vrste strojnog učenja najčešće je grupiranje podataka, asocijacija srodnih podataka ili onih s uzročno-posljedičnom vezom, ili neki slični zadatak za koji je u skupu podataka potrebno naći sličnosti i uzorke koji pomažu u davanju podatcima smisla.

Podržano strojno učenje je pristup gdje se model ponaša kao agent u nekoj okolini te iz pokušaja i pogrešaka saznaje kako i s kojim raspoloživim akcijama doći do cilja, uz pomoć navođenja nagradama za poželjne odluke i/ili kaznama za nepoželjne, na sličan način na koji se može trenirati kućne ljubimce, ili čak i ljude.

3.2.1. Izbor pristupa

Iz čiste prirode danog zadatka, za problem ostvarivanja automatskog upravljanja simulatora leta, gotovo odmah nam dolazi do izražaja podržani pristup strojnom učenju, iz nekoliko razloga.

S jedne strane, naučiti algoritam strojnog učenja da upravlja zrakoplovom praktički nemoguće bez ikakvih povratnih informacija o tome koliko je model uspješan, stoga se nenadzirano učenje čini kao najlošiji pristup zadatku.

S druge strane, zadatak nema jedno jedinstveno ispravno rješenje, zbog čega se nadzirano učenje inicijalno ne čini prikladno. Međutim, moguće je modelu problem upravljanja zrakoplovom predstaviti kroz skup podataka za učenje koji se sastoji od zapisanih podataka pokretanja simulacije leta kojom je upravljala iskusna ljudska osoba. Na taj način modelu bi na raspolaganju bili razni primjeri za učenje koji se sastoje od ulaza, to jest informacija o simulaciji za svaki okvir, i izlaza, to jest naredbi koje je ljudski pilot odlučio unijeti. Pretpostavivši da ljudski pilot zna što radi, cilj modela bio bi što bliže imitirati način na koji ta osoba upravlja zrakoplovom. Međutim, taj pristup ima nekoliko problema.

Model možemo naučiti upravljati poput ljudske osobe ali nemoguće je pouzdati se u bilo koju ljudsku osobu da će simulacijom upravljati optimalno, već s vlastitim greškama i pristranostima. Čak i ako pretpostavimo da ljudski pilot može upravljati simulatorom na optimalan način, postoji visoka vjerojatnost da model iz podataka za učenje jednostavno ne može naučiti generalizirati upravljanje zrakoplovom, i da na neviđenim poligonima neće imati dobre rezultate. Kako bi se to spriječilo, moguće je pokušati trenirati model na jako velikom skupu podataka s puno različitih situacija i nadati se da će model naučiti adekvatno generalizirati razne moguće situacije koje se mogu dogoditi pri simulaciji leta.

Ipak, veliki skupovi podataka zauzimaju puno memorijskog prostora i, važnije, treba puno ljudskog vremena da ih se prikupi. Model može, okvir po okvir, upravljati simulacijom daleko brže od normalne brzine simulacije, što znači da kad nemamo potrebu vizualno evaluirati performanse modela, možemo simulaciju kojom upravlja model pokretati koliko god brzo naše sklopovlje dozvoljava. S druge strane, to je daleko brže nego što bi ikakva ljudska osoba mogla procesirati podatke potrebne za upravljati takvom simulacijom, čineći proces pribavljanja skupa podataka mnogo puta sporijom nego u idealnom slučaju.

Ideja puštanja modela da kontrolira simulaciju bez prethodnog znanja, i nagrađivanje modela za poželjne akcije i kažnjavanje za nepoželjne čini se kao pristup koji

najviše odgovara prilici.

3.3. Umjetne neuronske mreže

3.3.1. Umjetni neuron

Jedan su od najkorištenijih alata u strojnom učenju - prva asocijacija velikog dijela populacije kad čuju riječ strojno učenje - umjetne neuronske mreže. Inspirirane prirodnom inteligencijom, umjetne su neuronske mreže matematički konstrukt koji možemo koristiti za predstavljanje sklopa za razmišljanje i donošenje odluka. Izgrađene su od mnoštva građevnih jedinica koje zovemo umjetni neuroni. Također inspirirani prirodom, to jest, prirodnim neuronima - građevnim jedinicama mozgova živih bića, umjetni su neuroni objekti koji primaju brojčane ulaze i na temelju njih računaju brojčane izlaze. To se ostvaruje na način da neuron ima brojčane vrijednosti koje zovemo težine (engl. *weights*), asocirane sa svakim ulazom, kao vezani parametri funkcije. Izlaz neurona računa se kao zbroj umnožaka svakog ulaza i njemu pridijeljene težine. Umjetni neuron također može imati, i u većini slučajeva i ima, još jednu vrijednost koju zovemo pomak (engl. *bias*), koja se zbroji s prethodno spomenutim zbrojem umnožaka ulaza i težina, to jest, težinskom sumom ulaza, kako bi se dobio konačan iznos izlaza neurona - upravo kao slobodni parametar funkcije.

Težine i pomak neurona mogu se mijenjati po želji, ovisno o željenom zadatku koji bi taj neuron trebao izvršiti. Time jednim umjetnim neuronom možemo modelirati linearnu jednadžbu s onoliko nepoznanica koliko i težina, odnosno ulaza u neuron i jednom slobodnom nepoznanicom. Sam po sebi, ovakav model umjetnog neurona mogao bi se koristiti za rješavanje mnogih matematičkih problema, no postoji nesretna činjenica da se svi izračuni koje možemo ostvariti takvim modelom svode na linearne, to jest, afine funkcije. Poznati primjer problema koji se ne mogu riješiti s linearnim matematičkim modelima jest razdvajanje linearno nerazdvojivih točaka funkcijama. Korištenjem modela umjetnog neurona kakav je zasad opisan, takav se zadatak ne bi mogao izvršiti, stoga bi bilo korisno tom modelu dodati nelinearnu komponentu kako bi se povećala njegova izražajnost.

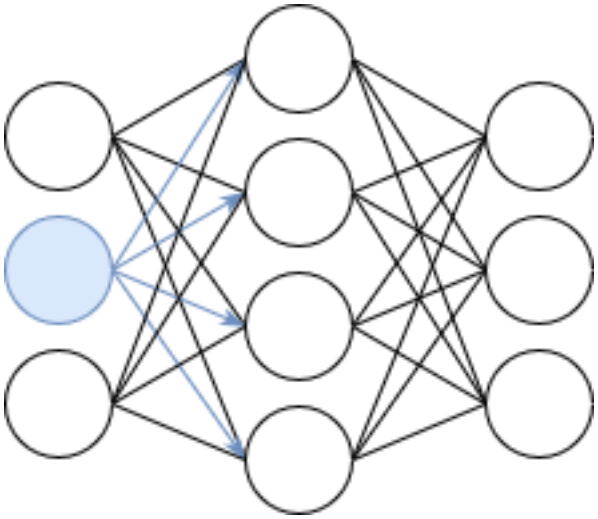
Iz navedenog razloga umjetni neuron sastoji se od još jednog dijela, a to je aktivacijska funkcija, nelinearna funkcija koja kao argument prima broj nastao ranije spomenutom težinskom sumom ulaza s pomakom, a kao izlaz daje također neku numeričku vrijednost, najčešće na praktičan način ograničenu, na primjer, između vrijednosti 0 i 1 ili -1 i 1. Zajedno, težine, pomak i aktivacijska funkcija definiraju jedan neuron.

Pojedinost zbog koje nam je umjetni neuron zanimljiv za potrebe strojnog učenja je ta da neurone možemo udružiti kako bismo stvorili model s još većom izražajnom moći od jednog umjetnog neurona, odnosno, možemo ih povezati u umjetne neuronske mreže.

3.3.2. Umjetne neuronske mreže

Umjetne neuronske mreže izgrađene su spajanjem izlaza jednih neurona na ulaze drugih neurona. Rezultirajuća struktura izgleda ne toliko različito prirodnim neuronskim mrežama koje sačinjavaju naš mozak i također ima sposobnost nečega što bismo mogli nazvati učenjem, čemu smo se i nadali. Sama za sebe je i dalje samo skup jednih ili drugih matematičkih funkcija ali ima potencijal biti puno više od samog zbrajanja i množenja brojeva.

Strukturu umjetne neuronske mreže možemo gledati kao prethodno opisan spoj pojedinačnih neurona u veliku mrežu, ali najčešće ćemo stvarati mreže koje se sastoje od organiziranih skupova neurona - slojeva. Slojevita struktura umjetne neuronske mreže postiže se organiziranim povezivanjem neurona kao na slici 3.3, i ako su izlazi jednog sloja neurona povezani samo s ulazima sljedećeg sloja neurona i tako dalje, to nazivamo unaprijednom (engl. *feedforward*) neuronskom mrežom. Ako su ulazi svih neurona spojeni na izlaze svih neurona prethodnog sloja, odnosno izlazi spojeni na ulaze svih neurona sljedećeg sloja, to zovemo potpuno povezanom umjetnom neuronskom mrežom (engl. *fully connected artificial neural network*, FCANN). Takva struktura korištena je u sklopu ovog diplomskog rada kako bi se ostvarilo automatsko upravljanje modelom letjelice u simulatoru leta, i sastoji se od ulaznog sloja od 3 neurona, jednog skrivenog sloja s 10 , i izlaznog sloja s 4 neurona. Skriveni slojevi su slojevi između ulaznog i izlaznog sloja a imaju takav naziv jer u normalnim okolnostima ne možemo vidjeti što se događa u njima, već samo koje podatke dajemo mreži kao ulaz, i koje nam mreža daje kao izlaz.



Slika 3.3: Slojevita struktura umjetne neuronske mreže - kružnicama su predstavljeni neuroni, a linijama, to jest vezama, ulazi i izlazi u neurone

Postoje naprednije strukture koje se mogu izgraditi na osnovu slojevitog modela, kao što su konvolucijske mreže (engl. *convolutional neural network*, CNN) i povratne mreže (engl. *recurrent neural network*). Takvi modeli ostvaruju vrhunske rezultate u područjima kao što su računalni vid i obrada prirodnog jezika, no sposobnosti i svojstva po kojima se ti modeli ističu u tim područjima nisu primjenjiva i prikladna na problemu upravljanja letjelicom u simuliranom 3D prostoru.

Slojeve umjetne neuronske mreže moguće je predstaviti kao vektore, u smislu da se računanje izlaza neurona vrši kao skalarni produkt ulaza neurona s njegovim težinama, zbrojen s pomakom neurona. Takva reprezentacija vrlo je korisna za notaciju i jednostavnost implementacije unaprijednog prolaza kroz mrežu i u ovom diplomskom radu slojevi će biti prikazani na taj način. Također, prvi sloj mreže naziva se ulazni sloj i nije sastavljen od neurona nego od ulaza u mrežu. Ulazni sloj iz tog razloga nema nikakve parametre koji se mogu mijenjati - on je naprosto vektor sastavljen od konstantnih vrijednosti.

3.3.3. Propagacija pogreške unatrag

Uz odlučenu funkciju gubitka i optimizacijski postupak - stohastički gradijentni spust, ostaje još samo jedno pitanje oko treniranja umjetne neuronske mreže. Algoritam stohastičkog gradijentnog spusta nalaže da se u svakoj iteraciji parametri modela inkrementalno pomiču u smjeru suprotnom od gradijenta funkcije gubitka po tom parametru, no kako izračunati derivaciju, odnosno gradijent funkcije gubitka po parametrima, što su u slučaju umjetne neuronske mreže zapravo težine i pomaci mreže? Odgovor na

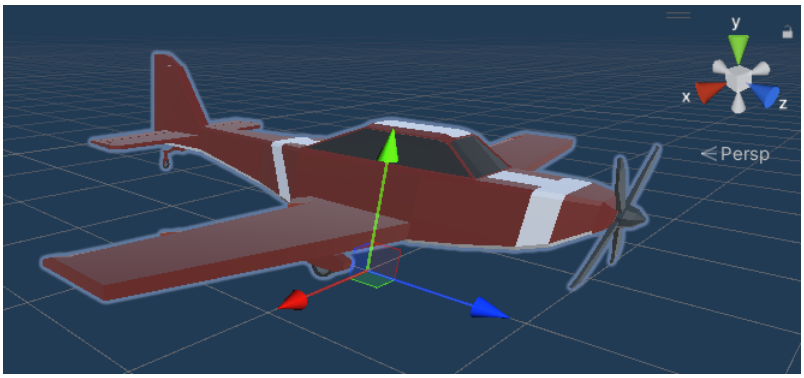
to pitanje dolazi u obliku postupka koji se naziva propagacija pogreške unatrag (engl. *backpropagation*, *backprop*), postupka čiju povijest, iako pod drukčijom nomenklaturom, možemo pratiti čak i do ranih 1960-ih (Kelley, 1960).

