VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
MATEMATINĖS INFORMATIKOS KATEDRA

Bakalauro baigiamasis darbas

Rankoje laikomų objektų atpažinimas naudojant TensorFlow.js

Atliko: 4 kurso 3 grupės studentas
Marek Rusakevič (parašas)

Darbo vadovas:
Irus Grinis (parašas)

Recenzentas:
Vytas Zacharovas (parašas)

Vilnius
2019

Turinys

Įvadas

Gilusis mokymasis per paskutinius metus labai išpopuliarėjo. Populiariausios ir dažniausiai naudojamos bibliotekos darbui su neuroniniais tinklais yra *TensorFlow* ir *Keras*. Šitie įrankiai įsitvirtino giliojo mokymo aplinkoje, nes yra galingi ir patikimi. Bet jie turi didelį trūkumą – jų nustatymas reikalauja nemažai laiko ir žinių, todėl naujokams šioje srityje nelabai patogu pradėti jais naudotis. Su šia problema padeda susidoroti neseniai pasirodęs *TensorFlow.js*.

Tikslas

Šio darbo tikslas yra sukonstruoti tinklą naudojant *Keras*, o po to naują biblioteką *TensorFlow.js* ir palyginti konstravimo procesus. Su *Keras* sukonstruosime paprasta tinklą, kuris atpažins mažas *„Lego“* detales. Apžvelgsime būdus normalizuoti paveikslėlius (duomenis), kad būtų pasiektas didesnis atpažinimo procentas. Taip pat tinklą konstruosime bandydami skirtingas architektūras ir palyginsime jų efektyvumą.

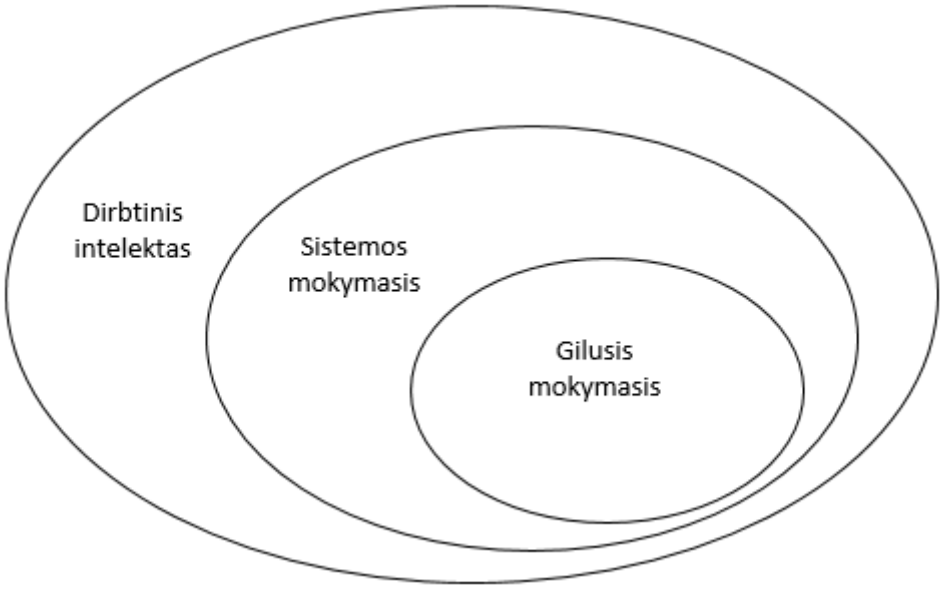
Su *TensorFlow.js* konstruosime tinklą, kuris leidžia atpažinti skirtingus objektus. Taip pat bandysime išspręsti papildomą problemą: tinklas gerai atpažįsta objektus, kurie neturi jokios papildomos informacijos paveikslėlio fone, bet jeigu kartu su atpažįstamu objektu yra dar vienas, atpažinimo tikimybė tampa kur kas mažesnė. Bandysime sukonstruot tinklą, kuris ignoruos visą foną ir bandys atpažinti objektą, kuris yra laikomas rankoje. Taip pat išbandysime naują neuroninių tinklų konstravimo būdą – mokymosi perkėlimą (*Transfer learning*). Su mokymosi perkėlimu neuroninio tinklo apmokymo laikas turėtų sumažėti keliskart.

Uždaviniai

- Sukonstruoti tinklą su *Keras* pagalba, kuris atpažįsta mažas *„Lego“* detales.
- Peržiūrėti visų sluoksnių tipus: sąsūkos sluoksnį, didžiausio atmetimo, tirštąjį sluoksnį, atmetimo sluoksnį.
- Palyginti skirtingas neuroninio tinklo architektūras.
- Sukonstruoti tinklą su *TensorFlow.js* naudojant mokymosi perkėlimą (*Transfer learning*), kurį galima bus apmokyti atpažinti skirtingus objektus, kurie bus laikomi rankoje.
- Panaudoti *HandTrack.js* biblioteką rankos atpažinimui vaizde.

1. Gilusis mokymasis

Jeigu norime suprasti kas yra gilusis mokymasis, iš pradžių turime suprasti kas yra dirbtinis intelektas, sistemos mokymasis ir kaip jie susiję tarpusavyje.



1 pav. Dirbtinis intelektas, sistemos mokymasis, gilusis mokymasis.

1.1. Dirbtinis intelektas (*AI*)

Dirbtinis intelektas buvo pradėtas tyrinėti po Antrojo pasaulinio karo, kai keletas pradedančiųjų kompiuterių srities pionierių pradėjo klausti, ar kompiuteriai galės mąstyti – ir į šį klausimą iki šiol niekas neatsakė.

Trumpas šios srities apibrėžimas būtų toks: pastangos automatizuoti intelektines užduotis, kurias paprastai atlieka žmonės.

AI yra bendra sritis, apimanti sistemos mokymąsi ir gilųjį mokymąsi, tačiau taip pat apima daug daugiau metodų, kurie nereikalauja jokio mokymosi. Pavyzdžiui, ankstyvose šachmatų programose dalyvavo tik programuotojų sukurtos taisyklės. Gana ilgą laiką daugelis ekspertų manė, kad dirbtinio intelekto žmogaus lygmeniu būtų galima pasiekti, sukuriant programuotojų pakankamai aiškių manipuliavimo žinių taisyklių. Šis metodas, žinomas kaip simbolinis AI, buvo dominuojantis nuo 1950 m. iki devintojo dešimtmečio pabaigos.

Nors simbolinis AI pasirodė esąs tinkamas išspręsti aiškiai apibrėžtas logiškas problemas, tokias kaip žaisti šachmatais Paaiškėjo, kad sunku išsiaiškinti aiškias taisykles, skirtas išspręsti sudėtingesnes, neaiškias problemas, tokias kaip vaizdo klasifikavimas, kalbos

atpažinimas ir kalbos vertimas. Sukurtas naujas požiūris į simbolinę AI vietą: sistemos mokymasis. [**1**]

1.2. Sistemos mokymasis

Sistemos, pagrįstos simboliniu dirbtiniu intelektu, negalėjo mokytis iš naujų duomenų. Tai paskatino atsirasti tyrimų sričiai, vadinamai sistemos mokymu, kuri apibrėžiama kaip „kompiuterio apmokymas" iš duotų duomenų, užuot jį tiksliai užprogramavus.

