TALLINNA TEHNIKAÜLIKOOL
INSENERITEADUSKOND
Tartu kolledž

TALLINNA TEHNIKAÜLIKOOL

KAAMERAPILDI JA NÄRVIVÕRGU ABIL ARVUTI TÖÖ JUHTIMINE

CONTROLLING A COMPUTER WITH A NEURAL NETWORK AND A CAMERA

RAKENDUSKÕRGHARIDUSE TÖÖ

Üliõpilane: Mihkel Killo

Üliõpilaskood: 178427EDTR

Juhendajad: Ants-Oskar Mäesalu, Merik Meriste

Tartu 2021

AUTORIDEKLARATSIOON

Olen koostanud lõputöö iseseisvalt.
Lõputöö alusel ei ole varem kutse- või teaduskraadi või inseneridiplomit taotletud.
Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

30. mai 2021

Autor: /allkirjastatud digitaalselt/

Töö vastab rakenduskõrghariduse töö esitatud nõuetele

“.......” .... 202…..

Juhendaja: ....
/ allkiri /

Kaitsmisele lubatud

“.......”....................202… .

Kaitsmiskomisjoni esimees ....
/ nimi ja allkiri /

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Mihkel Killo (sünnikuupäev: 19.08.1993)

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose Kaamerapildi ja närvivõrgu abil arvuti töö juhtimine,

mille juhendajad on Ants-Oskar Mäesalu ja Merik Meriste,

1.1 reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2 üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.

3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

---

[1]*Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil.*

/allkirjastatud digitaalselt/

30.05.2021

LÕPUTÖÖ ÜLESANNE

Üliõpilane: Mihkel Killo, 178427EDTR

Õppekava, peaeriala: EDTR17/17 - Telemaatika ja arukad süsteemid, peaeriala 2 - Küberfüüsikalised süsteemid

Juhendaja(d): Ants-Oskar Mäesalu, külalisõppejõud
Merik Meriste, dotsent

Lõputöö teema:

(eesti keeles) Kaamerapildi ja närvivõrgu abil arvuti töö juhtimine

(inglise keeles) Controlling a computer with a neural network and a camera

Lõputöö põhieesmärgid:

1. Närvivõrgu treenimine käemärkide tuvastamiseks
2. Treenitud mudeli kasutamine füüsilise objekti kontrollimiseks
3. Võimalikult väiksel süsteemil ülal oleva saavutamine
4. Hinnang sellele süsteemile töö- ja õppevahendina
5. Protsessi kirjeldus

Lõputöö etapid ja ajakava:

Nr	Ülesande kirjeldus	Tähtaeg
1.	Teema valitud	01.10.2020
2.	Praktiline töö tehtud	01.12.2020
3.	Lõputöö kirjalik osa valmis	31.05.2021

Töö keel: eesti keel **Lõputöö esitamise tähtaeg:** 31. mai 2021 a

Üliõpilane: Mihkel Killo */allkirjastatud digitaalselt/* 30. mai 2021 a
/allkiri/

Juhendaja: ………………………….. .... “.....”.............201….a
/allkiri/

Konsultant: ……………………………….. .... “.....”.............201….a
/allkiri/

Programmijuht: ……………………….............................. “.....”.............201….a
/allkiri/

Kinnise kaitsmise ja/või lõputöö avalikustamise piirangu tingimused formuleeritakse pöördel

SISUKORD

SISUKORD 5
SISSEJUHATUS 7
1 TAUST 9
1.1 ARVUTINÄGEMINE 9
1.1.1 ARVUTINÄGEMINE 20. SAJANDIL 9
1.1.2 ARVUTINÄGEMINE TÄNAPÄEVAL 9
1.2 NÄRVIVÕRGUD 10
1.2.1 NÄRVIVÕRKUDE TÖÖPÕHIMÕTE 10
1.3 KÄEMÄRKIDE TUVASTAMINE 11
1.3.1 VIIPEKEELE MASINATEGA TUVASTAMINE 11
2 LÄHTEÜLESANNE 13
2.1 NÄRVIVÕRGU TREENIMINE KÄEMÄRKIDE TUVASTAMISEKS 13
2.2 TREENITUD MUDELI KASUTAMINE FÜÜSILISE OBJEKTI KONTROLLIMISEKS 13
2.3 VÕIMALIKULT VÄIKSEL SÜSTEEMIL ÜLAL OLEVA SAAVUTAMINE 14
2.4 HINNANG SELLELE SÜSTEEMILE TÖÖ- JA ÕPPEVAHENDINA 14
2.5 PROTSESSI KIRJELDUS 14
3 KASUTATAV TEHNOLOOGIA 15
3.1 KASUTATAV RIISTVARA 15
3.1.1 NVIDIA JETSON NANO 15
3.1.2 USB-KAAMERA 17
3.2 KASUTATAV TARKVARA 17
3.2.1 SSH 17
3.2.2 *LINUX*-I PÕHINE *NVIDIA JETSON NANO* OPERATSIOONISÜSTEEM 17
3.2.3 DOCKER 18
3.2.4 JUPYTERLAB 18
3.3 KASUTATAVAD PROGRAMMEERIMISKEELED 18
3.3.1 PYTHON 18
3.3.2 BATCH 19
3.3.3 BASH 19
3.4 KASUTATAV NÄRVIVÕRK 19
4 RAKENDUSE LOOMINE 21
4.1 NANO SEADISTUS 21
4.2 NÄRVIVÕRGU TREENIMINE 23
4.2.1 KAAMERAPILT JUPYTERLAB-IS 23
4.2.2 ÜLESANDE PÜSTITUS 24
4.2.3 SISENDANDMETE KOGUMINE 25

4.2.4 MUDELI LOOMINE 26
4.2.5 NÄRVIVÕRGU TREENIMINE 27
4.2.6 LÕPPTULEMUSE HINDAMINE 29
5 ARUTELU 32
5.1 VÕIMALIKUD VÄLJUNDID JA RAKENDUSKOHAD 32
5.1.1 MASINÕPPE JA ARVUTINÄGEMISE KURSUS 32
5.1.2 INTERAKTIIVNE TÖÖTUBA 32
5.1.3 HOBIPROJEKTIDE PLATVORM 32
5.2 EDASIARENDAMISE VÕIMALUSED 33
KOKKUVÕTE 34
SUMMARY 35
VIITED 36
LISA 1. KASUTATUD KOOD 37

SISSEJUHATUS

Arvutinägemine ja tehisintellekt on tänapäeval väga suure tähelepanu osaks saanud, suuresti tänu viimase aastakümne jooksul aset leidnud läbimurretele nendes valdkondades. Selle töö peamine fookus on just nende valdkondade uurimiseks ja arendamiseks valmistatud Nvidia Jetson Nano (edaspidi Jetson Nano või Nano) arenduskomplekti võimaluste ja võimekuse uurimine. [1]

Töö esimeses pooles annan ülevaate arvutinägemisest ja närvivõrkudest. Lisaks tutvustan varasemaid arenguid käemärkide tuvastamise vallas. Hiljem näitan, kuidas kasutada Jetson Nano käemärkide tuvastamise ja nende abil arvuti töö juhtimise projektis. Selle saavutan treenides närvivõrku enda kogutud andmetega ja kasutades selle tulemusena tekkinud mudelit. Lõpuks annan omapoolse hinnangu selleks, mille jaoks see projekt kasulik on ja kuhu siit edasi võib areneda.

Selle projekti tarvis valisin arvutinägemise jaoks võrdlemisi lihtsa sisendi, milleks on käemärgid. Neid on võimalik pea igal inimesel lihtsalt näidata ja mõned käemärgid on läänemaailmas väga universaalsed. Sel põhjusel on väga suur ka rakenduste hulk, mille juures käemärke saaks kasutada. Näiteks on võimalus luua viipekeelt tõlgendav programm. Või robot, mis tegutseb vastavalt juhendava operaatori viibetele. Teises suunas minnes on võimalik luua nutiseade, mis on üks aste edasi puutetundlikkusest ja reageerib viibetele. Kõiki neid asju on selle töö kirjutamise ajaks kindlasti mingil määral arendatud, kuid selle töö fookus on odava arenduskomplekti võimekuse näitlikustamine selliste arenduste jaoks.

Selle projekti raames treenisin ma närvivõrke klassifitseerimisülesandeid täitma. Kuna Nano arenduskomplekti võimsus ei ole võrreldav suurte teadustööks kasutatavate arvutiserveritega, siis närvivõrgud mida kasutasin on ainult 18 kihi sügavused. Sügavamad närvivõrgud võivad olla rohkem kui 100 kihi sügavused, kasutatud on ka 1000-kihilisi närvivõrke [2]. Siiski annab selline sügavus juba väga häid tulemusi ja sügavama närvivõrgu kasutamine annab iga lisatud kihi kohta aina vähem kasu, suurendades samaaegselt vajaminevat arvutusvõimekust.

Jetson Nano on võrdlemisi uus arenduskomplekt. Selle potentsiaal iseseisvatele õppuritele ja koolidele masinõppe jaoks on veel suurel määral kasutamata. Loodan, et käesolev töö annab lugejale aimu kuidas ja miks Jetson Nano kasutada.

Võtmesõnad: Arvutinägemine, närvivõrk, Nvidia Jetson Nano, käemärkide tuvastamine, rakenduskõrghariduse töö

1 TAUST

Esimeses peatükis tutvustan arvutinägemise ajalugu ja tänapäeva. Teen ka põgusa ülevaate närvivõrkude kasutamise põhimõtetest. Lisaks annan ülevaate käemärkide masinatega tuvastamisest tänapäeval.

1.1 Arvutinägemine

Arvutinägemine on teadusharu, mille põhieesmärgiks on arvutile anda võime tuvastada pildil nähtu. Selle saavutamiseks on läbi aegade kasutatud erinevaid meetodeid ja võtteid.

1.1.1 Arvutinägemine 20. sajandil

Masinnägemise kui teadusuuringute subjekti alguseks võib lugeda 1960. aastate lõppu. Sel ajal hoogustus arvutite kasutuselevõtt ülikoolides ja teadustöös. Tegeleti aktiivselt tehisintellekti loomisega ning selle raames tekkis ka vajadus masinnägemiseks. Alguses arvati, et arvutinägemine on lihtne probleem, mille noored teadurid suudavad ühe suve jooksul projekti raames ära lahendada. [3]

Nii lihtne arvutinägemine siiski ei olnud ja toonase suveprojekti käigus ei jõutud püstitatud eesmärkideni. 1970. aastate jooksul loodi tugev alus tänapäevasele masinnägemisele. Sellel ajal tehtud uurimused ja katsetused on andnud valdkonnale vajalikud algoritmid, millest paljud on kasutusel tänapäevalgi. Järgnevatel aastakümnetel toimus edasine areng, mille käigus võeti kasutusele ka juba teiste valdkondade jaoks väljatöötatud meetodid ja tehnikad. 1990. aastate lõpus tõi suuremad muutused endaga kaasa arvutigraafika ja masinnägemise valdkondade varasemast suurem lõimumine.