Postupak propagacije pogreške unatrag počinje računanjem funkcije pogreške i računajući derivaciju odnosno gradijent funkcije pogreške po izlazu izlaznog sloja, to jest zadnjeg sloja mreže. Iz te vrijednosti možemo korištenjem pravila ulančavanja izračunati gradijent funkcije pogreške po parametrima, to jest, težinama zadnjeg sloja, čime dobijemo u kojoj mjeri sve specifične težine tog sloja utječu na funkciju gubitka. Ti gradijenti koristit će se za potrebe optimizacijskog algoritma kako bi se podesilo težine i pomak izlaznog sloja.

4. Simulator leta

4.1. Osnovne pretpostavke

Simulator leta razvijen u sklopu ovog diplomskog rada sastoji se od trodimenzijskog modela sportskog zrakoplova i niza nepomičnih lebdećih prstenova postavljenih na određene lokacije s ciljem da zrakoplov prođe kroz njih. Sustav je znatno jednostavniji od uvjeta stvarnoga života, s primjetnom razlikom da u prostoru ne postoji ništa osim zrakoplova i prstenova, dakle, nema nikakvog tla, zgrada, drveća ni planina. Također, ne postoji nikakva vrsta akceleracije u sustavu, već se zrakoplov kreće konstantnom brzinom unaprijed. Ulazima korisnika ili automatskog upravljanja, ovisno o vrsti simulacije koja se pokreće, zrakoplov se može okretati oko sve tri osi lokalnog koordinatnog sustava zrakoplova, to jest mijenjati nagib (engl. *pitch*, rotacija oko x-osi), repno skretati (engl. *yaw*, rotacija oko y-osi) i kotrljati se (engl. *roll*, rotacija oko z-osi). Ovdje je bitno spomenuti orijentacije koordinatnih osi jer postoji više različitih interpretacija trodimenzijskog koordinatnog sustava. Prateći konvencije pogonskog sustava Unity, koristi se lijevi koordinatni sustav, s x-osi kao horizontalnom osi i y-osi kao vertikalnom osi te z-osi protegnutom okomito na x-os u smjeru naprijed. Kraće rečeno, također prateći konvencije, orijentacije koordinatnih osi možemo prevesti kao *desno*, *gore* i *naprijed* redom za osi x, y i z.



Slika 4.1: Zrakoplov i orijentacije koordinatnih osi: x (crvena), y (zelena) i z (plava)

4.2. Korištene strukture podataka

Pozicije zrakoplova i prstenova u predstavljene su kao trodimenzionalni vektori, a njihove orijentacije u prostoru kao kvaternioni.

4.2.1. Kvaternioni

Kvaternion je matematički konstrukt često korišten u računalnoj grafici za zapisivanje rotacije zbog njegove sposobnosti da predstavlja rotaciju u trodimenzijskom prostoru bez slabosti pod engleskim imenom *gimbal lock*, fenomena koji se očituje usred specifičnih rotacija gdje usred poklapanja osi okretanja dolazi do gubitka jednog stupnja slobode, čineći rotaciju oko jedne od osi trenutačno nemogućom dok se problem ne otkloni.

Kvaternion je sačinjen od jedne realne komponente i tri imaginarne komponente. Standardni način zapisa kvaterniona je:

$$w + x_i + y_j + z_k \tag{4.1}$$

Način na koji kvaternioni rade je izvan područja ovog diplomskog rada, samo nam je važno što možemo postići koristeći taj konstrukt. Dva kvaterniona možemo međusobno pomnožiti čime se njihove rotacije zbrajaju. Također je moguće rotirati vektor pomoću kvaterniona, što ćemo u nastavku rada nazivati množenjem vektora kvaternionom. Ta operacija složenija je od običnog množenja i odvija se u dva koraka - množenje kvaternionom i zatim množenje inverznim kvaternionom, no budući da je i to izvan područja ovog diplomskog rada, i zbog činjenice da je ta operacija u pogonskom sustavu Unity predstavljena kao nadjačana operacija množenja između kvaterniona i vektora, dopustit ćemo takvo pojednostavljenje.

Dakle, rezultat "množenja" kvaterniona i vektora daje nam drugi vektor koji nastaje rotacijom prvotnog vektora u smjeru koji reprezentira kvaternion. Izvodi operacija s kvaternionima u kontekstu specifičnom za ovaj rad bit će navedeni kasnije, kad ih budemo trebali derivirati, a preuzeti su iz (Chudá, Hana, 2019) i izvornog koda pogonskog sustava Unity. Zadnja stvar koju valja napomenuti oko kvaterniona je ta da se u svoj nomenklaturi pogonskog sustava Unity koristi isključivo riječ *rotacija*, iako riječi *rotacija* i *orijentacija* nemaju isto značenje. Međutim, to ima više smisla nego na prvi pogled ako razmotrimo da kvaternioni doista opisuju samo rotaciju nekog tijela, a orijentacija se dobije primjenom te rotacije nad nekim vektorom. U tekstu ovog rada spominju se i rotacija i orijentacija, ali u izvornom kodu programskog dijela ovog rada

koristit će se isključivo riječ *rotation*.

4.2.2. Eulerovi kutovi

Jedan drukčiji način za numerički predstaviti rotaciju u trodimenzijskom prostoru su Eulerovi kutovi rotacije - tri vrijednosti koje predstavljaju kutove rotacije nekog tijela oko tri osi, odrađene jedna za drugom. Postoji više različitih konvencija koje su to tri osi, i kojim redom, ali pogonski sustav Unity koristi rotacije redom oko z-osi, pa x-osi, i na kraju oko y-osi. Takav način zapisivanja rotacije koristi se na nekoliko mjesta u ovom radu. Ulazi u simulator pomnoženi konstantama brzine rotacije zrakoplova predstavljaju Eulerove kutove rotacije zrakoplova u tome okviru. Također, podatci o orijentaciji prstenova u poligonima spremaju se u obliku Eulerovih kutova.

4.2.3. Predstavljanje stanja simulacije modelu

Kako bi model strojnog učenja mogao upravljati simulacijom leta potrebno je odrediti na koji način model vidi simulaciju, to jest, odrediti najvažnije informacije za koje smatramo da predstavljaju sve što je potrebno za donošenje odluka o upravljanju zrakoplovom. Jako zahtjevan pristup bio bi kao ulaz modelu davati slike ekrana na kojoj se odvija simulacija leta, upravo kao što bismo i ljudima predstavili simulator leta. Tada bi model imao dvije zadaće. Jedna bi bila prepoznati što se nalazi na slici, kontekstualizirati to, i po potrebi pamtiti određene podatke koji se ne daju iščitati s ekrana u svakom trenutku. Druga bi zadaća tek bila na temelju podataka o stanju simulacije leta donijeti odluku o sljedećim ulazima, to jest, komandama, koje unijeti u simulator leta kako bi se najbolje izvršilo zadatak prolaženja kroz sve prstenove. Ovaj pristup vrlo je komplicirani opsegom podosta širi od nečega prikladnog za diplomski rad, budući da, iz određenog gledišta, uključuje opis, dizajniranje i treniranje dvaju različitih modela - jednog modela računalnog vida koji bi prepoznavao i kontekstualizirao stanje simulacije leta iz danih slika, i jednog modela koji bi odlučivao o ulazima u simulator leta. Također, trenirati model nad takvim podatcima bilo bi resursno skupo i sporo.

Druga je moguća ideja preskočiti prepoznavanje stanja simulatora iz slike i umjesto toga postaviti virtualne senzore na tijelo zrakoplova, koji bi u stvarnom vremenu u simulaciji prenosili što taj zrakoplov može vidjeti. Što se tiče performansi i resursa, ovaj pristup bio bi znatno brži i jeftiniji od prethodnog, no ne bez svojih mana. Na primjer, postoji mogućnost da se model izgubi u prostoru u situacijama gdje sljedeći cilj, to jest, prsten, nije vidljiv u sceni pred zrakoplovom, ili možda da prstene kroz koje je

već prošao ne pamti i pokušava uvijek ponovno proći kroz njih. Također postoji problem kako predstaviti vidno polje modelu - bi li bilo najbolje koristiti skupi pristup ispaljivanja i praćenja zraka u prostoru? Što ako se neki daleki prsten nađe u području vidnog polja koje nije pokriveno nijednom ispaljenom zrakom, to jest, u području između zraka? Je li možda onda efikasnije ne naslijepo ispaljivati zrake nego ispitivati scenu nalazi li se neki objekt u smjeru u kojem zrakoplov otprilike ide? Na koji bi način to bilo onda implementirano? Također, ako nekako taj pristup djeluje u detektiranju prstenova korištenjem virtualnih senzora postavljenih na tijelu zrakoplova, kako možemo osigurati da zapamtimo kroz koje smo prstene prošli? Moguće je da bismo onda modelu ili trebali dati način da računa aproksimativnu lokaciju zrakoplova u svakom trenu i bilježiti lokacije svih prstenova kroz koje je zrakoplov prošao, ili u svakom okviru modelu javljati gdje je zrakoplov. U tom slučaju, bi li model bio zadužen za pamćenje lokacije prođenih prstenova, i što s prstenovima koji su primijećeni ali nisu trenutačni cilj kroz koji zrakoplov treba proći - bismo li trebali implementirati neki način da ih model "zapamti" i posjeti poslije?

Treći pristup bio bi da modelu na ulaz damo pozicije zrakoplova i najbližeg prstena, kao i orijentaciju zrakoplova u prostoru, i naučimo model kako interpretirati te podatke i kako formirati izlaze koji bi, korišteni kao ulaz u simulator leta, to jest, kao virtualne osi igraćeg kontrolera, upravljali zrakoplovom na najefikasniji način kroz prstenove. Model izrađen u sklopu ovog diplomskog rada ima upravo takve ulaze, točnije, prima tri realne vrijednosti koje predstavljaju poziciju zrakoplova u trodimenzijskom koordinatnom sustavu, kao i tri realne vrijednosti koje predstavljaju poziciju najbližeg prstena te tri realne vrijednosti koje predstavljaju orijentaciju zrakoplova u Eulerovim kutovima. Orijentacija se u računalnoj grafici inače bilježi korištenjem kvaterniona, no, kao što je prije spomenuto, za potrebe ulaza u model strojnog učenja odlučio sam sve orijentacije iz kvaterniona prevesti u Eulerove kutove, s ponovnim podsjetnikom da je u pogonskom sustavu Unity redoslijed rotacija Eulerovih kutova prvo oko z-osi, pa oko x-osi, i na kraju y-osi, gdje su orijentacije x, y i z-osi redom *desno*, *gore* i *naprijed*.

Dakle, prvotni ulaz u model sastojao se od devet realnih brojeva, a izlaz su tri realna broja koji predstavljaju intenzitet željene rotacije zrakoplova oko svake od tri osi u sljedećem okviru simulacije. Međutim, taj se ulaz ispostavio pretjerano kompliciranim jer model nije ništa naučio za vrijeme korištenja takvih ulaza.

Sljedeća ideja bila je modelu na ulaz dati vektor razlike pozicija zrakoplova i sljedećeg prstena, vektor brzine i orijentaciju zrakoplova. Broj ulaza bio bi isti, no odnos između zrakoplova i prstena jednostavniji je i predvidiv korištenjem vektora brzine. Ni

ta ideja nažalost nije davala zadovoljavajuće rezultate.

Zadnja ideja bila je potpuno pojednostavniti informacije koje se daju modelu na način da kao ulaze model prima samo koordinate sljedećeg prstena u koordinatnom sustavu zrakoplova, to jest, samo tri realna broja, umjesto dosadašnjih devet. U te tri vrijednosti implicitno je sadržana pozicija zrakoplova (uvijek je u ishodištu), udaljenost između zrakoplova i prstena - koja je jednaka poziciji prstena u ovom slučaju te rotacija zrakoplova u smislu da se koordinate prstena rotiraju oko ishodišta ako se zrakoplov rotira.

