Implementacija neuronske mreže na ZYBO razvojnom sustavu

Birka, Toni

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

Sveučilišni studij

IMPLEMENTACIJA NEURONSKE MREŽE NA ZYBO RAZVOJNOM SUSTAVU

Diplomski rad

Toni Birka

Osijek, 2019.

SADRŽAJ

1. UVOD

Istodobna dostupnost jeftinih računala visokih performansi, učinkovitih algoritama učenja i velikih baza podataka u posljednjih desetak godina, uzrokovala je značajan interes istraživača za rješavanje problema iz područja dubokog učenja, isto tako i brzi napredak u tom području. Tehnike dubokog učenja se sve više primjenjuju u području računalnog vida, a za testiranje tih tehnika se često koristi MNIST baza podataka koja se lako može interpretirati i koja omogućuje brzu usporedbu različitih tehnika [1]. Upravo ta baza podataka se koristi u ovom radu za treniranje neuronskih mreža u svrhu rješavanja problema klasifikacije rukom pisanih brojeva. Prepoznavanje rukom pisanih brojeva se široko koristi u automatskoj obrati bankovnih čekova, poštanskih adresa, itd. Postojeći sustavi koji rješavaju taj problem, uz neuronske mreže, uključuju neizrazitu logiku (engl. *fuzzy logic*), dok drugi sustavi mogu samo biti velike pregledne tablice koje sadrže moguće realizacije znamenki.

Prvo su, u drugom poglavlju, definirani pojmovi neuronske mreže, umjetne inteligencije, strojnog učenja, i dubokog učenja. Zatim su uspoređene struktura neurona ljudskog mozga i struktura umjetnog neurona. Opisani su slojevi neuronskih mreža i aktivacijske funkcije korištene u ovom radu te je navedena MNIST baza podataka na kojoj modeli izvode proces učenja. Navedene su najbitnije značajke ZYBO razvojnog sustava te arhitektura i način konfiguracije *Zynqa*, najbitnijeg elementa ZYBO platforme koji je sastavljen od FPGA i dvojezgrenog ARM *Cortex*-A9 procesora. Nakon toga je opisana *Keras* biblioteka za kreiranje i treniranje neuronskih mreža, *Kivy* biblioteka korištena za razvoj grafičkog korisničkog sučelja te ostale korištene biblioteke. Nadalje, navedeni su algoritmi za predobradu slike, kamera koja se koristi kao izvor HDMI signala te razvojna okruženja u kojima su se razvijala rješenja pojedinih elemenata sustava. U trećem poglavlju je predstavljena realizacija sustava, tako što je dijagramima toka opisana funkcionalnost dijela sustava implementiranog na ZYBO i grafičkog korisničkog sučelja. Osim toga, prikazan je i dizajn kućišta sustava. Na kraju je provedeno testiranje sustava, prikazani su dobiveni rezultati, ukazano je na nedostatke sustava te su predložene moguće metode za unaprjeđenje rada sustava.

2. TEORIJSKA PODLOGA I PRIMIJENJENE TEHNOLOGIJE

U ovom poglavlju je opisana ideja rada i, između ostalog, način rada neuronskih mreža (engl. *Neural Network*, skraćeno NN), korištena biblioteka za dizajniranje i treniranje istih, platforma na kojoj su implementirane, algoritmi korišteni pri obradi slike s kamere i ekstrakciji potencijalnih znamenaka s iste te programski alati korišteni pri izradi ovog diplomskog rada.

2.1. Idejno rješenje

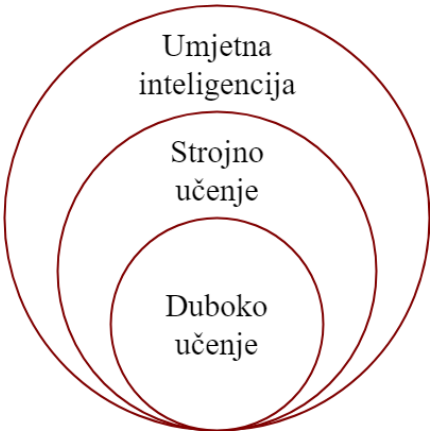
Ideja ovog rada je razvoj sustava za prepoznavanje rukom pisanih brojeva sa slike zabilježene kamerom. Sustav je dobio naziv ZYNN zbog povezivanja ZYBO platforme s NN, a zamišljen je kao skup dva podsustava.

Jedan dio ZYNN sustava se pogoni na ZYBO platformi na kojoj je implementirana NN za klasifikaciju rukom pisanih brojeva i razni algoritmi obrade slike s kamere. Interakcija s korisnikom se odvija preko drugog dijela sustava koji se pogoni na računalu, a korisniku omogućuje dizajniranje i treniranje modela NN-ova ili učitavanje već naučenih modela, te prikaz rezultata klasifikacije dobivenih od prvog podsustava. Osim toga, komunikacija podsustava se odvija preko UART sučelja (engl. *Universal Asynchronous Receiver/Transmitter*), koja između ostalog uključuje slanje parametara modela NN, rezultata klasifikacije i upravljačkih naredbi.

2.2. Neuronske mreže

NN, koja se još naziva i umjetna neuronska mreža (engl. *Artificial Neural Network*, skraćeno ANN), se definira kao skup međusobno povezanih jednostavnih procesnih elemenata čija se funkcionalnost temelji na biološkom neuronu koji su organizirani u slojeve i služe distribuiranoj paralelnoj obradi podataka [2]. NN se često povezuje s pojmovima umjetne inteligencije, strojnog učenja i dubokog učenja.

Umjetna inteligencija (engl. *Artificial Intelligence*, skraćeno AI) je vrlo veliko istraživačko područje, gdje strojevi pokazuju kognitivne sposobnosti kao što su učenje, proaktivna interakcija s okolinom, zaključivanje i dedukcija, računalni vid, prepoznavanje govora, rješavanje problema, percepcija i mnoge druge. Kolokvijalnije, AI označava bilo koju aktivnost u kojoj strojevi oponašaju inteligentno ponašanje koje ljudi obično pokazuju. AI crpi inspiraciju iz elemenata informatike, matematike i statistike [3].



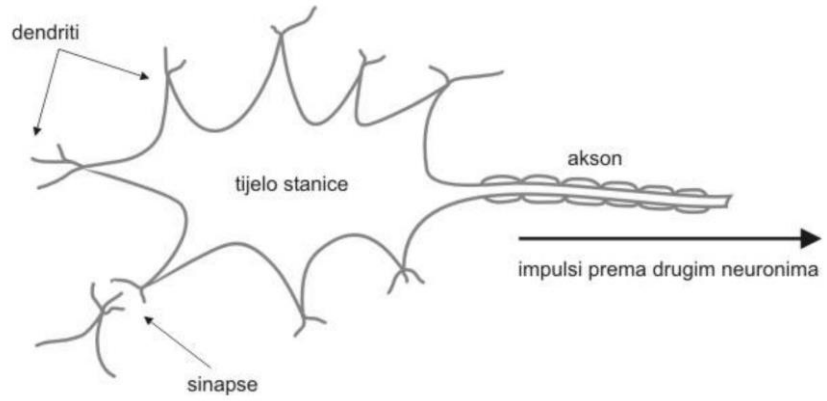
Slika 2.1.: Grafički prikaz povezanosti AI-ja, ML-a i DL-a.[3]

Strojno učenje (engl. *Machine Learning*, skraćeno ML) je podskup AI-ja koji se usredotočuje na podučavanje računala kako učiti bez potrebe za programiranjem određenih zadataka. Zapravo, ključna ideja ML-a je stvaranje algoritama koji uče iz podataka i vrše predikcije nad podacima. Postoje tri različite široke kategorije ML-a. U nadziranom učenju (engl. *supervised learning*) stroj uči iz ulaznih podataka kojima su pridruženi pripadajući izlazni podaci, s ciljem provođenja smislene predikcija na novim podacima. U nenadziranom učenju (engl. *unsupervised learning*) stroj uči na ulaznim podacima i sam pronalazi neku smislenu strukturu bez vanjskog nadzora. U podržanom učenju (engl. *reinforcement learning*) stroj djeluje kao agent koji ima interakciju s okolinom, traži ponašanja koja donose nagradu i na osnovu iskustva postaje sve uspješniji [3].

Duboko učenje (engl. *Deep Learning*, skraćeno DL) posebna je skupina ML metodologija koja koristi NN-ove. Neformalno se riječ duboko odnosi na prisutnost velikog broja slojeva u NN-u, ali to se značenje s vremenom promijenilo. Dok je prije nekoliko godina 10 slojeva bilo dovoljno da se mreža smatra dubokom, danas se mreža smatra dubokom kada ima stotinjak slojeva. DL je vrlo uspješno primijenjeno na različitim domenama (slika, tekst, video, govor i vizija), značajno poboljšavajući dostignuća proteklih desetak godina. Za uspjeh DL-a također je zaslužna relativno niska cijena grafičkih procesora (engl. *Graphics Processing Unit*, skraćeno GPU) za vrlo učinkovito numeričko računanje i dostupnost velike količine podataka za treniranje poput *ImageNet*-a, koji u bazi podataka sadrži preko 14 milijuna slika [4]. *Google, Microsoft, Amazon, Apple, Facebook* i mnogi drugi svakodnevno koriste tehnike DL-a za analizu ogromnih količina podataka. Međutim, ova vrsta ekspertize više nije ograničena samo na domenu akademskog istraživanja i na velike industrijske tvrtke, nego je postala neizostavnim dijelom moderne proizvodnje softvera [3].

Prva NN se pojavljuje 1943. godine kada su Warren McCulloch i Walter Pitts opisali svoje ideje i modelirali jednostavnu NN koristeći električne krugove [5]. Izvorni cilj bio je stvaranje računalnog sustava koji bi mogao rješavati probleme kao ljudski mozak. Međutim, vremenom su istraživači preusmjerili korištenje NN-ova na rješavanje specifičnih zadataka, što je dovelo do odstupanja od biološkog pristupa. Tako da se danas NN-ovi koriste u područjima medicine, robotike, marketinga, ekonomije i mnogim drugim.

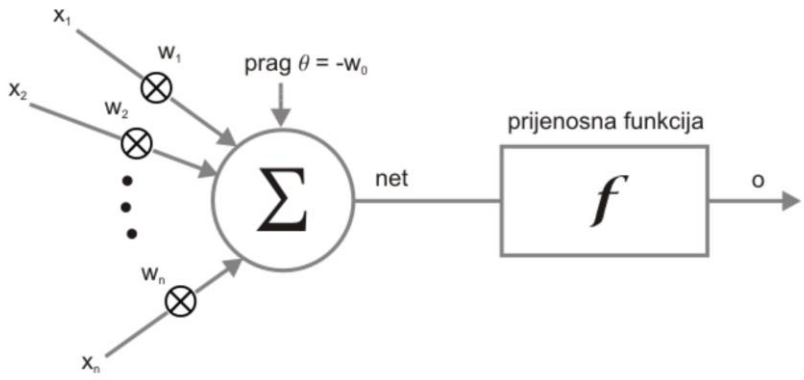
Osnovni element živčanog sustava je živčana stanica ili neuron. Prema [2] naš živčani sustav sastavljen je od oko 10^{11} neurona koji su raspoređeni po definiranom rasporedu, te je svaki u prosjeku povezan s 10^4 drugih neurona.



Slika 2.2.: Struktura neurona.[2]

Na slici 2.2. je prikazana struktura neurona, na kojoj se vidi da se neuron sastoji od:

- tijela stanice – sadrži informaciju predstavljenu električnim potencijalom između unutrašnjeg i vanjskog dijela stanice,
- dendrita – primaju informacije od drugih neurona koji utječu na potencijal stanice,
- aksona – dugačka cjevčica koja prenosi električke poruke tzv. akcijske potencijale u trajanju od 1 ms u trenutku kada ukupni potencijal tijela stanice pređe određeni prag,
- sinapsa – spojno sredstvo dvaju neurona kojim su prekriveni denditi.

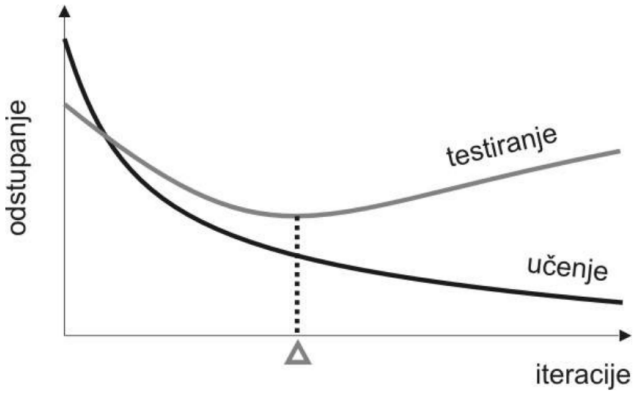


Slika 2.3.: Struktura umjetnog neurona.[2]

Slika 2.3. prikazuje strukturu umjetnog neurona koji se osim toga naziva i procesni element, čvor ili jedinica. Kod ovih neurona signali su numeričke vrijednosti, jakost sinapse se opisuje težinskim faktorom *w*, tijelo stanice je zbrajalo koje zbraja ulazne signale (x_1, x_2, ... x_n) pomnožene s težinskim faktorima (w_1, w_2, ... w_n), a na izlazu daje vrijednost *net*. Akson je prijenosna tj. aktivacijska funkcija f koja se primjenjuje na vrijednost *net* daje pripadajući izlazni signal ovisno o korištenoj funkciji. Često se pri zbrajanju dodaje vrijednost praga (engl. *bias*), što omogućava upravljanje vrijednosti pri kojoj se aktivacijska funkcija okida, drugim riječima utjecaj praga se može zamisliti kao translacija prijenosne funkcije po *x* osi.

NN-ovi se primjenjuju za rješavanje problema klasifikacije i predviđanja, tj. sve probleme kod kojih postoji nelinearna veza ulaza i izlaza. Kao što je opisano u [2], pri radu s NN-ovima postoje faza učenja i faza obrade podataka. Učenje je iterativni postupak predočavanja ulaznih primjera i eventualno očekivanih izlaza pri čemu se postupno prilagođavaju težine veza neurona.

Često se skup primjera za učenje dijeli na tri skupa: skup za učenje, skup za testiranje i validacijski skup. Prvi skup se koristi za podešavanje težinskih faktora. Primjeri iz drugog skupa služe za provjeru rada mreže s trenutnim težinskim faktorima tijekom učenja kako bi se postupak učenja zaustavio prilikom degradacije performanse mreže. Prenaučenost (engl. *overfitting*) je pojam koji se koristi kada mreža nakon određenog broja iteracija izgubi svojstvo generalizacije i dobro obrađuje samo podatke iz skupa primjera za učenje. To se može spriječiti kontinuiranim praćenjem izlaza mreže dobivenog pomoću skupa za testiranje (Slika 2.4.). Nad validacijskom skupu se na kraju određuje točnost i preciznost obrade podataka [2].



Slika 2.4.: Odstupanje stvarnog izlaza kroz iteracije.[2]

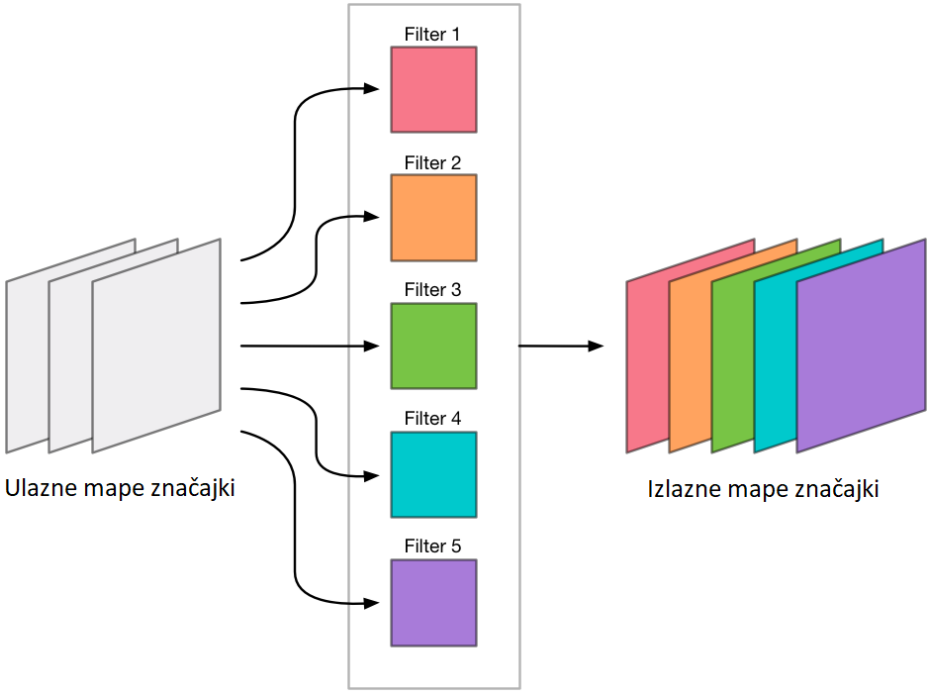
Pri učenju se susrećemo s pojmovima iteracije i epohe. Iteracija je korak u kojem se odvija podešavanje težinskih faktora, a epoha je jedno predstavljanje cijelog skupa za učenje. S obzirom na broj primjera predočenih mreži u jednoj iteraciji razlikujemo pojedinačno (engl. *on-line training*) i grupno učenje (engl. *batch training*). Pri pojedinačnom učenju se za svaki primjer podešavaju faktori, a pri grupnom učenju se za podešavanja faktora koristi cijela epoha koja predstavlja jedno predočavanje svih primjera, u tom slučaju se iteracije podudaraju s epohama. Znanje koje je mreža naučila tijekom faze treniranja je implicitno pohranjeno u težinama veza neurona [2].

2.2.1. Struktura neuronskih mreža

Radi jednostavnosti, u računarstvu se NN predstavlja kao skup slojeva. Ovi slojevi su kategorizirani u tri klase: ulazni, skriveni i izlazni. Svaka NN ima jedan ulazni i izlazni sloj. Broj neurona u ulaznom sloju jednak je broju ulaznih varijabli u podacima koji se obrađuju. Broj neurona u izlaznom sloju jednak je broju izlaza povezanih sa svakim ulazom. Skriveni slojevi su potrebni NN-u ako postoji nelinearnost u ulaznim podacima [6].

Postoji niz različitih slojeva iz klase skrivenih slojeva, a u nastavku su opisani najčešće korišteni, a ujedno i svi slojevi koji se koriste pri dizajniranju NN-ova u ovom radu.

Kolvolucijski sloj služi za izlučivanje značajki (engl. *features*) iz ulaznih podataka, što se postiže primjenom dvodimenzionalne konvolucije, po čemu je sloj dobio ime, osim toga se smanjuje rezolucija podataka tj. slike. Kod ove vrste slojeva ulaz je dvodimenzionalan pa se umjesto težina se pri procesu učenja treniraju jezgre (engl. *kernels*), koje se još nazivaju i detektorima značajki (engl. *feature detectors*) i filtrima (engl. *filters*). Izlaz sloja je također predstavljen u dvije dimenzije a naziva se mapama značajki (engl. *feature maps*), a broj mapa značajki na izlazu je definiran brojem primijenjenih filtera. Područje ulaza koje stvara samo jedan element u izlaznoj mapi značajki naziva se receptivnim poljem (engl. *receptive field*). Ilustracija konvolucijskog sloja je prikazana na slici 2.5. [7].



Slika 2.5.: Prikaz konvolucijskog sloja.[8]

Konvolucija se postiže prolaskom naučenog filtra kroz ulaznu mapu značajki, a vrijednost neurona izlazne mape značajki računa prema izrazu (2-1), u kojem su:

- σ – aktivacijska funkcija primijenjena u tom sloju,
- b – *bias* filtra,
- f – veličina filtra,
- $w_{l,m}$– težinski faktor na poziciji l, m u filtru,
- $a_{j+l,\,k+m}$ – vrijednosti ulazne mape značajki na poziciji j + l, k + m,
- j, k – pozicija na kojoj se nalazi filtar na mapi značajki.

$$\sigma\left(b + \sum_{l=0}^{f}\sum_{m=0}^{f} w_{l,m}a_{j+l,k+m}\right) \tag{2-1}$$

Postoje još dva parametra koja utječu na dimenzije izlaza, a to su korak (engl. *stride*) i dopuna (engl. *padding*). *Stride* označava korak filtera pri obilaženju ulazne mape značajki, tako da treba voditi računa da filter obuhvati cijelu mapu značajki kada je vrijednost ovog parametra veća od jedan. S ciljem očuvanja više informacija s rubova mape značajki se uvodi parametar *padding*, kojim se na rub dodaje zadani broj piksela s određenom vrijednošću, najčešće se dodaju nule. Padding se koristi kada se želi da izlaz iz sloja bude iste veličine kao i ulaz [8].

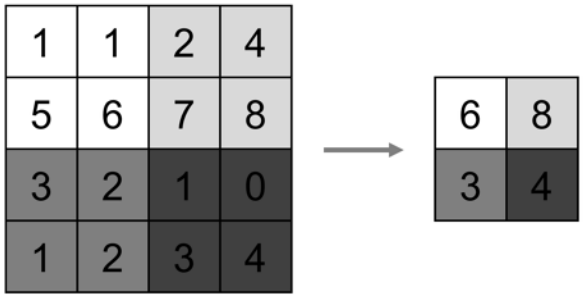
Prema [9], veličina izlaza konvolucijskog sloja se računa pomoću izraza (2-2), u kojem U predstavlja veličinu ulazne mape značajki, F veličinu filtra, P *padding* i S *stride*. Međutim, ako neki od elemenata nije kvadratnog oblika, tada se posebno mora računati i visina i širina izlaza.

$$veličina\ izlazne\ mape\ značajki = \frac{U - F + 2P}{S} + 1 \tag{2-2}$$

Sloj sažimanja (engl. *pooling layer*) služi za smanjenje rezolucije mape značajki te povećanje prostorne invarijantnosti NN-a, što znači da NN postaje neosjetljiva na manje pomake značajki. Postupak izračuna vrijednosti izlazne mape značajki je sličan onome kod konvolucijskog sloja, samo što ova vrsta slojeva nema učeće parametre, nego se na receptivno polje primjenjuje neka od metoda sažimanja. Važno je naglasiti da se u ovom sloju broj mapi značajki ne mijenja. Kao i kod konvolucijskog sloja, sloj sažimanje se definira s parametrima *stride* i *padding* [7].