1.1.2 Arvutinägemine tänapäeval

Tänapäeval on arvutinägemise põhifookuses masinõpe. Kasutatakse sügavaid närvivõrke ja keerulisi optimeerimisi. Praktiliselt kõigis arvutinägemise valdkondades annavad sügavatele närvivõrkudele optimeeritud algoritmid paremate omadustega mudelid kui varasemalt kasutusel olnud meetodid. See on võimaldanud viimase kümne aasta jooksul võtta kasutusele näiteks näotuvastuse põhised autentimissüsteemid nutiseadmetel.

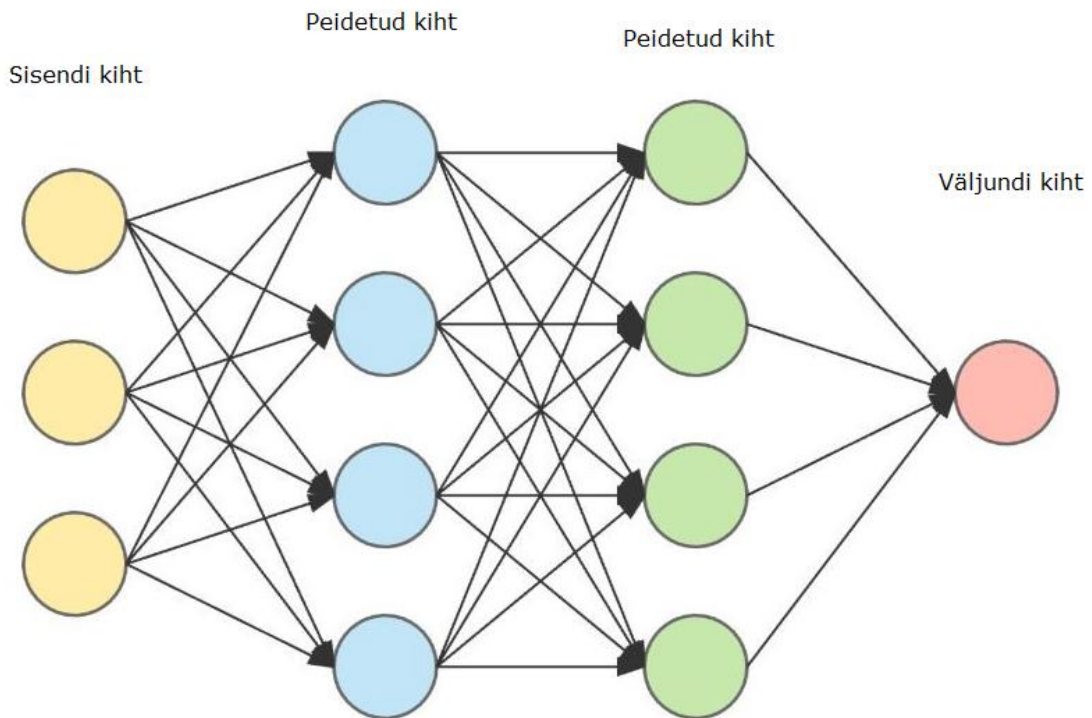
1.2 Närvivõrgud

Närvivõrgud on omavahel ühenduses olevatest neuronitest koosnevad võrgud. Närvivõrgud jagunevad bioloogilisteks ja tehislikeks närvivõrkudeks. Selles töös keskendun tehislikele närvivõrkudele, edaspidi lihtsalt närvivõrgud.

1.2.1 Närvivõrkude tööpõhimõte

Närvivõrgud koosnevad tehisneuronitest, mis enamasti jagunevad kihtidesse. Need neuronid on võimelised vastu võtma signaale. Lisaks suudavad neuronid töödelda saabunud signaale ja edastada töötlemise tulemust. Signaal, mis neuronisse tuleb, on mingi reaalarv. Neuron arvutab kõigi saabunud signaalide abil väljundi, mille see siis võib edastada kõigile temaga ühenduses olevatele neuronitele järgmises kihis. Nii neuronitel endil kui ka ühendustel neuronite vahel on kaalud. Need kaalud kas tugevdavad või nõrgendavad signaale neuronite vahel.

Närvivõrke on võimalik treenida, mis tähendab nende kaalude väärtuste muutmist. Üldiselt antakse närvivõrgu sisendi kihile ette suur hulk andmeid ja väljundkihis hinnatakse närvivõrgu tööd. Sõltuvalt iga iteratsiooni tulemusest muudetakse kaalude väärtuseid, tugevdades ühendusi, mis annavad positiivseid tulemusi, ja nõrgendades ühendusi, mis seda ei teinud. Peale treenimist saadud närvivõrku võib kutsuda mudeliks. Seda mudelit saab vastavalt vajadusele rakendustes kasutada, kui mudel osutub piisavalt heaks.



Joonis 1.1 Närvivõrgu illustratsioon

1.3 Käemärkide tuvastamine

Käemärkide tuvastamine on osa suuremast žestide tuvastamisest arvutinägemise valdkonnas. Žestide tuvastamise üks eesmärkidest arvutinägemise juures on näiteks kehakeele ja emotsioonide tuvastamine ning tundma õppimine arvutite jaoks. Käemärkide tuvastamise üks suur rakendus on kindlasti viipekeele arvutile õpetamine. Muuhulgas võimaldab see inimese ja arvuti vahelist suhtlust ning juhtimist ilma füüsiliste juhtseadmeteta. Seadmed, mida on võimalik žestide abil juhtida, on tuntud kui puutevabade kasutajaliidestega seadmed (ingl *touchless user interface* ehk TUI).

1.3.1 Viipekeele masinatega tuvastamine

Teatavasti on maailmas palju erinevaid räägitavaid keeli. Erinevad hinnangud pakuvad tänapäeval räägitavate keelte arvuks ümmarguselt 6500 keelt. Vähemtuntud on fakt, et maailmas on kasutusel ka umbes 300 erinevat viipekeelt. Sealhulgas on üks nendest keeltest Eesti viipekeel.

Viipekeele tuvastamine arvutite abil on valdkond, mida on uuritud pea sama kaua, kui arvutinägemise valdkond on vana. Aastakümnete jooksul on tehtud erinevate tehnoloogiliste lahenduste abil rohkem või vähem töötavaid süsteeme. Sellel aastatuhandel on põhiline rõhk olnud sügavate närvivõrkude kasutamisel viipekeelemärkide tuvastamiseks. Viimastel aastatel on järjest paremaid masinõppe jaoks suunatud viipekeele pildiandmekogusid lisandunud. See on võimaldanud ka teha suuri edusamme järjepideva viipekeele tõlkimise vallas. Siiski on veel pikk maa minna, enne kui on saavutatud seis, kus masin suudab ladusalt mistahes viipekeeles edastatud pikema sõnumi pädevalt kirjakeelde tõlkida. [4]

2 LÄHTEÜLESANNE

Käesoleva rakenduskõrghariduse töö eesmärgid:

- Närvivõrgu treenimine käemärkide tuvastamiseks
- Treenitud mudeli kasutamine füüsilise objekti kontrollimiseks
- Võimalikult väiksel süsteemil ülal oleva saavutamine
- Hinnang sellele süsteemile töö- ja õppevahendina
- Protsessi kirjeldus

Vastavalt ülal väljatoodud eesmärkidele sain Tallinna Tehnikaülikooli Tartu kolledži käest kasutada Nvidia Jetson Nano arenduskomplekti. Selle täpsem tutvustus asub järgmises peatükis. Need töö eesmärgid olen endale ise püstitanud ja need arenesid välja algsest soovist närvivõrke kasutada oma lõputöös.

2.1 Närvivõrgu treenimine käemärkide tuvastamiseks

See eesmärk kasvas välja minu soovist viia ennast paremini kurssi närvivõrkudega. Varasemalt õpingute käigus kasutasin ühe aine (Digitaalne tootmine, MET0340) raames närvivõrku, ilma et oleks erilist aimu kuidas see töötab. See aga oli piisavalt põnev ja salapärane, et paeluda huvi. Soov oma lõputöö teha viisil, et saaks närvivõrke selle juures kasutada tekkis juba tol ajal.

Algne plaan närvivõrgu jaoks oli seda kasutada hindamaks pilti kaamerast, mille ava on suunatud makettplaadile. Selle plaadi peal oleks olnud siis mõned elektroonikakomponendid, nagu näiteks üks valgusdiood ja mõned takistid. Töö eesmärk oleks olnud sellel närvivõrgul ära tuvastada millises konfiguratsioonis komponendid makettplaadil on, et selle järgi siis õigele klemmile vool suunata. Kuna aga üldiselt on halb mõte pinge all olevat vooluvõrku ringi ühendada sai see algne mõte asendatud käemärkide tuvastamisega. Nagu näha, jäid algsest ideest alles makettplaat ja valgusdioodid, nüüd aga närvivõrgu sisendi asemel väljundi näitlikustajana.

2.2 Treenitud mudeli kasutamine füüsilise objekti kontrollimiseks

Selle töö üks eesmärkidest on füüsilise maailma mõjutamine arvuti abil. Selleks on reaalsuses palju võimalusi, kuid siin töös piirdub see vaid paarile valgusdioodile voolu

andmisega. Etteruttavalt võin mainida, et kui juba on tehtud ühendus närvivõrgu väljundi ja füüsilise maailma klemmide vahel, ei ole raske vahetada need valgusdioodid millegi praktilisema vastu välja.

2.3 Võimalikult väiksel süsteemil ülal oleva saavutamine

See on eesmärk, mis sai küll osaliselt täidetud, kuid oleks saanud ka palju paremini seda teha. Täidetud sai see eesmärk, kuna kogu närvivõrke puudutav osa on tehtud Nvidia Jetson Nano peal. Aga algne eesmärk nägi ette närvivõrgu treenimist lauaarvutil ja selle tulemusena saadud mudeli jooksutamist oluliselt väiksemal süsteemil kui seda on Jetson Nano. Kuna Nano on oma võimekuselt võrreldav 10 aasta taguse tavakasutaja lauaarvutiga, siis ei saa ma päris rahul olla selle eesmärgi täitmisega. Projekti edasiarendamise raames eksisteerib muidugi võimalus kasutada treenitud närvivõrgu mudeleid väiksematel süsteemidel.