Ulazi formirani na taj način predstavljaju subjektivni pogled na sljedeći prsten iz pogleda zrakoplova, što je dobro, jer ulazi u simulator leta upravljaju simulacijom iz pogleda zrakoplova, to jest okreću zrakoplov oko njegovih vlastitih osi, a izlazi mreže se upravo koriste kao ulazi u simulator leta. Teoretski, kad simulator s ovakvim ulazima vidi prsten ispred sebe nadesno, kako god je orijentiran u globalnom koordinatnom sustavu, trebao bi idealno (repno) skrenuti udesno, to jest ili oko vlastite y-osi ili oko vlastite z-osi pa potom x-osi (zakotrljati se i potom nagnuti za gore - za veće kutove ta rotacija je brža zbog fizike zrakoplova)

4.3. Pristup učenju

Kad je u pitanju vrsta strojnog učenja koju želimo upotrijebiti na zadatku vezanom za ovaj rad, iz čiste prirode danog zadatka, za problem ostvarivanja automatskog upravljanja simulatora leta gotovo odmah nam dolazi do izražaja podržani pristup strojnom učenju, iz nekoliko razloga.

S jedne strane, naučiti algoritam strojnog učenja da upravlja zrakoplovom praktički je nemoguće bez ikakvih povratnih informacija o tome koliko je model uspješan, stoga se nenadzirano učenje čini kao najlošiji pristup zadatku.

S druge strane, zadatak nema jedno jedinstveno ispravno rješenje, zbog čega se nadzirano učenje inicijalno ne čini prikladno. Međutim, moguće je modelu problem upravljanja zrakoplovom predstaviti kroz skup podataka za učenje koji se sastoji od zapisanih podataka pokretanja simulacije leta kojom je upravljala iskusna ljudska osoba. Na taj način modelu bi na raspolaganju bili razni primjeri za učenje koji se sastoje od ulaza, to jest informacija o simulaciji za svaki okvir, i izlaza, to jest naredbi koje je ljudski pilot odlučio unijeti. Pretpostavivši da ljudski pilot zna što radi, cilj modela bio bi što bliže imitirati način na koji ta osoba upravlja zrakoplovom. Međutim, taj pristup ima nekoliko problema.

Model možemo naučiti upravljati poput ljudske osobe ali nemoguće je pouzdati

se u bilo koju ljudsku osobu da će simulacijom upravljati optimalno, već s vlastitim greškama i pristranostima. Čak i ako pretpostavimo da ljudski pilot može upravljati simulatorom na optimalan način, postoji visoka vjerojatnost da model iz podataka za učenje jednostavno ne može naučiti generalizirati upravljanje zrakoplovom, i da na neviđenim poligonima neće imati dobre rezultate. Kako bi se to spriječilo, moguće je pokušati trenirati model na jako velikom skupu podataka s puno različitih situacija i nadati se da će model naučiti adekvatno generalizirati razne moguće situacije koje se mogu dogoditi pri simulaciji leta.

Međutim, veliki skupovi podataka zauzimaju puno memorijskog prostora i, važnije, treba puno ljudskog vremena da ih se prikupi. Model može, okvir po okvir, upravljati simulacijom daleko brže od normalne brzine simulacije, što znači da kad nemamo potrebu vizualno evaluirati performanse modela, možemo simulaciju kojom upravlja model pokretati koliko god brzo naše sklopovlje dozvoljava. S druge strane, to je daleko brže nego što bi ikakva ljudska osoba mogla procesirati podatke potrebne za upravljati takvom simulacijom, čineći proces pribavljanja skupa podataka mnogo puta sporijom nego u idealnom slučaju.

Ideja puštanja modela da kontrolira simulaciju bez prethodnog znanja, i nagrađivanje modela za poželjne akcije i kažnjavanje za nepoželjne čini se kao pristup koji najviše odgovara prilici.

4.4. Ulaz modela i funkcija gubitka

Nakon izbora vrste strojnog učenja i pristupa za reprezentaciju stanja simulacije leta modelu strojnog učenja, potrebno je osmisliti kako će se podatci o letu prikazati modelu u obliku primjera za učenje. Budući da, kao što je ranije rečeno, nema jednog najtočnijeg načina za upravljanje zrakoplovom kroz poligon prstenova, model ne možemo podešavati u odnosu na odstupanje predikcije modela od ispravne oznake. Umjesto toga, u ovom radu koristi se podržani pristup učenju bez nagrada za poželjne odluke modela, već s jednom funkcijom gubitka čiji se iznos može izračunati u svakom trenutku simulacije, a čije se smanjenje može smatrati nagradom i glavni cilj modela je što više smanjiti iznos funkcije gubitka. Funkciju gubitka možemo bazirati na određenim metrikama relevantnim za donošenje odluka u simulatoru leta, na primjer, koliko je zrakoplov udaljen od sljedećeg prstena, koliki je kut razlike između smjera u kojem zrakoplov gleda i smjera u kojem bi gledao direktno u prsten, ili broj trenutačno neprođenih prstena. Postoji, doduše, par važnih stvari za držati na umu pri dizajniranju funkcije gubitka.

Prva je da bi funkcija gubitka trebala ovisiti o izlazima modela. Izlazi modela su tri realna broja u rasponu između -1 i 1 koji određuju intenzitet rotacije oko svake osi. Zapravo, određuju vrijednosti unosa u virtualne osi igraćeg kontrolera koji Unity koristi za obradu unosa korisnika, među ostalim i za vrijeme simulacije leta s ljudskim ulazima.

Na primjer, funkcija

$$||planePos - ringPos||$$

, to jest, norma vektora nastalog oduzimanjem lokacija zrakoplova i prstena, daje udaljenost zrakoplova od prstena ali kako ne ovisi o izlazima mreže, to jest ulazima u simulaciju, nemoguće je dobiti gradijent funkcije gubitka po izlazima mreže. Umjesto takve funkcije gubitka, mogli bismo izračunati novu poziciju zrakoplova, to jest poziciju u sljedećem okviru simulacije koja ovisi o trenutačnim izlazima modela, i koristiti tu poziciju u navedenoj formuli. Na taj način imamo definirani gradijent funkcije gubitka po izlazima modela.

Druga stvar za držati na umu je ta da bi vrijednost funkcije gubitka trebala biti informativna, to jest, da je njena derivacija u svakoj točki različita od 0, kako bi se moglo znati u kojem smjeru podesiti težine kako bi se postigao bolji učinak. Na primjer, funkcija

$$max(0, ||nextPos - ringPos|| - ||planePos - ringPos||)$$

bilježi promjenu u udaljenosti između zrakoplova i prstena između trenutnog i sljedećeg okvira u simulaciji leta. Ako primjenom izlaza modela kao ulaz u simulaciju udaljenost zrakoplova od prstena sljedeći okvir bude veća nego u trenutnom okviru, funkcija gubitka pozitivna je i jednaka razlici tih udaljenosti, a u suprotnom je slučaju funkcija gubitka 0. Isprve ima smisla ne kažnjavati model ako se zrakoplov pomiče bliže prstenu nego što je prije bio, ali u tom se slučaju pojavljuje problem da čim se zrakoplov na bilo koji način počne približavati prstenu, gradijent funkcije gubitka postane 0, i model ne dobiva nikakvu daljnju informaciju kako poboljšati putanju zrakoplova. To može dovesti do toga da model jednostavno, gotovo "lijeno", nauči kružiti oko prstena na konstantnoj udaljenosti.

Još jedan problem može se pojaviti ako za funkciju gubitka uzmemo, na primjer,

$$||nextPos - ringPos||$$

- udaljenost između zrakoplova i sljedećeg prstena, ili čak kvadratnu udaljenost između njih. Funkcija gubitka smanjuje se što se zrakoplov više približava prstenu, no

čim prođe kroz njega, funkcija gubitka smjesta poraste, gotovo uvijek za veći iznos nego da zrakoplov jedva zaobiđe prsten i nastavi kružiti oko njega. Iz tog razloga udaljenost između zrakoplova i prstena nije moguće koristiti samu za sebe kao funkciju gubitka, no mogli bismo dobiti bolje rezultate zbrojivši ju s nekim drugim funkcijama, na primjer onom koja jednostavno kažnjava model u ovisnosti o koliko je još prstena preostalo kroz koje zrakoplov nije prošao, ali bi još trebao.

Dakle, koristeći funkciju kao što je

$$||nextPos - ringPos|| + remainingRings$$

gdje je remainingRings broj prstenova kroz koje zrakoplov još nije prošao a trebao bi, dobivamo svojstvo funkcije gubitka da se pri prolazu kroz bilo koji prsten njen iznos nužno smanji, čineći prolazak kroz prsten najboljom opcijom za smanjiti iznos funkcije gubitka.

4.4.1. Regularizacija

Još jedna stvar koju bi bilo korisno spomenuti u vezi s funkcijom gubitka je regularizacija. Iz perspektive modela, ovaj zadatak, kao i mnoge druge zadatke u strojnom učenju, moguće je izvršiti na više načina. Kroz poligon prstenova moguće je proći direktno, matematički najkraćim mogućim putem, ili zaobilazno, uzimajući široke zaokrete. Osim činjenice da je moguće postaviti funkciju gubitka koja uzrokuje da je modelu puno "skuplje", to jest lošije poligon prolaziti indirektnim putevima, postoji način da dodatno poguramo model da preferira najkraći mogući put, pod nazivom regularizacija. Regularizacija se inače u strojnom učenju odnosi na tehniku korištenu radi sprječavanja prenaučenosti kroz metaforičko "zatezanje" parametara ka vrijednostima bližim nuli. To se može postići na nekoliko načina, a jedan od njih je u funkciju gubitka dodati komponentu koja nekako ovisi o parametrima modela, a najčešće je to norma vektora parametara pomnožena određenom konstantom koju zovemo regularizacijski faktor i često označavamo s λ.

No, s neuronskom mrežom je malo kompliciranije jer ima puno težina i pomaka, a ako imamo varijabilni broj slojeva s varijabilnim brojem neurona po sloju, ne možemo na lagan način samo odrediti neku mjeru po kojoj kazniti norme težina. Iz tog razloga je u sklopu ovog diplomskog rada testirana tehnika slična regularizaciji, koja u nekoj mjeri kažnjava normu izlaza mreže umjesto normu vektora težina, u nadi da će to natjerati model da ako nije potrebno raditi da ne radi, na primjer, oštra skretanja ili trzave manevre nalik klackalici koji samo rezultiraju kretanjem unaprijed. Isprobane su dvije varijante ove tehnike - kažnjavanje proporcionalno normi vektora izlaza

i kažnjavanje proporcionalno zasebnim apsolutnim vrijednostima svakog izlaza, gdje nisu svi izlazi jednako kažnjavani. Drugi pristup bio je pokušaj da se ne ograniči funkcionalne sposobnosti pokreta zrakoplova, a da se uz dovoljno sreće smanji količina kotrljanja koje je model izvodio u slobodno vrijeme. Nažalost, oba pristupa samo su rezultirala modelima koji su imali lošiji uspjeh po svim mjerama i ideja kažnjavanja modela proporcionalno njegovom izlazu je napuštena.

4.5. Izvod funkcije gubitka

Funkcija gubitka upotrijebljena u sklopu ovog diplomskog rada u nekoj mjeri kombinira ideje većine predstavljenih funkcija gubitka.

$$L = distDiff + rotDiffFactor * rotDiff + remainingRings \tag{4.2}$$

Ideja iza ove specifične funkcije gubitka je ta da se model kažnjava ako se iz okvira u okvir udaljava od sljedećeg prstena, ali također i ako se iz okvira u okvir okreće "dalje" od prstena nego što je prije bio. Varijabla distDiff ovdje predstavlja razliku između udaljenosti zrakoplova od prstena između dva susjedna okvira simulacije. Varijabla rotDiff predstavlja najmanji prostorni kut između sljedeće predviđene orijentacije zrakoplova i orijentacije pravca koji je spojnica između zrakoplova i prstena. Također, model se kažnjava ovisno o tome koliko još prstenova nije prošao. Motivacija za tu kaznu je ta da motiviramo model da prolazi kroz što više prstenova umjesto da prolazi vrlo blizu njima kako bi izbjegao nagli porast funkcije gubitka pri prolaženju kroz jedan.