Prema [3], navedene su neke od metoda sažimanja:

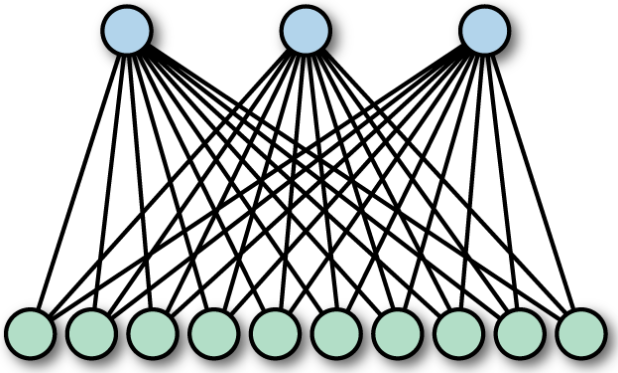
- sažimanje usrednjavanjem (engl. *Average pooling*) – za element izlazne mape značajki uzima se srednja vrijednost receptivnog polja,
- sažimanje maksimalnom vrijednošću (engl. *Max-pooling*) – iz receptivnog polja se izdvaja najveća vrijednost, kao što je prikazano na slici 2.6., pri čemu parametar *stride* iznosi dva.



Slika 2.6.: Sažimanje maksimalnom vrijednošću.[9]

U potpuno povezanom sloju (engl. *fully-connected layer*, skraćeno FC), kao što se može naslutiti po nazivu sloja, su svi neuroni povezani sa svim neuronima iz prethodnog sloja (Slika 2.7.). Broj parametara koje treba naučiti u sloju se izračunava izrazom (2-3). Parametar n_k se odnosi na broj neurona u k-tom potpuno povezanom sloju, n_{k-1} se odnosi na broj neurona prethodnog sloja, s tim da se na kraju se iznos povećava za jedan jer svaki potpuno povezani sloj ima po jedan *bias*. Ova činjenica ukazuje na potrebu za velikom količinom resursa za pohranu svih parametara, ako se trenira NN s potpuno povezanim slojevima koji sadrže velik broj neurona [10].

$$broj\ parametara = n_k * n_{k-1} + 1 \tag{2-3}$$



Slika 2.7.: Prikaz potpuno povezanog sloja. [10]

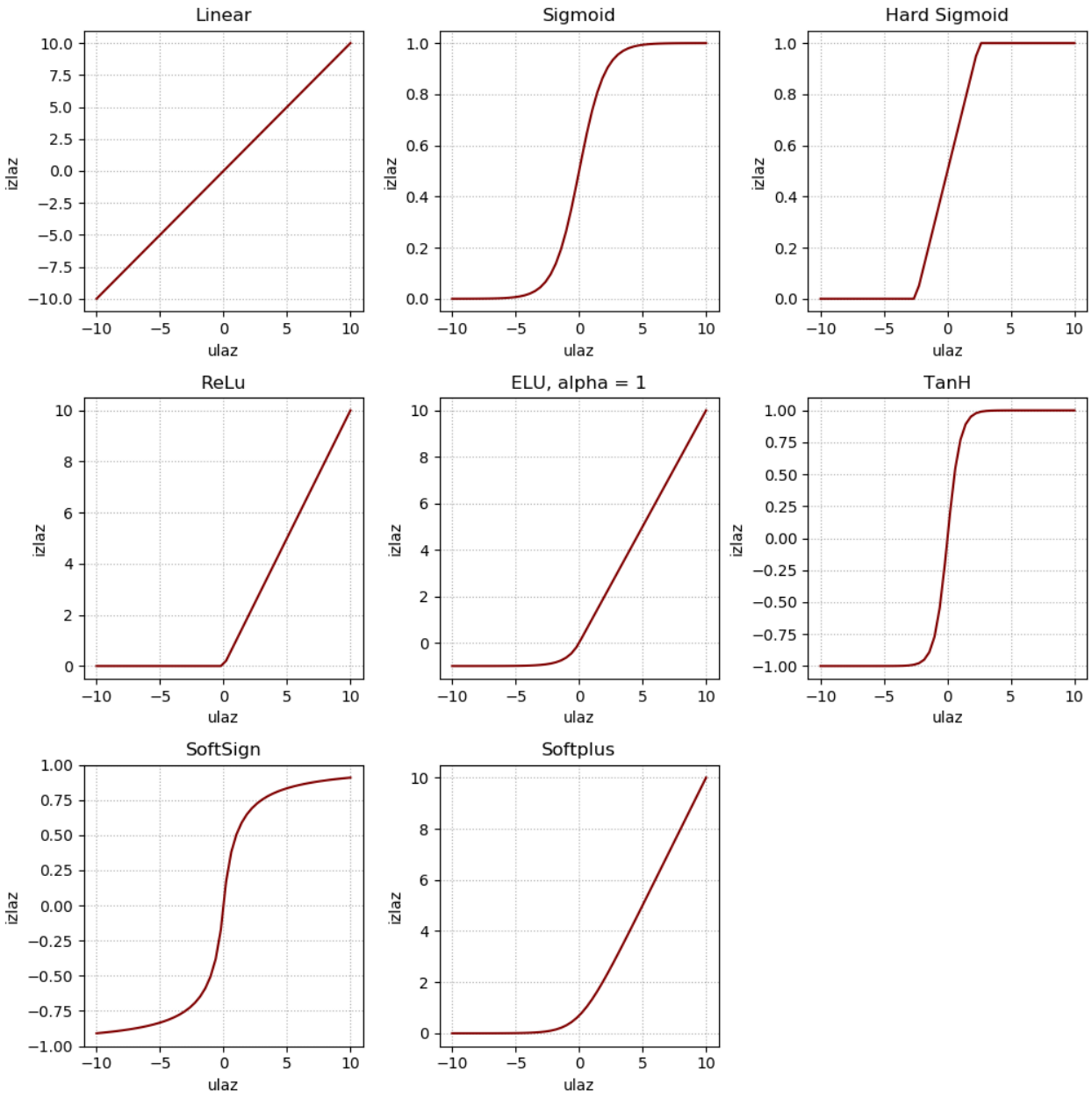
2.2.2. Aktivacijske funkcije

Aktivacijske funkcije (engl. *activation function*) imaju ključnu ulogu u radu NN-ova, jer određuju izlaz iz svakog neurona [11]. Slikom 2.8. su prikazani grafovi onih aktivacijskih funkcija koje je moguće koristiti za treniranje NN-ova na ZYNN-u.

Za rješavanje problema klasifikacije, najčešće se koristi *Sigmoid* zbog toga što je nelinearna, kontinuirano se razlikuje i ima fiksni izlazni raspon. Osim te funkcije, iste prednosti pruža *ReLu* aktivacijska funkcija, ali s boljim performansama od *Sigmoid* funkcije [12].

Softmax aktivacijska funkcija, čiji graf nije prikazan, izračunava raspodjelu vjerojatnosti događaja nad *n* različitih događaja. Općenito rečeno, ova funkcija računa vjerojatnost svake ciljne klase u odnosu na sve moguće ciljne klase, što je prikazano izrazom (2-4), pri čemu o_i označava i-ti izlaz sloja, dok se broj izlaza označava s *n*. Kasnije se izračunate vjerojatnosti koriste za utvrđivanje ciljne klase danih ulaza [12].

$$softmax(o_i) = \frac{exp\ (o_i)}{\sum_{j=0}^{n} exp\ (o_j)} \tag{2-4}$$

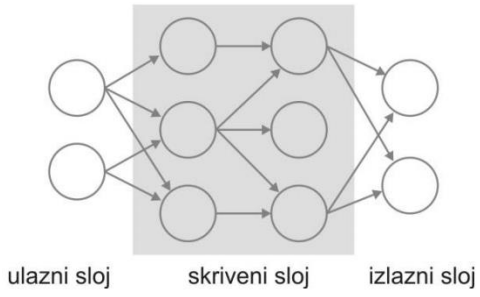


Slika 2.8.: Prikaz grafova aktivacijskih funkcija. [11]

2.2.3. Vrste neuronskih mreža

Postoje mnoge vrste NN-ova koje djeluju na različite načine kako bi se postigli različiti ishodi. Ovdje su navedene i ukratko opisane neke od najvažnijih vrsta i njihove primjene.

Aciklička NN (engl. *Feedforward* NN) je jedna od najjednostavnijih vrsta NN-ova, pri čemu je propagacija signala jednosmjerna odnosno podaci se kreću samo u jednom smjeru od ulaznog do izlaznog sloja (Slika 2.9.). Acikličke NN koriste se u tehnologijama poput prepoznavanja lica i računalnog vida, zato što je ciljne klase u tim aplikacijama teško klasificirati [13].

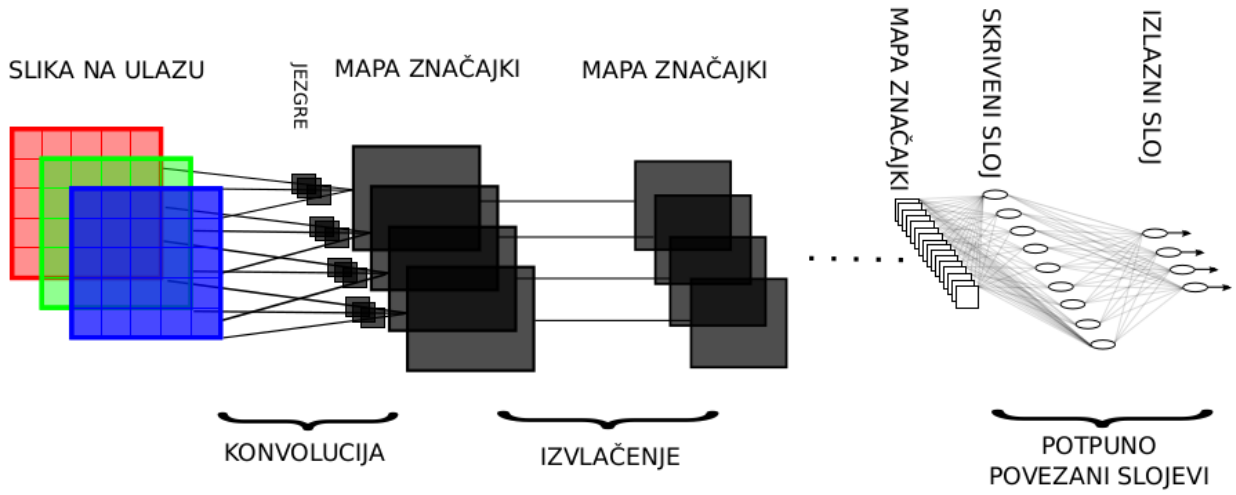


Slika 2.9.: Primjer acikličke NN.[2]

NN s višeslojnim perceptronom (engl. *Multilayer Perceptron*, skraćeno MLP) je vrsta acikličke NN koja za učenje koristi algoritam sa širenjem pogreške unazad (engl. *backpropagation*). Navedeni algoritam se koristi za brzo i efikasno učenje višeslojnih NN-ova, a radi na principu minimiziranja nastale pogreške metodom gradijentnog spusta [2]. MLP obično ima tri ili više slojeva, a primjenjuje se za klasifikaciju podataka koji se ne mogu linearno odvojiti. Ova vrsta se u praksi široko primjenjuje u prepoznavanju govora i tehnologijama strojnog prevođenja [13].

Kod NN s povratnom vezom (engl. *Recurrent* NN, skraćeno RNN) se izlaz određenog sloja sprema i vraća na ulaz, to pomaže u predviđanju ishoda sloja. Mreža se smatra RNN, ako jedan ili više neurona ima povratnu vezu. Ova vrsta je pogodna za sekvencijalne podatke jer svakim vremenskim korakom, neuron pamti neke informacije koje je imao u prethodnom koraku. Drugim riječima, neuroni s povratnom vezom djeluju kao memorijske ćelije tijekom računanja i izvođenja operacija. Zbog toga su vrlo učinkovite u prepoznavanju govora, pretvaranju teksta u govor, predviđanju dionica i slično [13].

Konvolucijska NN (engl. *Convolutional* NN, skraćeno CNN) je vrsta NN-ova specijalizirana za obradu podataka koji imaju rešetkastu topologiju (Slika 2.10.). Primjeri uključuju podatke vremenske serije, koji se mogu zamisliti kao jednodimenzionalnu rešetku koja uzima uzorke u pravilnim vremenskim intervalima, i slikovne podatke, koji se mogu zamisliti kao dvodimenzionalna rešetka piksela [14]. Tri važna svojstva CNN-a, koja mogu poboljšati rad ML sustava su raspršena povezanost, dijeljenje parametara i translacijska invarijantnost, zbog kojih su CNN-ovi vrlo učinkoviti u klasifikaciji slika, prepoznavanju slika i video zapisa, obradi prirodnog jezika i sustavima preporuka [13].



Slika 2.10.: Opća struktura CNN-a.[7]

Kod klasičnih NN-ova (primjerice MLP) postoji interakcija između svake ulazna jedinica svakom izlaznom jedinicom određenog sloja, što znači da je jedan izlaz sloja ovisan o svim ulazima sloja. S druge strane, konvolucijski slojevi imaju rijetku tj. raspršenu povezanost parametara. To se postiže filtrom manjih dimenzija od ulazne mape značajki. Na primjer, ulazna slika može imati tisuće ili milijune piksela, ali se unatoč tome, mogu detektirati male značajke, primjerice rubove nekog objekta. Time se treba pohraniti manje parametara, što smanjuje broj operacija potrebnih za izračun izlaza i povećava statističku učinkovitost [14].

U konvolucijskim slojevima se filtar s istim težinskim faktorima primjenjuje na cijeloj slici. Takvo dijeljenje parametara omogućava mrežama učenje relevantnih i diskriminativnih značajki. Jezgre se specijaliziraju za određenu funkciju, odnosno detekciju određenih značajki. Kada se ne bi dijelile težine mreža bi bila prenaučena na određeni detalj iz ulaznih podataka. S dijeljenjem se povećava općenitost naučene značajke i poboljšava se generalizacijska sposobnost mreže [7],[14].

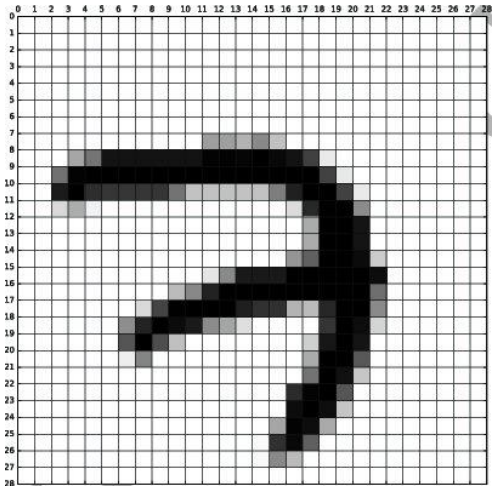
Svojstvo translacijske invarijantnosti omogućava CNN-u otpornost na male promjene položaja značajki. Primarno se to postiže slojevima sažimanja, a budući da se to postiže postepeno kroz više takvih slojeva, mreža i dalje uči položaje značajki, ali razvija otpornost na manje varijacije u položaju [7],[14].

2.2.4. MNIST

MNIST (*Mixed National Institute of Standards and Technology*) je baza podataka koju su uveli LeCun i suradnici 1998. godine [1]. Široko se koristi za provjeru novih tehnika u području računalnog vida, a posljednjih su godina mnogi autori istraživali performanse CNN-ova i drugih tehnika DL-a preko ove baze podataka. Danas je znatna količina radova, koji koriste ovu bazu podataka, postigla stopu pogreške manju od 1 %. MNIST sadrži ukupno 70.000 primjeraka, od kojih je 60.000 namijenjeno treningu, a ostatak testiranju. Skup za trening i test izabran je tako da isti pisac ne bi bio uključen u oba skupa. Skup za učenje sadrži primjerke od više od 250 osoba. Baza podataka je kreirana iz dva različita izvora:

- NIST *Special Database* 1 – primjerci prikupljeni među srednjoškolcima,
- NIST *Special Database* 3 – preuzeto od zaposlenika ureda za popis stanovništva [15].

Izvorne slike su prethodno obrađene, postupak obrade prvo uključuje normalizaciju slika kako bi se uklopile u okvir veličine 20×20 piksela uz očuvanje omjera slike (engl. *aspect ratio*). Zatim se primjenjuje filter za zaglađivanje (engl. *anti-aliasing filter*), i kao rezultat se dobije siva slika (engl. *grayscale image*). Na kraju se dobivena slika centrira u slici veličine 28×28 piksela, s obzirom na centar mase piksela. Primjer koji odgovara znamenki „7“ se nalazi na slici 2.11. Na slici 2.12. prikazan je veći broj primjeraka koji pripadaju skupu za učenje [15].



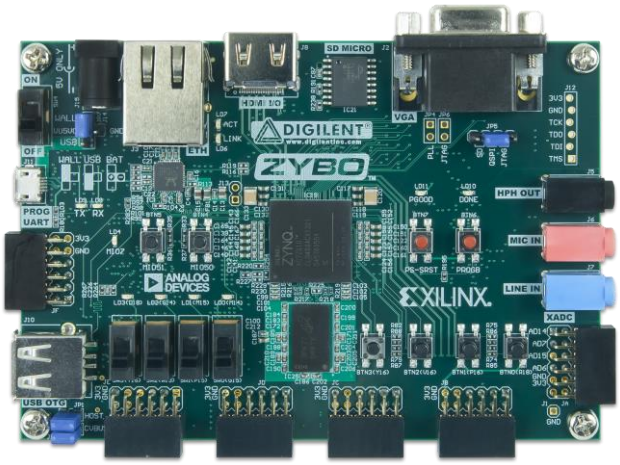
Slika 2.11.: Primjerak znamenke „7“.[15]

Slika 2.12.: 10 primjeraka svake znamenke iz MNIST-a. [15]

2.3. ZYBO razvojni sustav

ZYBO (*Zynq Board*) je platforma za razvoj softvera i digitalnih sklopova, sastavljena oko najmanjeg člana porodice *Xilinx Zynq*-7000, Z-7010, koji se temelji na *Xilinxovoj* AP SoC (engl. *All Programmable System on Chip*) arhitekturi [16]. Značajke ZYBO platforme:

- ZYNQ XC7Z010-1CLG400C,
- 512MB x32 DDR3,
- HDMI port,
- VGA port,
- OTG USB 2.0 PHY,
- eksterni EEPROM,
- audio ulaz/izlaz,
- *microSD* port,
- šest *Pmod* portova.



Slika 2.13.: ZYBO platforma. [16]

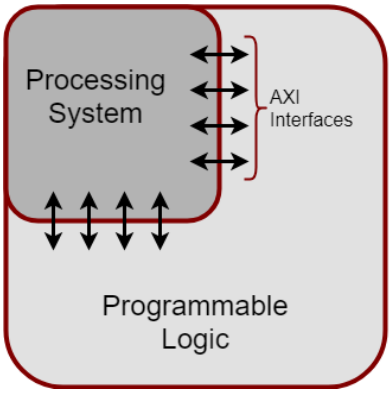
- periferni upravljači: 1G *Ethernet*, USB 2.0, SDIO, SPI, UART, CAN, I2C,
- JTAG programiranje i UART-USB konverter,
- GPIO: šest tipkala, četiri sklopke, pet LE dioda, Programska podrška

ZYBO je u potpunosti kompatibilan sa *Xilinxovim* setom alata visokih performansi *Vivado Design Suiteom*. Ovaj programski paket se može koristiti za projektiranje sustava bilo koje složenosti, od cjelovitog operativnog sustava (engl. *operating system*, skraćeno OS) koji pokreće više poslužiteljskih aplikacija, pa sve do jednostavnih aplikacija koje upravljaju LE diodama [16].

2.3.1. Arhitektura Zynq-a

Kombinacija dvojezgrenog ARM *Cortex*-A9 procesora s tradicionalnom FPGA (engl. *Field Programmable Gate Array*) je ključna značajka *Zynqa*. ARM *Cortex*-A9 je aplikacijski procesor, sposoban za pokretanje OS-a kao što je *Linux*, dok se programibilna logika temelji na *Xilinx* FPGA arhitekturi 7. serije. Arhitektura je upotpunjena industrijskim standardnim AXI sučeljima, koja pružaju veliku propusnost i veze s niskom razinom latencije između spomenuta dva dijela uređaja. To znači da se procesor i logika mogu koristiti za ono što najbolje rade, bez pretjeranog povezivanja dva fizički odvojena uređaja. U međuvremenu, koristi koje proizlaze iz pojednostavljenja sustava na SoC uključuju smanjenje fizičke veličine i ukupnih troškova [17].