2.4 Hinnang sellele süsteemile töö- ja õppevahendina

Kindlasti on üks selle töö suurematest väärtustest lugejale Jetson Nano arenduskomplekti tutvustamine. Üsna selgelt saab näitlikustatud vähemalt üks kasutusjuht, mille jaoks Nano on suurepärane. Lisaks on terve viies peatükk sellest tööst nii hinnang arenduskomplektile kui ka ettepanek selle kasutamiseks erinevates kohtades. See on kindlasti eesmärk mis sai selle töö tegemise käigus saavutatud.

2.5 Protsessi kirjeldus

Käesoleva kirjatöö neljas peatükk on kogu Nano seadistamise ja närvivõrkude kasutamise protsessi kirjeldus. Lisaks on seal vajadusel seletused, kus ja mida saaks teha teisiti. Seda kas see eesmärk sai edukalt täidetud suudab kõige paremini hinnata selle töö lugeja.

3 KASUTATAV TEHNOLOOGIA

Selles peatükis annan lühida ülevaate töös kasutatud riistvarast, tarkvarast ja programmeerimiskeeltest.

3.1 Kasutatav riistvara

Selles projektis põhilisteks riistvaralisteks vahenditeks olid mu kodune lauaarvuti, USB veebikaamera, Nvidia Jetson Nano arenduskomplekt ning hobielektroonika komponendid, mis selle arenduskomplekti juures kasutusel on. Lauaarvuti spetsifikatsioonid ei ole siinkohal tähtsad, kuna see oli vajalik vaid Nano-ga ühenduse loomiseks ja tollele instruktsioonide saatmiseks.

3.1.1 Nvidia Jetson Nano

Jetson Nano on arenduskomplekt, mis on Nvidia poolt välja töötatud arvuti- ja masinnägemise projektide jaoks. Selle väiksed mõõtmed ja suur jõudlus teevad sellest ideaalse platvormi uute ideede katsetamiseks ja prototüüpimiseks.

Joonis 3.1 Nvidia Jetson Nano arenduskomplekt (ilma lisadeta) [1]

Alltoodud tabelis on ülevaade Nano arenduskomplekti tähtsamatest tehnilistest andmetest. [1]

Tabel 3.1 Nano arenduskomplekti tehnilised andmed

GPU	128-tuumaline Maxwell graafikakaart
CPU	4-tuumaline ARM A57@1.43GHz
Mälu	4 GB 64-bit LPDDR4 @25.6 GB/s
USB	4x USB 3.0 Type-A, 1x USB 2.0 Micro-B
Võrguühendus	Gigabit Ethernet, M.2 Key E
Mõõtmed	100 mm x 80 mm x 30 mm
Lisad	GPIO, I^2C, I^2S, SPI, UART
Voolutarve	10W @ 5V * 2A toitega 20W @ 5V * 4A toitega

Nagu mõõtmetest näha, on tegemist võrdlemisi kompaktse arenduskomplektiga. Kõige tähtsam väärtus, mis teeb sellest hea masinnägemise ja tehisintellekti arenduskomplekti, on ehk 128-tuumaline graafikakaart. Kuigi need tuumad ise ei ole eriti suure jõudlusega, toimivad need väga hästi just närvivõrkude treenimisel. Tänapäeva tarkvara kasutades jaguneb närvivõrgu treenimine väikesteks juppideks, mida saab paralleelselt täita. Paljutuumaline graafikakaart võimaldab seda ideaalselt. *Jetson Nano*-l on ka 40-klemmiline ühendusriba, mis vastab mikrokontrollerite maailmas tuntud Raspberry Pi arenduskomplektidele. Nende nimi on GPIO klemmid ja nende abil on võimalik kasutada erinevaid elektroonikakomponente. Selles töös on need vajalikud valgusdioodide kontrollimiseks.

Voolutarve Nano-l sõltub sellest, mis pesa kaudu seda toita. Sobib 5V/2A Micro-USB toide või sama tugevusega DC-konnektori toide. Lisaks on võimalus kasutada 5V/4A tugevusega DC-konnektori toidet, mis annab kaks korda rohkem voolu ja selle tulemusel on ka arenduskomplekti jõudlus suurem. Viimast variant kasutades tuleb aga arvestada, et pikaaegse intensiivse töö korral võib komplekti kuuluvast radiaatorist väheks jääda. Ülekuumenemise vältimiseks piirab Nano oma graafikakaardi ja protsessori voolutarvet kõrgetel temperatuuridel. On võimalus radiaatori asemel kasutada tiivikuga jahutit, kuid sel juhul tuleb see ise hankida, kuna arenduskomplektis seda kaasas pole.

3.1.2 USB-kaamera

Selles projektis kasutasin kaamerana USB-veebikaamerat Tracer WEB007. Kaamera valikul lähtusin põhiliselt sellest, et selle pildikvaliteet oleks võimalikult sarnane tavakasutaja veebikaameraga pildikvaliteediga. Tegemist on võrdlemisi standardse laua peal seisva veebikaameraga. See kaamera edastab pilti kolmes erinevas piksliformaadis: MJPG, YUYV ja H264. MJPG ja H264 puhul oli valida kolme erineva resolutsiooni vahel: 1920x1080, 1280x720 ja 640x360. YUYV formaadis oli valikus vaid üks resolutsioon, 640x360. Kõikide nende valikute juures edastab kaamera videopilti sagedusega 30 kaadrit sekundis.

3.2 Kasutatav tarkvara

Projekti tarkvaraline pool koosnes üsna mitmest erinevast tarkvara kihist ja lahendusest. Esiteks oli mul kasutusel lauaarvuti, mille peal töötas Windowsi operatsioonisüsteem. Selles arvutis kasutasin SSH-d, et saada ühendus Nvidia Jetson Nano-ga, millel põhiline töö aset leidis. Nano peale installeerisin Nvidia Deep Learning Institute poolt loodud Linux-i põhise operatsioonisüsteemi. Selle peal omakorda kasutasin Docker-it, et jooksutada JupyterLab-i serverit. Sellele serverile sain kohalikus võrgus ligi oma lauaarvutiga, et seal veebipõhises keskkonnas projekti jaoks vajalikku koodi programmeerida.

3.2.1 SSH

SSH (ingl ***S****ecure* ***Sh****ell*), on krüptograafiline võrguprotokoll. Seda saab kasutada turvaliseks arvutite vaheliseks suhtluseks. Selles projektis kasutasin enda lauaarvutit *Jetson Nano*-le käskude edastamiseks. See kõik toimus tänu SSH-le. Esimese installatsiooni jaoks ei olnud võimalik SSH-d kasutada, selle jaoks toimus suhtlus *Putty* nimelise tarkvara abil jadaliidest (ingl *Serial Communication*) kasutades.

3.2.2 *Linux*-i põhine *Nvidia Jetson Nano* operatsioonisüsteem

Baastõmmis, mille peale *Nvidia Jetson Nano* operatsioonisüsteem on tehtud, on *Ubuntu* 18.04.5 LTS (*Long Term Support*) versioon. *Ubuntu* on üks populaarsemaid *Linux*-i versioone ja on laialdaselt kasutusel infotehnoloogia valdkondades. Selle konkreetse operatsioonisüsteemi tarvis on Ubuntu tõmmisele lisatud omajagu draivereid ja tarkvara, et algse installatsiooniga enamus asju juba kohe töötaks.

3.2.3 Docker

Nvidia Jetson Nano peal kasutasin *Docker*-i tõmmisest (ingl *image*) tehtud konteinerit. *Docker* on tarkvara, mis võimaldab luua tõmmiste abil kiirelt ja valutult virtuaalseid masinaid. Tõmmistest loodud masinad jooksevad selleks otstarbeks loodud ajutistes konteinerites, mida haldab *Docker*. *Docker*-i kasutamise kasuks on asjaolu, et varasemalt valmis tehtud tõmmiseid saab jooksutada praktiliselt igal masinal mis *Docker*-it toetab. Kõik vajalikud tarkvarad ja paketid ning konfiguratsioon konteineris oleva programmi või rakenduse töötamiseks pakitakse tõmmisesse kaasa ja see teeb konteinerid lihtsasti erinevate masinate vahel liigutatavaks. Lisaks on võimalik teha erinevaid avalikult saada olevaid tõmmiseid aluseks võttes või täiesti nullist enda tõmmiseid, kasutades selleks tõmmisfaile (ingl *Dockerfile*).

3.2.4 JupyterLab

JupyterLab on tarkvara, mis on osa suuremast *Project Jupyter*-i nimelisest projektist. Selle projekti eesmärk on pakkuda vabavaralist tarkvara, mida saab kasutada interaktiivseks programmeerimiseks. *JupyterLab* on selle projekti üks uuemaid kasutajaliideseid ja oma töö raames jooksutasin *Jetson Nano* peal *JupyterLab*-i serverit. See võimaldas mul oma arvuti brauseriaknast avada kasutajaliidese, mille abil sain kirjutada vajalikku koodi ja visualiseerida protsessi ning tulemusi.

3.3 Kasutatavad programmeerimiskeeled

Selle projekti jaoks vajamineva koodi kirjutasin valdavas osas Python-i programmeerimiskeeles. Siiski läks vaja veidi teadmisi ka teistest, operatsioonisüsteemide spetsiifilistest keeltest.

3.3.1 Python

Python on väga populaarne ja lihtsa süntaksiga kõrge-tasemeline programmeerimiskeel. Erinevalt mõnest teisest populaarsemast keelest nagu Java või C, on Python interpreteeritud keel. Sellel ei ole kompilaatorit, mis tähendab et koodi ei kompileerita enne jooksutamist, vaid käske täidetakse rida-rea kaupa, kuni kood otsa saab või mõni viga töö edasise jätkamise võimatuks teeb. JupyterLab-i keskkonnas tähendab see näiteks seda, et koodijuppe on võimalik käivitada vastavalt vajadusele ja esimesed osad saavad oma töö ära teha ilma, et viimased osad oleks veel valmiski

kirjutatud. Ühe sessiooni raames jooksutatud koodijuppide tulemused on teistele osadele koodis kasutatavad.

Lisaks eelmainitule on Python väga populaarne ka masinõppe valdkonnas. Selles keeles on hulgaliselt teeke ja vahendeid, mis hõlbustavad tööd närvivõrkudega. Kõige kuulsamad neist on näiteks PyTorch [5] ja TensorFlow, millest esimest ma ka siin projektis kasutan. Lisaks on siin projektis kasutatud veel teisigi teeke, nagu näiteks JetCam [6] kaamera kasutamiseks, IPython ja ipywidgets kasutajaliidese loomiseks.