4.5.1. funkcija distDiff

Izvodi funkcija koje sačinjavaju konačnu funkciju gubitka korištenu u radu navedeni su u nastavku. Prvo ćemo pobliže pogledati distDiff i funkcije od koje je ta komponenta funkcije gubitka sastavljena, a funkcijom rotDiff pozabavit ćemo se kasnije.

$$\begin{aligned} L &= distDiff + rotDiffFactor * rotDiff + remainingRings \\ distDiff &= ||nextPos - ringPos|| - ||planePos - ringPos|| \end{aligned} \tag{4.3}$$

Ovdje je ringPos pozicija sljedećeg prstena, planePos trenutna pozicija zrakoplova, a nextPos pozicija zrakoplova u sljedećem okviru animacije izračunata pomoću izlaza

modela: x, y i z. Varijabla thrust četvrti je izlaz modela, a sve ostalo osim funkcije nextPos smatra se konstantom. nextPos je funkcija čija se vrijednost računa izračunom funkcije nextRot:

$$\begin{gathered} nextPos = newForward \cdot velocity \cdot (1 + (thrust + 0.5)) \\ newForward = nextRot \cdot \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \end{gathered} \tag{4.4}$$

Pozicija zrakoplova u sljedećem okviru računa se skaliranjem jediničnog vektora unaprijed brojem koji se računa preko konstante velocity koja određuje normalnu brzinu zrakoplova i varijablom thrust koja, kao prethodno spomenuti četvrti ulaz u simulator kontrolira trenutnu brzinu zrakoplova. NextRot je funkcija koja se računa pomoću ostala tri ulaza u simulator, to jest, izlaza modela.

Rotacija vektora postiže se množenjem trodimenzijskog vektora kvaternionom koji predstavlja rotaciju u sljedećem okviru simulacije, a budući da za potrebe nextPos kvaternionom rotacije množimo uvijek jedinični vektor $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ taj izračun možemo pojednostaviti kao:

$$\begin{gathered} (w + x_i + y_j + z_k) \cdot \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} = \\ \begin{bmatrix} 2(x_i z_k + w y_j) & 2(y_j z_k - w x_i) & 1 - 2x_i^2 - 2y_j^2) \end{bmatrix} \end{gathered} \tag{4.5}$$

Slijedi način izračuna funkcije nextRot koja je potrebna za funkciju nextPos:

$$\begin{gathered} nextRot = \\ planeRot \cdot QE(x \cdot pitchSpeed, y \cdot yawSpeed, -z \cdot rollSpeed) \end{gathered} \tag{4.6}$$

planeRot je trenutna orijentacija zrakoplova u prostoru koju, kako bismo dobili orijentaciju u sljedećem okviru simulacije, rotiramo u ovisnosti o izlazima modela. To se ostvaruje množenjem kvaterniona koji predstavlja trenutnu orijentaciju kvaternionom koji predstavlja rotaciju dobivenu obradom ulaza u simulator. Umnožak dva kvaterniona (dolje: p i q) definiran je kao:

$$\begin{gathered} p \cdot q = (p_1 + p_2 i + p_3 j + p_4 k) \otimes (q_1 + q_2 i + q_3 j + q_4 k) = \\ (p_1 q_1 - p_2 q_2 - p_3 q_3 - p_4 q_4) + \\ i(p_1 q_2 + p_2 q_1 + p_3 q_4 - p_4 q_3) + \\ j(p_1 q_3 + p_3 q_1 + p_4 q_2 - p_2 q_4) + \\ k(p_1 q_4 + p_4 q_1 + p_2 q_3 - p_3 q_2) \end{gathered} \tag{4.7}$$

Promjena rotacije između dva susjedna okvira simulacije koja množi trenutnu orijentaciju zrakoplova u funkciji nextRot (izvod 4.6) dobiva se koristeći ulaze u simulator leta pomnožene konstantama brzina okretanja zrakoplova oko zasebnih osi kao

Eulerove kutove rotacije. Ovdje se može primijetiti da se za rotaciju oko z-osi koristi suprotna vrijednost ulazu z radi kompenzacije za činjenicu da se u Unityu koristi lijevi koordinatni sustav - sa suprotnom orijentacijom z-osi nego konvencionalno korišteni desni koordinatni sustav. Pretvorba Eulerovih kutova rotacije u kvaternion u narednim izvodima predstavljena je kao $QE(x, y, z)$ i za kasnije potrebe važno nam je znati točno kako se ta pretvorba odvija. Navedeni izvod u nastavku sadrži određene supstitucije radi sažetosti i razumljivosti:

$$\begin{aligned} xInput &= 0.5 \cdot x \cdot pitchSpeed \cdot \tfrac{\pi}{180} \\ yInput &= 0.5 \cdot y \cdot yawSpeed \cdot \tfrac{\pi}{180} \\ zInput &= -0.5 \cdot z \cdot rollSpeed \cdot \tfrac{\pi}{180} \end{aligned} \tag{4.8}$$

Definirane varijable koristit će se kao argumenti trigonometrijskih funkcija koje kao ulaz primaju vrijednosti polovina kutova u radijanima, zbog čega se skalirani ulazi koji predstavljaju rotaciju između dva okvira u stupnjevima množe sa dvije konstante. Zapis ove pretvorbe je dugačak i zato uvodimo dodatne pokrate:

$$\begin{aligned} sx = \sin(\tfrac{xInput}{2}) \quad sy = \sin(\tfrac{yInput}{2}) \quad sz = \sin(\tfrac{zInput}{2}) \\ cx = \cos(\tfrac{xInput}{2}) \quad cy = \cos(\tfrac{yInput}{2}) \quad cz = \cos(\tfrac{zInput}{2}) \end{aligned} \tag{4.9}$$

Ostatak postupka[1] glasi:

$$\begin{aligned} &QE(xInput, yInput, zInput) = \\ &(cx \cdot cy \cdot cz + sx \cdot sy \cdot sz) + \\ &(sx \cdot cy \cdot cz + cx \cdot sy \cdot sz)i + \\ &(cx \cdot sy \cdot cz - sx \cdot cy \cdot sz)j + \\ &(cx \cdot cy \cdot sz - sx \cdot sy \cdot cz)k \end{aligned} \tag{4.10}$$

4.5.2. funkcija rotDiff

Kao što je i prije spomenuto, rotDiff predstavlja najmanji prostorni kut između sljedeće predviđene orijentacije zrakoplova i orijentacije spojnice između zrakoplova i prstena, podijeljen s 360 kako bi vrijednost funkcije rotDiff uvijek bila između 0 i 1. Dodatno, rotDiff je u funkciji gubitka (izvod 4.2) skaliran konstantom rotDiffFactor zadane vrijednosti 0.3 kako bi važnost iznosa funkcije rotDiff unutar funkcije gubitka uvijek bila manja.

[1] Način izrade formule za pretvaranje Eulerovih kutova u bilo kojem redoslijedu u kvaternione preuzet s https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles

$$rotDiff = \angle(newForward, ringPos - planePos)/360 \tag{4.11}$$

Kut između dva vektora u 3D prostoru se u pogonskom sustavu Unity računa kao:

$$\angle(v1, v2) = \arccos(Clamp\left(\frac{v1 \cdot v2}{||v1||^2 \cdot ||v2||^2}\right), -1, 1)$$

$$Clamp(x, minValue, maxValue) = max(minValue, min(maxValue, x)) \tag{4.12}$$

Varijabla newForward iz ranijeg izvoda (4.4) ponovno se koristi i zato ju nije potrebno ponovno računati pri računanju vrijednosti rotDiff.

4.6. Propagacija pogreške unatrag

Nakon računanja izlaza modela unaprijednim prolazom kroz neuronsku mrežu i računanja vrijednosti funkcije gubitka za trenutačni okvir, potrebno je izračunati gradijente funkcije gubitka po svim parametrima modela. U specifičnom slučaju modela i funkcije gubitka iz ovog rada, gradijent funkcije gubitka po izlazima izlaznog sloja počinjemo računati deriviranjem funkcije gubitka po dvije komponente funkcije gubitka - razlici udaljenosti zrakoplova od prstena između dva susjedna okvira, distDiff, i najmanjem prostornom kutu između predviđene orijentacije zrakoplova i spojnice zrakoplova i prstena, rotDiff. Svaki od ta dva gradijenta pomnožit ćemo gradijentom funkcije kojoj pripada po komponentama te funkcije koje ovise o izlazima modela, pa zatim gradijentom tih komponenta po njihovim dijelovima koji ovise o izlazima modela, sve dok ne dođemo do same vrijednosti izlaza modela, kako bismo korištenjem pravila ulančavanja dobili gradijent funkcije gubitka po izlazu izlaznog sloja.

Budući da je funkcija gubitka podijeljena na dva dijela, prvo ćemo pogledati distDiff i gradijente njegovi komponenti, a onda rotDiff i gradijente njegovih komponenti. Dakle, gradijenti funkcije gubitka po komponentama funkcije distDiff računaju se ovako:

$$\frac{\partial distDiff}{\partial nextPos} = \frac{1}{||nextPos - ringPos||} \cdot (nextPos - ringPos)$$

$$\frac{\partial nextPos}{\partial newForward} = velocity \cdot (1 + (thrust + 0.5))$$

$$\frac{\partial nextPos}{\partial thrust} = newForward \cdot velocity \tag{4.13}$$

$$\frac{\partial L}{\partial thrust} = \frac{\partial L}{\partial distDiff} \cdot \frac{\partial distDiff}{\partial nextPos} \cdot \frac{\partial nextPos}{\partial thrust}$$

Nakon ovog koraka gradijent po izlazu thrust odvaja se od gradijenta po ostalim izlazima jer je on izravno određen, a za dobiti gradijente funkcije gubitka potrebno je dalje računati gradijente po funkcijama koje ovise o njima.

Gradijent varijable newForward po vrijednosti funkcije nextRot računa se kao gradijent množenja kvaterniona i vektora (izvod 4.5) :

$$\frac{\partial newForward}{\partial nextRot} = \begin{bmatrix} 2y & 2z & 2w & 2x \\ -2x & -2w & 2z & 2y \\ 0 & -4x & -4y & 0 \end{bmatrix} \tag{4.14}$$

Izračunom ovog gradijenta došlo je vrijeme da pogledamo gradijente funkcije rotDiff i njenih komponenti. Ovdje ćemo izraze kao $\arccos$ i $Clamp$ koristiti kao oznake vrijednosti funkcije $\arccos$ i $Clamp$

$$\frac{\partial rotDiff}{\partial \arccos} = 57.29578$$

$$\frac{\partial rotDiff}{\partial Clamp} = -\frac{\frac{\partial rotDiff}{\partial \arccos}}{\sqrt{Clamp(\frac{newForward \cdot (ringPos - planePos)}{||newForward||^2 \cdot ||(ringPos - planePos)||^2})^2}} \tag{4.15}$$

$$\frac{\partial rotDiff}{\partial newForward} = \frac{\partial rotDiff}{\partial Clamp} \cdot (ringPos - planePos)$$

Zadnjim izračunom ponovno smo dobili gradijent funkcije gubitka po varijabli newForward, koja je ista u oba dijela postupka. Iz te vrijednosti izračunat ćemo gradijent funkcije gubitka po funkciji nextRot, po formuli navedenoj prije (izvod 4.14) i rezultirajuću vrijednost pribrojit ćemo postojećem gradijentu funkcije gubitka po funkciji nextRot, čime smo ujedinili distDiff i rotDiff i dobili potpuni gradijent po funkciji nextRot

$$\frac{\partial L}{\partial nextRot} = \left(\frac{\partial distDiff}{\partial nextPos} \cdot \frac{\partial nextPos}{\partial newForward} + \frac{\partial rotDiff}{\partial newForward} \right) \cdot \frac{\partial newForward}{\partial nextRot} \tag{4.16}$$

Nakon izračuna prethodnih gradijenata, treba izračunati gradijent funkcije nextRot po promjeni rotacije, to jest, gradijent umnoška dvaju kvaterniona:

$$\frac{\partial nextRot}{\partial QE} = \begin{bmatrix} w & -x & -y & -z \\ x & w & -z & y \\ y & z & w & -x \\ z & -y & x & w \end{bmatrix} \tag{4.17}$$

Zatim se treba izračunati gradijent pretvorbe Eulerovih kutova u kvaternion, pri čemu ćemo ponovno koristiti skraćenu notaciju trigonometrijskih funkcija (izvod 4.9). Kao i pri pretvorbi Eulerovih kutova u kvaternion, argumenti trigonometrijskih funkcija trebaju biti izraženi u radijanima, što ćemo zbog jednostavnosti zapisa ovdje pretpostaviti da već jesu. Također radi sažetosti, notacija množenja skraćena je na način da, na primjer, $sxsysz$ predstavlja $sx \cdot sy \cdot sz$. Gradijent pretvorbe glasi:

$$\frac{\partial QE}{\partial Input} = 0.5 \cdot \begin{bmatrix} -sxcycz + cxsysz & -cxsycz + sxcysz & -cxcysz + sxsycz \\ cxcycz - sxsysz & -sxsycz + cxcysz & -sxcysz + cxsycz \\ -sxsycz - cxcysz & cxcycz + sxsysz & -cxsysz - sxcycz \\ -sxcysz - cxsycz & -cxsysz - sxcycz & cxcycz + sxsysz \end{bmatrix} \tag{4.18}$$

$$\frac{\partial Input}{\partial (x,y,z)} = 0.5 \frac{\pi}{180} \cdot \begin{bmatrix} pitchSpeed & yawSpeed, -rollSpeed \end{bmatrix} \tag{4.19}$$

$$\frac{\partial L}{\partial distDiff} = 1$$

$$\frac{\partial L}{\partial (x,y,z)} = \frac{\partial L}{\partial distDiff} \cdot \frac{\partial distDiff}{\partial nextPos} \cdot \frac{\partial nextPos}{\partial newForward} \cdot \tag{4.20}$$

$$\frac{\partial newForward}{\partial nextRot} \cdot \frac{\partial nextRot}{\partial QE} \cdot \frac{\partial QE}{\partial Input} \cdot \frac{\partial Input}{\partial (x,y,z)}$$

Time napokon dobivamo gradijent funkcije gubitka po izlazima iz mreže, što smo cijelo vrijeme htjeli. Ovome vektoru od tri vrijednosti nadodat ćemo četvrtu, gradijent funkcije gubitka po ulazu thrust koji smo ostavili sa strane u ranijem izvodu (4.13).