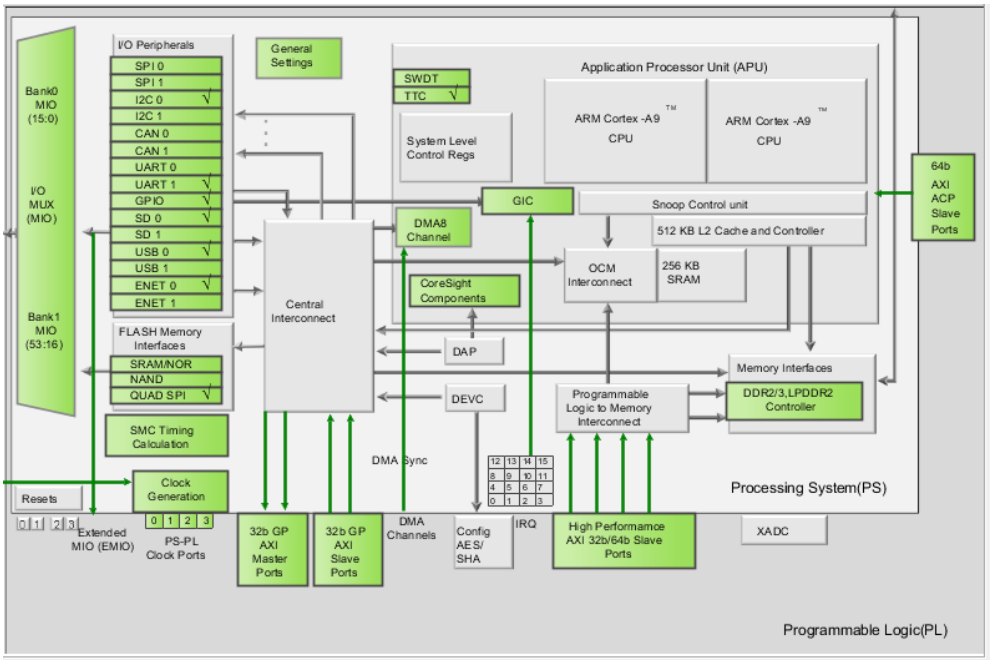
Zynq se sastoji od dva glavna dijela, procesnog sustava (engl. *Processing System*, skraćeno PS) formiranog oko dvojezgrenog ARM *Cortex*-A9 procesora i programibilne logike (engl. *Programmable Logic*, skraćeno PL), što je ekvivalentno FPGA (Slika 2.14.). Također sadrži integriranu memoriju, razne periferne jedinice i brza komunikacijska sučelja.



Slika 2.14.: Pojednostavljeni model Zynq arhitekture. [17]

PL je idealan za implementaciju brzih logičkih, aritmetičkih podsustava i podsustava protoka podataka, dok PS podržava programske rutine i/ili OS, što znači da se ukupna funkcionalnost bilo kojeg dizajniranog sustava može na odgovarajući način podijeliti između HW-a (engl. *hardware*) i SW-a (engl. *software*). Veze između PL i PS se ostvaruju preko standardnog AXI (engl. *Advanced eXtensible Interface*) sučelja [17].

Vivado IDE, koji je opisan u poglavlju 2.7.1, omogućuje detaljniji prikaz arhitekture *Zynq* AP SoC-a, dvostrukim klikom miša na *ZYNQ7 Processing System* blok u sklopu blok dizajna. Kao što se vidi na slici 2.15. PS se sastoji od mnogih komponenti, uključujući APU (engl. *Application Processing Unit*), AMBA (engl. *Advanced Microcontroller Bus Architecture*) interkonekcije, DDR3 upravljača memorije i razne periferne upravljače s njihovim ulazima i izlazima multipleksiranim na 54 namjenske nožice, tzv. MIO (engl. *Multiplexed I/O*) nožice. Periferni upravljači koji nemaju svoje ulaze i izlaze spojene na MIO nožice mogu ih umjesto toga usmjeriti kroz PL, preko EMIO (engl. *Extended MIO*) sučelja. Periferni upravljači povezani su s procesorima kao podređeni (engl. *slave*) preko AMBA interkonekcije i sadrže čitljive/upisive upravljačke registre koji su adresabilni u memorijskom prostoru procesora. PL također može generirati prekide na procesorima i izvesti DMA pristup DDR3 memoriji [16].



Slika 2.15.: Detaljni prikaz arhitekture Zynq AP SoC-a.

2.3.2. Konfiguracija Zynq-a

Za razliku od *Xilinx* FPGA uređaja, AP SoC uređaji poput *Zynq*-7010 dizajnirani su oko procesora, koji djeluje kao nadređeni (engl. *master*) PL-a i svih ostalih perifernih uređaja u PS-u. Zbog toga je postupak pokretanja *Zynqa* sličniji pokretanju mikroupravljača nego FPGA-a. Ovaj proces uključuje učitavanje procesora i izvođenje *Zynq* slike za pokretanja sustava (engl. *Boot Image*) se kreira u *Vivadu* i *Xilinx* SDK-u, a sadrži FSBL (engl. *First Stage Boot Loader*), sekvencu bitova (engl. *bitstream*) za konfiguraciju PL-a i korisničku aplikaciju.

ZYBO podržava tri različita načina pokretanja: *microSD*, QSPI *Flash* i JTAG. Način pokretanja odabran je pomoću *jumpera* JP5 na platformi koji se još naziva *Mode jumper*, koji utječe na stanje *Zynq* konfiguracijskih nožica nakon uključivanja.

Proces pokretanja sustava se, prema [16], može podijeliti u tri faze, a počinje nakon što se ZYBO uključi ili se *Zynq* resetira (unutar programa ili pritiskom na tipkalo PS-SRST). Jedan od procesora (CPU0) započinje s izvođenjem unutarnjeg dijela kôda koji se zove *BootROM*. Ako i samo ako je *Zynq* tek uključen, *BootROM* prvo sprema stanja nožica s *Mode jumpera* u registar koji određuje načina rada (engl. *mode register*). Ako se *BootROM* izvodi zbog resetiranja, tada se navedena radnja ne izvodi, nego se koristi prethodno određeni način rada. To znači da ZYBO treba isključiti i ponovno uključiti kako bi registrirao bilo kakve promjene na *Mode jumperu*. Zatim *BootROM* kopira FSBL iz nepromjenjive memorije, koju je *mode* registar specificirao, u 256 KB interne radne memorije (engl. *Read Access Memory*, skraćeno RAM) unutar APU-a, koja se još

naziva OCM (engl. *On-Chip Memory*). FSBL mora biti unutar Zynq *Boot Imagea* kako bi ga *BootROM* pravilno kopirao. Na kraju početne faze *BootROM* predaje izvođenje FSBL-u.

U ovoj fazi, FSBL prvo dovršava konfiguriranje PS komponenti, poput DDR upravljača. Zatim, ako je *bitstream* prisutan u *Zynq Boot Imageu*, on se čita i koristi za konfiguriranje PL-a. Nakon toga se korisnička aplikacija učitava u memoriju sa *Zynq Boot Image* i predaje joj se izvođenje. Posljednja faza uključuje izvođenje korisničke aplikacije koju je učitao FSBL.

2.4. Python biblioteke

U ovom potpoglavlju su opisane najvažnije biblioteke koje su korištene za razvoj grafičkog korisničkog sučelja (engl. *Graphical User Interface*, skraćeno GUI) ZYNN sustava.

2.4.1. Keras

Keras je API (engl. *Application Programming Interface*) za NN-ove visoke razine otvorenog kôda napisan u programskom jeziku *Python*, a može se koristiti preko *TensorFlow*, CNTK (*Microsoft Cognitive Toolkit*) ili *Theano* biblioteke. U izradi ovog rada je u pozadini korišten *TensorFlow*. Razvijen je kao dio projekta ONEIROS (*Open-end Neuro-Electronic Intelligent Robot Operating System*). s ciljem brzog eksperimentiranja s NN-ovima [18].

Prema [3] glavne značajke, zbog kojih je ova biblioteka odabrana za dizajniranje i treniranje NN-ova korištenih u ovom radu, su:

- modularnost – modeli se grade kao slijed ili graf samostalnih modula koji se mogu kombinirati poput LEGO blokova za izgradnju neuronskih mreža,
- minimalizam – biblioteka je napisana u *Pythonu* i svaki modul je kratak i samo-opisan,
- laka proširivost – jednostavno dodavanje novih funkcionalnosti i modula.

Naime, *Keras* unaprijed definira vrlo velik broj modula koji implementiraju različite vrste neuronskih slojeva, funkcije troška (engl. *cost function*), algoritama optimizacije (engl. *optimizer*), sheme inicijalizacije, aktivacijske funkcije i sheme regularizacije [3]. Navedene pozadinske biblioteke su sposobne izvoditi učinkovite simboličke izračune s tenzorima, koji nisu ništa drugo nego višedimenzionalni niz ili matrica. a ujedno su i temeljni blokovi za stvaranje neuronskih mreža. Osnovna struktura podataka, koju koristi *Keras*, je model, koji označava način na koji su slojevi organizirani. Prema [18] postoje dvije vrste modela:

- sekvencijalni – jednostavniji tip modela u kojem su slojevi sekvencijalno složeni slično kao kod stogova i redova,
- funkcionalni – koristi se za složenije arhitekture NN-ova kao što su usmjereni aciklički grafovi, modeli sa zajedničkim slojevima i modeli s više izlaza [3].

Slika 2.16. prikazuje kako se sekvencijalni model može kreirati prosljeđivanjem liste instanci slojeva konstruktoru ili dodavanjem slojeva pozivanjem *add()* metode. Prvi sloj u modelu mora dobiti informaciju o obliku ulazne informacije, dok ostali slojevi znaju u kojem obliku će dobiti podatke na ulazu na osnovu izlaza iz prethodnog sloja.

Linija	Kôd
1:	`from keras.models import Sequential`
2:	`from keras.layers import Dense, Activation`
3:	
4:	`model = Sequential([`
5:	`    Dense(30, input_shape=(5,)),`
6:	`    Activation('relu'),`
7:	`    Dense(2),`
8:	`    Activation('softmax'),`
9:	`])`
10	
11:	`model2 = Sequential()`
12:	`model2.add(Dense(30, input_shape=(5,)))`
13:	`model2.add(Activation('relu'))`
14:	`model2.add(Dense(2))`
15:	`model2.add(Activation('softmax'))`

Slika 2.16.: Dva načina dodavanja modela slojeva sekvencijalnom modelu.

Tim kôdom se kreiraju jednostavni identični sekvencijalni modeli NN-a s po jednim ulaznim, skrivenim, i izlaznim slojem. Ulazni sloj ima 5 ulaza, što je definirano sa svojstvom *input_shape*. Prvi navedeni *Dense*, koji se je detaljnije opisan u sljedećem potpoglavlju, se odnosi model potpuno povezanog sloja koji sadrži 30 neurona s *relu* aktivacijskom funkcijom. Izlazni sloj modela je druga *Dense* instanca, a budući da koristi *softmax* aktivacijsku funkciju i sastoji se od dva neurona, ovaj model NN-a može klasificirati podatke s dvije klase. Prije treniranja modela, treba se konfigurirati proces učenja, što se vrši metodom *compile()*. Navedenoj metodi se predaju:

- algoritam optimizacije – to može biti identifikator postojećeg algoritma u obliku teksta (npr. *adam*, *rmsprop* ili *adagrad*) ili instanca klase *Optimizer*,
- funkcija troška – cilj koji model pokušava minimizirati uz pomoć algoritma optimizacije, također može biti zadana identifikatorom postojeće funkcije ili ugrađenom funkcijom troška, za treniranje NN-ova za ovaj sustav se koristi pogreška unakrsne entropije (engl. *cross entropy loss*), koja se često koristi u rješavanju problema klasifikacije s više klasa,
- popis metrika – za problem klasifikacije ovaj parametar se postavlja na '*accuracy*', koji se izračunava kao odnos točno klasificiranih slika i ukopnog broja slika, osim toga, može biti zadan identifikatorom postojeće metričke funkcije ili prilagođenom metričkom funkcijom.

Korištena funkcija troška, točnog naziva *sparse categorical crossentropy*, služi za izračun pogreške pri klasifikaciji s više klasa, a *sparse* se odnosi na izlaze definirane integer vrijednošću, kod obične unakrsne entropije izlaz predstavlja vektor veličine broja izlaza, pri čemu broj jedan poprima vrijednost na indeksu klasificirane klase, dok ostale vrijednosti poprimaju vrijednost nula. Vrijednost funkcije troška se računa prema izrazu (2-5), pri čemu *N* označava broj primjeraka unutar skupa, a $p_{i,y}$ dobivenu vjerojatnost klasifikacije za stvarnu labelu *y* [12].

$$-\frac{1}{N}\sum_{i=0}^{N} ln\,(p_{i,y}) \tag{2-5}$$

Model se trenira metodom na *fit()* koja kao argumente prima ulazne podatke i label iz skupa za treniranje u obliku *Numpy* nizova, osim toga je moguće zadati broj epoha, validacijski skup podataka, listu funkcija povratnog poziva (engl. *callback function*), *verbosity* način rada, i tako dalje. Metoda vraća objekt klase *History* gdje su zabilježene vrijednosti pogreške i metričke vrijednosti uspješnih epoha uz iste te vrijednosti dobivene na validacijskom skupu.

Istrenirana mreža se može procijeniti pozivom metode *evaluate()* kojoj se kao osnovni argumenti predaju ulazni podatci i labele iz skupa za testiranje u obliku *Numpy* nizova, a kao vraća vrijednosti pogreške u obliku skalara.

Generiranje izlaznih predikcija za ulazne uzorke se koristi metoda *predict()*, kojoj se, u osnovnom slučaju, predaje *Numpy* niz novih ulaznih podataka. Metoda vraća *Numpy* niz predviđenih labela [18].

Postoje unaprijed izgrađeni slojevi *Kerasa*, od kojih su u ovom radu korišteni:

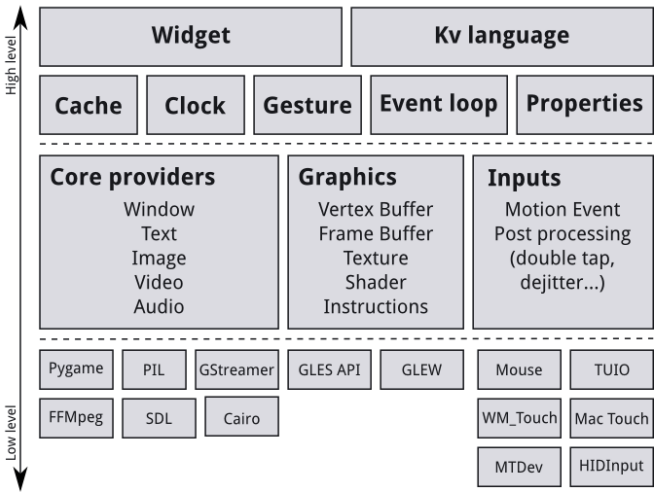
- *Dense* – model potpuno povezanog sloja,
- *Activation* – ovaj sloj primjenjuje zadanu aktivacijsku funkciju na izlaz prethodnog sloja
- *Flatten* – ovaj sloj služi za ispravljanje tenzora, drugim riječima višedimenzionalni tenzor pretvara u jednodimenzionalni,
- *Conv2D* – model konvolucijskog sloja,
- *MaxPooling2D* – model sloja sažimanja maksimalnom vrijednošću [18].

2.4.2. Kivy

Kivy je višeplatformska *Python* biblioteka otvorenog kôda koja se može koristiti za brzi razvoj korisničkih sučelje (engl. *User Interface*, skraćeno UI). *Kivy* se koristi preko *Pythona* verzije 2.7 ili novije, tako da postoje sve uobičajene *Python* značajke, ali s dodatnom *Kivy* funkcionalnošću. *Kivy* je napisan u *Pythonu* i *Cythonu*. *Cython* je optimizirajući statički kompajler (engl. *compiler*) za programski jezik *Python* [20], a u ovoj biblioteci se koristi za ubrzanje vremenski zahtjevnih komponenti biblioteke uključujući komponente povezane s grafičkim

prikazivanjem, izračunavanjem matrice, upravljanje događajima i svojstvima. *Kivy* podržava rad na *Android*, *iOS*, *Windows*, *Linux*, OS X platformama i na *Raspberry Pi*-ju [19].

Kao što je opisano u [19] i prikazano slikom 2.17., ključna ideja za razumijevanje *Kivyja* je ideja o modularnosti i apstrakciji. Osnovni zadatci (engl. *core tasks*) poput otvaranja prozora, prikazivanja slika i teksta, reprodukcije zvuka, dobivanja slika s fotoaparata, ispravljanja pravopisa, itd. su apstrahirani, što čini API jednostavnim za korištenje i lako proširivim.



Slika 2.17.: Prikaz arhitekture Kivy biblioteke. [19]

Dio kôda koji koristi specifične API-je za razgovor s OS-om s jedne strane i *Kivyjem* s druge, može se reći posredni komunikacijski sloj, se naziva osnovni davatelj usluga (engl. *core provider*). Prednost upotrebe specijaliziranih *core providera* za svaku platformu je ta što se mogu u potpunosti iskoristiti funkcionalnosti OS-a i djelovati što je moguće efikasnije. Nadalje, korištenjem biblioteka bilo koje platforme učinkovito se smanjuje veličina *Kivy* distribucije.

Isti koncept je primijenjen na upravljanje ulazima. Davatelj unosa (engl. *input provider*) je dio kôda koji daje podršku određenom ulaznom uređaju, kao što je miš, tipkovnica, *Apple Trackpad*, TUIO (engl. *Tangible User Interface Objects*) ili emulator miša. Po potrebi se može dodati podrška za novi ulazni uređaj, jednostavno se kreira nova klasa koja čita ulazne podatke s uređaja i pretvara ih u osnovne *Kivy* događaje.

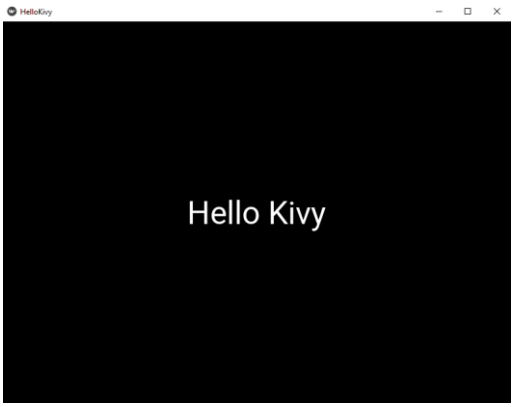
Grafički API je apstrakcija *OpenGL*-a [21], kojemu se na najnižoj razini izdaju hardverski ubrzane naredbe za crtanje. Svi dostupni *widgeti* koriste ovaj API, a implementiran je u C-u zbog boljih performansi.

Kôd u osnovnom paketu pruža najčešće korištene značajke, što uključuje:

- *Widget* – UI elementi koji se dodaju radi pružanja neke funkcionalnosti,
- *Kivy Language* – *Kivy* jezik se koristi za jednostavno i efikasno opisivanje UI-ja,
- *Cache* – pohrana *Python* objekata, može se zadati vremensko ili količinsko ograničenje,
- *Clock* – zakazivanje periodičnih i aperiodičnih događaja,
- *Gesture* – prepoznavanje pokreta, kao što su krugovi ili pravokutnici,
- *Event loop* - obrađuje ažuriranje ulaznih događaja i prosljeđivanje zainteresiranim stranama,
- *Properties* – klasa svojstava koja povezuju kôd *widgeta* s opisom UI-ja.

Osnovna klasa za kreiranje *Kivy* aplikacija je *App* klasa, koja služi za pokretanje *Kivy* petlje. Aplikacija *Kivy* ne kreira se instanciranjem klase *App* već instanciranjem podređene klase koja proširuje klasu *App*. Aplikacija se zatim pokreće pozivom metode *run()* pomoću stvorene instance. *Kivy* se koristi za izgradnju UI-ja koji se sastoji od niza vizualnih elemenata koji se nazivaju *widgeti*. Između instanciranja klase i njezinog pokretanja moraju se definirati *widgeti* i njihov izgled. Metoda *build()* koja je nadjačana (engl. *override*) iz klase *App*, vraća stablo koje sadrži sve ostale *widgete* UI-a, a glavni *widget* stabla se još naziva i *root widget* [6],[19].

Slika 2.18. prikazuje kôd osnovne *Kivy* aplikacije. Na početku se uvoze potrebni moduli iz *Kivyja*. Klasa *App* je već opisana, a klasa *Label* koja služi kao *widget* za prikaz teksta. Metodom *build()* se kreira instanca labele kojoj se zatim zadaju svojstva *text* i *font_size*. Nakon stvaranja podređene klase *HelloKivyApp* stvara se instanca iz te klase. Zatim se instancom poziva metoda *run()*, te se otvara prozor aplikacije prikazan na slici 2.19.

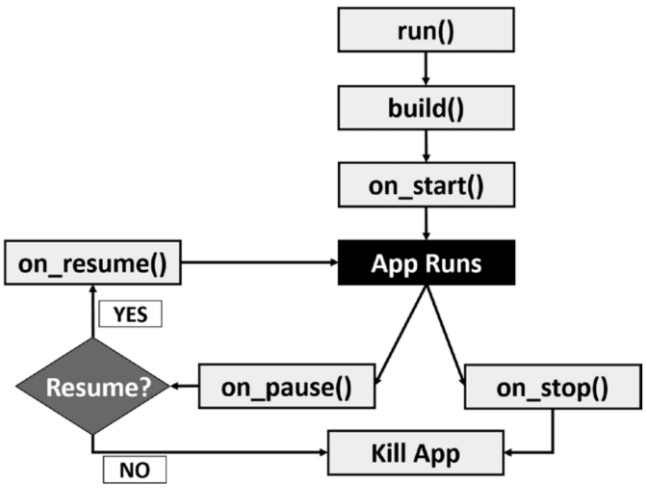


Slika 2.19.: Izgled osnovne aplikacije.

Linija	Kôd
1:	`import kivy`
2:	`kivy.require('1.11.1')`
3:	
4:	`from kivy.app import App`
5:	`from kivy.uix.label import Label`
6:	
7:	`class HelloKivyApp(App):`
8:	`    def build(self):`
9:	`        widget = Label()`
10:	`        widget.text = "Hello Kivy"`
11:	`        widget.font_size = 50`
12:	`        return widget`
13:	
14:	`helloKivy = HelloKivyApp()`
15:	`helloKivy.run()`

Slika 2.18.: Kôd osnovne aplikacije.