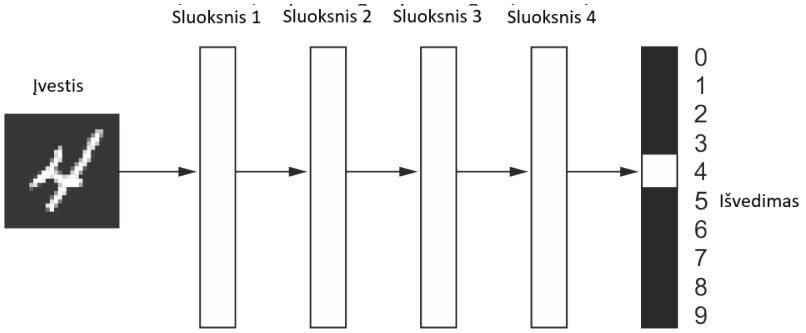
Nors sistemos mokymasis atsirado tik dešimtajame dešimtmetyje, jis greitai tapo daugiausiai naudojama dirbtinio intelekto atmaina. Tą nulėmė padidėjusios duomenų bazės ir galingesni kompiuteriai.

2 pav. Klasikinis programavimas prieš sistemos mokymąsi.

Klasikiniame programavime programai yra paduodami duomenys ir taisyklės, pagal kurias šie duomenys yra nagrinėjami ir gaunami norimi atsakymai. Su sistemos mokymosi žmonės įveda duomenis, taip pat atsakymus, kurių tikimasi iš duomenų, o gaunamos yra taisyklės. Tada šios taisyklės gali būti taikomos naujiems duomenims pateikti originalius atsakymus. [**1**]

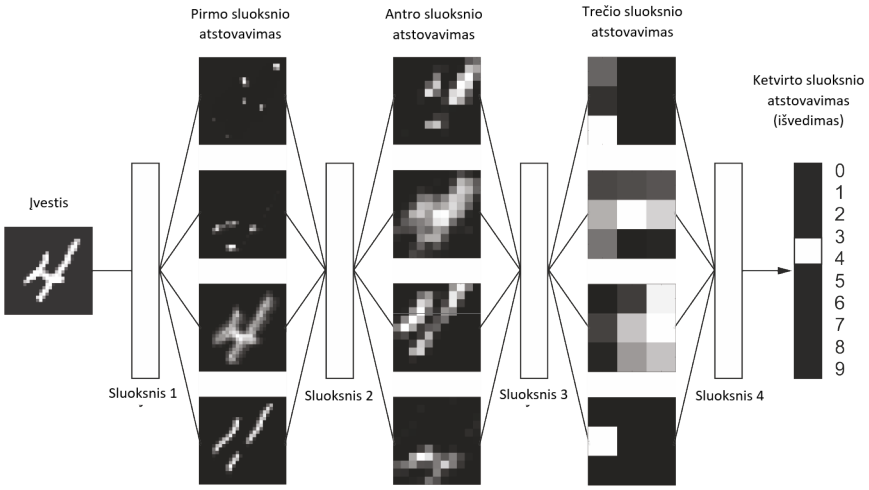
1.3. Gilusis mokymasis

Gilusis mokymasis yra sistemos mokymosi atšaka, kuri naudoja neuroninius tinklus. Neuroninis tinklas yra sudarytas iš kelių sudėtų vienas ant kito sluoksnių, kur kiekvienas sluoksnis išmoksta vis abstraktesnius duomenų bruožus. Gilusis mokymasis vadinamas giliu (*depth of the model*), nes turi daugiau nei kelis sujungtus sluoksnius. Modernus gilusis mokymas dažniausiai turi tarp dešimties ir kelių šimtų sluoksnių, kurie išmoksta bruožus tiesiai iš duomenų.



3 pav. Gilus neuroninis tinklas skaitmenų klasifikacijai.

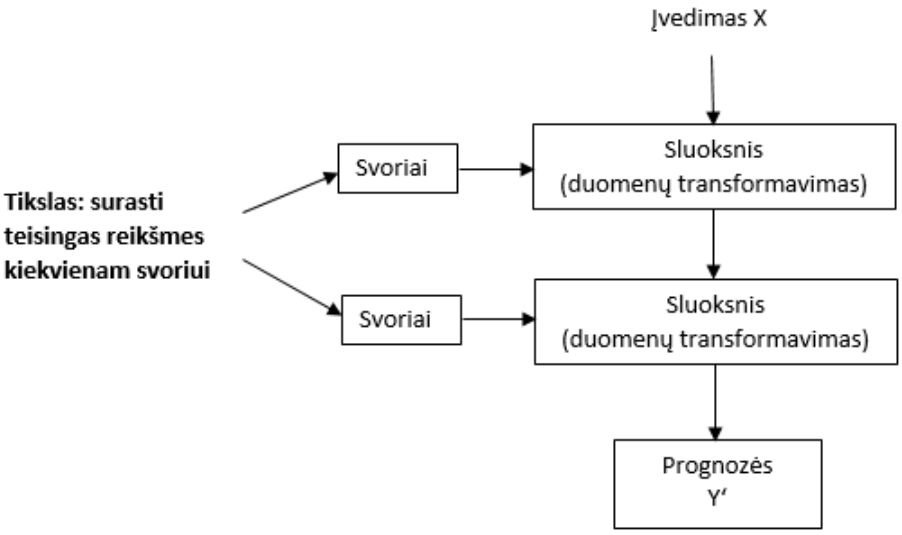
Kaip matome paveikslėlyje 3, tinklas transformuoja pradinį paveikslėlį į reprezentacijas, kurios skiriasi nuo pradinio paveikslėlio ir yra kur kas informatyvesnės.



4 pav. Gilios reprezentacijos, įgytos pagal skaitmenų klasifikacijos modelį.

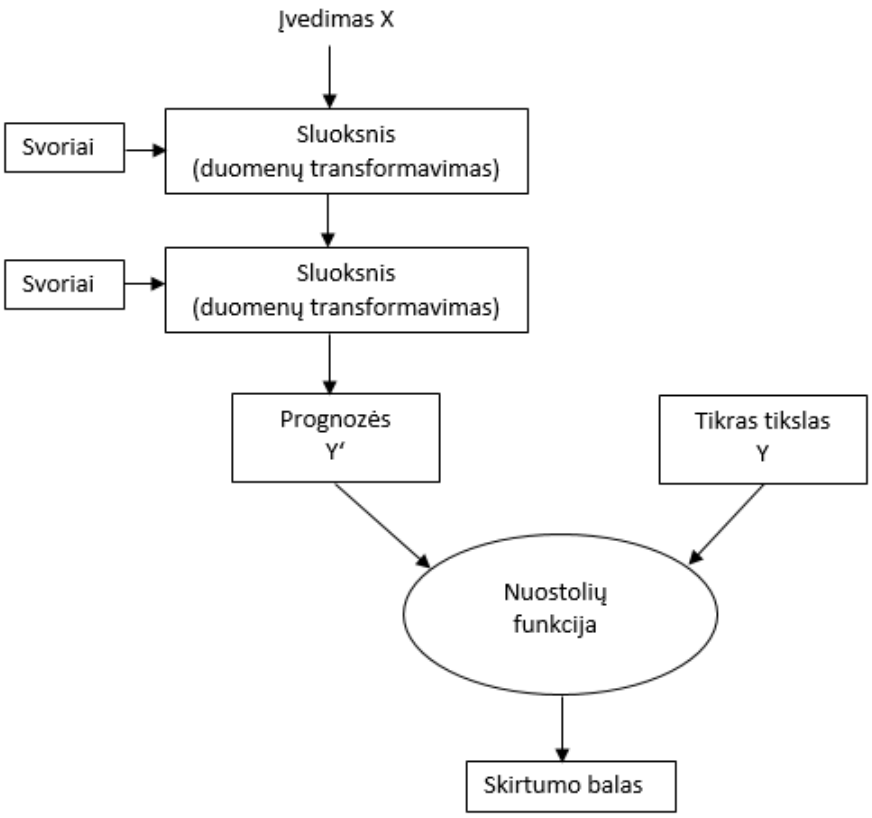
Apie gilų tinklą galima galvoti kaip apie daugiapakopę informacijos filtravimo operaciją, kurioje informacija pereina per filtrus ir yra „išvaloma“.