Tako dobiven gradijent funkcije gubitka po izlazu aktivacijske funkcije izlaznog sloja iskoristit ćemo za "propagaciju pogreške unatrag" tako da se izračunom gradijenta izlaza izlaznog sloja po izlazima prethodnog sloja, koji služe kao ulazi u izlazni sloj, i množenjem te vrijednosti gradijentom funkcije gubitka po ulazu izlaznog sloja, ponovno po pravilu ulančavanja, dobiva gradijent funkcije gubitka po ulazima predzadnjeg sloja. Gradijenti izlaza slojeva uključuju i gradijente aktivacijskih funkcija, u ovom slučaju funkcije SoftSign (Szandała, 2021).

$$\begin{gathered} h = SoftSign(s) = (x, y, z) \\ \frac{\partial SoftSign(s)}{\partial s} = \frac{1}{(|s|+1)^2} \\ \frac{\partial L}{\partial s} = \frac{\partial L}{\partial h} \cdot \frac{\partial SoftSign(s)}{\partial s} \end{gathered} \tag{4.21}$$

Iz gradijenta funkcije gubitka po izlazu sloja, kako je prethodno navedeno, mogu se izračunati gradijenti funkcije gubitka po težinama i pomaku. Ovdje se s x označava ulaz u trenutni sloj mreže.

$$\begin{aligned} \frac{\partial L}{\partial W} &= \frac{\partial L}{\partial s} \cdot \frac{\partial s}{\partial W} & s &= Wx + b \\ \frac{\partial s}{\partial W} &= x & \frac{\partial L}{\partial W} &= \frac{\partial L}{\partial s} \cdot x \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial s} \cdot \frac{\partial s}{\partial b} & \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial s} \cdot 1 \end{aligned} \tag{4.22}$$

Ti gradijenti se u slučaju stohastičkog gradijentnog spusta u svakom koraku optimizacijskog postupka računaju nanovo, množe sa stopom učenja, i oduzimaju od trenutne vrijednosti težina s ciljem što većeg smanjenja pogreške modela.

$$\begin{aligned} W' &= W - \eta \cdot \frac{\partial L}{\partial W} \\ b' &= b - \eta \cdot \frac{\partial L}{\partial b} \end{aligned} \tag{4.23}$$

Ranije izračunati gradijent funkcije gubitka po izlazu sloja koristi se kako bi se propagiralo gubitak sve do početka mreže, dajući nam informacije o tome koliko točno svaka težina ili pomak doprinosi funkciji gubitka.

$$\frac{\partial L}{\partial h_{n-1}} = \frac{\partial L}{\partial s_n} \cdot \frac{\partial s_n}{\partial h_{n-1}}$$

Ideja je ta da računanjem gradijenata funkcije pri svakom okviru animacije za vrijeme treniranja modela korak po korak pomoću stohastičkog gradijentnog spusta dolazimo do parametara koji dovode do najmanjeg iznosa funkcije gubitka, to jest, do optimalnih parametara za zadani problem.

4.7. Skaliranje ulaza u model

Pri treniranju prvotno dizajniranog modela za automatsko upravljanje simulatorom leta pojavio se problem da je aktivacijska funkcija na izlazu mreže uvijek bila u zasićenju, to jest, da su izlazi mreže uvijek bile vrijednosti izrazito blizu -1 ili 1, a razlog tome bio je veliki iznos ulaza u tu aktivacijsku funkciju.

Posljedica toga bila je da je model zrakoplovom upravljao skrećući oko svake osi najviše moguće u nekom smjeru. Najčešće se to očitovalo kao nekontrolirani, kontinuirani let spiralnom putanjom, ali u rijetkim situacijama kad bi model mijenjao smjer, to bi bilo izrazito naglo jer bi se zrakoplov u jednom okviru okretao najbrže što može u jednu stranu, i onda odmah sljedeći okvir najbrže što može u drugu stranu. U još rjeđim situacijama u kojim bi model upravljao zrakoplov ravno, to bi postigao trzavim uzastopnim skretanjem lijevo-desno jer očito nije imao sposobnost postaviti išta osim vrijednosti -1 i 1 na izlaze.

Pojava navedena gore bila je dovoljno problematična posljedica zasićenja aktivacijske funkcije, ali još gora je bila druga posljedica - ta da se derivacijom zasićene aktivacijske funkcije dobije izrazito mala vrijednost blizu nuli što ometa proces propagacije pogreške unatrag govoreći modelu da je doprinos bilo čega što je prethodilo toj funkciji gotovo nikakav, što nadalje uzrokuje da se sve težine mreže u procesu optimizacije mijenjaju vrlo, vrlo sporo ili nikako.

Zbog gore navedenih razloga nije bilo moguće nastaviti trenirati model prije dovođenja u red barem iznose ulaza u aktivacijsku funkciju, a možda i sve iznose ulaza i međurezultata u mreži. Ideja koja prva nameće je ta da su ulazi u mrežu preveliki i da ih se nekako treba smanjiti. Ulazi vezani za pozicije objekata u prostoru nerijetko su svojim iznosima prelazili vrijednosti reda veličine 100, a iznosi rotacija su bili u stupnjevima, između 0 i 360.

Prvi pokušaj skaliranja odvio se tako da su se svi kutovi rotacije koji su se davali modelu podijelili s 360 kako bi se dobile vrijednosti između 0 i 1, a svi iznosi koordinata pozicija podijelili s nekim drugim brojem koji možemo zvati faktorom skaliranja udaljenosti. Razni pokušaji odabira faktora skaliranja uključuju: 200 (posljedica prvotnog automatskog generiranja poligona s prstenovima na nasumičnim lokacijama u kocki dimenzija 400 * 400 sa središtem u ishodištu, vrijednost najveće koordinate (x, y ili z) između svih pozicija prstenova, prosječna vrijednost norme radij-vektora pozicije svih prstenova i najveću vrijednost normi radij-vektora pozicije svih prstenova.

Sa skaliranjem vrijednosti svih ulaza u mrežu otprilike između -1 i 1, sustav je pokazivao nade za ulaz koji neće zasititi funkciju aktivacije na kraju mreže. Među-

tim, to se nije dogodilo. Još gore od toga, kao posljedica implementacije unaprijeđenog generatora poligona, poligoni su postali izduženi i skaliranje svih dimenzija istom konstantom bio bi preveliki problem - u ekstremnom slučaju, moguća pozicija zadnjeg prstena je 0.1 po x i y-osi i 1000 po z-osi. Ne samo to, nego skaliranje pozicija koje ovisi o rasporedu prstena u poligonu radi drukčije za svaki zasebni poligon, što znači da modelu možemo predstaviti dva prstena koja se nalaze na skaliranim koordinatama (0.1, 0.1, 0.1), iako se jedan u neskaliranom prostoru nalazi na koordinatama (1, 1, 1) a drugi na koordinatama (10, 10, 10). Rezultat toga bio bi da se model ne zna snalaziti u realnom prostoru jer je treniran samo na relativnim mjerama udaljenosti umjesto na apsolutnima.

Skaliranje pozicija na spomenute načine jednostavno nije izvedivo, stoga je izbačeno iz implementacije. Rješenje za problem zasićenja aktivacijske funkcije riješeno je umjesto toga na način da se umjesto ulaza smanji težine. Prethodno su svi parametri mreže - i težine i pomaci - bili inicijalizirani na slučajne vrijednosti između -1 i 1, ali ograničavanjem na manji raspon, na primjer između -0.1 i 0.1 na ulaz u aktivacijsku funkciju u većini slučajeva dolaze ulazi koji ne dovode do zasićenja.

4.8. Automatska generacija poligona

Kako bi model strojnog učenja s trenutnom konfiguracijom imao ikakve prilike naučiti izvršavati zadani zadatak potrebno je pripremiti dovoljno primjera za treniranje kako bi model mogao steći sposobnost generaliziranja problema umjesto da se nauči savršeno i "napamet" prolaziti jedan ili dva poligona na kojima se trenirao. Rješenje za to je napraviti barem deset poligona s različitim konfiguracijama prstenova. Ručno upisati poziciju i lokaciju svakog prstena mukotrpno je i vremenski neisplativo, stoga je za potrebe ovog diplomskog rada izrađen i generator poligona za razvijeni simulator leta.

Početni algoritam za generaciju poligona sastojao se od nasumičnog generiranja točki u kocki u prostoru dimenzija 400*400 sa središtem u ishodištu, to jest, generiranja N trodimenzijskih vektora s iznosima x, y i z-koordinata između -200 i 200, slučajno odabrano uniformnom razdiobom. Rotacije prstena zadane su u potpunosti nasumično. To rezultira s kaotičnim i nelogičnim poligonima, od kojih se nadamo da će model za vrijeme treniranja dovesti u mnoge različite pozicije u odnosu na sljedeći prsten - oštro desno, blago dolje, dijagonalno gore lijevo, ili čak u potpuno suprotnom smjeru...

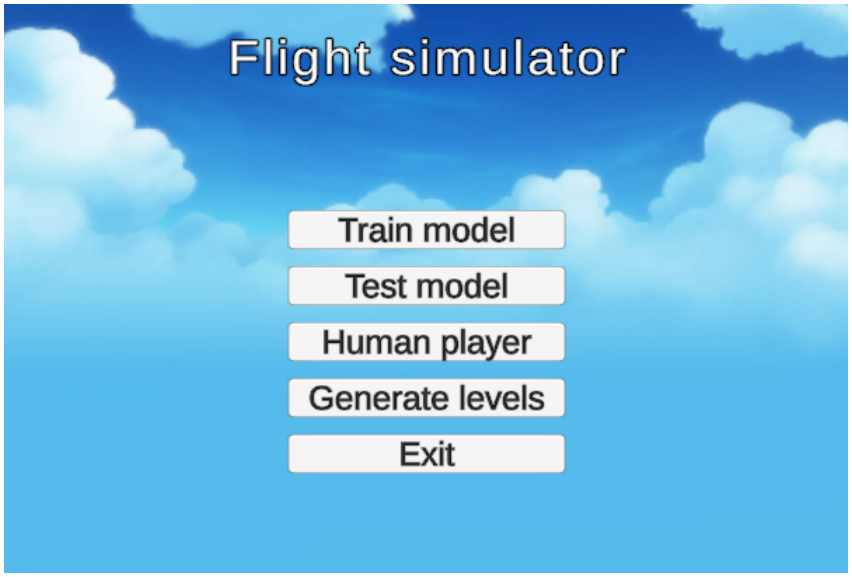
Nažalost, sve što su ti poligoni uspjeli postići je zbuniti model, koji nije uspio naučiti apsolutno išta iz njih. Za izbjeći taj učinak bilo je potrebno popraviti genera-

tor poligona na način da ne generira toliko kaotične poligone, već takve koji sadrže određeni kontinuitet u pozicijama susjednih prstenova. Varijable izmijenjenog generatora poligona su broj prstenova u poligonu, prosječna udaljenost između prstenova, varijacija udaljenosti između prstenova, maksimalni kut između prstenova, kao i broj različitih poligona koji generator u jednom pozivu generira. Pseudokod generatora poligona predstavljen je u sljedećem poglavlju

5. Implementacija

5.1. Korisnička aplikacija

Programsko rješenje ovog rada u potpunosti je sadržano u jednom projektu razvojnog okvira Unity, u obliku aplikacije s grafičkim sučeljem iz kojeg korisnik može pristupiti svim mogućnostima aplikacije, a one su, kao što je prikazano na slici 5.1: generiranje poligona za simulaciju leta, pokretanje simulatora leta kojim upravlja korisnik aplikacije na odabranom postojećem poligonu, treniranje modela strojnog učenja i pokretanje simulatora leta s automatskim upravljanjem na odabranom postojećem poligonu i s određenom konfiguracijom spremljenih parametara modela.