3.3.2 Batch

Batch on Windowsi käsurea keel. See ei ole kõrgetasemeline programmeerimiskeel nagu Python, vaid mõnest lihtsamast käsust koosnev interpreteeritud skriptimise keel. Batch keeles Windows-il jooksutamiseks mõeldud skriptid on laiendiga '.bat' ja tuntud kui batch failid. Siin projektis kasutasin batch-i vähe, põhiliselt ainult käsurealt Jetson Nano-ga SSH ühenduse loomiseks.

3.3.3 Bash

Bash on Linux-i vaste Windowsi batch keelele. See ei ole kindlasti ainus keel, mis erinevate Linux-i versioonide peal kasutusel, kuid Ubuntu-s on see vaikimisi kasutusel. Kuna Jetson Nano kasutasin headless viisil (sellest pikemalt järgmises peatükis), oli bash käsurida ainuke viis Nano peal midagi ära teha. Selle keele abil navigeerisin failisüsteemis, tegin shell skripte ja kirjutasin Dockerfile-i, tekitasin tõmmiseid nendest samadest failidest ja käivitasin neid.

Väga rangelt võttes ei saa ei batch-i ega ka bash-i arvestada kui programmeerimiskeelt. Tehniliselt on tegemist siiski käsurea liidesega, mille põhiline eesmärk on võimaldada käsurea kaudu jooksutada erinevaid vajalikke programme kasutaja poolt määratud argumentidega. Siiski tõin need siinkohal välja, kuna minimaalne teadmine neist on selle projekti jaoks tarvilik.

3.4 Kasutatav närvivõrk

Käesolevas osas kirjeldan enda töös kasutusel olevat närvivõrku. Selleks on ResNet18 nimeline närvivõrk. Oma nimetuse saab see enda tüübi järgi, milleks on residuaalne

sügav närvivõrk (ingl residual neural network). Number 18 selle nimes tähistab selle konkreetse närvivõrgu puhul tema kihtide arvu.
Residuaalsed närvivõrgud teeb eriliseks see, kuidas signaalid nendes liiguvad. Kui tüüpilise närvivõrgu puhul signaal rändab kiht-kihi haaval väljundkihi poole, siis residuaalsetes närvivõrkudes võivad osad signaalid mõned kihid täielikult vahele jätta. See kiirendab algse treenimise protsessi, mille jooksul närvivõrk õpib kõigepealt robustsemad tunnused selgeks. Osade kihtide vahele jätmine võimaldab lihtsamini treenida sügavaid närvivõrke, mille treenimine muidu võib osutuda keeruliseks. Lihtsa näitena põhjustavad väikesed variatsioonid algsetes kihtides suuri muutusi hilisemates kihtides. Peale treenimise algfaasi võib närvivõrk esialgu vahele jäetud kihtide osad kasutusele võtta. Algselt vahele jäänud osasid kasutades laieneb treenimise käigus tunnuste hulk, millega närvivõrk arvestab. Selline närvivõrgu treenimine võimaldab luua tugevama põhja mudelile, mis on vähem häiritud väikestest muudatustest ning mitteolulistest tunnustest ja üldiselt saab paremini oma ülesannetega hakkama.

Käesoleva töö praktilises osas katsetasin ka teisi närvivõrke. Minu valikus olid veel ResNet34, AlexNet ja SqueezeNet. ResNet34 on samasugune residuaalne närvivõrk nagu ResNet18, lihtsalt peaaegu kaks korda sügavam. AlexNet ja SqueezeNet on vanemad närvivõrgud, mille ühiseks nimetajaks on konvolutsioonilised närvivõrgud. Töö käigus aga ei õnnestunud mul AlexNet-i ja SqueezeNet-i närvivõrkudele mudeli treenimine tarkvarade versioonide mitteühilduvuse tõttu. ResNet34 kasutades õnnestus närvivõrgu treenimine kasutatavaks mudeliks, kuid selle treenimisaeg oli mitu korda suurem kui ResNet18 oma. Samas märgatavat vahet nende kahe mudeli võimekuses ei eksisteerinud. Seega loen kasutatavaks närvivõrguks ResNet18 ja kõik järgnevad protsessid ning tulemused on saadud seda kasutades.

PyTorch-i arvutinägemise orienteeritud teek Torchvision võimaldab valida ResNet18-le eeltreenitud ja treenimata varianti. Eeltreenitud on see närvivõrk selle projekti raames piltide klassifitseerimise ülesannete jaoks. Muuhulgas nõuab eeltreenitud mudel rangemalt paika pandud sisendandmeid. Nende kohta on täpsemalt kirjas neljandas peatükis. Selles töös kasutasin nii eeltreenitud kui ka treenimata varianti ResNet18 närvivõrgust.

4 RAKENDUSE LOOMINE

4.1 Nano seadistus

Et kasutada Nano arenduskomplekti täiel võimsusel, valisin sellele 5V/4A toite. Kuna tahtsin ka jätta võimalikult palju ressurssi närvivõrkude treenimiseks ja piltide töötlemiseks, kasutasin seda *headless* viisil. See tähendab, et ma ei installeerinud sinna graafilist kasutajaliidest ja kasutasin teist arvutit sellega suhtlemiseks. Lisaks ühendasin Nano külge USB veebikaamera ja CAT 5e internetikaabli, mille teine ots ühendus ruuteri külge. Esmase seadistuse ja operatsioonisüsteemi paigaldamise tegin micro-SD kaardi ja USB A – USB Micro-B kaabli abil. Selle kaabli kaudu lõin *serial*-ühenduse arvuti ja *Nano* vahel, tänu millele sain installeeritud operatsioonisüsteemi ja kätte *Nano* lokaalse võrgu IP-aadressi. Kogu edasise töö jaoks USB kaablit enam vaja ei olnud, kogu vajalik suhtlus arvuti ja *Nano* vahel hakkas toimima läbi SSH.

Sellisena üles seatult kasutas *Nano* jõudeolekus oma olemasolevast 4 GB mälust ainult 400 MB, ehk umbes 10%. Veel ~140 MB kulus *Docker*-i konteineri jooksutamisele, mille sees kogu närvivõrkude treenimine aset leidis. Selle konteineri kujutise algne versioon pärineb *Nvidia Deep Learning Institute* (NvDLI) kursuste lehelt [7] ja selle eesmärk on jooksutada *JupyterLab*-i serverit. Sellel serveril on üles seatud JupyerLab Notebook keskkond ja mõned abistavad koodijupid. Käsk selle konteineri jooksutamiseks on järgnev:

```
sudo docker run --runtime nvidia -it --rm --network host \
        --volume ~/nvdli-data:/nvdli-nano/data \
        --volume /etc/udev/rules.d:/etc/udev/rules.d \
        --volume /dev:/dev \
        --volume /sys/class/gpio:/sys/class/gpio \
        --volume /sys/devices:/sys/devices \
        --device /dev/video0 \
        --device /dev/gpiochip0:/dev/gpiochip0 \
        --device /dev/gpiochip1:/dev/gpiochip1 \
        jupyterlab-killo:latest
```

Käsu esimene rida ütleb *Nano*-le, et see jooksutaks *Docker*-i konteinerit *Nvidia runtime*-is ja laseb konteineril kasutada hosti (Nano) võrguühendust. Lipud (ingl *flags*) --it ja --rm on vastavalt konteineri interaktiivseks jooksutamiseks ja peale konteineri kasutamise lõppu selle eemaldamiseks. Teine rida ühendab minu poolt eelnevalt tekitatud Nano kausta nimega /nvdli-data konteinerisisese kaustaga /nvdli-nano/data. See lubab salvestada konteineri töö jooksul tekkinud andmeid Nano SD-kaardile. Järgmised neli '--volume' algusega rida on vajalikud selleks, et konteineri seest saaks kontrollida GPIO klemme. Samal eesmärgil on käsus kaks viimast '--device' algusega rida. Esimene '--device' algusega rida on USB videokaamera konteineriga ühendamiseks.

Käsu viimane rida on minu tehtud Docker-i tõmmis, mis võttis baasiks Nvidia kursuste lehelt Docker-i tõmmise. [8] Minu versioonis on sellele kujutisele lisatud üks Git-i vabavaraline tarkvarateek [9], mis võimaldab kontrollida GPIO klemme Docker-i konteineri seest Pythoni koodi abil. Dockerfile-i sisu:

```
FROM nvcr.io/nvidia/dli/dli-nano-ai:v2.0.1-r32.4.4
RUN pip3 install git+https://github.com/Heerpa/jetson-gpio
```

Selle Dockerfile-i abil tõmmise tegemine käis Jetsonis käsuga:

```
docker build -t jupyterlab-killo .
```

Et eelmainitud pikka käsku ei peaks iga kord kui serverit vaja jooksutada sisse trükkima, tegin sellest käsureaskripti nimega docker_dli_run.sh ja andsin sellele õigused jooksmiseks käsuga:

```
chmod +x docker_dli_run.sh
```

Edaspidi sain seda serverit jooksutada vaid käsuga:

```
./docker_dli_run.sh
```

Sama efekti oleks saanud ka kirjutades vastavasisulise *Docker Compose* faili, kuid selles töös ma seda teed ei läinud. Docker Compose on programm, mis lihtsustab Docker-i tõmmiste kasutamist, eriti suure hulga argumentide korral.