Specifiškai ką sluoksnis daro su galutiniais duomenimis yra laikoma sluoksnio svoryje. Galima pasakyti, kad sluoksnio įgyvendinta transformacija yra parametrizuojama pagal jo svorius kaip 5 paveikslėlyje. Mokymas reiškia surasti tokias reikšmes kiekvieno sluoksnio svoriams, kad tinklas tinkamai pažymėtų įvedimą į jam priskirtą tikslą. Sudėtingumas yra tas, kad gilus neuroninis tinklas gali turėti daugybę parametrų, o keičiant vieną iš jų, bus paveikta visų kitų elgsena.



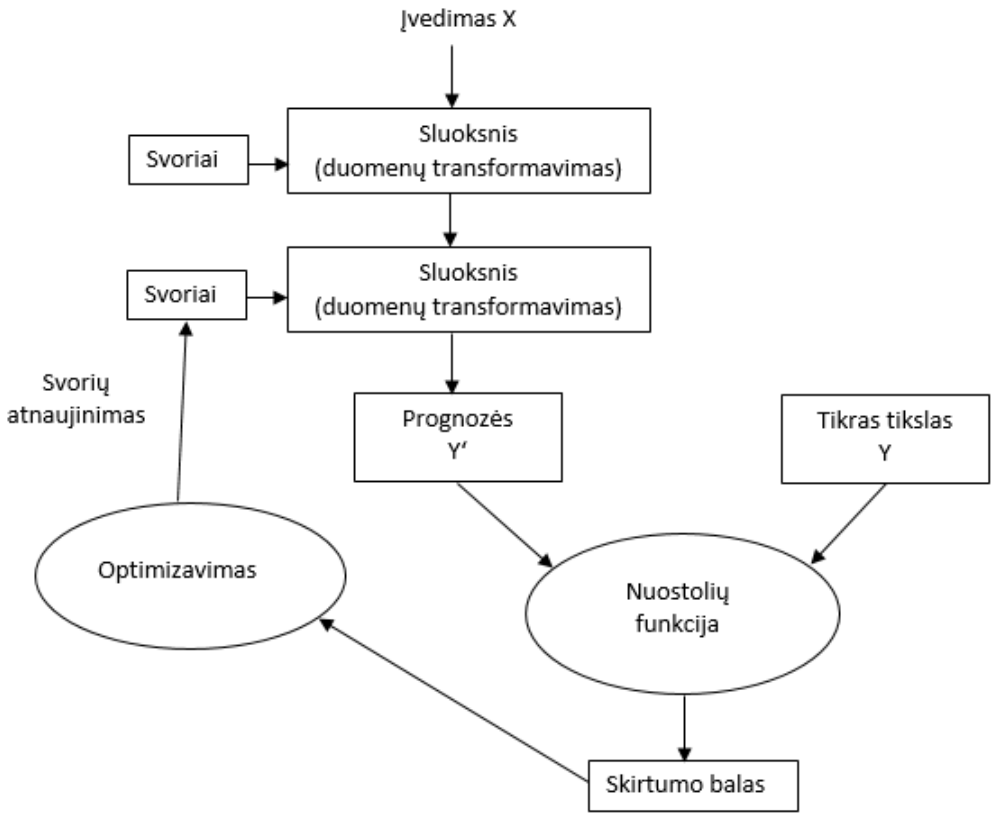
5 pav. Neuroninis tinklas yra parametrizuojamas pagal svorius.

Kad kontroliuotume neuroninio tinko išvedimą, reikia mokėt apskaičiuoti kaip toli išvedimas buvo nuo laukiamo rezultato. Tai atlieka nuostolių funkcija (Loss function, pav. 6). Nuostolių funkcija atsižvelgia į tinklo prognozes ir norimą tikslą, ir apskaičiuoja skirtumo balą, užfiksuodama kaip gerai tinklas atliko savo darbą.



6 pav. Nuostolių funkcija nustato tinklo išvedimo kokybę.

Giliojo mokymosi pagrindinis triukas yra naudoti šį balą kaip atsako signalą, kad šiek tiek pakoreguoti svorių vertę (Pav. 7), tokiu būdu mažinant dabartinio pavyzdžio nuostolių balą. Šis koregavimas yra optimizatoriaus, kuris įgyvendina „skleidimo atgal“ („*Backpropagation*“) algoritmą: pagrindinį giliojo mokymosi algoritmą, darbas.



7 pav. Skirtumo balas nusako kaip reikia keisti svorius.

Iš pradžių tinklo svoriams yra priskiriamos atsitiktinės vertės, todėl tinklas tiesiog įgyvendina porą atsitiktinių transformacijų. Aišku, kad pirmas išvedimas bus toli nuo to, ko mes norime ir nuostolių balas bus didelis. Bet su kiekvienu pavyzdžiu, kurį tinklas apdoroja, svoriai bus šiek tiek koreguojami ir nuostolių balas mažės. Tai bus kartojama reikiama kiekį kartų ir galiausiai gausime svorių vertes, su kuriomis nuostolių funkcija bus mažiausia. Tinklas su minimaliu nuostoliu yra tas, kurio rezultatai yra arčiausiai norimo tikslo. [**1**]

Gilusis mokymasis pasiekė:

- žmogaus gebėjimo lygio vaizdo klasifikacijos;
- žmogaus lygio kalbos atpažinimo;

- žmogaus lygio rankraščio transkripcijos;
- pagerintos konversijos iš teksto į kalbą;
- žmogaus lygio automatino vairavimo;
- patobulinto skelbimų taikymo, kurį naudoja „Google“, „Baidu“ ir „Bing“;
- geresnių paieškos rezultatų žiniatinklyje;
- gebėjimo atsakyti į natūralios kalbos klausimus;

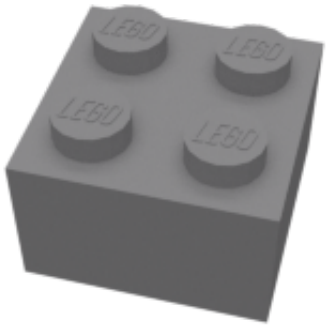
2. *Keras*

Šiame skyriuje konstruosime neuroninį tinklą, kuris atpažins *„Lego“* detales, su *Keras* pagalba. Iš pradžių normalizuosime duomenų rinkinį, kuris yra sudarytas iš 6000 *„Lego“* detalių paveikslėlių. Tam naudosime *ImageJ* vaizdo apdorojimo programą. Toliau konstruosime tinklą. Tinklo konstravimui naudosime skirtingas architektūras, sudarytas iš skirtingų sluoksnių tipų, kurias palyginsime ir išsirinksime efektyviausią.