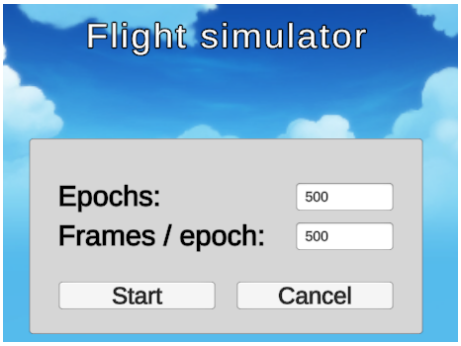


Slika 5.1: Sve mogućnosti aplikacije

Pritiskom na gumbe koji predstavljaju navedene opcije otvaraju se odgovarajući izbornici gdje korisnik može podesiti parametre potrebne za izvršavanje odabrane akcije, kao što su odabir poligona pri pokretanju simulacije, odabir broja epoha i broja iteracija po epohi pri pokretanju treniranja modela, odabir udaljenosti i kutova između

prstenova pri pokretanju generiranja poligona te odabir naučenog modela umjetne inteligencije i poligona kojeg treba preći. Svi izbornici prikazani su na slikama 5.2.

(a) Odabir poligona pri pokretanju simulacije



(b) Odabir broja epoha i broja iteracija po epohi pri pokretanju treniranja modela

(c) Odabir udaljenosti i kutova između prstenova pri pokretanju generiranja poligona

(d) Odabir naučenog modela umjetne inteligencije i poligona kojeg treba preći

Slika 5.2: Izbornici u grafičkom sučelju aplikacije

Opisano grafičko sučelje dio je početne scene u Unity projektu pod nazivom MenuScene. MenuScene osim naslova, gumbova i kontekstnih izbornika sadrži i objekt koji ne sadrži ikakve vizualne komponente već služi kao spremnik za skriptu UIController.cs u kojoj se nalazi klasa `UIController`, iz koje počinje sva interakcija korisnika s aplikacijom. U `UIController` su ostvarene sve funkcionalnosti glavnog izbornika, uključujući povezivanje gumbova s funkcijama za koje su zaduženi, prikazivanje i skrivanje kontekstnih izbornika po potrebi, dohvaćanje vrijednosti unesenih u okvire za unos podataka u kontekstnim izbornicima i mijenjanje scene pri početku simulacije leta.

5.1.1. Prenošenje podataka između scena

Pri početku vizualizirane simulacije leta potrebno je promijeniti scenu iz MenuScene u drugu scenu imena SimulationScene. Bez ikakve intervencije programera, svi objekti prisutni u prvoj sceni nestat će učitavanjem nove scene, kao i informacije koje oni sadrže. Budući da je cilj pokrenuti simulaciju s parametrima određenim u glavnom izborniku, kao što su poligon na kojemu se pokreće simulacija leta, informacija upravlja li simulacijom korisnik ili model strojnog učenja i ako simulacijom upravlja model, koju konfiguraciju parametara koristi, potrebno je ostvariti mehanizam prijenosa tih informacija između scena. U aplikaciji izrađenoj u sklopu ovog rada to je ostvareno koristeći `ScriptableObject`, posebnu vrstu objekta u Unityu [1] koja služi za pohranjivanje podataka koji perzistiraju unutar jednog pokretanja aplikacije u objektu dostupnom bilo kojem drugom objektu u bilo kojoj sceni, jer se `ScriptableObject` zapravo ne nalazi ni u jednoj sceni, već u direktoriju projekta, gotovo kao pomoćna datoteka za privremeno zapisivanje podataka.

`ScriptableObject` stvara se stvaranjem nove C# skripte koja sadrži vlastitu klasu koja nasljeđuje klasu `ScriptableObject`. U ovom projektu ta klasa zove se `SimulationContext`, i sadrži tri prethodno spomenute vrijednosti - niz znakova u kojemu je sadržano ime poligona koji se koristi za simulaciju, instancu enumeracije `InputType` koja označava upravlja li simulacijom korisnik aplikacije ili model strojnog učenja, i niz znakova u kojemu je sadržano ime datoteke u kojoj su spremljeni parametri umjetne neuronske mreže koja se koristi za automatsko upravljanje simulacijom, ako njom upravlja model.

5.2. Poligoni

Poligoni koji se koriste u simulatoru leta određeni su kolekcijom prstenova kroz koje zrakoplov treba proći kako bi uspješno prošao poligon. Početna pozicija zrakoplova uvijek je u ishodištu, a početna je orijentacija uvijek u pozitivnom smjeru globalne z-osi, s orijentacijama vlastite x-osi i y-osi koje se podudaraju s orijentacijama globalne x-osi i y-osi. Drugim riječima, zrakoplov je uvijek usmjeren u smjeru `NAPRIJED`,u kojem se kreće, dok mu je krov orijentiran u smjeru `GORE`, to jest u pozitivnom smjeru globalne y-osi, a kotači `DOLJE`, to jest u negativnom smjeru globalne y-osi. Zbog toga nije potrebno navoditi ikakve podatke o zrakoplovu pri opisu poligona.

[1] https://docs.unity3d.com/Manual/class-ScriptableObject.html, (Unity, 2019)

U aplikaciji se podatci o svakom poligonu predstavljaju instancom strukture podataka Level, čija je jedina varijabla lista prstenova u poligonu, poredana redoslijedom u kojem ih se mora proći. Prsteni su instance strukture podataka `Ring`, koja sadrži poziciju i orijentaciju prstena, zapakirane u strukturu podataka Pose razvojnog okvira Unity. Pose sadržava instancu klase `Vector3` koja predstavlja poziciju objekta i instancu klase Quaternion koja predstavlja orijentaciju.

Kako bi bilo moguće pristupiti podatcima o poligonima i nakon gašenja aplikacije, oni se spremaju u tekstualne datoteke u direktorijima projekta, točnije u Assets/Levels/, na način da svaki red tekstualne datoteke predstavlja jedan prsten, počevši s koordinatama pozicije prstena odvojenim razmacima nakon kojih je znak točka-zarez koji služi kao razdjelnik, i na kraju reda nalazi se orijentacija prstena, zapisana u obliku Eulerovih kutova rotacije, također odvojenih razmacima.

5.2.1. Generacija, spremanje i učitavanja poligona

Kao što je prethodno navedeno, poligone je moguće stvoriti ručno ali u sklopu ovog rada razvijena je klasa LevelGenerator za automatizaciju procesa generacije poligona. LevelGenerator je statička klasa sa statičkom metodom `generateLevels()` koja stvara niz poligona sa željenim svojstvima, a poziva se iz glavnog izbornika aplikacije.

Algoritam 1: Generiranje poligona

generateLevels ($levelCount, ringDist, ringDistVar, maxAngle, ringCount, prefix$) **:**

- inicijaliziraj listu prstenova
- **za** *i od 0 do levelCount* **:**
  - $pozicija = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$
  - $rotacija = 1$
  - isprazni listu prstenova
  - **za** *j od 0 do ringCount* **:**
    - $\Delta dist$ = nasumičan broj između $-ringDistVar$ i $ringDistVar$
    - $pozicija = pozicija + rotacija * (\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \cdot (ringDist + \Delta dist))$
    - **ako** *j > 0* **onda**
      - $randRot =$ nasumična rotacija ograničena na $maxAngle$
      - $rotacija = rotacija \cdot randRot$
    - dodaj u listu novi prsten $(pozicija, rotacija)$
  - spremi poligon pod imenom <$prefix$><$i + 1$>.txt

Parametre koje ta metoda prima su: $levelCount$ broj poligona koje treba generirati, $ringCount$ - broj prstena koje treba generirati po poligonu, $ringDist$ - prosječna udaljenost između susjednih prstenova, $ringDistVar$ - maksimalno odstupanje od te prosječne udaljenosti (u izvornom kodu ringDistVariance), $maxAngle$ - maksimalni kut rotacije između susjednih prstenova i $prefix$ - niz znakova koji služi kao prefiks imena datoteka te skupine poligona. Valja ponovno napomenuti da notacija $kvaternion*vektor$ predstavlja pojednostavljeni zapis operacije rotiranja vektora kvaternionom. Kvaternion $1 + 0i + 0j + 0k$, ili kraće, 1, zove se jedinični kvaternion i odgovara rotaciji od nula stupnjeva po svim osima.

Spremanje i učitavanje podataka o poligonima radi se preko druge statičke klase, `LevelLoader`. Klasa `LevelLoader` sadrži četiri statičke metode: jednu za dohvaćanje liste imena postojećih poligona spremljenih u Assets/Levels, drugu za učitavanje poligona s danim imenom, treću - koja je građevni blok prethodne - za stvaranje instance prstena iz retka datoteke i četvrtu, koja služi za stvoriti instancu klase `PlaneSimulator` s učitanim određenim poligonom za svrhe treniranja modela - više o tome kasnije.

5.2.2. Postavljanje scene

Pri prelasku iz glavnog izbornika u scenu za vizualiziranu simulaciju, potrebni se podatci dohvate preko instance objekta `SimulationContext`. Slično kako je glavnim izbornikom upravljao `UIController`, scenom upravlja objekt koji sadrži skriptu SimulationSceneController.cs u kojoj je klasa `SimulationSceneController`. Pri učitavanju, scena je potpuno prazna, s iznimkom već postavljenog objekta zrakoplova u ishodištu, a `SimulationSceneController` iz podatka o tome koji poligon se koristi, dohvaćenog iz konteksta stvara prstenove tako da instancira Unity *prefab* Assets/Prefabs/Ring s pozicijama i orijentacijama svakog prstena spremljenog u strukturi podataka učitanog poligona.

Zrakoplov u letu prati kamera pomoću vlastite skripte za praćenje, CameraFollowsPlane.cs. Moguće je, ali samo iz Unity *editora* promijeniti udaljenost kamere od zrakoplova, visinu kamere iznad zrakoplova i vertikalni kut pogleda u odnosu na zrakoplov. Kamera cijelo vrijeme ostaje na istoj relativnoj poziciji i orijentaciji u odnosu na zrakoplov, što znači da je na ekranu zrakoplov uvijek na istoj poziciji i jednako orijentiran, čak i ako je u globalnom koordinatnom sustavu okrenut naopako.

Slika 5.3: Primjer scene simulacije. Izvori modela zrakoplova i tekstura neba: https://assetstore.unity.com/packages/3d/vehicles/air/planes-choppers-polypack-194946, https://assetstore.unity.com/packages/2d/textures-materials/sky/fantasy-skybox-free-18353

5.3. Scena simulacije

5.3.1. Klasa PlaneController

5.3.2. Kontrole simulatora leta

Sučelje `PlaneInput` sadrži četiri metode za dohvaćanje ulaza za simulator leta - `getHorizontal()`, `getVertical()`, `getRudder()` i `getThrust()`. Te metode vraćaju realne vrijednosti između -1 i 1 koje odgovaraju željenom ulazu na virtualnim osima igraćeg kontrolera koje upravljaju redom: rotacijom zrakoplova oko z, x i y-osi te brzinom kretanja zrakoplova.

U slučaju ljudskog korisnika, četiri osi kojima se upravlja simulacijom leta mapirane su na tipke na tipkovnici ili igraćem kontroleru. Mijenjanje nagiba, rotacija oko x-osi ovisi o vertikalnoj osi, to jest prvoj osi. Na tipkovnici to odgovara tipki "S" za pozitivni i tipki "W" za negativni smjer osi, a na igraćem kontroleru vertikalnoj osi glavnog (lijevog) analognog štapića. Repno skretanje, rotacija oko y-osi, ovisi o osi *Rudder*, drugoj osi. Na tipkovnici to odgovara tipkama "Q" i "E", a na igraćem kontroleru stražnjim tipkama "RB/R1" i "LB/L1". Kotrljanje, rotacija oko z-osi, ovisi o horizontalnoj osi, trećoj osi. Na tipkovnici to odgovara tipkama "A" i "D", a na igraćem kontroleru horizontalnoj osi glavnog (lijevog) analognog štapića. Kontroliranje brzine zrakoplova ovisi o osi *Thrust*, četvrtoj osi. Na tipkovnici to odgovara tipkama "Shift" i "Ctrl", a na igraćem kontroleru vertikalnoj osi sporednog (desnog) analognog štapića.