JupyterLab-i server avalikustas *Nano*-l pordi 8888, mille kaudu sain lokaalses võrgus enda arvuti veebibrauserist sellele ligi. Kogu edasine töö leidis aset minu enda arvutil, mille abil kirjutasin *JupyterLab*-i keskkonnas *Python*-i keeles koodi. [10]

4.2 Närvivõrgu treenimine

Eelkirjeldatud moel valmis seatud Nano-l klassifitseeriva närvivõrgu treenimine on võrdlemisi sirgjooneline protsess. Osa minu projekti jaoks vajalikust koodist oli juba JupyterLabi Notebookis olemas. Selle projekti jaoks vajalikud muudatused ja lisamised tegin koodis oma võimetele vastavalt. Kogu treenimiseks ja hindamiseks kasutatud kood on kirjas lisas 1. Minu lisatud või muudetud kood on seal tähistatud

Treenimine koosneb järgmistest suurematest osadest:

- Kaamerapildi kättesaamine
- Ülesande püstitus
- Sisendandmete kogumine
- Mudeli loomine
- Närvivõrgu treenimine
- Lõpptulemuse hindamine

4.2.1 Kaamerapilt Jupyterlab-is

Et kasutada Nano külge ühendatud kaamerat, kasutasin Python-i teeki nimega JetCam [6]. See teek on tehtud just nimelt Jetson Nano jaoks, et kasutada USB ja CSI (ingl *camera serial interface*) kaameraid. Peale kaamera Jetsoni külge ühendamist uurisin järgnevate käskude abli välja, mis formaadis see kaamera suudab pilte ja videot edastada:

```
sudo apt-get install v4l-utils
v4l2-ctl --list-formats-ext
```

Esimese käsu abil installeerisin vajaliku tarkvara, milleks on v4l/v4l2 ja selle tööriistad. Need lühendid tähendavad Video4Linux tarkvara kollektsiooni, kus v4l2 tähendab selle tarkvara teist versiooni. Nende tarkvarade abil on käsurealt võimalik uurida vajalikku informatsiooni ja kontrollida kaameraid. Teise käsuga kuvasin kõik formaadid ja suurused, mida kasutatav kaamera on võimeline edastama. Neid oli vaja selleks, et JetCam-i teegile ette anda õigete parameetritega infot. JetCam-i jaoks oli ainus sobilik

piksliformaat sellel kaameral YUYV suurusega 640*360 pikslit ja seda formaati suutis kaamera edastada 30 kaadrit sekundis. Kaamera valmis sättimine JupyterLab-is käis järgnevate käskudega:

```
from jetcam.usb_camera import USBCamera

camera = USBCamera(width=224, height=224, capture_width=640, capture_height=360,
capture_device=0)
```

Esimene rida sellest käsust on käsk teegi impordiks, teine seadistab muutujale nimega 'camera' vastavaks Jetsoni küljes oleva kaamera. 'USBCamera' parameetrid on järjekorras kuvatava pildi laius ja kõrgus, kaamera poolt edastatava pildi laius ja kõrgus, ning viimaseks seadme tunnus. See viimane vastab varasemalt konteineri käivitamisel kasutatud parameetrile:

```
--device /dev/video0 \
```

4.2.2 Ülesande püstitus

Ülesande püstitamine on koodiblokk, kus panin paika kategooriad, mida närvivõrk peab suutma tuvastada. Nendeks kategooriateks määrasin neli erinevat käemärki:

- pöial üles
- pöial alla
- käsi rusikas
- peopesa

Kaamerapildi kasutamiseks närvivõrgus on seda vaja eelnevalt töödelda. Närvivõrkude treenimisel väikse koguse treeningandmetega on oht teha mudel liialt spetsiifiline. Selle vältimiseks on mõistlik iga kord enne kui pilt närvivõrku sisendiks anda seda teatud määral muuta. Selle saavutamiseks kasutasin *transform*-i reegleid *Torchvision*-i jaoks.

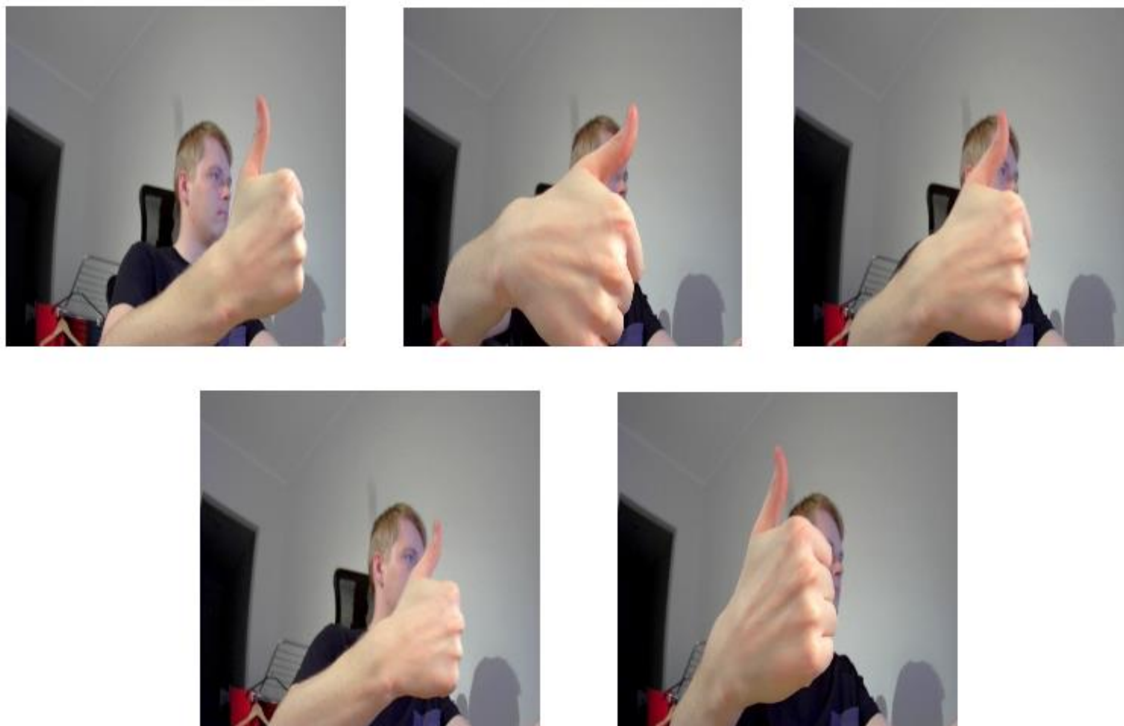
Transformid, mida sisendpiltide peal kasutatakse iga kord kui neid treenimiseks kasutada:

```
TRANSFORMS = transforms.Compose([
    transforms.ColorJitter(0.2, 0.2, 0.2, 0.2),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

Siin on näha nelja erinevat transformi reeglit. Esimene neist, ColorJitter, muudab pildi heledust, kontrasti, saturatsiooni ja värvust juhuslikult kuni defineeritud määrani. 0.2 tähendab siin kontekstis kuni 20% muutust. Teine transform teeb kõik pildid 224*224 pikslit suureks. Kolmas transform teeb kõigist piltidest tensorid ja neljas normaliseerib need tensorid mis ühes hulgas eksisteerivad. Normaliseerimiseks kasutatavad parameetrid on pärit TorchVisioni-i enda dokumentatsioonist. Esimeses blokis normaliseerimise funktsioonist on tensorite kanalite aritmeetilised keskmised, teises blokis on nende kanalite standardhälbed. Siin kasutatavad normaliseerimise väärtused on need, mida nõuavad Torchvision-i eeltreenitud närvivõrgud. Kui kasutasin mitte eeltreenitud närvivõrku, oleksin saanud neid väärtusi muuta, kuid tegemist on üldkasutatavate väärtustega kuna need annavad häid tulemusi.

4.2.3 Sisendandmete kogumine

Sisendandmed närvivõrgule on kaamerapildid. Et neid koguda, tekitasin *JupyterLab*-i selleks otstarbeks kasutatava teegi *IPyWidgets*-i abil graafilise kasutajaliidese [11]. Selle liidese abil sain tekitada vajaliku hulga kategoriseeritud pilte. Iga kategooria jaoks kogusin 50 pilti enda tehtud käemärkidest. Piltide tegemisel üritasin varieerida käe asendit ja kaugust kaamerast. Kui tegemist oleks seadmega, mis peab olema igas olukorras töökindel, oleks mõistlik olnud varieerida nii paljusid elemente sisendpiltidest kui võimalik. Näiteks varieeruv valgustus, taust, erinevad inimesed ja nii edasi. Kuna käesolev töö pidi töötama vaid minu arvutilaua taga, on ka pildid vastavad.



Joonis 4.1. Pildid kategooria pöial üles treenimiseks

4.2.4 Mudeli loomine

Mudeli loomine on selle töö raames protsess, millega pannakse paika treenimisseadme arhitektuur ja närvivõrgu tüüp. Nagu eelnevalt mainitud, kasutasin siin töös närvivõrku ResNet18. Selle paika panemine käib Torchvisioni abil järgnevalt:

```
import torch
import torchvision

model = torchvision.models.resnet18(pretrained=True)
model.fc = torch.nn.Linear(512, len(dataset.categories))