Životni ciklus (Slika 2.20.) započinje pokretanjem aplikacije primjenom metode *run()*. Nakon toga se izvodi metoda *build()*, koja vraća *widgete* za prikaz. Prije pokretanja aplikacije poziva se metode *on_start()*. Aplikacija se može pauzirati ili stopirati. Ako je pauzirana, tada se poziva metoda *on_pause()*, a pri nastavljanju aplikacije se poziva metoda *on_resume()*. Ako se ne nastavi, aplikacija se zaustavlja. Aplikacija se također može izravno zaustaviti bez pauziranja, a u tom slučaju se poziva metoda *on_stop()*. Sve spomenute metode se mogu nadjačati iz *App* klase i koristiti za izvođenje željenih radnji.



Slika 2.20.: Životni ciklus Kivy aplikacije. [6]

Kada je podređena klasa na kraju naziva ima „*App*“, *Kivy* ostatak gleda kao naziv aplikacije, što je vidljivo na primjeru jednostavne *Kivy* aplikacije (Slika 2.19.), pri čemu je naslov prozora *HelloKivy*, a naziv klase *HelloKivyApp*. Samo treba imati na umu da „*App*“ mora početi s velikim slovom, u suprotnom se cijeli naziv klase uzima kao naziv aplikacije. Naslov se također može zadati kao argument konstruktora klase, koji osim toga prihvaća i putanju do slike koja bi se koristila kao ikona [6].

Iako se *Kivy* aplikacije mogu pisati u cijelosti na *Pythonu*, *Kivy* također nudi vlastiti dizajnerski jezik, tzv. KV jezik, posebno usmjeren prema jednostavnom i skalabilnom UI dizajnu i tako olakšava odvajanje dizajna sučelja od logike aplikacije. Jezik kreira datoteke s „*kv*“ ekstenzijom, tzv. KV datotekama, što se često koristi pri razvoju složenijih aplikacija, odnosno kada se koristi velik broj *widgeta*. KV jezik gradi stablo *widgeta* na jednostavan način koji je lako čitljiv u odnosu na njegovo dodavanje unutar *Python* kôda, što također olakšava uklanjanje pogrešaka [6],[19].

KV datoteka sastoji se od skupa pravila sličnih CSS pravilima koji definiraju *widgete*. Pravila se sastoje od klasa *widgeta* i skupa svojstava s njihovim vrijednostima. Nakon naziva klase dodaje se dvotočka kako bi označio početak sadržaja tog *widgeta*. Sadržaj pod danim *widgetom* je uvučen jednako kao što se u *Pythonu* definira sadržaj blokova. Između naziva svojstva i njegove vrijednosti nalazi se dvotočka. Na primjer, pokretanjem kôda (Slika 2.21.) se postiže isti učinak kao s kôdom na slici 2.18. Svi redovi koji su uvučeni ispod naziva klase pripadaju tom *widgetu*. Naziv KV datoteke postavlja se na ime klase podređene *App* klasi izostavljanjem „*App*“ dijela naziva, kako bi se KV datoteka učitala pri pokretanju *main.py* skripte. Drugi način na koji se može učitati KV datoteka je korištenje modula *Builder* i metode *load_file()* kojoj se kao argument predaje putanja do KV datoteke [6].

Linija	Kôd: main.py	Kôd: HelloKivy.kv
1:	`import kivy`	`#:kivy 1.11.1`
2:	`kivy.require('1.11.1')`	
3:		`<Label>:`
4:	`from kivy.app import App`	`    text: "Hello Kivy"`
5:	`from kivy.uix.label import Label`	`    font_size: 50`
6:		
7:	`class HelloKivyApp(App):`	
8:	`    def build(self):`	
9:	`        return Label()`	
10:		
11:	`helloKivy = HelloKivyApp()`	
12:	`helloKivy.run()`	

Slika 2.21.: Kôd osnovne aplikacije uz korištenje KV datoteke.

2.4.3. Ostale biblioteke

Osim navedenih biblioteka, ostali važniji korišteni moduli su:

- *PySerial* [22] – za serijsku komunikaciju,
- *Threading* [23] – omogućuje rad aplikacija u više niti (engl. *thread*),
- *NumPy* [24] – pruža alate za rad s višedimenzionalnim nizovima visokih performansi,
- *OpenCV* [25] –funkcije usmjerene uglavnom na računalni vid u stvarnom vremenu.

2.5. Obrada slike

Obrada slike (engl. *image processing*) je područje koje se koristi u širokom rasponu aplikacija, prvenstveno u dvije pomalo različite svrhe:

- poboljšanje vizualnog izgleda slika za ljudskog promatrača, uključujući njihov ispis i prijenos,
- pripremanje slika za mjerenje i analizu svojstava i struktura koje otkrivaju [26].

U ovom radu su primjenjuju algoritmi iz ovog područja u svrhu predobrade slike s kamere, što uključuje uklanjanje šuma, binarizaciju, segmentaciju, skaliranje i filtriranje slike.

2.5.1. Sigma filtar

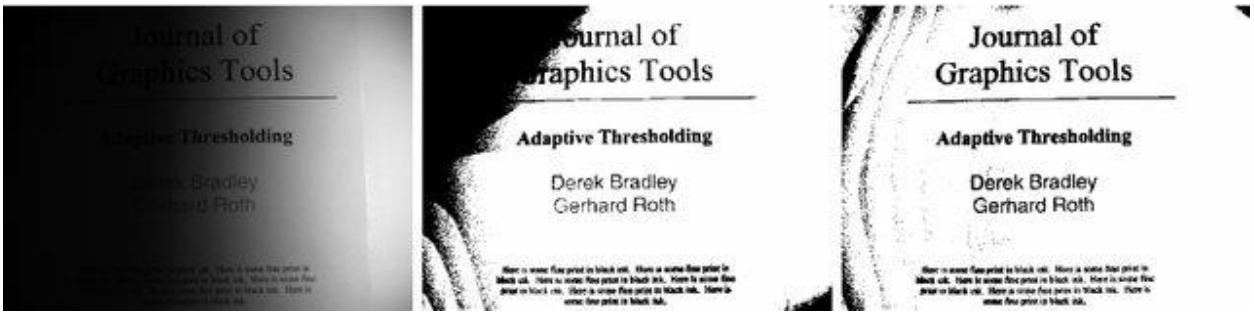
Sigma filtar, koji je detaljno opisan u [27], smanjuje zašumljenost slike na isti način kao i filtar usrednjavanja. Ideja *sigma* filtra je usporediti samo one intenzitete u kliznom prozoru (engl. *sliding window*) koji se razlikuju od intenziteta središnjeg piksela za ne više od fiksnog parametra tolerancije. Prema ovoj ideji, *sigma* filter smanjuje Gaussov šum i zadržava rubove slike nezamućenima, sličan je bilateralnom filtru koji je otprilike četiri puta sporiji od *Sigma* filtra.

Filtar radi tako što se traži skup piksela unutar kliznog prozora čije vrijednosti odstupaju od intenziteta središnjeg piksela u granicama tolerancije. Nakon toga se usrednjavanjem vrijednosti pronađenog skupa određuje intenzitet središnjeg piksela. Postupak je ubrzan korištenjem lokalnog histograma. Histogram je niz u kojem svaki element sadrži broj pojava odgovarajuće vrijednosti ili intenziteta boje u prozoru. *Sigma* filter izračunava histogram za svaku lokaciju kliznog prozora postupkom ažuriranja. Na početku izvođenja, izračunava se histogram kliznog prozora koji se nalazi na početku reda slike, nadalje se izračunavaju granične vrijednosti intenziteta piksela s obzirom na parametar tolerancije te se na kraju intenzitet središnjeg piksela odredi na osnovu srednje vrijednosti piksela između graničnih vrijednosti. Za ostatak reda se u histogram dodaju nove vrijednosti obuhvaćene prozorom, a uklanjaju vrijednosti koje više nisu unutar prozora, nakon čega se određuje intenzitet središnjeg piksela. Za svaki slijedeći red slike se primjenjuje isti postupak.

2.5.2. Bradley Adaptive Tresholding

Metoda praga (engl. *thresholding*) uobičajeni je zadatak u mnogim aplikacijama računalnog vida, a koristi se za binarizaciju slike, odnosno klasifikaciju piksela na svijetle ili tamne, odnosno bijele ili crne. Zbog loših rezultata dobivenih binarizacijom slike globalnim pragom (engl. *global threshold*), implementiran je adaptivni algoritam za binarizaciju slike, koji u obzir uzima prostornu varijaciju osvjetljenja [28].

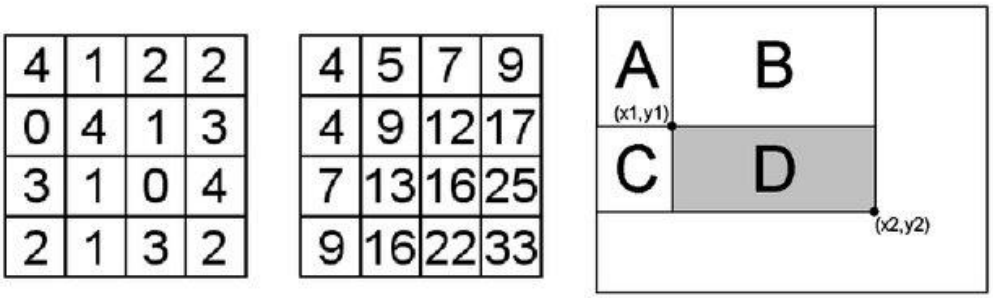
Bradley Adaptive Tresholding algoritam je jednostavno proširenje Wellnerove metode [29], čija usporedba je prikazana je na slici 2.22. Wellnerova metoda je bazirana na izračunu pomičnog prosjeka prolaskom kroz sliku, a prednost joj je što prolazi kroz sliku samo jednom. Međutim, problem kod Wellnerove metode je ovisnost o redoslijedu skeniranja piksela, a i pomični prosjek nije dobar za predstavljanje susjednih piksela na svakom koraku, jer uzorci iz susjedstva nisu ravnomjerno raspoređeni u svim smjerovima. Iz tog razloga *Bradley Adaptive Tresholding* algoritam koristi integralnu sliku, s tim da dodatno obilazi sliku kako bi se ista izračunala, koja je detaljnije opisana u nastavku. Umjesto izračunavanja pomičnog prosjeka, u ovom se algoritmu izračunava prosjek prozora centriranog oko svakog piksela, što se vrši u konstantnom vremenu pomoću integralne slike. Ako je vrijednost trenutnog piksela za zadani postotak manja od prosjeka prozora, tada se piksel postavlja na crnu boju, u protivnom se postavlja na bijelu.



Slika 2.22.: Usporedba Wellnerove metode (sredina) i Bradley Adaptive Threshold algoritma (desno) primijenjenih na slici (lijevo).[28]

Integralna slika (engl. *integral image*) je alat koji se može koristiti za računanje zbroja neke funkcije piksela (npr. intenzitet piksela) u pravokutnoj regiji unutar slike. Za računanje integralne slike, na lokaciju *I(x, y)* se sprema zbroj *f(x, y)* vrijednosti svih piksela lijevo i iznad piksela *(x, y)*, što je prikazano izrazom (2-6). Ilustracija izračuna je prikazana slikom 2.23., pri čemu se vidi ulazna slika te dobivena integralna slika.

$$I(x,y) = f(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1) \tag{2-6}$$



a) *b)* *c)*

Slika 2.23.:Integralna slika: a) primjer ulazne slike, b) izračunate vrijednosti integralne slike, c) izračun zbroja vrijednosti unutar pravokutnika D pomoću integralne slike u konstantnom vremenu.[28]

Jednom kada se dobije integralna slika, zbroj funkcija bilo kojeg pravokutnika (npr. pravokutnik *D* na slici 2.23.) s gornjim lijevim kutom *(x1, y1)* i donjim desnim kutom *(x2, y2)* može se izračunati u konstantnom vremenu pomoću izraza (2-7), što je ekvivalentno računanju zbroja vrijednosti unutar pravokutnika *(A+B+C+D) - (A+B) - (A+C)+A*.

$$\sum_{x=x_1}^{x_2}\sum_{y=y_1}^{y_2} f(x,y) = I(x2,y2) - I(x2,y1-1) - I(x1-1,y2) + I(x1-1,y1-1) \qquad (2\text{-}7)$$

2.5.3. Median filtar

Median filtar spada u klasu nelinearnih filtara koji sortira intenzitet piksela kliznog prozora i mijenja vrijednost središnjeg piksela s medijanom odnosno centralnom vrijednošću sortirane sekvence. *Median* filtri se koriste za uklanjanje impulsnog šuma (engl. *impulsive noise*) uz očuvanje rubova i detalja slike [27].

2.5.4. Dilatacija

Dilatacija (engl. *dilation*) je, uz eroziju (engl. *erosion*), temeljna operacija za morfološku obradu slike. Ova operacija se koristi za povećavanje objekata na sivoj ili binarnoj slici, pri čemu je način i opseg povećanja zadan oblikom i veličinom prozora. Kada se primjenjuje na sivoj slici izlazna slika se računa prema izrazu (2-8), pri čemu *f* predstavlja ulaznu sliku, *b* predstavlja prozor kojim se prelazi preko slike, dok je $\hat{b}$ prikazan izrazom (2-9), element izlazne slike na lokaciji *(x, y)* poprima vrijednost maksimuma iz prozora centriranog oko tog piksela na ulaznoj slici [30].

$$[f \oplus b](x,y) = \max_{(s,t)\in\hat{b}}\{f(x-s,y-t)\} \qquad (2\text{-}8)$$

$$\hat{b}(x,y) = b(-x,-y) \qquad (2\text{-}9)$$

2.5.5. Erozija

Za razliku od dilatacije, ova operacija se koristi za smanjivanje objekata na sivoj ili binarnoj slici. Operacija je prikazana izrazom (2-10) te funkcionira na isti način kao dilatacija, jedino što se umjesto maksimuma traži minimalna vrijednost unutar prozora.

Na temelju dvije spomenute morfološke operacije, razvijeni su razni morfološki algoritmi, primjerice, otvaranje (engl. *Opening*) i zatvaranje (engl. *Closing*) [30].

$$[f \ominus b](x, y) = \min_{(s,t)\in b}\{f(x+s, y+t)\} \tag{2-10}$$

2.5.6. Bilinearna interpolacija

Interpolacija je postupak procjene vrijednosti na nepoznatim lokacijama na osnovu poznatih podataka, a koristi se u zadacima kao što su zumiranje, smanjivanje, rotiranje i geometrijsko ispravljanje digitalnih slika [30].

Najjednostavnija metoda je interpolacija najbližeg susjeda (engl. *nearest neighbor interpolation*), jer svakoj novoj lokaciji dodjeljuje intenzitet najbližeg susjeda na izvornoj slici. Ovaj je pristup jednostavan, ali ima tendenciju stvaranja nepoželjnih artefakata, poput ozbiljnih izobličenja ravnih rubova, ovaj fenomen se još naziva *aliasing*.

Prikladniji pristup je bilinearna interpolacija (engl. *bilinear interpolation*) u kojoj se koristi četiri najbliža susjeda za estimaciju intenziteta na određenoj lokaciji. Neka su *(x, y)* oznake koordinata točke kojoj se želi dodijeliti vrijednost intenziteta i neka je *v(x, y)* vrijednost intenziteta. Za bilinearnu interpolaciju, dodijeljena vrijednost dobiva se prema (2-11), pri čemu su četiri koeficijenta određena iz četiri jednadžbe s četiri nepoznanice koje se mogu napisati pomoću četiri najbliža susjeda točke *(x, y)*. Ovaj pristup daje mnogo bolje rezultate od interpolacije najbližeg susjeda, uz skromno povećanje računalnog opterećenja. Suprotno onome što ime sugerira, bilinearna interpolacija nije linearna operacija jer uključuje množenje koordinata [30].

$$v(x, y) = ax + by + cxy + d \tag{2-11}$$

2.5.7. CCL algoritam

Jedan od najčešćih problema koji se susreću u analizi slike je utvrđivanje fizički povezanih dijelova slike. CCL algoritam (engl. *Connected Component Labeling*) su 1966. godine dizajnirali Rosenfeld i Pfalz [31], a koristi *union-find* strukturu podataka za učinkovito rješavanje ovog problema. *Union-find* je struktura koja podržava spajanje dviju grupa i upit jesu li neka dva elementa u istoj grupi.

Algoritam dva puta prolazi slikom. Pri prvom prolasku algoritam prolazi kroz svaki piksel, provjeravajući piksel iznad i piksel lijevo, te na osnovu oznaka tih piksela dodjeljuje oznaku trenutnom pikselu. Drugi prolazak slikom služi za čišćenje nereda koji je nastao pri dodjeli oznaka, poput više različitih oznaka za iste komponente [32].

2.6. SJCAM SJ4000 WiFi

SJCAM SJ4000 *WiFi* (Slika 2.24.) je akcijska kamera kineske proizvodnje, pristojnih specifikacija, koja se koristi kao izvor HDMI signala rezolucije 640×480, a neke od bitnijih značajki su:

- senzor od 12 MP - snimanje oštrih fotografija i videozapisa u *Full* HD rezoluciji,
- 170° širokokutna leća,
- 1.5" 4:3 LCD,
- 4x digitalni zum,
- mikro HDMI izlaz - video prijenos uživo i reprodukcija medija,
- *WiFi* – omogućuje pristup u stvarnom vremenu, daljinsko snimanje i konfiguraciju [33].

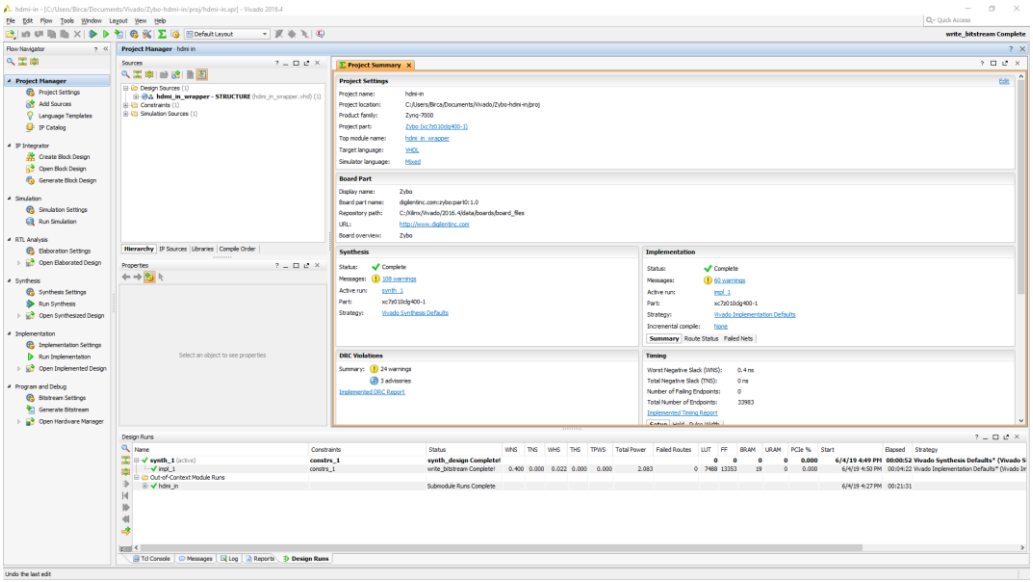
Slika 2.24.: SJCAM SJ4000 WiFi.[33]

2.7. Razvojna okruženja

Za konfiguriranje ZYBO platforme korištena su dva razvojna alata iz *Vivado Design Suite* programskog paketa. Navedeni paket sadrži usluge koje podržavaju sve faze FPGA dizajna, počevši od unosa dizajna, simulacije, sinteze, razmještaj i povezivanje, generiranja *bitstreama*, uklanjanja pogrešaka i provjere, kao i razvoja SW-a namijenjenog tim FPGA-ima. IP (engl. *Intellectual property*) je osnovni element za dizajniranje sustava, a pruža jednostavan mehanizam za uključivanje složene logike u dizajn. IP-evi, koje je razvio *Xilinx*, su optimizirani i testirani za rad s FPGA resursima, uvelike ubrzavajući razvoj dizajna [34].

U *Vivado* IDE alatu je dizajniran sustav za PL, dok je u *Xilinx* SDK alatu razvijena aplikacija za PS. Za razvoj GUI-ja je korišten *Visual Studio Code*, osim toga za vizualizaciju podataka je korišten *Spyder* IDE zbog svojih ugrađenih mogućnosti za uvid u varijable.

2.7.1. Vivado IDE

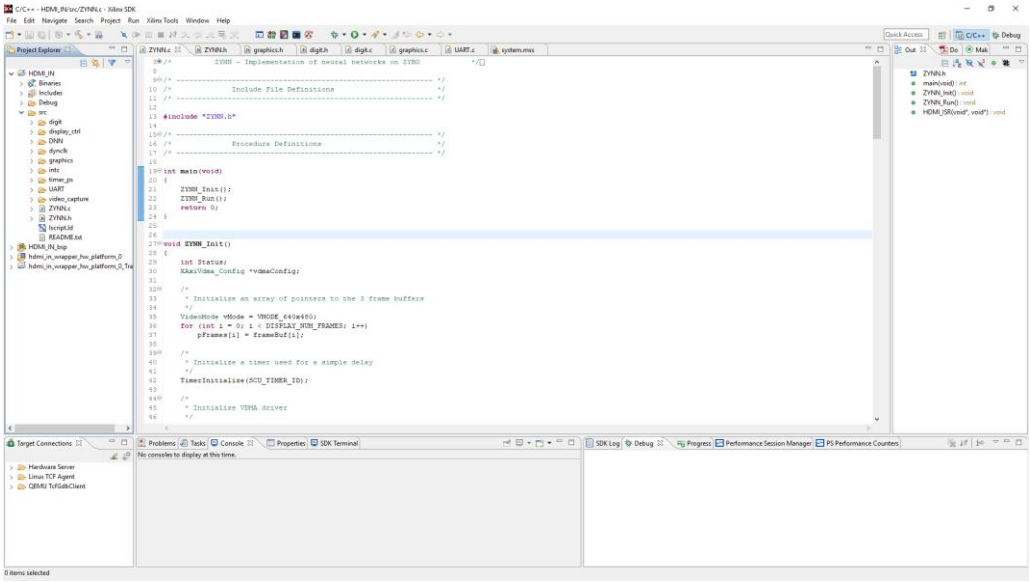


Slika 2.25.: Prikaz Vivado IDE-a.