U slučaju automatskog upravljanja, vrijednosti prethodno navedene četiri virtualne

osi iščitavaju se iz četiri izlaza neuronske mreže modelirane klasom `NeuralNet`, sadržane u instanci klase `AIPlaneController`. Više o neuronskoj mreži bit će spomenuto kasnije.

5.3.3. Klasa PlaneSimulator

Klasa `planeSimulator` prati trenutno stanje simulacije leta, uključujući pozicije i orijentacije svih objekata u simulaciji. Nužno je spomenuti da to ne smije imati veze s bilo kojom scenom u Unity aplikaciji, iz ranije navedenih razloga povezanih s treniranjem modela. To nažalost onemogućava korištenje klase `Transform`, jer instance te klase ne mogu postojati izvan scene. Ta klasa sadrži poziciju i orijentaciju nekog objekta, ali nudi i mnogo važnih metoda bez kojih ćemo morati modificirati svoj pristup, na primjer metoda za translaciju i rotaciju objekta.Te vrijednosti morat ćemo računati ručno, a za pamtiti podatke o poziciji i orijentaciji objekata koristit ćemo drugu klasu pogonskog sustava Unity nazvanu `Pose`, koja je samo plitki omotač oko te dvije vrijednosti.

Klasa `planeSimulator` nudi javnu metodu `tick()` kojom se računa sljedeći okvir simulacije leta, na sljedeći način:

Algoritam 2: Računanje okvira simulacije leta

tick `()`**:**

- **ako** *vrsta ulaza = AI* **onda**
  - $localRingPos = getLocalRingPos(pozicija, rotacija)$
  - postavi $localRingPos$ kao ulaz neuronske mreže
- $rotacija = calculateNextRot(rotacija, ulaz.xyz)$
- $pozicija = calculateNextPos(pozicija, rotacija, ulaz.thrust)$
- **ako** *je poligon pređen* **onda**
  - izađi iz metode
- **ako** *tijelo zrakoplova siječe granice sljedećeg prstena* **onda**
  - prsten je pređen, aktiviraj sljedeći
  - **ako** *nema više prstenova* **onda**
    - svi su pređeni, *poligon je pređen*

U metodi `tick()` klase `PlaneSimulator` koriste se metode triju pomoćnih klasa: `UpdatePosition`, `UpdateRotation` i `RingLocalizer`. U slučaju automatskog upravljanja, metoda `getLocalRingPos(position, rotation)` klase `RingLocalizer` prima poziciju i orijentaciju zrakoplova i pretvara globalne

koordinate sljedećeg prstena u lokalni koordinatni sustav zrakoplova kako bismo to postavili kao ulaz u neuronsku mrežu modela koji upravlja simulacijom. Metoda `calculateNextRot(rotation, x, y, z)` klase `UpdateRotation` prima trenutnu orijentaciju zrakoplova i prva tri ulaza u simulator leta, a vraća orijentaciju zrakoplova u sljedećem okviru simulacije, koju također pamti za ikakve buduće potrebe. Metoda `calculateNextPos(position, rotation, thrust)` klase `UpdateRotation` prima trenutnu poziciju i orijentaciju zrakoplova, kao i četvrti ulaz u simulator, a vraća poziciju zrakoplova u sljedećem okviru simulacije, koju također pamti za ikakve buduće potrebe.

Navedene metode za računanje pozicije i rotacije zapravo su jednake onima navedenim u poglavlju s izvodima funkcije gubitka modela i propagacije pogreške unatrag (izvodi 4.6, 4.4), a izdvojene su u zasebne klase upravo zbog potrebe za izračunom njihovih gradijenata i njihovog korištenja na tri mjesta u izvornom kodu.

Klasa `PlaneSimulator` osim metode za izračun sljedećeg stanja simulacije također sadrži i različite pomoćne javne metode, uključujući metodu kojom mijenja poziciju i rotaciju 3D modela zrakoplova čiju mu referencu u pozivu metode preda klasa `PlaneController`, metodu za ponovno pokretanje simulacije na istom poligonu, metodu za mijenjanje poligona i metode za vraćanje broja pređenih, odnosno nepređenih prstenova.

5.3.4. Detekcija prolaska kroz prsten

Klasa `PlaneSimulator`, kao što je već u pseudokodu prikazano, zadužena je i za provjeru kolizije između zrakoplova i sljedećeg prstena, na način da izračuna prostore koji zauzimaju aproksimacija trupa zrakoplova i aproksimacija unutrašnjosti prstena, to jest zone kroz koju zrakoplov treba proći. Oba prostora aproksimiraju se volumenom kvadra, samo što se dimenzije ta dva kvadra razlikuju. Za to se koristi klasa `Bounds` pogonskog sustava Unity, koja opisuje "granice" tijela u prostoru. Sama kolizija se detektira pomoću metode `Bounds.Intersects(otherBounds)` koja jednostavno vraća odgovor nalaze li se navedena dva kvadra u koliziji. Klasa `PlaneSimulator` sadrži metodu za ažuriranje granica tijela zrakoplova pri izračunu svakog okvira simulacije i metodu za ažuriranje granica tijela sljedećeg prstena svaki put kad se taj prsten promijeni, to jest, kad zrakoplov prođe kroz trenutni prsten. Također, ako je pokrenuta simulacija leta uz vizualizaciju, kao pri ispitivanju naučenog modela strojnog učenja ili ako simulacijom upravlja korisnik aplikacije, pri prolasku kroz pravilni prsten mijenjaju se boje prstenova u sceni. Prsten kroz koji je zrakoplov prošao boja se zelenom

bojom, a sljedeći prsten kroz koji zrakoplov treba proći boja se crvenom bojom.

5.4. Implementacija automatskog upravljanja

5.4.1. Klasa NeuralNet

Glavna komponenta programa za ostvarivanje automatskog upravljanja zrakoplovom je klasa `NeuralNet`, koja je sadržana u klasi `AIPlaneInput`. Ta klasa predstavlja unaprijednu potpuno povezanu umjetnu neuronsku mrežu s 3 neurona u ulaznom sloju, varijabilnim brojem skrivenih slojeva, svaki od kojih varijabilne širine, i 4 neurona u izlaznom sloju. Klasa `NeuralNet` sadrži javne metode korištene za unaprijedni prolaz kroz mrežu, računanje vrijednosti funkcije gubitka, i vršenje postupka propagiranja pogreške unatrag.

Klasa `NeuralNet` ima dva konstruktora. Jedan prima niz znakova koji predstavlja put do datoteke sa spremljenim parametrima neuronske mreže. Korištenjem ovog konstruktora poziva se metoda `LoadWeights(path)` kojom se težine i pomaci mreže inicijaliziraju se na vrijednosti iz te datoteke.

Drugi konstruktor ne prima nikakve parametre i njegovim korištenjem težine i pomaci mreže inicijaliziraju se na nasumične vrijednosti. Te nasumične vrijednosti kontrolirane su minimalnom i maksimalnom mogućom generiranom vrijednosti, konstantama `minWeight` i `maxWeight`, zadanih vrijednosti -0.1 i 0.1. Postoji iznimka da se, koje su god vrijednosti te dvije konstante, pri generaciji težina za prvi skriveni sloj mreže njihova vrijednost efektivno podijeli s 10, što se čini radi izbjegavanja zasićenosti aktivacijskih funkcija mreže.

5.4.2. Sučelje Layer

Svaki sloj mreže zasebni je entitet, instanca klase koja implementira `Layer`, sučelje koje sadrži metode `forward(input)`, za unaprijedni prolaz, to jest, izračunavanje izlaza sloja i `backward(grads)`, za unatražni prolaz, to jest, računanje gradijenata funkcije gubitka po vlastitim parametrima i propagaciju pogreške unatrag.

5.4.3. Klasa FCLayer

Generalni, potpuno povezani slojevi, instance su klase `FCLayer`. Svaki potpuno povezani sloj u svojoj strukturi podataka sadrži svoje težine, svoje pomake, dimenziju

prethodnog sloja, vlastitu dimenziju, svoje posljednje ulaze, izlaze i gradijente funkcije gubitka po svojem ulazu, težinama i pomacima, kao i težine i pomake za koje je optimizacijski postupak odlučio da su najbolji dosad. Unaprijedni prolaz potpuno povezanih slojeva računa se kao težinska suma njegovih ulaza i težina s pomacima:

$$outputs = inputs \cdot weights + bias \tag{5.1}$$

Izvod za unatražni prolaz nalazi se u prethodnom poglavlju (izvod 4.22). Klasa `FCLayer` također sadrži javnu metodu `optimStep()` koja služi za obavljanje koraka algoritma stohastičkog gradijentnog spusta u kojoj prima stopu učenja i modificira sve svoje težine i pomake po formuli za korak algoritma stohastičkog gradijentnog spusta (3.1), koristeći zapamćene gradijente težina i pomaka.

Klasa `FCLayer` sadrži i javne metode za pretvaranje svojih težina i pomaka, kako trenutnih, tako i najboljih, u listu znakovnih nizova po pravilu da jedan element liste predstavlja težine između svih neurona tog sloja i jednog neurona prethodnog sloja, odvojene zarezom i razmakom. Zadnji element liste je vektor pomaka tog sloja.

Osim toga, `FCLayer` sadrži i javnu metodu koja interpretira takvu listu kao težine i pomake, i zatim postavi vrijednosti na vrijednosti primljenih parametara.

5.4.4. Učitavanje i spremanje parametara

Prethodno spomenuta metoda `LoadWeights(path)` klase `NeuralNet` za učitavanje težina pri stvaranju mreže zapravo čita datoteku s parametrima i poziva metodu `importWeights()` svakog od svojih slojeva redom nad onoliko redova te datoteke koliko odgovara dimenziji svakog sloja, no radi i malo više od toga. Svaka datoteka koja sadrži parametre neuronske mreže u prvom redu sadrži konfiguraciju, to jest, dimenzije svakog sloja, te neuronske mreže, u obliku brojeva odvojenih znakom "x". Na primjer, konfiguracija "3x10x4" označava da su u toj datoteci sačuvani parametri neuronske mreže s 3 ulazna neurona, jednim skrivenim slojem od 10 neurona i 4 izlazna neurona. Pri pozivu metode `LoadWeights(path)` zapravo se na mjestu stvaraju i svi potpuno povezani slojevi mreže i pune se njihovi parametri.

Operacija suprotna toj operaciji je spremanje težina i pomaka neuronske mreže, koje se obavlja pozivom metode `saveWeights(prefix)` za trenutne parametre ili `saveBestWeights(prefix)` za najbolje parametre dosad. te funkcije spremaju težine i pomake svakog sloja zaredom, red po red u datoteku, gdje je jedan red ekvivalentan jednom elementu ranije navedene liste znakovnih nizova parametara odvojenih zarezom i razmakom. Naravno, u prvi red datoteke se zapiše konfiguracija mreže čiji se

parametri spremaju, a spremaju se pod imenom koje je određeno prefiksom dobivenom kao argument funkcije, konkatenirano s trenutnim datumom i vremenom. Prefiks će se kasnije koristiti za identificirati svojstva mreže - jesu li spremljeni parametri inicijalni (samo za svrhe ispitivanja valjanosti koda), finalni ili najbolji.

Do liste svih datoteka sa spremljenim parametrima mreže moguće je doći pozivom statičke metode `getSavedWeights()` klase `NeuralNet` koja pretraži direktorij predodređen za spremanje parametara modela i vrati imena svih tekstualnih datoteka iz tog direktorija.