device = torch.device('cuda')
model = model.to(device)
```

Esimesed kaks rida sellest koodist impordivad PyTorchi Pythoni teegid. Edasi toimub Torchvisioni teegist mudeli valik, sealhulgas ka valik eeltreenitud või treenimata mudeli

vahel. Järgmisel real defineerin mudelile väljundi eelse täielikult ühendatud (ingl fully-connected) kihi, kus eelnevatelt kihtidelt sisse tulnud 512 tunnusest ühendatakse need lineaarselt minu defineeritud nelja kategooriaga. Peale seda defineerin ära PyTorch-i jaoks kasutatava arhitektuuri, milleks on CUDA [12]. Selle annan ka *model.to(device)* käsu abil mudeli parameetriks.

4.2.5 Närvivõrgu treenimine

Võttes mudeli aluseks ja transformeeritud pildid sisendiks, saab treenida närvivõrku. Selle protsessi haldamiseks on kasutajaliidesel mõned väljad. Nendeks on treeningtsüklite (ingl epochs) arvu määramiseks väli, progressiriba mis näitab iga tsükli progressi, väljad käesoleva täpsuse ja vea näitamiseks, ning nupud treenimise alustamiseks ja hindamisprotsessi alustamiseks. Lisasin sellele liidesele ka välja, mis töötas stopperina. See võimaldas mul treenimiseks kulunud aja kohta täpseid tulemusi mõõta. Lisaks tekitasin sinna välja, kus märkisin peale treenimise lõppu keskmise tsükli peale kulunud aja.

Et saada võimalikult täpseid mõõtmistulemusi, mis ei ole sõltuvad süsteemi kasutusel olnud ajast, treenisin enne ajavõtu mudeleid mitu erinevat mudelit soojenduseks. Minu lootus on see, et kuna Jetson Nano reguleerib oma võimsust vastavalt temperatuurile, saavutasin nende mudelite treenimise ajaks juba töötemperatuuri. See ehk välistas võimaluse, et üks mudel treenis kiiremini tänu sellele, et süsteem alustas madalamalt temperatuurilt suurema võimsusega.

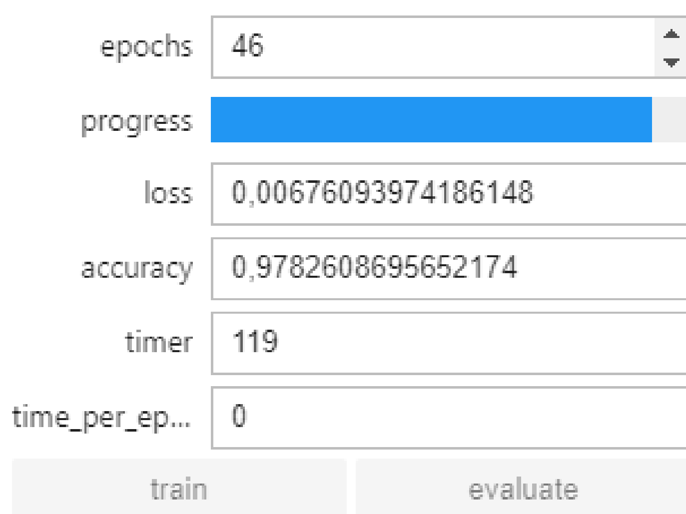
Treenimise jaoks oli minu andmekogus nelja kategooria kohta kokku 200 pilti. Iga treeningtsükli alguses jaotati need 200 pilti 8 kaupa puntidesse (ingl batch). Üks punt läks korraga transformeerimisse ning seejärel närvivõrku sisendiks. Kui iga punt oli närvivõrgust läbi käinud, lõppes ka treeningtsükkel. 200 pilti kaheksa kaupa pundis teeb kokku 25 punti. Treenimise ajal progressiriba jälgides arvutus klappis, sest kasutajaliides uuendas ennast peale iga pundi treenimiseks kasutamist. Oli näha, et see progressiriba teeb 25 sammu, enne kui lõppu jõuab ja algab uus tsükkel. Kahesaja pildi korral võttis iga tsükli treenimine veidi üle 20 sekundi aega. Täpsemad tulemused on toodud tabelis 4.1.

Tabel 4.1. Treenimiseks kulunud aeg ja kao-funktsiooni tulem

Mudel	Tsükleid	Aeg s	Aeg/tsükleid s	Kadu
Eeltreenitud ResNet18	-	-	-	0.1845
Eeltreenitud ResNet18	10	235	23.5	0.0002
Eeltreenitud ResNet18	30	653	21.77	0.00018
Eeltreenitud ResNet18	50	1030	20.6	0.00001
Eeltreenimata ResNet18	50	1041	20.8	0.00003

Selles tabelis on ära toodud erinevad minu treenitud mudelid ja nende treenimiseks kulunud aeg. Lisaks on viimases tabeli tulbas ka kaofunktsiooni arvutuslik tulemus viimase treeningtsükli järel. Esimesel real on hindamise režiimis ühe korra kõik pildid närvivõrgust läbi lastud. Nagu näha, on kadu ääretult suur. Eeltreenitud, kuid mitte minu poolt enda andmetega treenitud, ResNet18 närvivõrgu mudel ei olnud mingilgi määral võimeline hindama mis käemärki talle näidati.

Iga rida peale esimest on närvivõrgu mudel, mille ma treenisin siis vastavalt kas eeltreenitud või viimasel juhul eeltreenimata mudelist. Nagu näha, siis keskmine treenimisele kulunud aeg tsükli kohta lühenes veidi kui korraga treenida rohkem tsükleid. Ei olnud ka suurt vahet treenimisele kulunud ajas eeltreenitud ja eeltreenimata närvivõrgul. Ootuspärane on ka see, et treenides mudeleid rohkem tsükleid kadu vähenes.

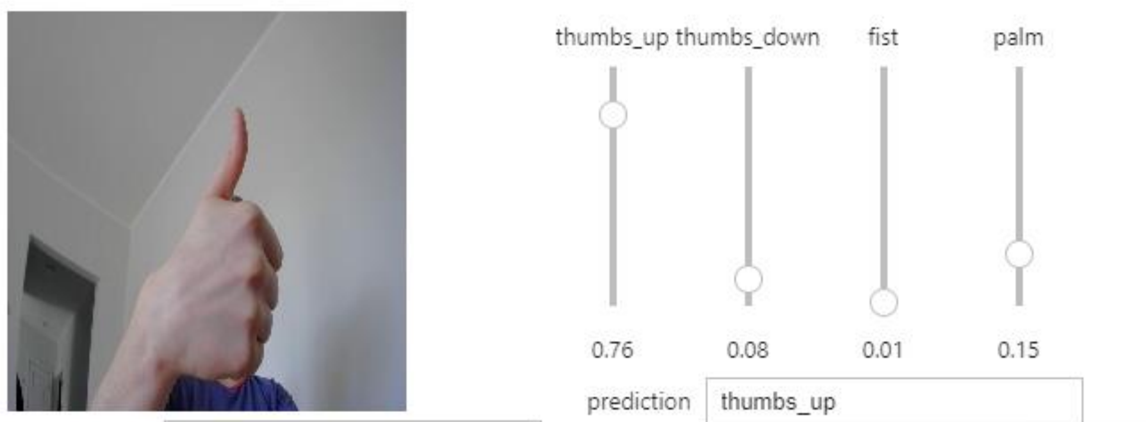


Joonis 4.2. Närvivõrgu treenimise osa kasutajaliidesest nähtuna treenimise ajal

4.2.6 Lõpptulemuse hindamine

Lõpptulemuse hindamine toimus peale mudeli treenimist. Selle jaoks oli mitu erinevat moodust. Esiteks, reaalajas näitas loodud kasutajaliides skoore igale toodud kategooriale, nagu näidatud joonisel 4.3. Teine variant seisnes GPIO klemmidega ühendatud kahe valgusdioodi abil tulemuse näitlikustamises. Kolmas hindamise kriteerium oli kao-funktsiooni väljund.

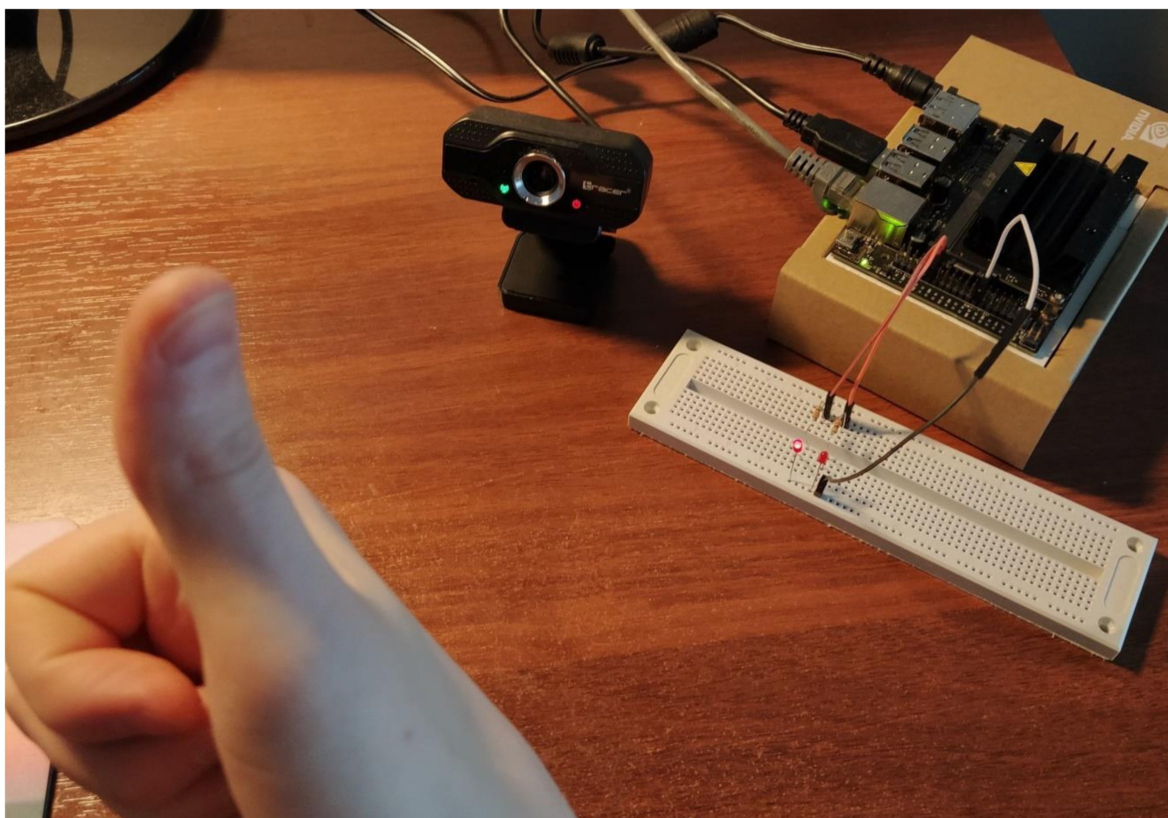
Varasemalt Nano seadistuse alapeatükis kirjeldatud moel andsin konteineris jooksvatele programmidele õiguse kasutada GPIO klemme. Nende juhtimiseks närvivõrgu väljundite abil koostasin eraldi programmijupi, mis võrdles iga programmitsükli tulemusel hinnatud skoore. Kui nendest kõige kõrgem skoor ületas 50% piiri, siis kasutasin selle skoori kategooriale vastavat klemmide konfiguratsiooni, et valgusdioode kontrollida.



Joonis 4.3. Kasutajaliides tulemuse hindamise faasis nähtuna

Kui kasutada dioode binaarsetena (sees - väljas), on võimalik kahe dioodiga näidata nelja erinevat konfiguratsiooni. Selles töös sättisin käemärgi „pöial üles" võrduma dioodidel sellega, et esimene oli sisse lülitatud, teine välja. „Pöial alla" oli sellele täpselt vastupidine. „Rusikas" pani ideaalis põlema mõlemad dioodid ja „peopesa" lülitas mõlemad välja. Veidi raskendav ainult dioodide abil hindamise puhul oli see, et mõlemad dioodid panin kustuma ka juhul, kui mudel vähemalt 50%-se kindlusega ei suutnud ühte käemärki tuvastada. See on koht, kus oleks võinud kasutada veidi keerulisemat eristavat meetodit, näiteks dioodide kiire sisse-välja lülitamine.

Programm reageeris muutuvatele käemärkidele üllatavalt kiirelt, näiteks õnnestus mul mõnikord näidata sekundis mitut käemärki, millele programm ka reageeris dioodide vastava ümberlülitusega. Kasutajaliideses muutusid asjad Nano mälupuuduse tõttu tihtipeale viivitusega.



Joonis 4.4. Mudeli hindamine peale treenimist. Pildil näha Nano, kaks dioodi, kaamera ja „pöial üles” käemärk.

Tabelis 4.1 välja toodud mudelid said käemärkide klassifitseerimisega hakkama erinevalt. Eeltreenitud ResNet18 närvivõrgu mudel ilma minupoolse treenimiseta ei olnud mingil määral võimeline minu tehtud käemärke klassifitseerima. Mis on ka täiesti ootuspärane tulemus. Natuke huvitavamaks läheb asi siis, kui uurida erinevate tsüklite arvu jagu treeninud mudeleid.

Eeltreenitud närvivõrk, mida treenisin 10 tsüklit, sai hakkama ainult kolme käemärgiga. Kõige edukamalt ja kindlamalt tuvastas närvivõrk ära „pöial alla‟ käemärgi. Enam-vähem hästi sai see hakkama ka „pöial üles‟ ja peopesa käemärkidega. Aga mitte kordagi ei tuvastanud mudel ära rusikas kätt.

Eeltreenitud närvivõrk, mida treenisin ise 30 tsüklit, oli töökindluselt väga sarnane 10 tsüklit treeninud närvivõrgule. Endiselt ei saanud mudel rusikas käe tuvastamisega hakkama, teiste käemärkide puhul oli märgata veidi kindlamat tuvastamist.

Kõige parema tulemuse andis 50 tsüklit minu poolt treenitud ResNet18 eeltreenitud närvivõrk. Selle töökindlus oli tunduvalt parem kui eelmise kahe mudeli oma. See klassifitseeris õigesti kõik neli käemärki, kuigi rusikas käe osas ei olnud hinnang tavaliselt üle 70% kindlusega. Teiste käemärkide puhul oli mudel enamasti täiesti kindel, et ma neid parasjagu näitan.

Eeltreenimata ResNet18 puhul olid tulemused peale 50 treeningtsüklit vaevu rahuldavad. Mudel sai peaaegu 100 protsendilise kindlusega hakkama „pöial üles" ning „pöial alla" klassifitseerimisega, kuid ei suutnud tuvastada rusikas kätt. Lahtise peopesa korral oli mudel umbes 50% kindel, et tegemist on peopesaga, sõltuvalt täpsest käe asetusest. Oma töökindluse poolest meenutas see mudel kõige enam 10 tsüklit treeninud eeltreenitud närvivõrgu mudelit.

5 ARUTELU

Kaamera ees pöidla näitamise teel valgusdioodi põlema panemine erilist väärtust ei oma. Siiski on selles töös kasutatavaid tehnoloogiaid ja protsesse võimalik rakendada suure hulga ülesannete täitmiseks. Selles peatükis toon mõne neist näiteks. Lisan ka mõne mõtte, kuhu selle projektiga edasi areneda.

5.1 Võimalikud väljundid ja rakenduskohad

Arvutinägemine on valdkond, kus areng on pidev. Nagu esimeses peatükis sai mainitud, on see valdkonnana eksisteerinud 1960. aastatest saadik. Käesoleva töö eesmärk ei olnud rajada masinnägemise teadustippude teed, vaid uurida selle kasutuselevõtmise keerukust ühel kindlal platvormil. Olles selle töö ära teinud, on selge et nii väikese projektiga autor Nano arenduskomplekti täit võimekust ära ei kasutanud. Järgnevalt aga mõned rakenduskohad selle töö põhjal.