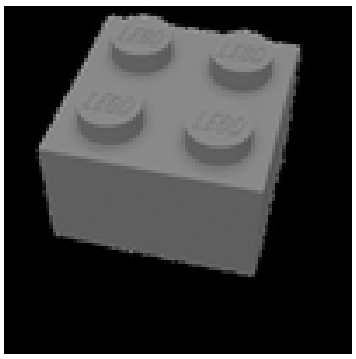
2.1. Duomenų rinkinys

1. Tinklo apmokymui naudosime 28×28 dydžio paveikslėlius. Visą duomenų rinkinį padalinsime į mokymo ir validavimo dalis. Mokymui skirsime 70%, o validavimui – 30%.

   Prieš apmokant neuroninį tinklą normalizuosime visus mokymo duomenis. Iš pradžių konvertuosime pradinį apmokymo paveikslėlį (Pav. 8) iš RGB į pilkos spalvos (*Grayscale*) ir pakeisime dydį į 100×100 (Pav. 9).

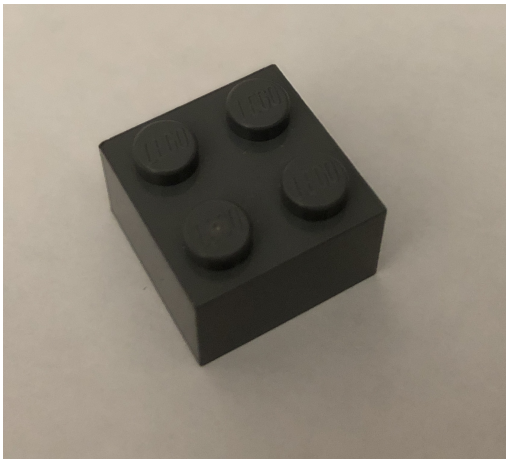


8 pav. Nenormalizuotas paveikslėlis.



9 pav. Pilkos spalvos, 100×100 apmokymo paveikslėlis.

Taip pat turime normalizuoti testavimo duomenis. Pradžioje iš pradinio paveikslėlio (Pav. 10) pašalinsime foną (Pav. 11). Tada taip pat konvertuosime iš RGB į pilkos spalvos atspalvius (*Grayscale*) ir pakeisime dydį į 100×100 (Pav. 12).

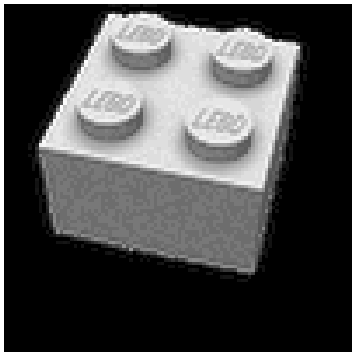


10 pav. Testavimo paveikslėlis.

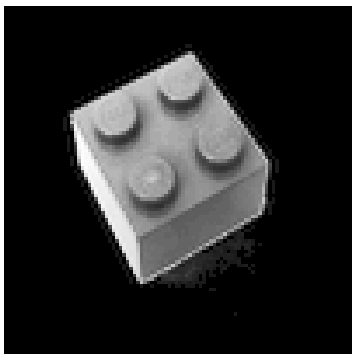
11 pav. Testavimo paveikslėlis be fono.

12 pav. Pilkos spalvos, 100×100 testavimo paveikslėlis.

2. Panaudosime *ImageJ* metodą „Padidinti kontrastą“. Gavome tokius paveiklėlius:

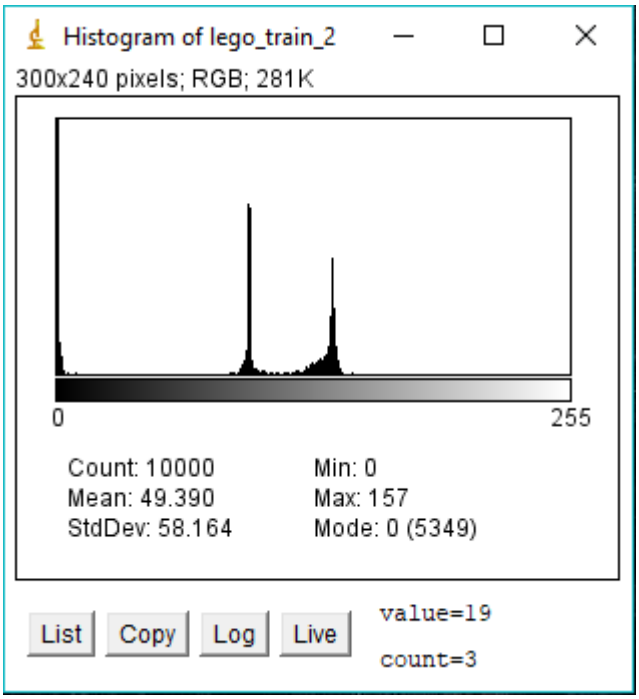


13 pav. Apmokymo paveikslėlis.

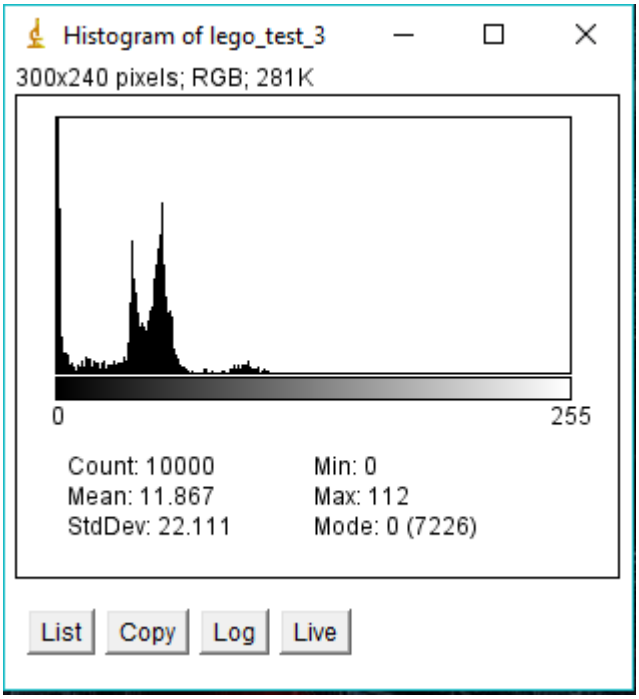


14 pav. Testavimo paveikslėlis.