Vivado IDE (engl. *Integrated Design Environment*) pruža niz značajki za sintezu sustava na FPGA i pomoću ovog alata se definira struktura HW-a na *Zynq* (Slika 2.25.). Sve što se radi u GUI-ju ovog alata, može se učiniti i korištenjem naredbenog jezika TCL (engl. *Tool Command Language*) pokretanjem TCL skripte iz datoteke ili izravnim unošenjem naredbi u TCL terminal. Najbitnije značajke koje podržava *Vivado* IDE:

- IP Integrator – sinteza sustava grafičkim povezivanjem perifernih IP-eva, konfiguriranje postavki HW-a,
- RTL (engl. *Register-transfer level*) – dizajniranje u HDL (engl. *Hardware Description Language)* jezicima VHDL, *Verilog* i *SystemVerilog*,
- *Simulation* – simuliranje ponašanja sustava,
- *Synthesis* – alat za sintezu uređaja 7 serije i novijih,
- *Power Analyser* – procjena potrošnje energije,
- *Logic Analyser* – otklanjanje pogrešaka,
- generiranje *bitstreama* [35].

2.7.2. Xilinx SDK

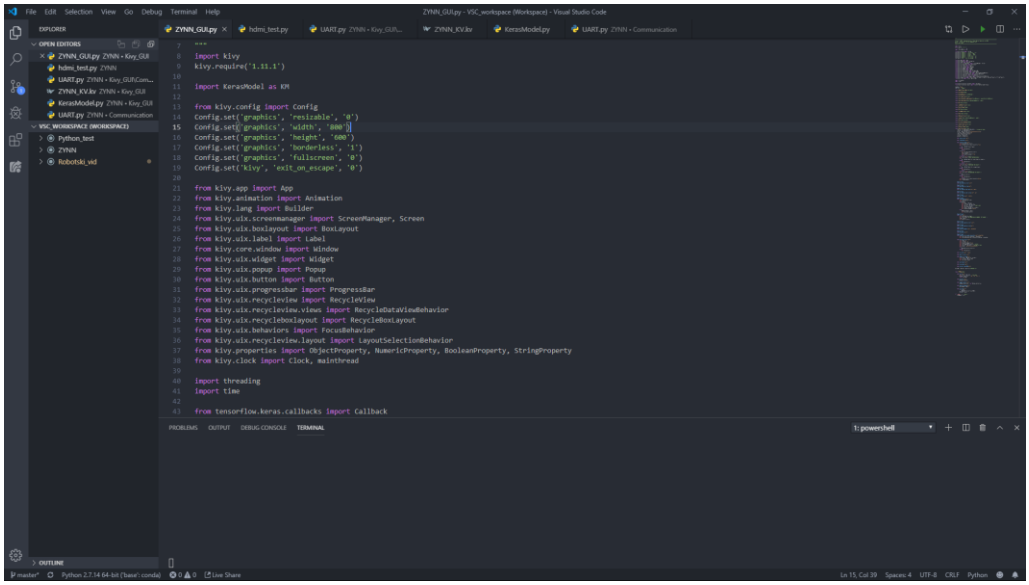


Slika 2.26.: Prikaz Xilinx SDK-a.

Xilinx SDK (engl. *Software Development Kit*) je zasnovan na standardnom *Eclipse* IDE-u koji pruža dobro poznati GUI (Slika 2.26.). Za razliku od običnog *Eclipse* IDE, *Xilinx* SDK je visoko specijalizirano okruženje za razvoj softvera za *Xilinx* ugradbene procesore, uključujući *Zynq*. Podržava samostalne (engl. *bare metal*) aplikacije i aplikacije bazirane na OS-u. Najvažnije značajke ovog alata:

- C/C++ uređivač kôda i okruženje za kompilaciju s praćenjem pogrešaka,
- upravljanje projektima i arhiviranje,
- XMD (engl. *Xilinx Microprocessor Debugger*) koji se koristi za komunikaciju sa *Xilinx* ugradbenim procesorima pomoću JTAG-a,
- SoC programator koji se koristi za programiranje *Xilinx* SoC-eva s *bitstreamom*,
- *flash* programator i generator FPBL-a, koji se koriste za automatsko pokretanje sustava iz *flash* memorije,
- generator skripte za povezivanje (engl. *linker script*) za mapiranje slike aplikacije u memorijski prostor HW-a [35].

2.7.3. Visual Studio Code



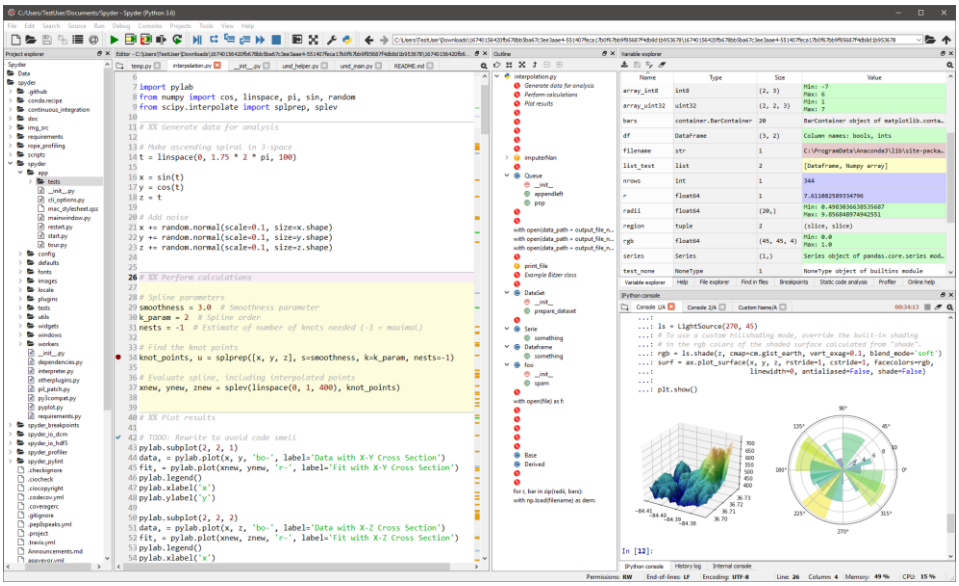
Slika 2.27.: Prikaz Visual Studio Code-a.

Visual Studio Code je uređivač teksta otvorenog kôda sa značajkama IDE-a koji je razvio *Microsoft* 2015. godine za *Windows*, *Linux* i *MacOS* operacijske sustave i nije u nikakvoj povezanosti s *Visual Studio* IDE-om (Slika 2.27.). Vrlo je prilagodljiv i omogućuje korisnicima mijenjanje teme, prečaca na tipkovnici, postavki i instaliranje proširenja za dodatne funkcionalnosti. Dolazi s ugrađenom podrškom za *JavaScript*, *TypeScript* i *Node.js* i ima bogat sustav proširenja za druge jezike kao što su C ++, C #, *Java*, *Python*, PHP, *Go* i *runtime* kao što su .NET i *Unity*. Značajke ovog razvojnog sučelja su:

- *IntelliSense* – skup značajki koje kodiranje čine praktičnijim, uključujući dovršavanje, nagovještavanje kôda i davanje informacija o funkcijama i argumentima funkcije,
- *terminal* – ima ugrađeni terminal koji je isti kao i zadani terminal,
- *git* integracija – prate se promjene izvornog kôda za svaki projekt koji je unutar *git* repozitorija,
- ispravljanje pogrešaka – ugrađeni *debugger* ubrzava proces uređivanje kôda i ispravljanje pogrešaka [36].

Ove godine je na anketi za programere *StackOverflowa* ovaj alat rangiran kao najpopularnije razvojno okruženje [37].

2.7.4. Spyder IDE



Slika 2.28.: Prikaz Spyder IDE-a. [39]

Spyder (engl. *Scientific Python Development Environment*) je moćno znanstveno okruženje napisano na *Pythonu*, za *Python*, a dizajnirano od strane znanstvenika, inženjera i analitičara podataka 2.28. Sadrži jedinstvenu kombinaciju napredne funkcije uređivanja, analize, uklanjanja pogrešaka i profiliranja sveobuhvatnog razvojnog alata s istraživanjem podataka, interaktivnim izvođenjem, dubinskim pregledom i mogućnostima vizualizacije znanstvenog paketa [38]. Ključne značajke ovog IDE-a su:

- *editor* – isticanje sintakse, dovršavanje kôda, interaktivno izvođenje kôda,
- *IPython* konzola – pokretanje više individualnih konzola,
- *Variable Explorer* – omogućuje uvid u varijable i promjenu istih iz GUI-ja,
- *Help* – pruža dokumentaciju ili izvorni kôd za bilo koji *Python* objekt,
- *Static Code Analysis* – detektira potencijalne greške i druge probleme s kôdom,
- *Profiler* – mjeri učinak svake funkcije kako bi se pronašle slabosti i postigla optimizacija,
- *History log* - kronološki bilježi svaku naredbu upisanu u bilo koju konzolu *Spydera* vremenskim oznakama, isticanjem sintakse s mogućnosti ponovnog odabira naredbe [39].

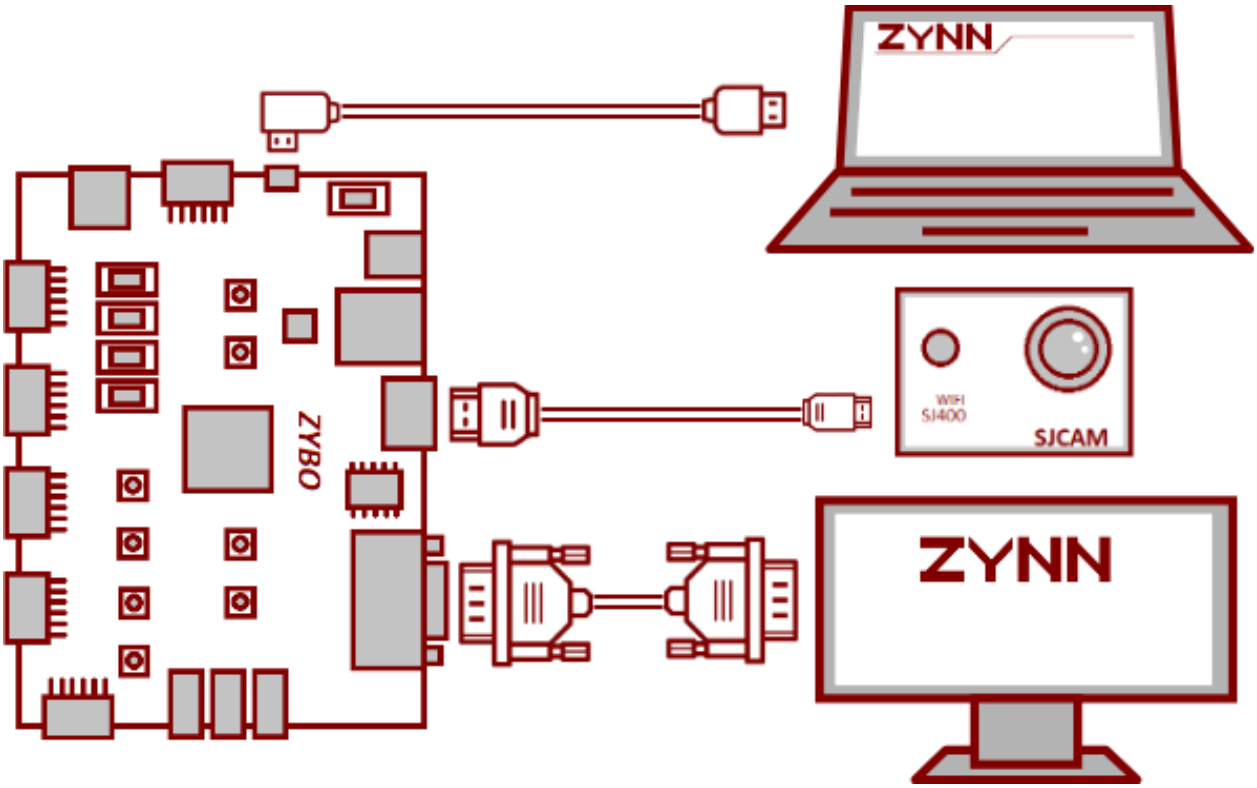
2.8. Ostali programski alati

Ostali programski alati koji su korišteni pri izradi ovog rada su:

- *Tera Term* [40] – za serijsku komunikaciju sa ZYBO platformom,
- *Terminal* [41] – također za serijsku komunikaciju, ali ovaj alat ima dodatnu mogućnost prikaza primljenih paketa u binarnom obliku

3. REALIZACIJA ZYNN SUSTAVA

ZYNN sustavom se upravlja preko GUI-ja na računalu, pomoću kojeg se dizajniraju i treniraju *Keras* modeli ili učitavaju parametri postojećih modela, koji se zatim šalju na ZYBO platformu. ZYBO u međuvremenu preko HDMI kabela prima sliku s kamere, koju ujedno prikazuje na monitoru koji je spojen preko VGA sučelja. Nakon što su primljeni svi parametri *Keras* modela, ZYBO čeka na upravljačke naredbe od strane GUI-ja, kojima se daje signalizacija za početak procesa traženja i prepoznavanja znamenaka na slici s kamere. Nakon što ZYBO izvede taj proces, dobivene rezultate šalje računalu te na monitoru prikazuje sliku u binarnom obliku s detektiranim znamenkama. Nakon primanja rezultata, korisnik može ponoviti postupak za detekciju znamenaka u bilo kojem trenutku. Slika 3.1. prikazuje način na koji su povezani najbitniji dijelovi ZYNN-a. U nastavku ovog poglavlja je detaljnije objašnjen način realizacije ovog sustava.



Slika 3.1.: Grafički prikaz ZYNN sustava.

3.1. Implementacija na ZYBO

Glavna komponenta ZYBO platforme, *Zynq*, je sastavljena od PL dijela, koji sadrži FPGA, i PS dijela, koji sadrži dvojezgreni ARM *Cortex*-A9. U *Vivado* IDE-u se sintetizira cijeli sustav, odnosno konfigurira se korišteni HW, dok se u *Xilinx* SDK razvija aplikacije u C programskom jeziku koja se pogoni na PS-u. Oba rješenja su objašnjena u nastavku ovog potpoglavlja, a datoteke projekta su dostupne u prilogu P.1.

3.1.1. Ugradbeni sustav

Osnova za razvoj ZYNN sustava je *HDMI Input* projekt koji demonstrira upotrebu HDMI i VGA portova na ZYBO platformi. U tom projektu se video podaci s HDMI porta prosljeđuju na VGA port, a preko UART sučelja se konfigurira prikaz [42].

Budući da su je to sve što je potrebno za rad ZYNN sustava, identičan blok dizajn, koji je prikazan u prilogu P.3, je korišten pri razvoju ovog sustava, a najvažniji IP-evi dizajna su:

- *ZYNQ7 Processing System* – omogućava konfiguraciju i kontrolu ZYNN sustava, dok se DDR memorija koristi kao međuspremnik okvira (engl. *frame buffer*),
- *DVI to RGB Video Decoder*– pretvara HDMI video rezolucije 640×480 u 24-bitni RGB video s odgovarajućim vertikalnim i horizontalnim signalima sinkronizacije,
- *AXI GPIO* – izlaz koji se koristi za otkrivanje signala na HDMI portu (engl. *hot plug detect*),
- *Video Timing Controller* – instanca „*v_tc_0*“ je konfigurirana kao izvor vremena, dok instanca „*v_tc_1*“ detektira način rada videozapisa primljenog od HDMI izvora,
- *Video In to AXI4-Stream* – pretvara paralelni video signal u *AXI4-Stream* sučelje,
- *AXI Video Direct Memory Access* – omogućuje spremanje *AXI4-Stream* videa u PS DDR memoriju u obliku mapirane AXI memorije i čitanje za dobivanje izlaznog videa,
- *AXI4-Stream to Video In* – suprotno od *Video In to AXI4-Stream* IP-a
- *RGB to VGA output* – 24-bitni RGB video pretvara u VGA video format [43].

Kako bi se mogla razvijati aplikacija u *Xilinx* SDK-u, prvo se generira *bitstream*, što može potrajati nekoliko desetaka minuta, i na kraju se izvozi HW platforma za SW razvojne alate uključujući generirani *bitstream*.

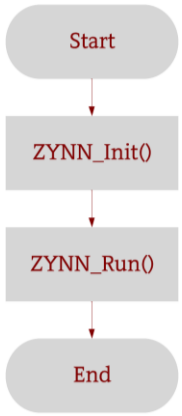
3.1.2. Programska podrška

Osnova aplikacije je izmijenjeni *HDMI Input* projekt. Budući da je razvijana kompleksna aplikacija, napisano je nekoliko pomoćnih biblioteka kako bi se olakšalo pisanje kôda i kako bi taj kôd bio pregledniji:

- DNN (skraćeno od *Deep* NN) – u ovoj biblioteci se izvode sve radnje u vezi modela NN, od primanja parametara modela, pa sve do izračuna krajnjih vjerojatnosti klasifikacije,
- UART – preko ove biblioteke se obavlja sva komunikacija između računala i ZYBO platforme, a sadrži funkcije za primanje i slanje *float* varijabli i tri tipa *integera*, osim toga sadrži i funkciju za slanje binarne slike, pri čemu se u jednom paketu šalju vrijednosti od osam piksela, što ubrzava slanje osam puta u odnosu na slanje svakog piksela pojedinačno,

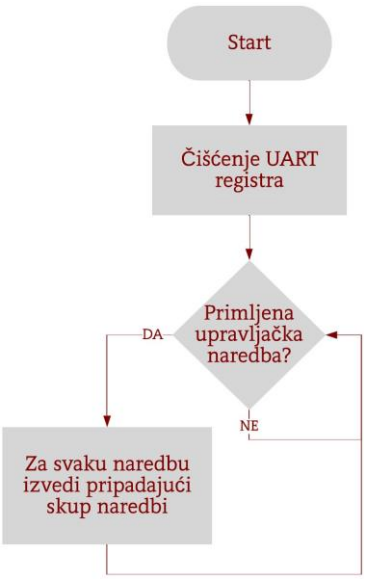
- *graphics* – ova biblioteka sadrži funkcije vezane za obradu slike iz *frame buffera*, između ostalog, binarizaciju, *Sigma* filtar, *Bradley Adaptive Thresholding* algoritam, *Median* filtar i funkciju za crtanje pravokutnika,
- *digit* – u ovoj biblioteci su implementirani algoritmi vezani za pronalazak znamenaka na slici te pripremu slika znamenaka za NN, uključujući modificirani CCL algoritam, skaliranje i *Sigma* filtar za *grayscale* slike.

Glavne datoteke aplikacije se „*ZYNN.c*“ i „*ZYNN.h*“, u kojima se uključuju sve korištene biblioteke, uključujući navedene i biblioteke iz *HDMI Input* projekta. Dijagram toka *main()* funkcije je poprilično jednostavan jer se u noj izvode „samo“ dvije funkcije. Prvo se poziva *ZYNN_Init()* funkcija kojom se redom inicijaliziraju dva *buffera* za pohranu *frameova*, *timer*, VDMA *driver*, upravljač zaslona, upravljač prekidima, upravljač snimanja videa, UART sučelje i alocira se memorija za *buffer* za pohranu ulazne slike NN-a. Osim toga, pri inicijalizaciji se zadaje i funkcija koja se pozva pri detekciji HDMI signala te se na zaslon monitora ispisuje natpis „ZYNN“. Nakon toga se poziva *ZYNN_Run()*, funkcija kojom se pokreće ZYNN sustav i koja teoretski nikad ne završi jer sadrži beskonačnu petlju.



Slika 3.2.: Dijagram toka main() funkcije.

Prema slici 3.3. koja prikazuje pojednostavljeni dijagram toka *ZYNN_Run()* funkcije, nakon pozivanja funkcije se prvo čisti registar za pohranu podataka primljenih preko UART-a. Zatim se pokreće beskonačna *while* petlja unutar koje se čeka na upravljačke naredbe, što uključuje i promjenu stanja na HDMI konekciji. Upravljačkim naredbama se može uspostaviti ili prekinuti komunikacija s GUI-jem, dati znak za slanje konfiguracije i parametara modela mreže, izbrisati model, započeti proces detekcije znamenaka i odbaciti rezultati koji su prikazani u tom trenutku na monitoru. Ovisno o naredbi, nakon što se određeni skup naredbi odradi, UART-om se šalje povratna informacija.



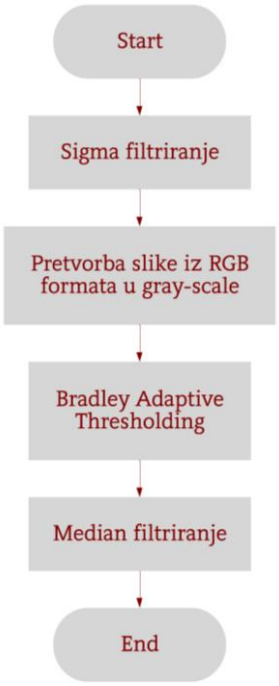
Slika 3.3.: Dijagram toka ZYNN_Run() funkcije.