5.1.1 Masinõppe ja arvutinägemise kursus

Jetson Nano saaks võtta kasutusele kõrghariduses õppekavadel ja ainetes, millel on tegemist tehisintellekti, asjade interneti ja targa taristuga. Selle töö raames selgus, et sarnase raskusastmega töö sobiks rühmaprojektiks. Seda küll eeldusel, et on olemas ka juhis, kuidas vajalikke seadistamisi teha. Lisaks peaks siis igale rühmale, kes sellist projekti teeb, andma ka konkreetse eesmärgi.

5.1.2 Interaktiivne töötuba

Sarnaselt eelmisele punktile saaks Jetson Nano peale tehtud rakendust kasutada töötubades ja huviringides. Võimalusi, kuidas seda täpselt realiseerida, on palju ja need sõltuvad suuresti sihtgrupi vanusest. Üks plusspunkt Nano sellisel otstarbel kasutamiseks on tema väikesed mõõtmed. Seda on lihtne transportida ja kasutajaliidesega variandis on talle vaja lihtsalt külge ühendada monitor ja klaviatuur/hiir, mille abil saab kogu tegevust vastavalt vajadusele kontrollida.

5.1.3 Hobiprojektide platvorm

Kindlasti on Jetson Nano ka suurepärane platvorm asjaarmastajatele, kes soovivad teha mingi põneva lelu või katsetada oma viimaseid ideid. Peale esmast seadistamist on selle

arenduskomplekti kasutamine võrdlemisi meeldiv kogemus. Kuna komplekt vajab tugevat voolu tööks, võib olla selle kasutamine liikuval seadmel keerulisem kui näiteks Arduino või Raspberry arendusplaatide kasutamine. Aga autori arvamus siinkohal on, et Nano võimekus ja kasutusmugavus trumpavad selle miinuse üle.

5.2 Edasiarendamise võimalused

Võttes selles projektis tehtud töö aluseks on võimalik areneda mitmesse suunda. Piisava aja ja tahtmise juures saaks GPIO klemmide külge ühendada midagi kasulikumat kui paar valgusdioodi. Sellel platvormil on piisavalt potentsiaali, et sellega ära teha väga huvitavaid masinõppe/tehisintellekti projekte ja lahendusi. Mõned nendest lahendustest on leitavad Nvidia Community Jetson-i lehel [13].

Teisest küljest on variant treenida Jetson Nano abil enda vajadustele vastav närvivõrk ja siis selle treenimise tulemusena saadud mudelit kasutada eesmärgipäraselt rakenduses. On võimalus teha endale kas näiteks arvuti veebikaamera ees multimeedia kasutamiseks puutevaba kasutajaliides. Või teha sellest viipekeelt teksti või kõnekeelde tõlkiv robot. Baas on selle jaoks olemas, vaja on ainult visiooni ja pealehakkamist.

KOKKUVÕTE

Arvutinägemine on suur ja keeruline valdkond. Selles töös puudutasin vaid selle valdkonna ühte pinnapealset osa, milleks on piltide ja videote klassifitseerimine. Siiski selle töö käigus õppisin palju ja saavutasin eesmärgid, mille endale algselt seadsin. Omandasin hulgaliselt teadmisi arvutinägemisest ja masinõppest, närvivõrkudest ja nende kasutamisest. Väga hea oli ka minu kogemus Jetson Nano-ga töötamisel.

Töö põhiliseks eesmärkiks oli närvivõrgu treenimine ja selle kasutamine käemärkide tuvastamiseks. Lisaks sellele oli eesmärgiks tuvastatud käemärke kasutada sisendina füüsilise objekti konfiguratsiooni muutmiseks. Need mõlemad eesmärgid said täidetud. Kasutasin alustena residuaalseid närvivõrke, mille treenisin siis enda kogutud andmetega käemärke klassifitseerima. Saadud mudelit rakendasin, et kaamerapilti sisendina kasutades lülitada sisse-välja valgusdioode vastavalt sellele, mis käemärgiga parasjagu tegu oli.

Nvidia Jetson Nano arenduskomplekt sobib sellise raskusastmega töö tegemiseks suurepäraselt. Tegemist on hea tööriistaga masinõppe ja tehisintellekti valdkondades katsetamiseks. Nano-l on piisavalt võimsust, et saada võrdlemisi lühikese ajaga hakkama minu ette antud koguse treeningandmete abil närvivõrgu treenimisega. Sellel arenduskomplektil on potentsiaali olla väga heaks õppevahendiks kõrgkoolis ja huviringides.

Selle töö raames valminud mudeleid oleks saanud teha paremaks, kasutades suuremat kogust treeningandmeid. Kuna tegemist on ühe autori tööga, siis kõigil närvivõrkude sisendandmetel on tegemist minu enda kätega. See suurendab ohtu selleks, et teiste inimeste jaoks see närvivõrgu mudel nii hästi ei tööta. Kui sellest tööst teha edasiarendus, on see üks ohukoht, mida tuleb kindlasti jälgida.

SUMMARY

Computer vision is a massive and complicated field of study. During this project, I only managed to come in contact with a tiny part of this field, namely classifying pictures and videos. Nonetheless I learned a lot and achieved my initial goals. I accumulated a lot of knowledge about the fields of computer vision and machine learning, also of neural networks and how to use them. Using Nvidia Jetson Nano for my project was also a pleasant experience.

The main goal of this project was to train a neural network and use it to classify hand signs. An extra goal was to use these hand signs as an input to change a physical objects configuration. Both goals were met. I used residual neural networks as a base to work on. I then trained these networks with data that I collected by myself to classify hand signs. I put the model that was created to work when I used live feed from a camera to signal patterns with LEDs, according to the hand signal shown.

Nvidia Jetson Nano Development Kit is a very good fit for tasks of this nature and of such complexity. It is a great tool for practice in machine learning and artificial intelligence fields. Nano has enough power to complete training of neural networks with the supplied data in a relatively short amount of time. This development kit has the potential to be a great learning tool in higher education and in extra-curricular studies.

The models that were created during this project could have been better, if bigger and more varied data were used. Since it is a project done by one author, every piece of input data features my own hands doing the signals. This creates a distinct possibility that the models created will have a poorer performance for other people. Should this project ever be used as a basis for further work, this is a potential problem that should be looked out for.

VIITED

[1] Nvidia, „Jetson Nano Developer Kit," [Võrgumaterjal]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit. [Kasutatud 03 12 2020].

[2] H. Kaiming, Z. Xiangyu, R. Shaoqing ja S. Jian, „Deep Residual Learning for Image Recognition," 2016.

[3] S. A. Papert, „MIT Libraries," 01 07 1966. [Võrgumaterjal]. Available: https://dspace.mit.edu/handle/1721.1/6125. [Kasutatud 22 5 2021].

[4] R. Rastgoo, K. Kiani ja S. Escalera, „Sign Language Recognition: A Deep Survey," *Expert Systems with Applications,* kd. 164, nr 113794, 2021.

[5] PyTorch, „PyTorch veebileht," [Võrgumaterjal]. Available: https://pytorch.org/vision/stable/transforms.html. [Kasutatud 23 05 2021].

[6] Nvidia AI IOT, „JetCam GitHub veebileht," Nvidia, 21 06 2019. [Võrgumaterjal]. Available: https://github.com/NVIDIA-AI-IOT/jetcam. [Kasutatud 05 01 2021].

[7] Nvidia, „Deep Learning Institute - Getting Started with AI on Jetson Nano," Nvidia, [Võrgumaterjal]. Available: https://courses.nvidia.com/courses/course-v1:DLI+S-RX-02+V2/about. [Kasutatud 03 12 2020].

[8] Nvidia, „DLI Nano AI," [Võrgumaterjal]. Available: https://ngc.nvidia.com/catalog/containers/nvidia:dli:dli-nano-ai. [Kasutatud 3 12 2020].

[9] GitHub, „A Python library that enables the use of Jetson's GPIOs," [Võrgumaterjal]. Available: https://github.com/Heerpa/jetson-gpio-nanohard. [Kasutatud 3 12 2020].

[10] Project Jupyter, „Project Jupyter," [Võrgumaterjal]. Available: https://jupyter.org/. [Kasutatud 3 12 2020].

[11] Jupyter Widgets, „Jupyter Widgets GitHub'i leht," [Võrgumaterjal]. Available: https://github.com/jupyter-widgets/ipywidgets. [Kasutatud 23 05 2021].

[12] Nvidia, „CUDA Zone," Nvidia, [Võrgumaterjal]. Available: https://developer.nvidia.com/cuda-zone. [Kasutatud 29 05 2021].

[13] Nvidia, „Nvidia Jetson Community projektid," [Võrgumaterjal]. Available: https://developer.nvidia.com/embedded/community/jetson-projects. [Kasutatud 23 05 2021].

LISA 1. KASUTATUD KOOD

Siin lisas on ära toodud töös närvivõrkude treenimiseks kasutatud kood, koos endapoolsete muudatustega. Enda tehtud muudatused on märgitud rohelise värviga. Iga koodiblokk on mõeldud jooksutamiseks ülalkirjeldatud viisil üles seatud Docker-i konteineris asuvas JupyterLab-i Notebook-is.

Kaamera ühendamine:

```
from jetcam.usb_camera import USBCamera

camera = USBCamera(width=224, height=224, capture_width=640, capture_height=360,
capture_device=0)

camera.running = True
```

Ülesande püstitus:

```
import torchvision.transforms as transforms
from dataset import ImageClassificationDataset

TASK = 'camera_gpio'
SUBTASK = 'resnet_pre30'
CATEGORIES = ['thumbs_up', 'thumbs_down', 'fist', 'palm']
DATASETS = ['A']
TRANSFORMS = transforms.Compose([
    transforms.ColorJitter(0.2, 0.2, 0.2, 0.2),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
datasets = {}
for name in DATASETS:
    datasets[name] = ImageClassificationDataset('camera_gpio/' + TASK + '_' + name,
CATEGORIES, TRANSFORMS)
```

Andmete kausta loomine:

```
DATA_DIR = '/nvdli-nano/data/camera_gpio/'
!mkdir -p {DATA_DIR}
```

Andmete kogumise osa loomine:

```
import ipywidgets
import traitlets
from IPython.display import display
from jetcam.utils import bgr8_to_jpeg

dataset = datasets[DATASETS[0]]

camera.unobserve_all()
camera_widget = ipywidgets.Image()

traitlets.dlink((camera, 'value'), (camera_widget, 'value'), transform=bgr8_to_jpeg)

dataset_widget = ipywidgets.Dropdown(options=DATASETS, description='dataset')
category_widget = ipywidgets.Dropdown(options=dataset.categories,
description='category')
count_widget = ipywidgets.IntText(description='count')
save_widget = ipywidgets.Button(description='add')
count_widget.value = dataset.get_count(category_widget.value)

def set_dataset(change):
    global dataset
    dataset = datasets[change['new']]
    count_widget.value = dataset.get_count(category_widget.value)
dataset_widget.observe(set_dataset, names='value')

def update_counts(change):
    count_widget.value = dataset.get_count(change['new'])
category_widget.observe(update_counts, names='value')

def save(c):
    dataset.save_entry(camera.value, category_widget.value)
    count_widget.value = dataset.get_count(category_widget.value)
save_widget.on_click(save)

data_collection_widget = ipywidgets.VBox([
    ipywidgets.HBox([camera_widget]), dataset_widget, category_widget, count_widget,
save_widget
])
```

Mudeli valik ja loomine:

```
import torch
import torchvision

model = torchvision.models.resnet18(pretrained=True)
model.fc = torch.nn.Linear(512, len(dataset.categories))

device = torch.device('cuda')
model = model.to(device)

model_save_button = ipywidgets.Button(description='save model')
model_load_button = ipywidgets.Button(description='load model')
model_path_widget = ipywidgets.Text(description='model path', value='/nvdli-
nano/data/camera_gpio/camera_gpio_model.pth')

def load_model(c):
    model.load_state_dict(torch.load(model_path_widget.value))
model_load_button.on_click(load_model)

def save_model(c):
    torch.save(model.state_dict(), model_path_widget.value)
model_save_button.on_click(save_model)

model_widget = ipywidgets.VBox([
    model_path_widget,
    ipywidgets.HBox([model_load_button, model_save_button])
])
```

GPIO klemmide seadistus:

```
import RPi.GPIO as GPIO
import time

output_pin1 = 17  # BCM pin 17, BOARD pin 11
output_pin2 = 18  # BCM pin 18, BOARD pin 12

output_pins = [output_pin1, output_pin2]
GPIO.setmode(GPIO.BCM)
for pin in output_pins:
    GPIO.setup(pin, GPIO.OUT, initial=GPIO.LOW)

thumbs_up = [GPIO.HIGH, GPIO.LOW]
thumbs_down = [GPIO.LOW, GPIO.HIGH]
fist = [GPIO.HIGH, GPIO.HIGH]
palm = [GPIO.LOW, GPIO.LOW]

pin_configurations = [thumbs_up, thumbs_down, fist, palm]
```

Treenitud mudeli hindamise osa loomine:

```
import threading
from utils import preprocess
import torch.nn.functional as F

state_widget = ipywidgets.ToggleButtons(options=['stop', 'live'], description='state',
value='stop')
prediction_widget = ipywidgets.Text(description='prediction')
score_widgets = []
scores = []
for category in dataset.categories:
    score_widget = ipywidgets.FloatSlider(min=0.0, max=1.0, description=category,
orientation='vertical')
    score_widgets.append(score_widget)
    scores.append(0)

def live(state_widget, model, camera, prediction_widget, score_widget):
    global dataset
    while state_widget.value == 'live':
        image = camera.value
        preprocessed = preprocess(image)
        output = model(preprocessed)
        output = F.softmax(output, dim=1).detach().cpu().numpy().flatten()
        category_index = output.argmax()
        prediction_widget.value = dataset.categories[category_index]
        for i, score in enumerate(list(output)):
            score_widgets[i].value = score
            scores[i] = score
        if max(scores) > 0.5:
            max_conf = scores.index(max(scores))
            GPIO.output(output_pin1, pin_configurations[max_conf][0])
            GPIO.output(output_pin2, pin_configurations[max_conf][1])
        else:
            GPIO.output(output_pin1, GPIO.LOW)
            GPIO.output(output_pin2, GPIO.LOW)

def start_live(change):
    if change['new'] == 'live':
        execute_thread = threading.Thread(target=live, args=(state_widget, model, camera,
prediction_widget, score_widget))
        execute_thread.start()

state_widget.observe(start_live, names='value')

live_execution_widget = ipywidgets.VBox([
    ipywidgets.HBox(score_widgets),
    prediction_widget,
    state_widget
])
```

Treenimise osa loomine:

```
BATCH_SIZE = 8

optimizer = torch.optim.Adam(model.parameters())

epochs_widget = ipywidgets.IntText(description='epochs', value=1)
eval_button = ipywidgets.Button(description='evaluate')
train_button = ipywidgets.Button(description='train')
loss_widget = ipywidgets.FloatText(description='loss')
accuracy_widget = ipywidgets.FloatText(description='accuracy')
progress_widget = ipywidgets.FloatProgress(min=0.0, max=1.0, description='progress')
timer_widget = ipywidgets.IntText(description="timer")
time_per_epoch_widget = ipywidgets.FloatText(description="time_per_epoch")

def train_eval(is_training):
    global BATCH_SIZE, LEARNING_RATE, MOMENTUM, model, dataset, optimizer, eval_button, train_button, accuracy_widget, loss_widget, progress_widget, state_widget, timer_widget, time_per_epoch_widget

    try:
        train_loader = torch.utils.data.DataLoader(
            dataset,
            batch_size=BATCH_SIZE,
            shuffle=True
        )

        state_widget.value = 'stop'
        train_button.disabled = True
        eval_button.disabled = True
        timer_widget.value = 0
        time_per_epoch_widget.value = 0
        time.sleep(1)
        t0 = time.time()
        epochs_init = epochs_widget.value

        if is_training:
            model = model.train()
        else:
            model = model.eval()
        while epochs_widget.value > 0:
            i = 0
            sum_loss = 0.0
            error_count = 0.0
            for images, labels in iter(train_loader):
                # send data to device
                images = images.to(device)
                labels = labels.to(device)

                if is_training:
                    # zero gradients of parameters
                    optimizer.zero_grad()
```

```
            outputs = model(images)
            loss = F.cross_entropy(outputs, labels)

            if is_training:
                loss.backward()
                optimizer.step()

            error_count += len(torch.nonzero(outputs.argmax(1) - labels).flatten())
            count = len(labels.flatten())
            i += count
            sum_loss += float(loss)
            progress_widget.value = i / len(dataset)
            loss_widget.value = sum_loss / i
            accuracy_widget.value = 1.0 - error_count / i
            timer_widget.value = time.time() - t0

        if is_training:
            epochs_widget.value = epochs_widget.value - 1
        else:
            break
    time_per_epoch_widget.value = round( timer_widget.value / epochs_init, 2 )
  except e:
    pass
  model = model.eval()

  train_button.disabled = False
  eval_button.disabled = False
  state_widget.value = 'live'

train_button.on_click(lambda c: train_eval(is_training=True))
eval_button.on_click(lambda c: train_eval(is_training=False))

train_eval_widget = ipywidgets.VBox([
  epochs_widget,
  progress_widget,
  loss_widget,
  accuracy_widget,
  timer_widget,
  time_per_epoch_widget,
  ipywidgets.HBox([train_button, eval_button])
])
```

Kõik osad koos tööle:

```
all_widget = ipywidgets.VBox([
    ipywidgets.HBox([data_collection_widget, live_execution_widget]) ,
    ipywidgets.HBox([train_eval_widget, model_widget])])
])
display(all_widget)
```

Lõpetamine:

```
import os
import IPython

GPIO.cleanup()
os._exit(00)
```