5.4.5. Aktivacijske funkcije

Osim potpuno povezanih slojeva, postoje još i aktivacijske funkcije, od kojih se po jedna nalazi direktno poslije svakog potpuno povezanog sloja, uključujući i izlazni sloj. Aktivacijske funkcije direktno implementiraju sučelje `Layer`, što znači da sve sadrže metode `forward(inputs)` i `backward(grads)`, i nemaju specifičniju zajedničku nadklasu. U trenutačnoj implementaciji neuronske mreže sve su aktivacijske funkcije $SoftSign(x)$, to jest, instance klase `SoftSign`, iz želje da se izlazi svakog sloja, ali najvažnije posljednjeg, ograniče na vrijednost između -1 i 1, a da se pritom izbjegne "oštrina" standardne aktivacijske funkcije $\tanh(x)$, koja za ulaze relativno malene apsolutne vrijednosti poprima vrijednosti vrlo blizu -1 ili 1, to jest, dolazi u zasićenje. Vrijednost funkcije $SoftSign(x)$ računa se na sljedeći način:

$$SoftSign(x) = \left(\frac{x}{|x| + 1} \right) \tag{5.2}$$

Iako je $SoftSign(X)$ jedina aktivacijska funkcija koja je na kraju prisutna u mreži, u direktorijima projekta postoje još tri potpuno implementirane aktivacijske funkcije: $ReLU(X)$, $Softmax(x)$ i $\tanh(x)$

5.4.6. Unaprijedni prolaz kroz mrežu

Unaprijedni prolaz kroz mrežu svodi se na to da klasa `NeuralNet` ulaz u mrežu proslijedi funkciji `forward(inputs)` prvog potpuno povezanog sloja i izlaze te funkcije proslijedi funkciji `forward(inputs)` aktivacijske funkcije toga sloja, pa zatim te izlaze iskoristi kao ulaze sljedećeg sloja, i tako dalje do izlaznog sloja. Klasa `NeuralNet` tim putem unutar svoje funkcije `forward(inputs)` ulančava slojeve i zapamti izlaz iz zadnjeg sloja, koji također koristi kao povratnu vrijednost svoje funkcije `forward(inputs)`.

5.4.7. Izračunavanje funkcije gubitka

Funkcija `loss()` klase `NeuralNet` kao parametre prima trenutnu poziciju zrakoplova, instance klasa `UpdatePosition` i `UpdateRotation` za izračunati poziciju i rotaciju zrakoplova u sljedećem okviru simulacije, poziciju sljedećeg prstena i broj prstena koje zrakoplov još nije prošao. Rezultat pokretanja te funkcije je računanje funkcije gubitka upravo onako kako je opisano u prethodnom poglavlju (izvod 4.2), uz dodatno čuvanje najnovijeg, kao i najnižeg iznosa funkcije gubitka.

5.4.8. Propagacija pogreške unatrag

Pozivom metode `backward` klase `NeuralNet` računaju se gradijenti funkcije gubitka po svim parametrima sustava, počevši od komponenta funkcije gubitka, naime `distDiff` i `rotDiff`, zatim njihovih varijabli `nextPos`, to jest instance klase `UpdatePosition` i `nextRot`, to jest instance klase `UpdateRotation`. Ovaj postupak detaljno je razrađen u dijelu prošlog poglavlja posvećenom propagaciji pogreške unatrag, počevši od izvoda (4.13). Ove četiri klase implementiraju sučelje `Layer` i svaka zasebno implementira metodu `backward()` na prikladan način kako bi se proces propagacije pogreške unatrag odvio jednostavnije. Nažalost, zbog nekompatibilnosti vrsta ulaza, nijedna klasa nema implementiranu metodu `forward()` i zbog toga ih ne možemo potpuno automatizirano koristiti kao bilo koji sloj, iako postoji i drugi razlog, a to je grananje funkcije gubitka na distDiff i rotDiff.

Izvršavanje unatražnih koraka gore navedenih klasa rezultira gradijentom funkcije gubitka po izlazima mreže, što znači da je proces propagacije pogreške unatrag napokon došao do mreže, odakle stvari postaju brže i više automatizirane. Pozivom funkcije `backward()` klase `NeuralNet` gradijent funkcije gubitka po izlazu mreže daje se kao argument funkciji `backward` aktivacijske funkcije izlaznog sloja mreže. Nakon toga se u petlji gradijent propagira unatrag, modificirajući se na svakom koraku, i usput se izračunaju gradijenti svih parametara potpuno povezanih slojeva, sve sukladno izvodu algoritma propagacije pogreške unatrag iz prethodnog poglavlja (izvod 4.22).

5.4.9. Korak optimizacijskog algoritma

Nakon unatražnog prolaza kroz cijeli sustav tijekom kojeg su izračunati gradijenti svih parametara modela, slijedi poziv metode `optimStep()` klase `NeuralNet`. Ta metoda najjednostavnije samo pozove funkciju `optimStep()` svakog potpuno povezanog sloja mreže, i ti slojevi sami sebi podese parametre po načelu stohastičkog gradi-

jentnog spusta, kako je opisano ranije.

5.5. Treniranje modela

5.5.1. Klasa AITrainer

Opisane funkcionalnosti modela koriste se pri treniranju, a klasa koja je zadužena za pokretanje treniranja modela zove se `AITrainer`. Ta klasa ima samo jednu javnu metodu, `startTrainingSimulation(numEpochs, numFrames)`, koja je namijenjena pokretanju direktno pritiskom na gumb u glavnom izborniku aplikacije, a pokreće treniranje modela strojnog učenja nad svim dostupnim poligonima, kroz `numEpochs` epoha, gdje je jedna epoha jedan pokušaj prolaženja kroz sve prstenove nekog poligona, s najvećim dopuštenim brojem okvira simulacije po pokušaju koji se računa kao:

```
frameNo * (1 + 5 * epoch / epochNo)
```

Ideja iza promjenjivog broja iteracija, to jest, okvira simulacije, po epohi je ta da modelu isprve ne damo puno vremena da leti, pod pretpostavkom da će većina tog vremena biti uzaludno potrošena na neinformirani let u prazno ili daleko od sljedećeg prstena u trenutnom stanju podnaučenosti. No, s većim brojem odrađenih epoha očekujemo da model može zrakoplovom dulje upravljati s uvjerljivim uspjehom.

Svaka nova epoha pokrenuta je na nasumično odabranom poligonu iz liste svih dostupnih poligona dohvaćene pozivom statičke metode `getLevelNames()` klase `LevelLoader`

5.5.2. Klasa IndependentFlightSimulation

Sama simulacija odvija se unutar klase `IndependentFlightSimulation`. Budući da se proces treniranja kako ga je pokrenula klasa `AITrainer` odvija u potpunosti odvojeno od vizualizacije, klasa `IndependentFlightSimulation` zadužena je za poziv računanja svakog okvira simulacije pomoću vlastite instance klase `PlaneSimulator`, što u vizualiziranoj simulaciji uz pomoć *callback* funkcije pozvane pri iscrtavanju svakog okvira na ekran radi klasa `PlaneController`. Klasa `IndependentFlightSimulation` ima javnu metodu `reset(numIter)` kojom se zove istoimena metoda klase `PlaneSimulator` kako bi se ponovno pokrenula simulacija, a klasa `AITrainer` koja poziva tu metodu kao njen argument šalje novi trenutni broj maksimalno dopuštenih okvira simulacije.

Klasa `IndependentFlightSimulation` od klase `AITrainer` pri stvaranju i pri ranije spomenutom pozivu metode `reset(numIter)`, dobiva trenutni broj maksimalno dopuštenih okvira simulacije i u petlji poziva vlastitu funkciju `tick()` toliko puta, ili manje u slučaju ranije završenog poligona. Funkcija `tick()` poziva istoimenu funkciju klase `PlaneSimulator` koja računa novo stanje, to jest, sljedeći okvir simulacije, ali također poziva vlastitu funkciju `networkStep()` koja vrši redom unaprijedni prolaz, izračun funkcije gubitka, unatražni prolaz i korak optimizacijskog algoritma, sve kao što je opisano ranije u ovom poglavlju.

6. Zaključak

U okviru ovog diplomskog rada istražene su osnove strojnog učenja i kako se strojno učenje može primijeniti u simulaciji leta. Razmatrani su različiti načini predstavljanja stanja simulacije leta modelu strojnog učenja, kao i različiti pristupi strojnom učenju. Za rješenje danog zadatka odabran je podržani pristup strojnom učenju, gdje model poprima ulogu inteligentnog agenta koji upravlja simulacijom, a uči na način da dobiva povratne informacije o tome koliko dobro ili loše napreduje u zadatku.

Kao završni proizvod ovog rada razvijena je aplikacija u pogonskom sustavu Unity kojom se, kroz grafičko korisničko sučelje, može pokrenuti simulacija leta kojom može upravljati ili korisnik aplikacije preko tipkovnice ili igraćeg kontrolera, ili naučeni model strojnog učenja, koristeći izlaze neuronske mreže čije je težine moguće spremati i učitavati iz programa, u obliku tekstualnih datoteka. Razvijena aplikacija ima razne mogućnosti, uključujući generiranje poligona s mnogo različitih opcija pri generaciji i treniranje modela s varijabilnim brojem epoha i iteracija po epohi podesivim u aplikaciji i mnogim ostalim mogućnostima podesivim iz izvornog koda, kao što su stopa učenja, stopa kažnjavanja razlike u rotaciji, konfiguracija slojeva neuronske mreže i odabir aktivacijskih funkcija mreže.

Rezultati treniranja modela strojnog učenja zadovoljavajući su i uz najuspješniji skup parametara modela ima sposobnost samostalno uspješno proći veliku većinu poligona na kojima je ispitan. Najuspješniji skup parametara dobiven je treniranjem modela unaprijedne potpuno povezane umjetne neuronske mreže s 3 ulazna neurona, jednim skrivenim slojem od 10 neurona, i 4 izlazna neurona, kroz 1000 epoha, s između 200 i 1200 iteracija, to jest, okvira simulacije, po epohi i stopom učenja 0.001.

LITERATURA

B. Bijelić. Simulator leta letjelice drone. 2018.

Chudá, Hana. Universal approach to derivation of quaternion rotation formulas. *MATEC Web Conf.*, 292:01060, 2019. doi: 10.1051/matecconf/201929201060. URL https://doi.org/10.1051/matecconf/201929201060.

CHIHYUNG JEON. The virtual flier: The link trainer, flight simulation, and pilot identity. *Technology and Culture*, 56(1):28–53, 2015. ISSN 0040165X, 10973729. URL http://www.jstor.org/stable/24468693.

Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

Jinia Konar, Prerit Khandelwal, i Rishabh Tripathi. Comparison of various learning rate scheduling techniques on convolutional neural network. U *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*, stranice 1–5, 2020. doi: 10.1109/SCEECS48394.2020.94.

Seth M. Nielsen. A visually realistic simulator for autonomous evtol aircraft. *Theses and Dissertations*, 9340, 2021.

Shital Shah, Debadeepta Dey, Chris Lovett, i Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. 05 2017. doi: 10.48550/arxiv.1705.05065.

Tomasz Szandała. *Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks*, stranice 203–224. Springer Singapore, Singapore, 2021. ISBN 978-981-15-5495-7. doi: 10.1007/978-981-15-5495-7_11. URL https://doi.org/10.1007/978-981-15-5495-7_11.

ed. Tolliver, Judy. Bruce artwick is still flying. *Computer Science Alumni News*, 1(6):10–11, 1996.

Unity. Unity - manual: Scriptableobject, 2019. URL https://docs.unity3d.com/Manual/class-ScriptableObject.html.

Sustav za automatsko upravljanje pojednostavljenim modelom letjelice u 3D prostoru

Sažetak

Automatsko upravljanje podrazumijeva sposobnost vozila da se kroz prostor kreće ciljano i informirano bez ikakve interakcije i intervencije ljudske osobe. Bilo u obliku sustava autopilota u zrakoplovima ili samovozećeg automobila, automatsko je upravljanje vozilima područje razvoja umjetne inteligencije koje pokazuje u isto vrijeme uspjeh u prošlim, završenim projektima i neizmjeriv potencijal u budućim projektima koji će tek dostići svoj vrhunac. U sklopu ovog rada razvijen je jedan model za automatsko upravljanje pojednostavljenim modelom zrakoplova u simuliranom 3D prostoru.

Ključne riječi: Strojno učenje, umjetne neuronske mreže, optimizacija, automatsko upravljanje, simulacija leta, Unity

Autonomous control system for a simplified aircraft model in 3D space

Abstract

Autonomous vehicles are vehicles with the ability to traverse the environments with purpose and in an informed manner without any interaction or intervention by humans whatsoever. Be it in the shape of autopilot systems in airplanes or self-driving automobiles, autonomous vehicles are an area in the development of artificial intelligence which, at the same time, boasts great success in past projects that are finished, and immeasurable potential in future ones that are yet to reach their full potential. This thesis documents the development of one such system intended for autonomous control of a simplified aircraft model in a simulated 3D environment.

Keywords: Machine learning, artificial neural networks, optimization, autonomous control, flight simulation, Unity