3. Prieš normalizavimą, paveikslėlių histogramos atrodė taip:

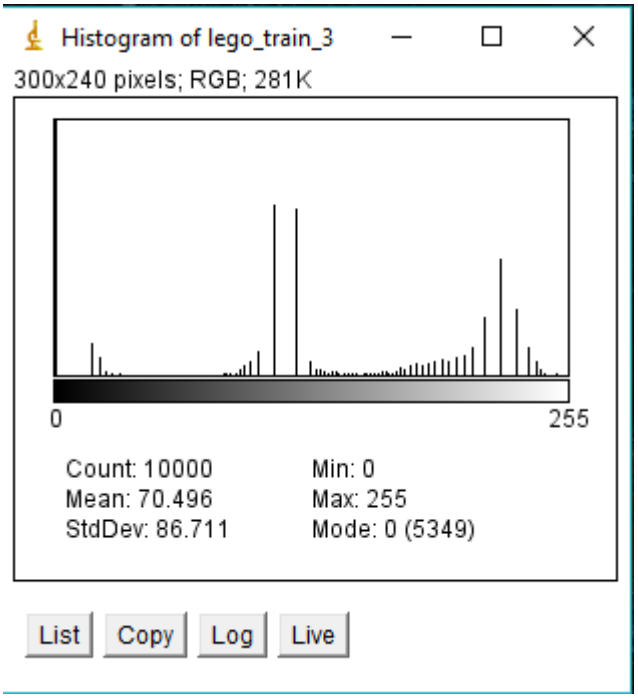


15 pav. Nenormalizuoto apmokymo paveikslėlio histograma.

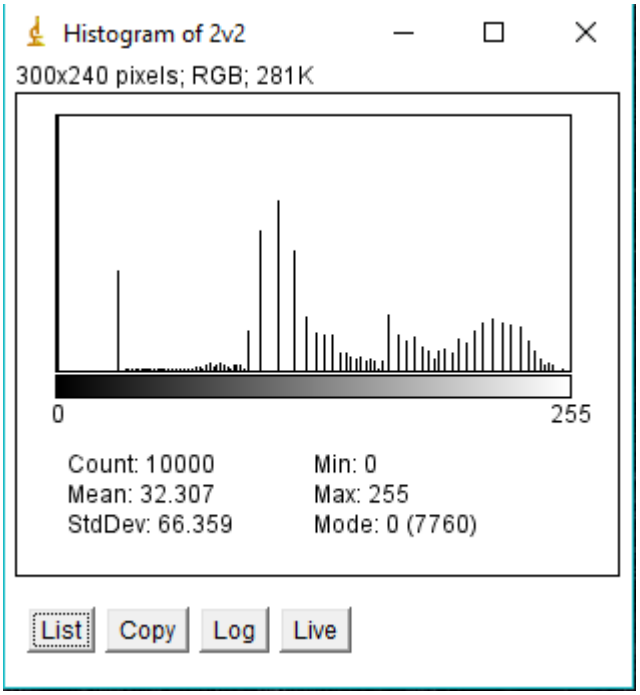


16 pav. Nenormalizuoto testavimo paveikslėlio histograma.

Po normalizavimo gavome tokias histogramas:



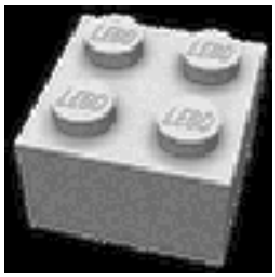
17 pav. Normalizuoto apmokymo paveikslėlio histograma.



18 pav. Normalizuoto testavimo paveikslėlio histograma.

Matome, kad po normalizavimo histogramos tapo panašesnės.

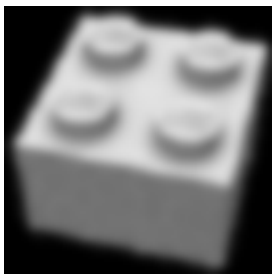
4. Panaudodami parašytą *ImageJ* įskiepį (*plugin*), centruosime duomenis.



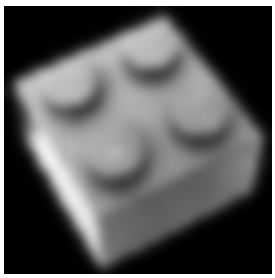
19 pav. Centruotas apmokymo paveikslėlis.

20 pav. Centruotas testavimo paveikslėlis.

5. Pritaikysime paveikslėliams *ImageJ* Gauso filtrą.



21 pav. Centruotas apmokymo paveikslėlis su Gauso filtru.



22 pav. Centruotas testavimo paveikslėlis su Gauso filtru.

6. Pakeisime paveikslėlių dydžius į 28×28. Taip pat panaudosime *ImageJ* funkciją „Rasti kraštus“.

23 pav. Apmokymo paveikslėlis dydžio 28×28 po funkcijos „Rasti kraštus“ panaudojimo.

24 pav. Testavimo paveikslėlis dydžio 28×28 po funkcijos „Rasti kraštus“ panaudojimo.

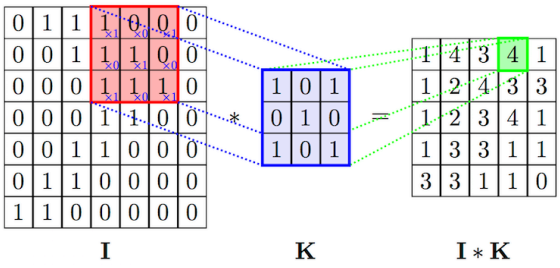
2.2. Modelis

Prieš konstruojant neuroninį tinklą, apžiūrėkime sluoksnių tipus.

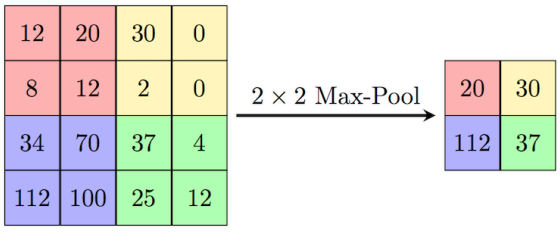
2.2.1. Sluoksniai

Mūsų modelyje naudosime [**3**]:

- Sąsūkos sluoksnis (*Convolutional layer*) – Sluoksnis, kuris uždeda filtrą (mūsų atveju 3×3) ir taip iš paveikslėlio yra išgaunami 3×3 dydžio sritys, su poslinkiu 1. (25 pav.)
- Didžiausio atmetimo sluoksnis (*Max-Pool layer*) – Sluoksnis, kuris sumažina paveikslėlio reprezentaciją dvigubai (naudojamas 2×2 filtras). (26 pav.)
- Tirštasis sluoksnis (*Dense layer*) – Sluoksnis, kuris jungia visus neuronus iš praeito sluoksnio, su visais šio sluoksnio neuronais.
- Atmetimo sluoksnis (*Dropout layer*) – dalis neuronų yra ignoruojama, kad išvengtume permokymo (*overfitting*).



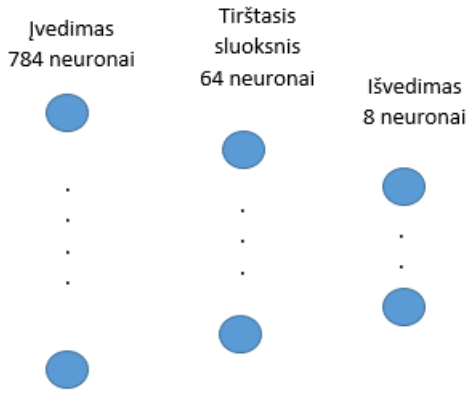
25 pav. Sąsūkos sluoksnis.