Proces detekcije znamenaka, čiji se dijagram toka vidi na slici 3.4., započinje s predobradom slike spremljene u *frame buffer*. Prvo se primjenjuje *Sigma* filtar kako bi se uklonio šum sa slike, parametri filtra su određeni eksperimentalnom metodom. Zatim se dobivena slika pretvara u *grayscale* format tako što se intenzitet piksela *I* izračuna prema izrazu (3-1), koja se još naziva i metoda svjetlosti (engl. *luminosity method*) [44]. Najmanji koeficijent se nalazi uz kanal

plave boje zato što se upravo u tom kanalu pojavljuje najviše šuma. Svaki piksel je tipa *integer* veličine 8 b, a najveća vrijednost koju može poprimiti je 255.

$$I = 0.21R + 0.72G + 0.07B \quad (3\text{-}1)$$

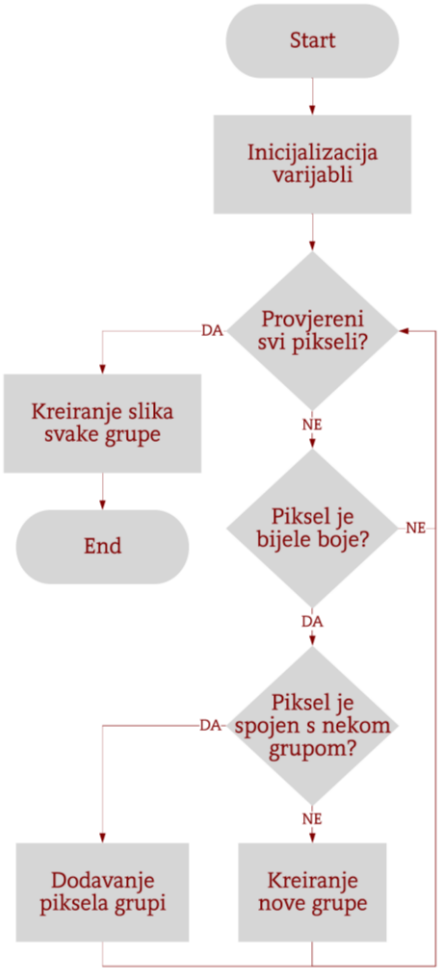
Nakon toga se primjenjuje *Bradley Adaptive Thresholding* metoda kojom se *grayscale* slika pretvara u binarnu sliku, tako što se pozadina predstavlja nulom (crnom bojom), a sve ostalo jedinicom (bijelom bojom). Na kraju je dodano *Median* filtriranje, kako bi se riješio problem nakupina malog broja bijelih piksela, koje su se u velikim količinama pojavljivale na slici. Slika dobivena predobradom se pretvara nazad u RGB format te se sprema u *frame buffer* kako bi se prikazala na monitoru.



Slika 3.4.: Dijagram toka procesa predobrade slike.

Modificirani CCL algoritam se primjenjuje na binarnoj slici kako bi se povezani pikseli grupirali, a dijagram toka ovog algoritma je prikazan na slici 3.5. Prvo se inicijaliziraju varijable potrebne za rad algoritma. Zatim se samo jednom prolazi kroz sliku, za razliku od originalnog CCL algoritma opisanog u 2.5.7. Način provjere piksela je identičan, samo što se grupe piksela pohranjuju u posebne strukture odnosno korisnički definirane tipove podataka. Te strukture podataka sadrže polje indeksa piksela grupe, broj piksela grupe te minimalne i maksimalne vrijednosti u smjeru *x* i *y* osi, u svrhu određivanja lokacije grupe i dimenzija okvira koji obuhvaća sve piksele grupe.

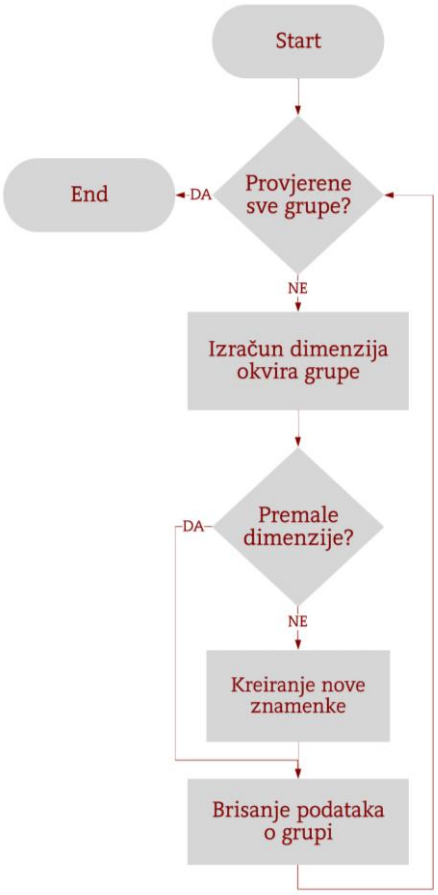
Prolaskom kroz sve piksele kreiraju se grupe, koje se trebaju na kraju pretvoriti u konkretne slike koje će se koristiti za klasifikaciju. To je izvedeno funkcijom *Labels2Digits()*, čiji je dijagram toka na slici 3.6., koja redom prolazi kroz sve grupe piksela, te provjerom dimenzija okvira odlučuje koje grupe će se koristiti za klasifikaciju znamenaka. Ako su obje dimenzije okvira manje od 20 piksela tada se ta grupa odbacuje, u suprotnom



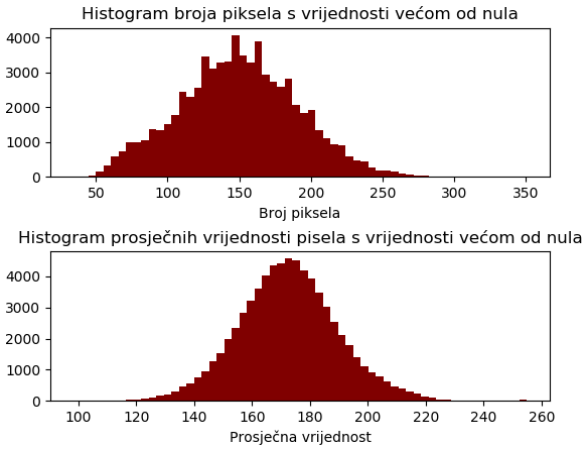
Slika 3.5.: Dijagram toka modificiranog CCL algoritma.

se kreira nova znamenka pozivanjem funkcije *AddDigit()*, nakon koje se također brišu podaci, uključujući oslobađanje alocirane memorije.

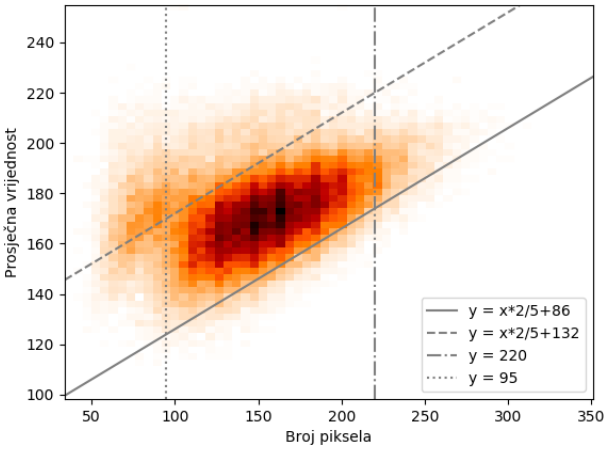
Prije kreiranja potencijalnih znamenki, provedena je statistička analiza slika MNIST baze podataka kako bi se normalizirale ulazne slike NN-a. Varijable koje su se proučavale su broj piksela slike koji imaju vrijednost različitu od nule i prosječni intenzitet tih piksela. Analiza je izvedena nad svim ulaznim podacima odnosno nad skupom za učenje i skupom za testiranje koji ukupno sadrže 70 tisuća slika, a dobiveni rezultati su prikazani u obliku histograma (Slika 3.7.) iz kojih se može zaključiti da najviše slika sadrži oko 150 piksela s vrijednosti većom od nule s prosječnim intenzitetom od oko 172. Osim toga, dvodimenzionalnim histogramom je prikazana ovisnost spomenutih varijabli (Slika 3.8.), pri čemu su tamnijim bojama označene veće nakupine podataka. Zatim su određena četiri pravca oko najveće nakupine podataka, čije su jednadžbe također prikazane na slici, kojima je obuhvaćeno 73.1 % podataka. Unutar tog područja, s mogućim odstupanjima, će se nalaziti svaka slika kreirana funkcijom *AddDigit()*.



Slika 3.6.: Dijagram toka Labels2Digits() funkcije.



Slika 3.7.: Histogrami broja i prosječne vrijednosti piksela.

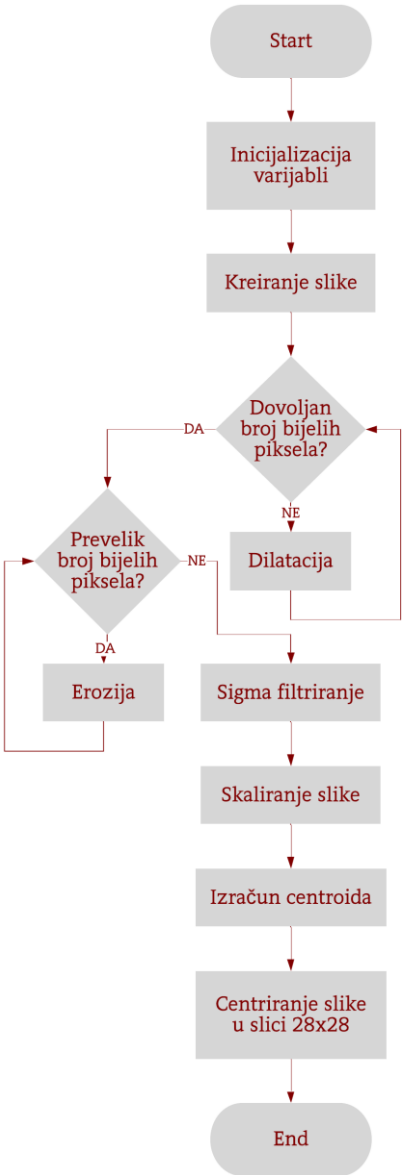


Slika 3.8.: Dvodimenzionalni histogram ovisnosti proučavanih varijabli.

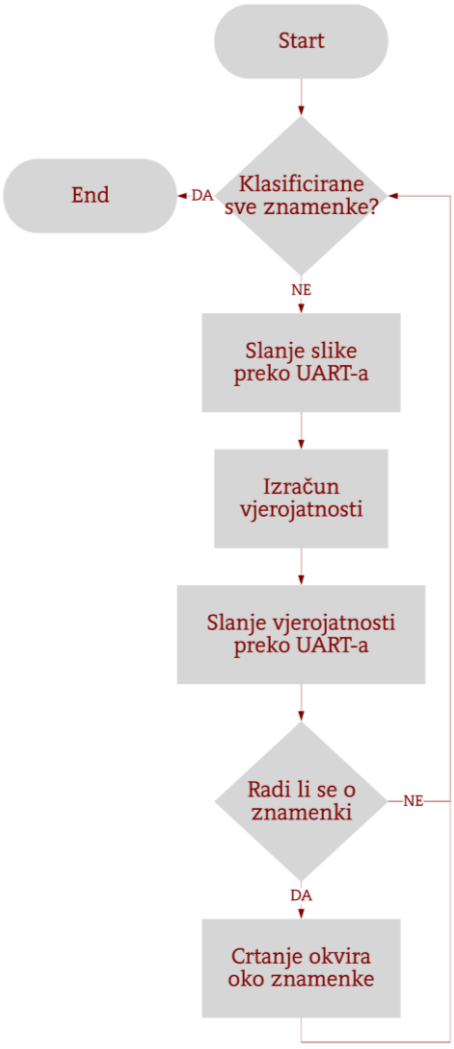
Funkcijom *AddDigit()* kreira se nova potencijalna znamenka. Funkcija, osim što kreira novu sliku s pikselima zadane grupe, kreira sliku koju će NN koristiti za klasifikaciju, što je izvedeno na sličan način na koji je nastala MNIST baza podataka, što je opisano u 2.2.4. Zbog

velikih gubitaka informacija pri skaliranju s visokim faktorom skaliranja (engl. *scale factor*), dodane su funkcije dilatacije i erozije kako bi broj piksela slike bio u granicama određenim prethodnim analizom podataka. *Sigma* filtriranjem se postiže smanjenje prosječnog intenziteta piksela, a parametri filtra, točnije, veličina prozora je određena eksperimentalnom metodom kako bi konačna slika bila u području zadanom pravcima na slici 3.8. Nakon skaliranja bilinearnom interpolacijom, izračunava se centar mase piksela uzimajući u obzir intenzitet kao težinski faktor. Konačna slika se dobije centriranjem unutar slike veličine 28×28, s obzirom na izračunati centroid. Slika 3.9. prikazuje dijagram toka opisane funkcije.

Nakon što su pronađene sve potencijalne znamenke, trebaju se izračunati vjerojatnosti za pojedinu znamenku (Slika 3.10.). Prije toga se 28×28 slika znamenke kopira u *buffer* za ulaznu sliku NN, te se ista šalje računalu preko UART-a. Zatim se ulazna slika predaje funkciji



Slika 3.9.: Dijagram toka AddDigit() funkcije.



Slika 3.10.: Dijagram toka procesa klasifikacije znamenaka.

CalculateProbs() koja izračunava vjerojatnosti za deset klasa odnosno za brojeve od nula do devet. Nakon toga se računalu šalju dobivene vjerojatnosti, te se na kraju provjerava postoji li vjerojatnost veća od 55 %, na osnovu čega se zaključuje je li je znamenka pronađena i na slici koja se nalazi u *frame bufferu* se crta okvir oko iste.

NN modeli su realizirani u C programskom jeziku pomoću sedam definiranih struktura podataka, od kojih je najvažnija struktura *deep_neural_network* jer sadrži sve informacije o mreži:

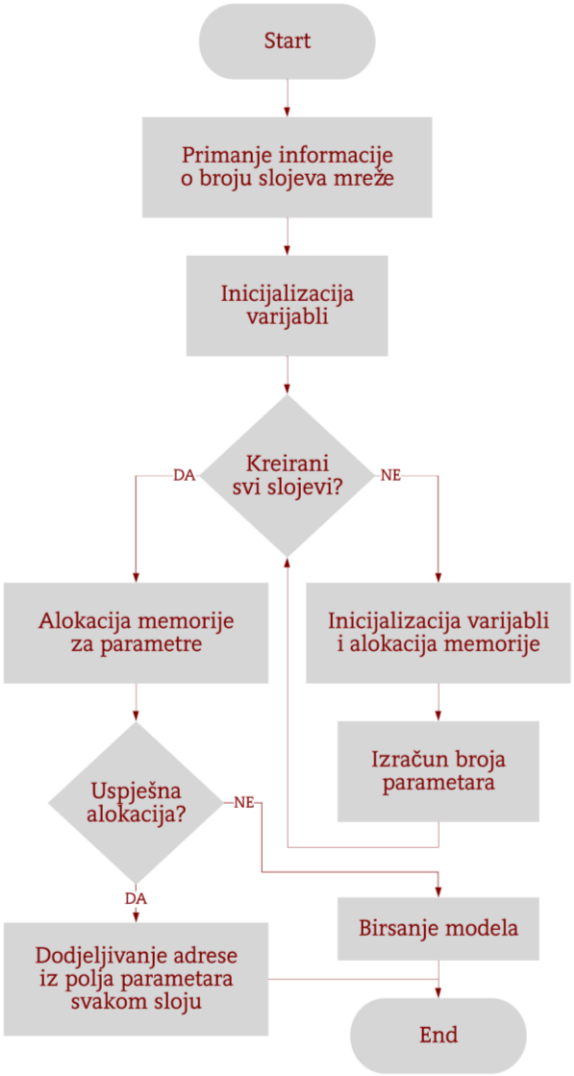
- *input_size* – dimenzija ulaznih podataka,
- *channels* – broj kanala ulaza, npr. RGB ima tri kanala,
- *input_tensor* – pokazivač na 3D polje gdje su pohranjene vrijednosti ulaznog tenzora,
- *num_of_layers* – broj slojeva u mreži,
- *num_of_parameters* – broj parametara mreže,
- *all_parameters* – pokazivač na 1D polje koje je namijenjeno za pohranu svih parametara,
- *layers* – pokazivač na 1D polje slojeva.

Budući da ima više vrsta slojeva, definiran je novi tip podatka koji je nazvan *layer*, koji sadrži informaciju o vrsti sloja i *void* pokazivač, odnosno pokazivač nedefiniranog tipa podatka, koji služi za pohranu podataka određenog sloja.

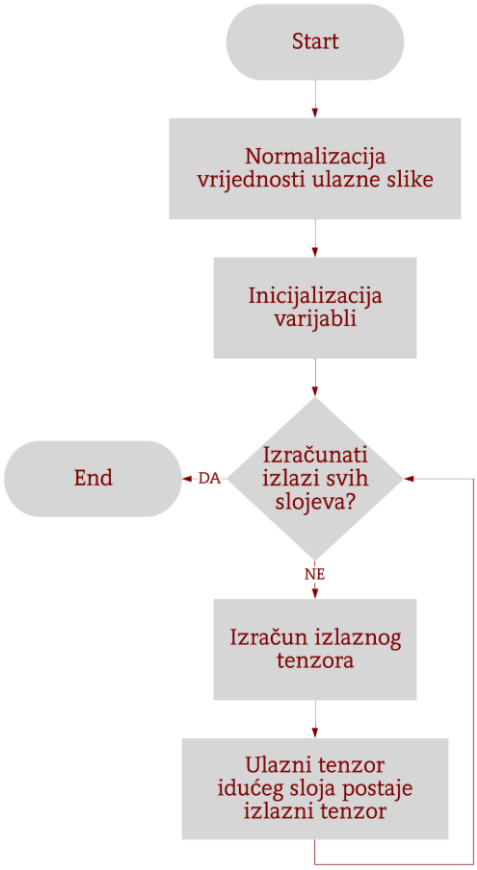
Ostalih pet struktura je namijenjeno za pohranu informacija raznih vrsta slojeva, a te strukture su nazvane: *convolutional_layer*, *max_pooling_layer*, *fully_connected_layer*, *flatten_layer* i *softmax_layer*. Svaka struktura sadrži varijable specifične za sloj NN-a koji opisuju, a te varijable pohranjuju informacije o, između ostalog, veličini ulaznih i izlaznih podataka sloja, broj neurona ili filtera i mapi značajki, parametre *padding* i *stride*, pokazivač na polje težinskih faktora, pokazivač na aktivacijsku funkciju, pokazivač na polje izlaznih podataka, itd.

Konfiguracija modela NN-a se prima pozivanjem funkcije *RecvConfiguration()*, koja kreira model mreže kao varijablu tipa *deep_neural_network*. Prema slici 3.11., prvo se primaju informacije o broju slojeva i informacije o svakom sloju, a u međuvremenu se inicijaliziraju varijable, alocira potrebna memorija i izračunava se broj parametara sloja. Nakon čega se alocira memorija za polje u koje će se spremati svi parametri mreže, ako alokacija ne uspije, najčešće jer nema dovoljno resursa za pohranu parametara, brišu se podaci o modelu.

Koncept pohrane težinskih faktora je realiziran tako što se svi parametri mreže spremaju u polje čiju adresu ima spomenuta varijabla *all_parameters*. Pri kreiranju modela NN-a, svakom sloju koji radi s parametrima se dodjeljuje adresa elementa iz polja parametara, koji odgovara prvom predodređenom parametru tog sloja. Time se postiglo to da se ne mora za svaki sloj alocirati memorija za polje parametara te je ujedno olakšano primanje težinskih faktora.



Slika 3.11.: Dijagram toka funkcije RecvConfiguration().



Slika 3.12.: Dijagram toka funkcije CalculateProbs().

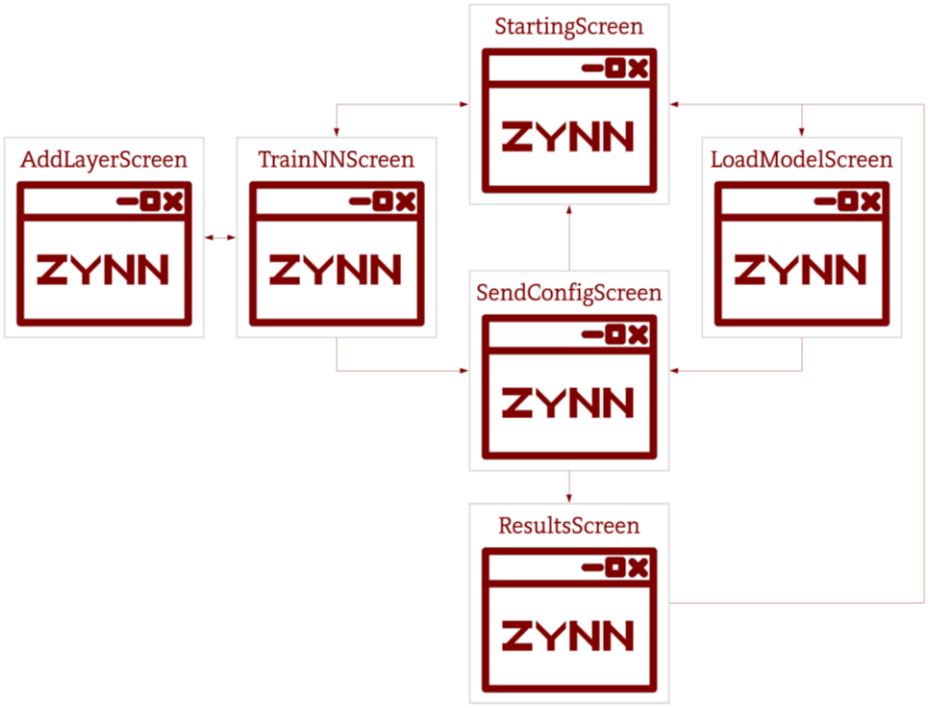
Nakon kreiranja modela NN-a, poziva se funkcija *RecvParameters()*, koja je sastavljena od jednostavne *for* petlje unutar koje se primaju vrijednosti težinskih faktora mreže.

Klasifikacija znamenaka se ostvaruje pozivom funkcije *CalculateProbs()*, čiji je dijagram toka prikazan na slici 3.12. Prvo se izvodi normalizacija vrijednosti ulazne slike na vrijednosti između nula i jedan, dijeljenjem originalne vrijednosti s 255. Zatim se inicijaliziraju varijable potrebne za izračun vjerojatnosti. Nakon čega se prolazi kroz sve slojeve modela te se izračunava izlazni tenzor svakog sloja, s tim da se izlazni tenzor koristi kao ulazni tenzor slijedećeg sloja. Na kraju se izračuna izlaz iz zadnjeg sloja koji su zapravo rezultati klasifikacije.

Identični rezultati se dobiju na računalu *Kerasovom predict()* metodom. Jedino je prije treniranja *Keras* modela trebalo postaviti tip ulaznih podataka na 32 bitni *float*, jer *Cortex*-A9 ne može raditi sa 64 bitnim *floatom*, u suprotnom se ne dobiju isti rezultati.

3.2. Grafičko korisničko sučelje

GUI aplikacija je razvijena u *Pythonu*, koristeći *Kivy* biblioteku, a sastoji se od skupa zaslona aplikacije na kojima se korisniku ponuđene razne mogućnosti. Svi zasloni su instance podklase *Kivy* klase *Screen* i prikazani su na slici 3.13. Strelicama su prikazane povezanosti između zaslona, drugim riječima, prikazuje na koje je zaslone moguće prijeći s trenutnog zaslona. Također, smjerom strelice definiran je i smjer tranzicije (primjerice, tranzicija sa *StartingScreena* na *LoadModelScreen* se izvodi s desna na lijevo). Svrha svakog zaslona se može zaključiti po nazivu klase. Zasloni sadrže određeni skup *widgeta*, koji su definirani u KV datoteci.

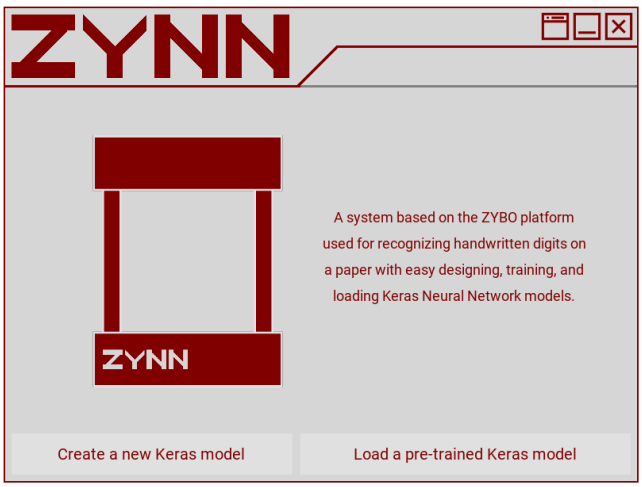


Slika 3.13.: Grafički prikaz GUI aplikacije.

3.2.1. Upute za rad sa ZYNN korisničkim sučeljem

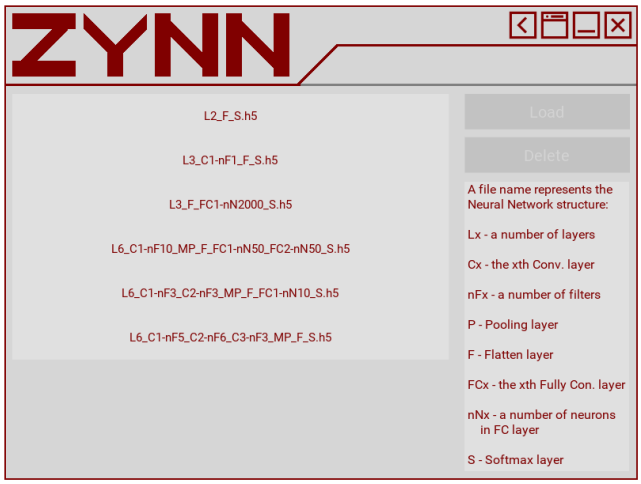
U ovom potpoglavlju je detaljnije opisan GUI, navođenjem i pojašnjenjem namjene pojedine klase sa slike 3.13.

StartingScreen je početni zaslon GUI aplikacije, koji pruža korisniku neke informacije o ZYNN sustavu te odabir između kreiranja novog *Keras* modela i učitavanja postojećeg modela (Slika 3.14.). Osim toga, pri vrhu prozora s desne strane su dostupne tri tipke, kojima se prozor aplikacije može



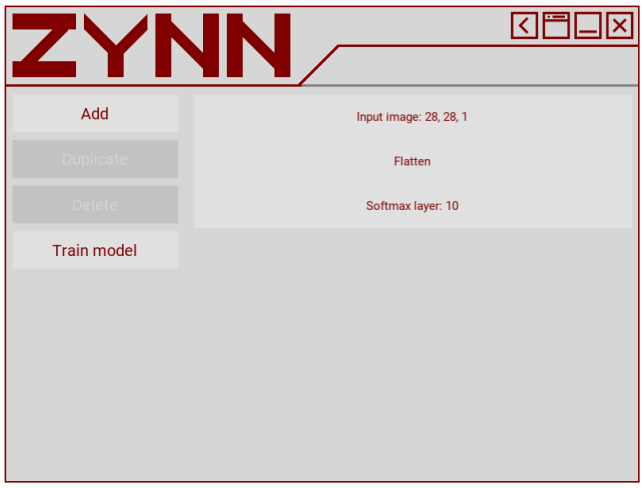
Slika 3.14.: Prikaz StartingScreen zaslona.

zatvoriti, minimizirati i dodati klasična *Windows* traka, kojom se prozor aplikacije može pomicati po zaslonu računala.



Slika 3.15.: Prikaz LoadModelScreen zaslona.

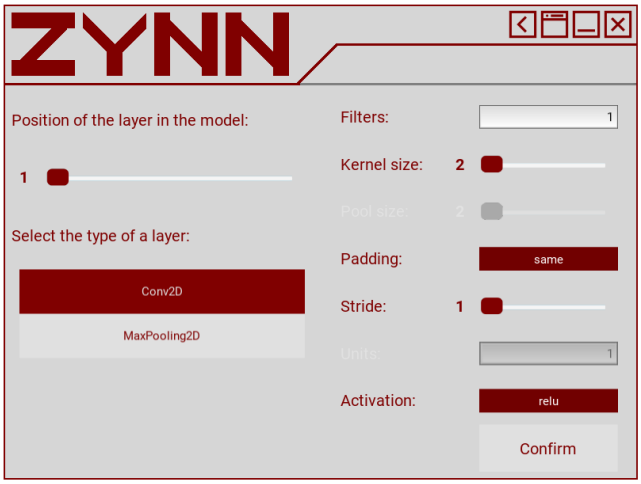
Na *LoadModelScreen* zaslonu je moguće učitati postojeći *Keras* model iz direktorija „*SavedModels*“ koji se nalazi u direktoriju aplikacije. Modeli su pohranjeni u HDF5 datotekama pomoću *Kerasove save()* metode, a naziv datoteka je određen sukladno strukturi mreže, kao što je prikazano na desnoj strani slike 3.15. Osim učitavanja modela, moguće je i obrisati model, čime se briše označena datoteka. Dok nije označena niti jedna datoteka, obje tipke su onemogućene, kao što se vidi na slici. Na ovom zaslonu je dodana još jedna tipka u gornjem desnom kutu kojom se vraća na *StartingScreen*. Učitavanje modela se izvodi u drugoj niti kako bi se *Kivy widgeti* nastavili osvježavati, jer bi u suprotnom cijela aplikacija zablokirala dok se model ne bi učitao. Osim toga, pri učitavanju se pokreće animacija *progress bar widgeta* koji se nalazi pri vrhu prozora.



Slika 3.16.: Prikaz TrainNNScreen zaslona.

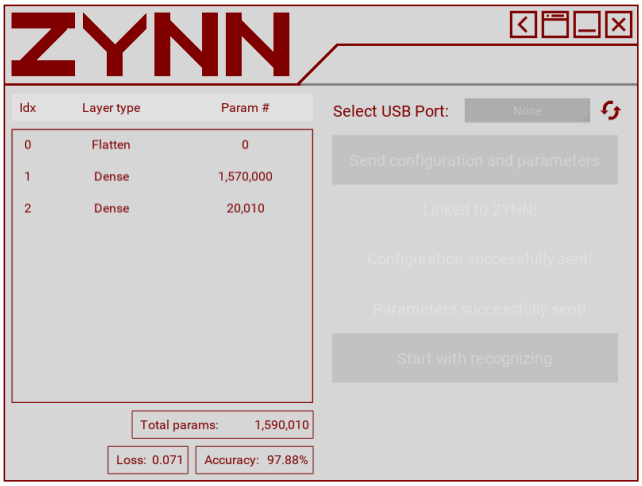
Otvaranjem *TrainNNScreena* se prikazuje najjednostavnija NN koju je moguće kreirati. Iako su prikazana tri sloja, ova NN se zapravo sastoji od jednog sloja, jer se „*Input image*“ odnosi na ulaznu sliku, a „*Flatten*“ služi samo ravnanje te slike. Tako da ostaje „*Softmax layer*“, koji je specifičan po tome što ima 10 neurona sa *softmax* aktivacijskom funkcijom. Korisnik ima na raspolaganju opcije kojima može dodati novi sloj, čime se prebacuje na *AddLayerScreen*, dupliciratи ili obrisati označeni sloj i može pokrenuti proces učenja. Dupliciranje i brisanje sloja je onemogućeno ako nije odabran niti jedan sloj ili je odabran jedan od tri sloja prikazanih na slici 3.16. ili ako bi se dupliciranjem konvolucijskog sloja ili sloja sažimanja slika reducirala na veličinu jedan ili manje piksela. Za vrijeme učenja, koje se izvodi u drugoj niti, su sve tipke onemogućene uključujući tipku za povratak na *StartingScreen*, a na *progress baru* se prikazuje animacija, pri čemu se vidi postotak učenja, koji je dobiven na osnovu završenih epoha. Za osvježavanje vrijednosti *progress bara* koristi se *callback* funkcija koja se predaje *fit()* metodi.

Za dodavanje slojeva pri dizajniranju NN, koristi se *AddLayerScreen* zaslon (Slika 3.17.). Prvo je potrebno odabrati poziciju na kojoj će se sloj dodati, koja je na početku postavljena na poziciju označenog sloja na *TrainNNScreenu*. Sa svakom promjenom pozicije mijenja se popis dostupnih vrsta slojeva, koje ovise o tome je li je odabrana pozicija ispred ili nakon *flattena*. Pritiskom na jednu od ponuđenih vrsta, omogućuje se odabir opcija vezanih za tu vrstu sloja, dok su ostale onemogućene. Promjenom veličine jezgre ili *padding* parametra, mijenja se maksimalna vrijednost na koju se može postaviti *stride*. Tipom „*Confirm*“ potvrđuje se dodavanje sloja i otvara se *TrainNNScreen* s dodanim novim slojem.



Slika 3.17.: Prikaz AddLayerScreen zaslona.

SendConfigScreen, kao što i ime zaslona govori, služi za slanje konfiguracije i parametara modela na ZYBO platformu Osim toga, ovaj zaslon pruža korisniku informacije o broju parametara svakog sloja, ukupnog broja parametara. Nadalje, na zaslonu su prikazane i vrijednosti pogreške te postotak točnosti modela, što je prikazano na slici 3.18., a izračunava se na podacima iz skupa za testiranje.

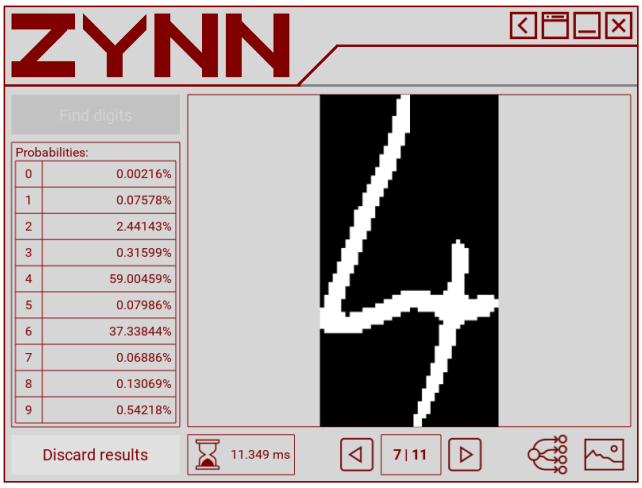


Slika 3.18.: Prikaz SendConfigScreen zaslona.

S desne strane zaslona je prije slanja potrebno odabrati serijski port na kojem je ZYBO platforma spojena, ako je ZYBO spojen nakon otvaranja ovog zaslona, klikom na tipku pored se osvježava popis dostupnih COM portova. Odabirom porta, omogućuje se slanje konfiguracije i parametara mreže, a pritiskom na tipku započinje se proces slanja, koji se također izvodi u drugoj niti, uz ispisivanje pripadnih poruka stanja i izvođenje animacije *progress bara*. Nakon što se slanje završi, pritiskom na tipku „*Start with recognizing*“ prelazi se na *ResultsScreen*. Međutim, ako se pojavi problem pri slanju, otvara se *popup* s obavijesti te postupak treba ponoviti.

ResultsScreen služi za prikaz rezultata klasifikacije znamenaka (Slika 3.19.). Tipkom „*Find digits*“ se daje znak ZYBO platformi za početak procesa detekcije znamenaka. Zatim se u drugoj niti primaju poruke o stanju procesa, uz izvođenje animacije *progress bara*. Ako nijedna znamenka nije pronađena ili se pojavi problem pri izvođenju procesa, otvara se *popup* s pripadnom

porukom, nakon čega se postupak može ponoviti. Korisniku se, nakon primljenih rezultata, predočuje prva znamenka, s lijeva su ispisane vjerojatnosti, ispod slike se prikazuje vrijeme izračuna vjerojatnosti i redni broj znamenke u odnosu na ukupni broj. Strelicama u donjem dijelu prozora može se promijeniti znamenka koja se prikazuje, čime se ujedno mijenjaju vjerojatnosti i vrijeme izvođenja. Osim toga, može se prikazati *frame* u binarnom obliku, skalirana slika trenutne znamenke s ulaza NN-a i odbaciti rezultate, nakon čega se može ponoviti proces.



Slika 3.19.: Prikaz ResultsScreen zaslona.

3.2.2. Pomoćne Python skripte

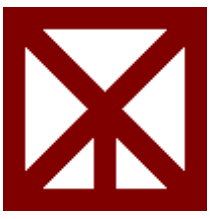
Zbog bolje preglednosti koda, kreirane su pomoćne *Python* skripte za određene funkcije GUI aplikacije, a sve korištene *Python* skripte se nalaze u prilogu P.2. Osim toga, većina korištenih ikona pri dizajniranju aplikacije je preuzeta s izvora [45], a neke ilustracije su preuzete s [46].

U *KerasModel.py* skripti su napisane funkcije vezane za operacije nad *Keras* modelima, između ostalog: učitavanje i treniranje modela, dohvaćanje podataka o slojevima modela, izračun veličine izlaza, izračun maksimalne veličine filtera, dodavanje slojeva, slanje konfiguracije i parametara te obrada rezultata.

Za učitavanje, spremanje, brisanje i provjeru postojećih *Keras* modela koristi se *SaveAndLoadModels.py* skripta koja se koristi u sklopu *KerasModel.py*.

UART.py, koja se također koristi u *KerasModel.py*, služi za razmjenu informacija sa ZYBO platformom, omogućuje slanje upravljačkih naredbi, slanje parametara mreže i slično.

Iako se *Images_for_GUI_cv.py* skripta ne koristi izravno u radu aplikacije, ovom skriptom se je, primjerice, kreirala pozadina sa ZYNN natpisom te ikona aplikacije (Slika 3.20.). Ikona je dizajnirana tako što su slova „Z“, „Y“ i „N“ složena jedno preko drugog. U ovoj skripti se koristi i *OpenCV* biblioteka za crtanje nekih oblika i spremanje kreiranih slika.

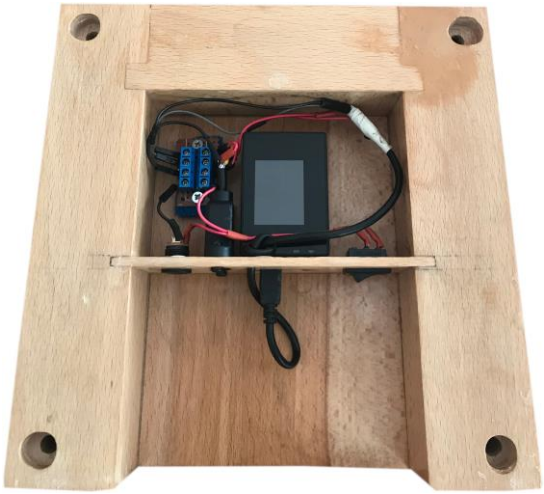

Slika 3.20.: Ikona GUI-ja.

Još jedna skripta koja se ne koristi izravno u radu GUI-ja je *MNIST_stats.py*, kojom je provedena statistička analiza nad slikama iz MNIST baze podataka. Osim toga, generirani su i histogrami koji su prikazani u poglavlju 3.1.2.

3.3. Kućište sustava

Slika 3.21.: Kućište ZYNN sustava.

Kućište ZYNN sustava je izrađeno od drvenog materijala te je prikazano na slici 3.21. U gornjem dijelu kućišta nalazi se kamera, LED traka koja osvjetljava područje na koje dolazi papir i sklopka kojom se taj gornji dio uključuje/isključuje (Slika 3.22.). Elementi se napajaju preko 5,5×2,1 mm konektora na koji se spaja 5 V adapter. U donjem dijelu, koje je prikazano na slici 3.23., je pričvršćena ZYBO platforma te konektor na koji se spaja USB kabel za napajanje i komunikaciju s GUI-jem.



Slika 3.22.: Gornji dio kućišta.

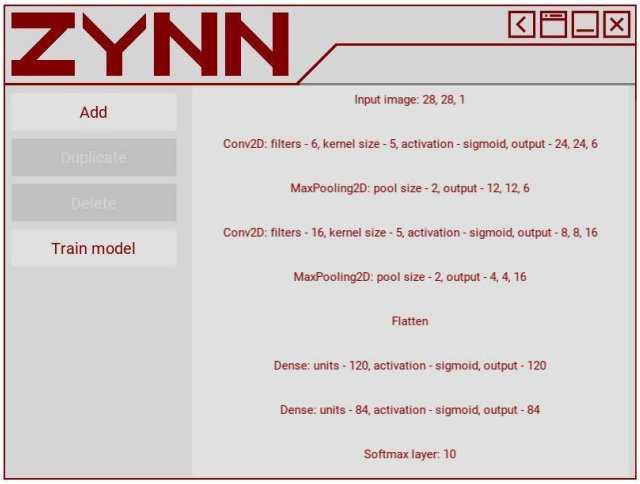
Slika 3.23.: Donji dio kućišta.

4. TESTIRANJE I REZULTATI

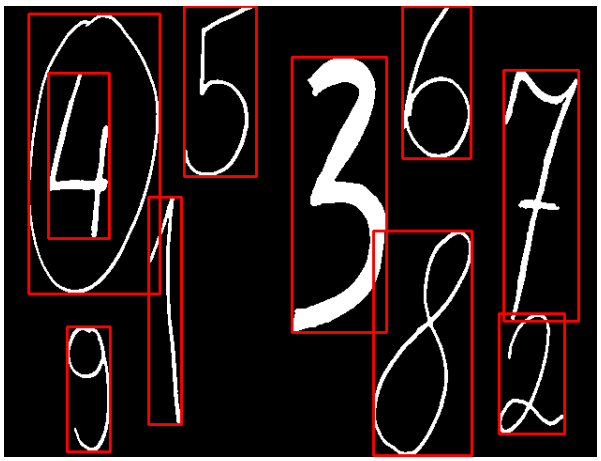
Prva testiranja su provedena pri pisanju biblioteke *UART* i *Python* skripte *UART.py*, kojima je uspješno ostvarena komunikacija između računala i ZYBO platforme.

Nakon toga se je testirao rad implementiranog modela NN, tako što su se MNIST slike slale preko UART sučelja na ZYBO platformu, koja je vraćala rezultate. Time su se provjeravali izlazi iz svakog sloja kako bi se pronašle greške u radu modela. Na kraju se postiglo to da se modelom na ZYBO dobiju identični rezultati onima dobivenim *Keras* metodom *predict()*. Kasnije su još uspješno provedena testiranja ostalih biblioteka koje se koriste u radu, primjerice crtanje oblika na *framebuffer* koji se prikazuje na monitoru, algoritmi za obradu slike s kamere, itd.

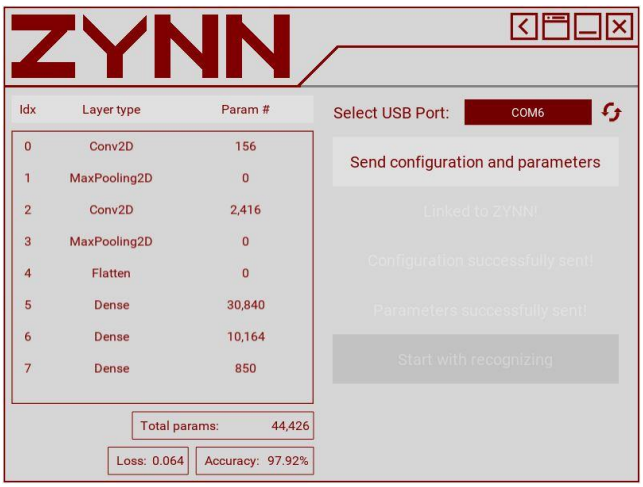
Za testiranje rada ZYNN sustava istrenirana je mreža slična *LeNet* mreži, koja je jedna od prvih objavljenih CNN-ova, koju je predstavio Yann Lecun u svrhu prepoznavanja rukom pisanih brojeva, čiji je model tada postigao izvanredne rezultate [47]. Istrenirana mreža se razlikuje od *LeNet* mreže po tome što koristi sažimanje po maksimalnoj vrijednosti umjesto sažimanja po prosječnoj vrijednosti te u prvom sloju vrijednost *paddinga* je nula. Slika 4.1. prikazuje slojeve mreže nakon dodavanja na *TrainNNScreenu*. Treniranje mreže kroz pet epoha je trajalo oko 45 minuta, pri čemu je postignuta točnost od 97.92 % uz vrijednost pogreške unakrsne entropije 0.064 (Slika 4.2.). Nakon što su poslani parametri, izvedena je detekcija znamenaka sa slike. Na slici 4.3. su prikazane detektirane znamenke, što je vidljivo na monitoru spojenom preko VGA sučelja. Iz priloženog se može zaključiti da se znamenke



Slika 4.1.: Prikaz slojeva LeNet mreže.