26 pav. Didžiausio atmetimo sluoksnis.

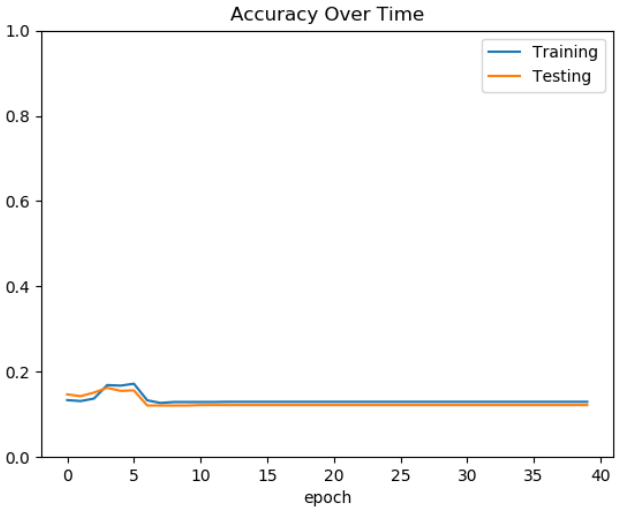
2.2.2. Architektūra

1. Iš pradžių sukonstruosim tinklą su vienu tirštuoju sluoksniu, kuris turės 64 neuronus. Mokymas vyks 40 iteracijų.



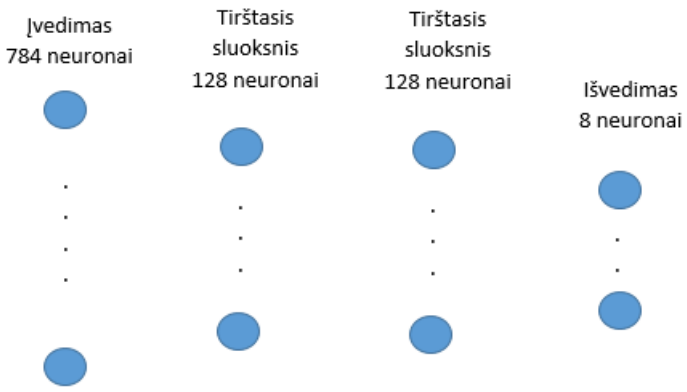
27 pav. Pirmas tinklas.

Toks tinklas pasiekė tik 18.0% atpažinimo tikimybę.



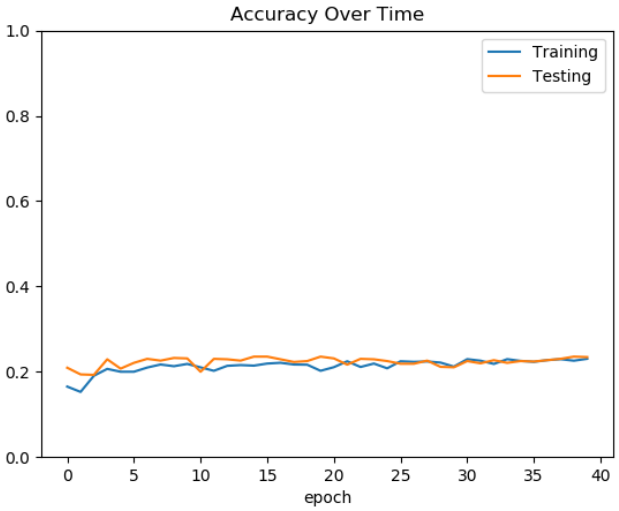
28 pav. Pirmo tinklo grafas.

2. Naujas tinklas turės du tirštuosius sluoksnius po 128 neuronus. Mokymas vyks 40 iteracijų.



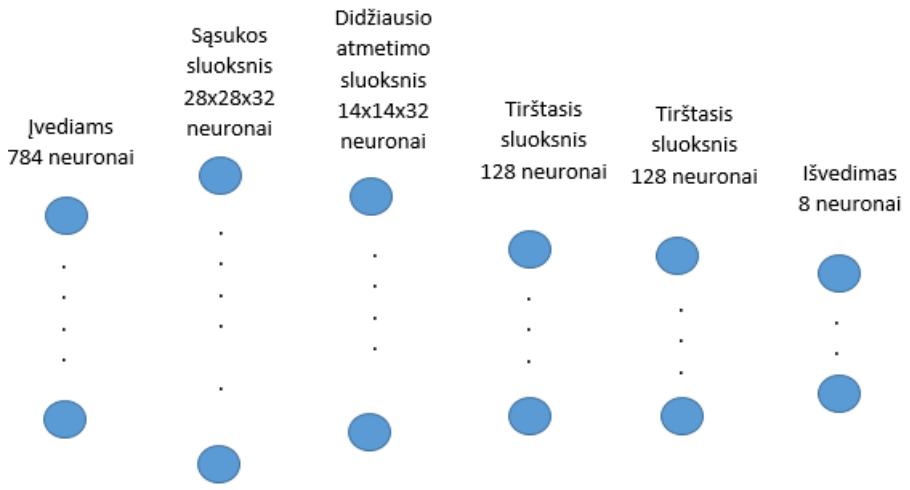
29 pav. Antras tinklas.

Toks tinklas pasiekė 22.44% atpažinimo tikimybę.



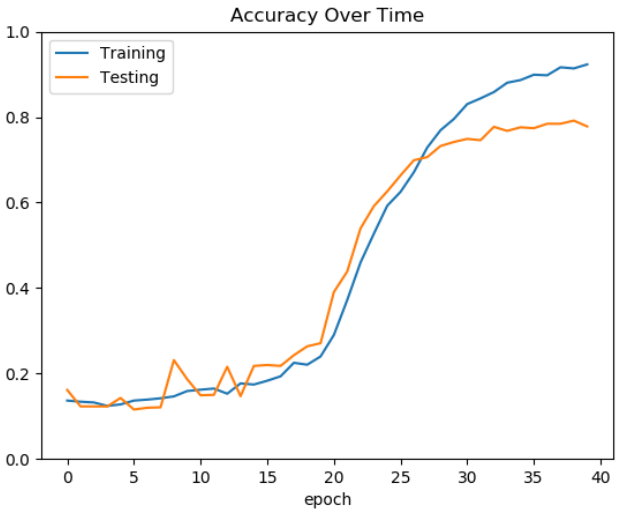
30 pav. Antro tinklo grafas.

3. Prieš tirštuosius sluoksnius įdėsim vieną sąsūkos sluoksnį, kuris turės 32 filtrus, ir išgaus iš paveikslėlio 3×3 dydžio sritys, su poslinkiu 1. Po sąsūkos sluoksnio bus didžiausio atmetimo sluoksnis (2×2 filtras). Mokymas vyks 40 iteracijų.