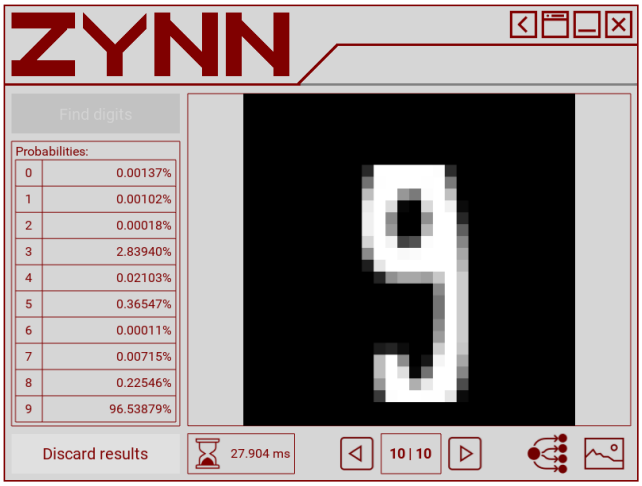


Slika 4.3.: Prikaz detektiranih zamenaka LeNet mrežom.



Slika 4.2.: Prikaz broja parametara mreže i postignutih vrijednosti točnosti i pogreška.

mogu nalaziti jedna unutar druge ili jedna u blizini druge, jedino problem predstavljaju spojene znamenke. Osim toga, ako papir nije dobro namješten, može doći do detekcije znamenaka u kutovima slike. Vrijeme izvođenja cijelog procesa detekcije je 5.9 sekundi, dok je vrijeme prolaska slike kroz mrežu oko 28 ms, što je posljedica velikog broja slojeva mreže. Sve znamenke su supješno klasificirane, a primjer je prikazan na slici 4.4. Prema tablici 4.1. kreirane su mreže sastavljene od:



Slika 4.4.: Primjer prikaza detektirane znamenke.

- NN1 – jednog FC-a s 2000 neurona,
- NN2 – šest FC-a s po 20 neurona,
- CNN1 – konvolucijskog sloja s 10 filtera veličine 4, *Max-pooling* i FC sloja sa 100 neurona,
- CNN2 – konvolucijski sloj s 5 filtera veličine 10, *Max-pooling* sloj.

Tablica 4.1.: Informacije o korištenim mrežama.

Mreža	Broj parametara	Broj skrivenih slojeva	Točnost [%]	Pogreška unakrsne entropije
LeNet	44,426	6	97.92	0.064
NN1	1,590,010	1	97.88	0.071
NN2	18,010	6	90.18	0.406
CNN1	26,370	3	98.39	0.051
CNN2	1,315	2	96.78	0.102

Mreže su testirane na istoj slici kao i *LeNet* (Slika 4.3.). Tablica 4.2. prikazuje rezultate klasifikacije, pri čemu se t_u odnosi na ukupno vrijeme izvođenja procesa detekcije, na koje najviše utječe broj znamenki napisanih na papiru, a t_z na prosječno vrijeme potrebno za izračun vjerojatnosti jedne znamenke. Uzme li se u obzir zauzeće memorije u odnosu na dobivenu točnost i pogrešku unakrsne entropije, može se zaključiti da mreže s konvolucijskim slojevima postižu dosta bolje rezultate uz manje potrebnih resursa za pohranu težinskih faktora.

Tablica 4.2.: Rezultati klasifikacije.

Mreža	t_u [s]	t_z [ms]	Dobivene vjerojatnosti za znamenke: [%]									
			0	1	2	3	4	5	6	7	8	9
LeNet	5.941	27.9	99.67	99	98.04	99.68	74.06	99.59	99.7	75.05	95.41	96.54
NN1	6.075	41.06	99.97	98.14	99.52	99.82	84.18	99.76	97.24	42.95	95.92	93.81
NN2	5.77	10.2	94.22	22.74	55.96	84.53	2.52	69.41	89.44	77.87	31.33	4.47
CNN1	5.772	13	99.92	98.68	99.09	99.87	85.92	99.88	99.91	72.69	93.27	99.53
CNN2	5.796	12.95	96.51	99.84	98.32	99.68	66.69	95.53	99.27	92.89	96.09	92.09

Za određene postavke strukture mreže može doći do pogrešne detekcije, zbog ne razvijanja klasifikacijskih svojstava tijekom procesa učenja, zbog prenaučenosti mreže ili zbog odstupanja napisane znamenke od znamenaka te klase iz MNIST baze podataka. Također, sustav je osjetljiv na orijentaciju znamenaka te ista ne smije biti pretanka kako ne bi nestala pri otklanjanju šuma. S druge strane, sustav nije osjetljiv na veličinu znamenke, jedino što ne smije biti manja od 20 piksela, te broj znamenaka na papiru, osim što utječe na vrijeme izvođenja procesa detekcije.

Idealno bi bilo postići prepoznavanje znamenaka u stvarnom vremenu, za što se treba sklopovski ubrzati što je više moguće operacija, primarno operacije vezane za obradu slike, kako bi se više toga paralelno izvodilo na FPGA-u i time ubrzalo proces detekcije znamenaka. Osim toga, moguće je implementirati druge tipove mreža (primjerice RNN) i prilagoditi sustav za rad s drugim bazama podataka (primjerice *Fashion*-MNIST, CIFAR10). Nadalje, kada bi se na ZYBO instalirao *Linux* OS, sve bi se izvodilo na platformi te ne bi bilo potrebe za povezivanjem s računalom. Međutim, u tom slučaju bi trebalo voditi računa o resursima platforme.

5. ZAKLJUČAK

Kreiranjem ZYNN sustava se implementirala NN na ZYBO platformi, čime je ostvaren cilj ovog rada. Dio ZYNN-a koji se izvodi na računalu služi za dizajniranje, treniranje, učitavanje i brisanje spremljenih *Keras* NN modela. Pored toga, ostvaruje se komunikacija s ZYBO platformom, te se šalju parametri mreže, uz uvid u podatke o mreži. Nakon što se pošalju podaci može se početi proces detekcije znamenaka s papira, koji se odvija na ZYBO platformi. Pri tom procesu se prvo otklanja šum i binarizira se slika, nakon čega se primjenjuje CCL algoritam za pronalazak nakupina piksela. Zatim se kreiraju slike znamenki po uzoru na MNIST slike. Dobivene slike se provode kroz NN čiji parametri su prethodno poslani te se na kraju rezultati šalju računalu te se prikazuju korisniku.

Rezultati su pokazali kako ZYNN sustav radi zadovoljavajuće. Za testiranje je korištena mreža slična *LeNet*-u, te su uspješno prepoznate znamenke s papira, koje moraju biti veće od 20 piksela i ne smiju biti spojene, što znači da, primjerice, mogu biti napisane jedna unutar druge.

Rad sustava se može unaprijediti optimizacijom kôda, sklopovskim ubrzanjem operacija ili cijelih algoritama, primjerice algoritama za obradu slike ili operacije konvolucije pri radu CNN-a. Dovoljnim ubrzanjem postiglo bi se prepoznavanje znamenaka u stvarnom vremenu. Sustav se može proširiti dodavanjem drugih tipova NN ili adaptiranjem za rad s drugim bazama podataka.

LITERATURA

[1] Y. LeCun, C. Cortes, C. J. C. Burges, *The MNIST database of handwritten digits*, 1998., dostupno na: http://yann.lecun.com/exdb/mnist/, [22.8.2019.]

[2] B. D. Bašić, M. Čupić, J. Šnajder, Umjetne neuronske mreže, diplomski rad, Fakultet elektrotehnike i računarstva, Zagreb, svibanj 2008., dostupno na: https://www.fer.hr/_download/repository/UmjetneNeuronskeMreze.pdf, [16.8.2019.]

[3] A. Gulli, S. Pal, *Deep Learning with Keras*, *Pact Publishing Ltd.*, Birmingham, travanj 2017.

[4] *ImageNet* baza podataka, *Stanford Vision Lab*, *Stanford University*, *Princeton University*, 2016., dostupno na: http://www.image-net.org/, [16.8.2019]

[5] W. McCulloch, W. Pitts, *A Logical Calculus of Ideas Immanent in Nervous Activity*, 1943.

[6] A. F. Gad, *Practical Computer Vision Applications Using Deep Learning with CNN*s, *Apress*, prosinac 2018.

[7] V. Vukotić, Raspoznavanje objekata dubokim neuronskim mrežama, diplomski rad, Fakultet elektrotehnike i računarstva, Zagreb, lipanj 2014., dostupno na: http://www.zemris.fer.hr/~ssegvic/project/pubs/vukotic14ms.pdf, [22.8.2019.]

[8] B. Prijono, *Student Notes: Convolutional Neural Networks (CNN) Introduction*, dostupno na: https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/, [22.8.2019.]

[9] M. Mishra, *Convolutional Neural Networks, Explained*, *Oracle*, 7. ožujka 2019., dostupno na: https://www.datascience.com/blog/convolutional-neural-network, [22.8.2019.]

[10] I. Lieder, Y. S. Resheff, T. Hope, *Learning TensorFlow: A Guide to Building Deep Learning Systems*, *O'Reilly Media*, kolovoz 2017.

[11] R. Maksutov, *Deep study of a not very deep neural network. Part 2: Activation functions*, *Towards Dana Science*, 1. svibnja 2018., dostupno na: https://towardsdatascience.com/deep-study-of-a-not-very-deep-neural-network-part-2-activation-functions-fd9bd8d406fc, [22.8.2019.]

[12] *Machine Learning Cheatsheet*, dostupno na: https://ml-cheatsheet.readthedocs.io/en/latest/index.html, [22.8.2019.]

[13] A. Mehta, A Complete Guide to Types of Neural Networks, Digital Vidya, 25. siječnja 2019., dostupno na: https://www.digitalvidya.com/blog/types-of-neural-networks/, [22.8.2019.]

[14] I. Goodfellow, Y. Bengio i A. Courville, *Deep Learning*, MIT Press, 2016., dostupno na: http://www.deeplearningbook.org/, [22.8.2019.]

[15] A. Baldominos, Y. Saez, P. Isasi, *A Survey of Handwritten Character Recognition with MNIST and EMNIST*, 4. kolovoza 2019., dostupno na: https://www.researchgate.net/publication/334957576_A_Survey_of_Handwritten_Character_Recognition_with_MNIST_and_EMNIST, [22.8.2019.]

[16] *ZYBO FPGA Bord Reference Manual*, *Digilent*, 11. travnja 2016., dostupno na: https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf, [17.8.2019]

[17] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, R. W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*, *Strathclyde Academic Media*, 2014.

[18] *Keras Documentation*, dostupno na: https://keras.io/, [17.8.2019]

[19] *Kivy Documentaion*, dostupno na: https://kivy.org/doc/stable/, [17.8.2019]

[20] *Cython – C-Extensions for Python*, dostupno na: https://cython.org, [17.8.2019]

[21] *OpenGL*, dostupno na: https://www.opengl.org/, [17.8.2019]

[22] *PySerial Documentation*, dostupno na https://pythonhosted.org/pyserial/, [23.8.2019.]

[23] *Python Documentation: Threading -Thread-based parallelism*, dostupno na: https://docs.python.org/3/library/threading.html, [23.8.2019.]
[24] *NumPy*, dostupno na: https://www.numpy.org/, [23.8.2019.]
[25] *OpenCV*, dostupno na: https://opencv.org/, [23.8.2019.]
[26] J. C. Russ, F. B. Neal, *The Image Processing Handbook*, CRC *Press*, 2016.
[27] V. Kovalevsky, *Modern Algorithms for Image Processing: Computer Imagery by Example Using C#*, *Apress*, 2019.
[28] D. Bradley, G. Roth, *Adaptive Thresholding using the Integral Image*, siječanj 2007., dostupno na: https://pdfs.semanticscholar.org/8d74/418ec3c4e2ff45b72e723ac0fbe5fcd58620.pdf, [23.8.2019.]
[29] P. D. Wellner, *Adaptive Thresholding for the DigitalDesk*, 1993., *Rank Xerox*, dostupno na: https://pdfs.semanticscholar.org/ea59/dc10e8cf6f2d13088415d72b48f417f817bd.pdf, [23.8.2019.]
[30] R. C. Gonzalez, R. E. Woods, *Digital Image Processing*, *Pearson Education*, 2018.
[31] A. Rosenfeld, J. L. Pfalz, *Sequential Operations in Digital Picture Processing*, JACM, 4. listopada 1966.
[32] U. Sinha, *Connected Component Labelling*, *AI Shack*, dostupno na: http://aishack.in/tutorials/connected-component-labelling/, [23.8.2019.]
[33] SJCAM SJ4000 *WiFi*, dostupno na: https://sjcam.com/product/sj4000/, [18.8.2019.]
[34] S. Churiwala, *Designing with Xilinx FPGAs Using Vivado*, Springer, 2017.
[35] H. J. Butt, *Hardware acceleration of an evolutionary algorithm on Xilinx Zynq-7000*, diplomski rad, Sveučilište Oslo, 10. kolovoza 2015., dostupno na: https://pdfs.semanticscholar.org/a731/9e09d2761e354319cb254342756f5a2c12d3.pdf, [23.8.2019.]
[36] *Visual Studio Code*, dostupno na: https://code.visualstudio.com/, [18.8.2019.]
[37] *Developer Survey Results* 2019, dostupno na: https://insights.stackoverflow.com/survey/2019, [18.8.2019.]
[38] *The Scientific Python Development Environment*, *Anaconda Cloud*, dostupno na: https://anaconda.org/anaconda/spyder, [18.8.2019.]
[39] *Spyder: The Scientific Python Development Environment — Documentation*, dostupno na: https://docs.spyder-ide.org/, [19.8.2019.]
[40] *Tera Term Open Source Project*, dostupno na: https://ttssh2.osdn.jp/index.html.en, [19.8.2019.]
[41] *Terminal*, dostupno na: https://sites.google.com/site/terminalbpp/, [19.8.2019.]
[42] *Zybo HDMI Input Demo*, *Digilent*, dostupno na: https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-hdmi-input-demo/start, [24.8.2019.]
[43] Intellectual Property, Xilinx, dostupno na: https://www.xilinx.com/products/intellectual-property.html, [24.8.2019.]
[44] J. D. Cook, Three algorithms for converting color to grayscale, 24. kolovoza 2009., dostupno na: https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/, [25.8.2019.]
[45] *Icons8*, dostupno na: https://icons8.com/, [27.8.2019.]
[46] *The Noun Project*, dostupno na: https://thenounproject.com/, [27.8.2019.]
[47] *Dive into Deep Learning*, *Convolutional Neural Networks(LeNet)*, dostupno na: https://www.d2l.ai/chapter_convolutional-neural-networks/lenet.html, [8.9.2019.]

POPIS KRATICA

NN	neuronska mreža (engl. *Neural Network*)
UART	*Universal Asynchronous Receiver/Transmitter*
ANN	umjetna NN (engl. *Artificial Neural Network*)
AI	umjetna inteligencija (engl. *Artificial Intelligence*)
ML	strojno učenje (engl. *Machine Learning*)
DL	duboko učenje (engl. *Deep Learning*)
FC	potpuno povezani sloj (engl. *fully-connected layer*)
MLP	višeslojni perceptron (engl. *Multilayer Perceptron*)
CNN	konvolucijska NN (engl, *Convolutional* NN)
RNN	NN s povratnom vezom (engl. *Recurrent* NN)
SoC	sustav na čipu (engl. *System on Chip*)
AP SoC	sve programibilni SoC (engl. *All Programmable SoC*)
OS	operacijski sustav (engl. *operating system*)
FPGA	*Field Programmable Gate Array*
PS	procesni sustav (engl. *Processing System*)
PL	programibilna logika (engl. *Programmable Logic*)
HW	*hardware*
SW	*software*
AXI	*Advanced eXtensible Interface*
AMBA	*Advanced Microcontroller Bus Architecture*
RAM	radna memorija (engl. *Read Access Memory*)
OCM	memorija na čipu (engl. *On-Chip Memory*)
GUI	grafičko korisničko sučelje (engl. *Graphical User Interface*)
UI	korisničko sučelje (engl. *User Interface*)
API	aplikacijsko programsko sučelje (engl. *Application Programming Interface*)
TCL	*Tool Command Language*
IP	*Intellectual property*
IDE	*Integrated Design Environment*

SAŽETAK

Cilj rada bila je implementacija neuronske mreže na ZYBO razvojnom sustavu, što je postignuto razvijanjem ZYNN sustava. Grafičko korisničko sučelje sustava, koje je razvijeno koristeći *Kivy framework*, omogućuje dizajniranje i treniranje novih *Keras* modela i učitavanje postojećih modela prije slanja parametara mreže na ZYBO platformu. Nakon slanja parametara, slijedi prepoznavanje znamenaka sa slike kamere, koja je prenesena u memoriju platforme HDMI kabelom. Na platformi se prvo izvodi proces predobrade slike, kojim se otklanja šum, lociraju nakupine piksela, odnosno potencijalne znamenke, koje se zatim skaliraju i prilagođuju tako da budu što sličnije onima iz MNIST baze podataka. Nakon toga se svaka nakupina piksela provlači kroz model neuronske mreže čiji su parametri prethodno poslani. Rezultati klasifikacije se šalju i prikazuju na korisničkom sučelju na računalu. Osim toga, na monitoru, koji je spojen preko VGA sučelja, se prikazuje obrađena slika s kamere na kojoj su uokvirene nakupine piksela koje su klasificirane kao znamenka, odnosno one kojima je vjerojatnost pripadanja nekoj klasi veća od 55 %. Korisnik zatim može klasificirati drugi skup znamenaka ili odabrati ili kreirati neki drugi *Keras* model za klasifikaciju.

Ključne riječi: ZYBO, *Zynq*, *Keras*, neuronske mreže, MNIST, *Kivy*.

ABSTRACT

Title: Implementation of a neural network on the ZYBO development system

The aim of the paper was implementation of a neural network on the ZYBO development system, which was achieved by developing the ZYNN system. The graphical user interface of the system, developed using the *Kivy* framework, provides designing and training new *Keras* models and loading existing models before sending the network parameters to the ZYBO platform. After sending the parameters, the digits are recognized from the camera image, which is transferred to the platform memory via an HDMI cable. The platform first preprocesses an image that eliminates noise, locates pixel clusters, or potential digits, which are then scaled and adjusted to be as similar as possible to those of the MNIST database. Subsequently, each potential digit is passed through a neural network model whose parameters have been previously sent. The classification results are sent and displayed on the computer user interface. In addition, the monitor, which is connected via the VGA interface, displays a processed image from a camera with framed pixel clusters that are classified as a digit, that is, those that have a probability belonging to a class of more than 55 %. The user can then classify another set of digits or select or create another *Keras* model for classification.

Keywords: ZYBO, *Zynq*, *Keras*, neural networks, MNIST, *Kivy*.

ŽIVOTOPIS

Toni Birka rođen je 16. rujna 1995. godine u Virovitici. Živi u Končanici, u blizini Daruvara. Završio je Češku osnovnu školu J. A. Komenskog u Daruvaru, zatim upisuje Tehničku školu Daruvar, smjer tehničar za računarstvo. Radom „RGB Display“ 5. svibnja 2014. godine je osvojio treće mjesto na Državnoj smotri radova u Obrazovnom sektoru elektrotehnika i računarstvo za školsku godinu 2013./2014., koja je bila održana na FER-u u Zagrebu. Istim radom je 10. svibnja 2014. godine osvojio zlato za uspješan nastup na izložbi INOVA MLADI 2014. na FSB-u u Zagrebu. Nakon završetka srednjoškolskog obrazovanja je izravno upisan na preddiplomski sveučilišni studij Računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Dobiva priznanje i nagradu za postignut uspjeh u studiranju 20. svibnja 2016. godine. Sudjelovao je 22. listopada 2016. godine na dvadesetčetverosatnom natjecanju u programiranju *IEEE Extreme*. Od početka studiranja je primao stipendiju Bjelovarsko-bilogorske županije, a od ožujka 2017. godine je stipendist tvrtke *Comping* iz Zagreba. Sudjelovao je na natjecanju iz informatike na Elektrijadi u Budvi 2017. godine. Bio je dva puta demonstrator na kolegiju Digitalna elektronika. Po završetku preddiplomskog studija 2017. godine, upisuje diplomski sveučilišni studij, izborni blok Procesno računarstvo. Natjecao se u programiranju na *STEM Games*-ima 2018. i 2019. godine. Povodom obilježavanja 41. godišnjice FERIT-a dodijeljeno mu je priznanje za postignut uspjeh u izvannastavnim aktivnostima za objavljivanje znanstvenog rada na međunarodnom znanstvenom skupu ZINC 2018., kao koautor rada pod naslovom „*E2LP Extension Board for Teaching Basic Digital Electronics*“ te razvoj digitalne ploče za E2LP razvojni sustav koja se koristi u nastavi iz kolegija Digitalna elektronika.

PRILOZI

P.1 ZYNN Vivado projekt, nalazi se na CD-u.
P.2 Direktorij s *Python* skriptama, nalazi se na CD-u.
P.3 Blok dijagram iz Vivado projekta.