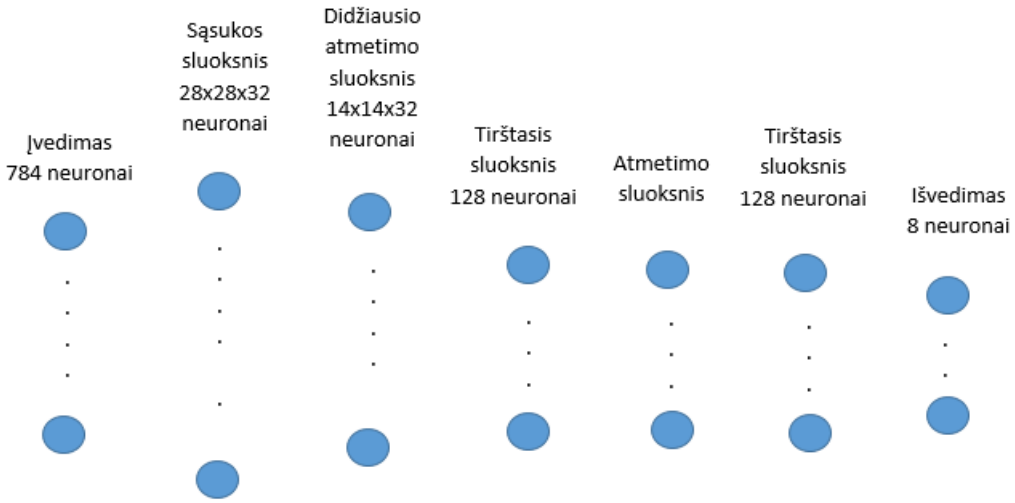
31 pav. Trečias tinklas.

Toks tinklas pasiekė 79.4% atpažinimo tikimybę.



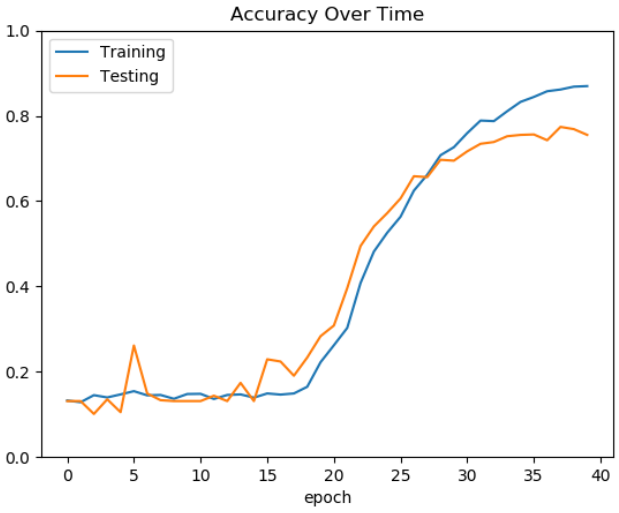
32 pav. Trečio tinklo grafas.

4. Kad išvengtume permokymo, tarp tirštųjų sluoksnių įdėsim atmetimo sluoksnį su koeficientu 0.5.



33 pav. Ketvirtas tinklas.

Toks tinklas pasiekė 77.81% atpažinimo tikimybę.

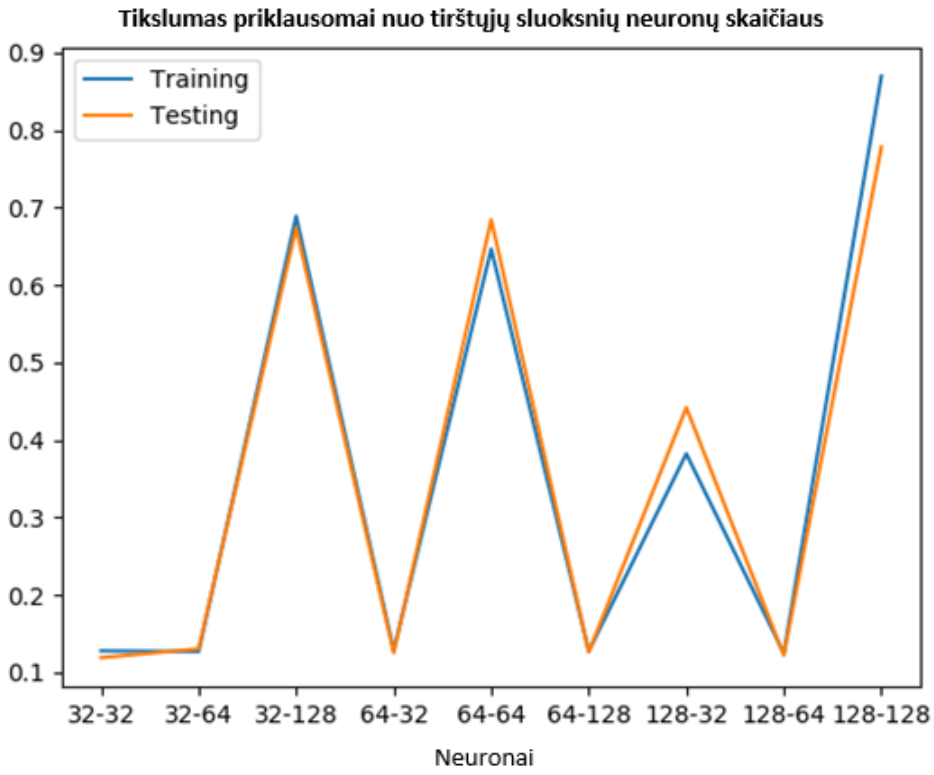


34 pav. Ketvirto tinklo grafas.

5. Patikrinkime, kaip keičiasi tinklo atpažinimas pakeitus tirštųjų sluoksnių neuronų skaičių.

Neuronų sk.	accuracy	Value_accuracy
32-32	0.1276	0.1187
32-64	0.1267	0.1302
32-128	0.6888	0.6739
64-32	0.1281	0.125
64-64	0.6464	0.6843
64-128	0.1272	0.126
128-32	0.3821	0.4416
128-64	0.1245	0.1218
128-128	0.8696	0.7781

35 pav. Tirštųjų sluoksnių neuronų sk. lentelė.



36 pav. Tirštųjų sluoksnių neuronų sk. grafas.

Kaip matome didžiausią atpažinimo tikimybę tinklas pasiekė, kai jis buvo sudarytas iš vieno sąsūkos sluoksnio, kuris turi 32 filtrus ir išgauna iš paveikslėlio 3×3 dydžio sritis, su poslinkiu 1. Po sąsūkos sluoksnio eina didžiausio atmetimo sluoksnis (2×2 filtras). Tada eina 2 tirštieji sluoksniai (turintys po 128 neuronus), tarp kurių yra atmetimo sluoksnis su koeficientu 0.5.

3. *TensorFlow.js*

Vis daugiau kūrėjų naudoja *Keras* ir *TensorFlow* savo giliojo mokymosi projektuose. Viena didžiausių kliūčių žmonėms, kurie nori išbandyti gilųjį mokymą, yra visas reikalingos įrangos nustatymas, kuris gali užimti nemažai laiko. *TensorFlow.js* neturi tokio trūkumo, nes tinklą, apmokytą naudojant *TensorFlow.js*, galima paleisti tiesiogiai per naršyklę naudojant *JavaScript*.[**12**]

3.1. Istorija

Prieš rašant apie *TensorFlow.js*, reikėtų papasakoti apie patį *TensorFlow*. *TensorFlow* buvo sukurtas 2011 m. *„Google“*, kaip mašininio mokymosi/gilaus mokymosi programų biblioteka. *TensorFlow* yra parašytas naudojant *C++*, kuris leidžia dirbti žemam lygyje. *TensorFlow* turi ryšį su kitom kalbom, pvz., *Python*, *R* ir *Java*. Taigi, akivaizdus klausimas: kas gi apie *JavaScript*? Tradiciškai, *JavaScript*, gilusis mokymasis buvo atliekamas naudojant API. API buvo sukurtas naudojant tam tikrą karkasą, o modelis buvo įdiegtas serveryje. Klientas siuntė užklausą naudodamas *JavaScript*, kad gauti rezultatus iš serverio.

2017 m. pasirodė projektas vadinamas *Deeplearn.js*, kurio tikslas – leisti gilųjį mokymą atlikti *JavaScript* be API vargo. Tačiau buvo klausimų apie greitį. *DeepLearn.js* kodas negalėjo būti leidžiamas ant GPU. Kad išspręsti šią problemą, buvo pritaikytos *WebGL* (*JavaScript* API, skirta interaktyviosios 2D ir 3D grafikos vaizdavimui bet kurioje suderinamoje interneto naršyklėje, nenaudojant papildinių) galimybes *DeepLearn.js*, taip gimė *TensorFlow.js*. [**12**]

3.2. *TensorFlow.js* naudojimas

Ką galima daryti su *TensorFlow.js*?

- Galima importuoti esamą, iš anksto parengtą modelį. Jei turime *TensorFlow*, arba *Keras* modelį, kuris buvo anksčiau apmokytas, galime jį išsaugoti plačiai žinomu *Json* (*JavaScript Object Notation*) formatu, po to konvertuoti jį į *TensorFlow.js* formatą ir įkelti į naršyklę, kad galėtumėme daryti išvadas.
- Galima iš naujo apmokyti importuotą modelį. Galime naudoti perkėlimo mokymą, kad patobulinti esamą modelį, naudojant nedidelį kiekį duomenų, surinktų naršyklėje, naudojant techniką, vadinamą *„Image Retraining“*.

- Taip pat galima naudoti *TensorFlow.js*, norėdami apibrėžti, mokyti ir paleisti modelius visiškai naršyklėje naudojant *Javascript* ir aukšto lygio sluoksnių API (*LayerAPI*). Aukšto lygio sluoksnių API yra labai panašus į *Keras* API.

Tensorflow.js suteikia[**12**]:

- *CoreAPI* – susitvarko su žemo lygio kodu.
- *LayerAPI* – pastatytas ant *CoreAPI* ir palengvina mūsų gyvenimą, padidinant abstrakcijos lygį.

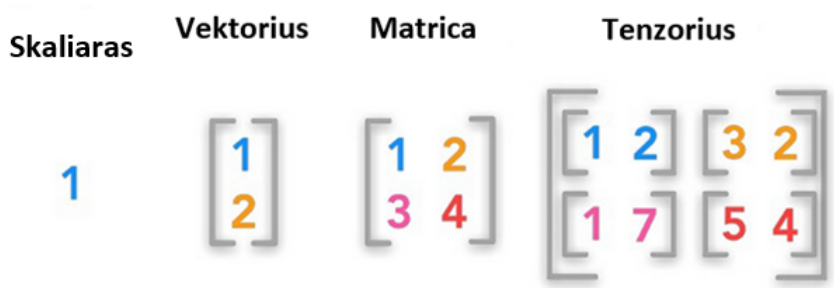
3.3. *CoreAPI*

TensorFlow.js yra biblioteka, leidžianti nustatyti ir paleisti skaičiavimus naudojant *JavaScript* tenzorius. Tenzorius yra vektorių ir matricų apibendrinimas į aukštesnes dimensijas.

TensorFlow.js centrinis duomenų vienetas yra *tf.Tensor*: reikšmių rinkinys, suformuotas į masyvą, kurio dimensija yra vienas, arba daugiau. *tf.Tensors* yra labai panašūs į daugiamatį masyvą.[**9**]

tf.Tensor taip pat turi šias savybes:

- *rank*: nustato, kiek matmenų yra tenzoriuje.
- *shape*: apibrėžia duomenų matmenį.
- *dtype*: kuris apibrėžia tenzoriaus duomenų tipą.



37 pav. Skaliaras, vektorius, matrica ir tenzorius.

- Skaliaras yra vienas numeris. Pavyzdžiui, x = 1.
- Vektorius yra skaičių masyvas. Pavyzdžiui, x = [1,2].
- Matrica yra 2-D masyvas ([[1, 2], [3, 4], [5, 6]]).

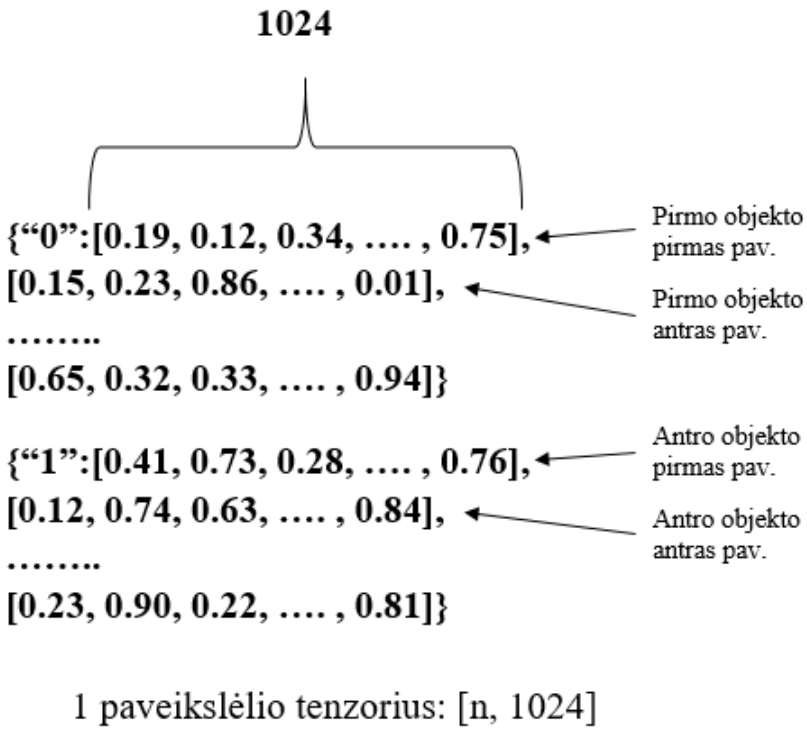
- Tenzorius yra n-dimensijos masyvas su n > 2.

TensorFlow.js turi naudingų funkcijų įprastiems atvejams, pvz., *„Scalar“*, *„1D“*, *„2D“*, *„3D“* ir *„4D“* tenzoriams, taip pat daug funkcijų, skirtų inicijuoti tenzorius gilioje mokymosi tikslais.

3.3.1. Vaizdų aktyvavimų išsaugojimas ir įkėlimas.

TensorFlow.js yra pakankamai naujas projektas, todėl jis dar neturi viso reikalingo funkcionalumo. Vienas iš jų yra vaizdų aktyvavimų išsaugojimas/įkėlimas. Tai buvo reikalingas funkcionalumas mūsų projekte, todėl mums teko jį realizuoti patiems. Visi vaizdų aktyvavimai buvo saugomi kaip tenzoriai, todėl teko panaudoti *TensorFlow.js* funkcijas skirtas darbui su tenzoriais.

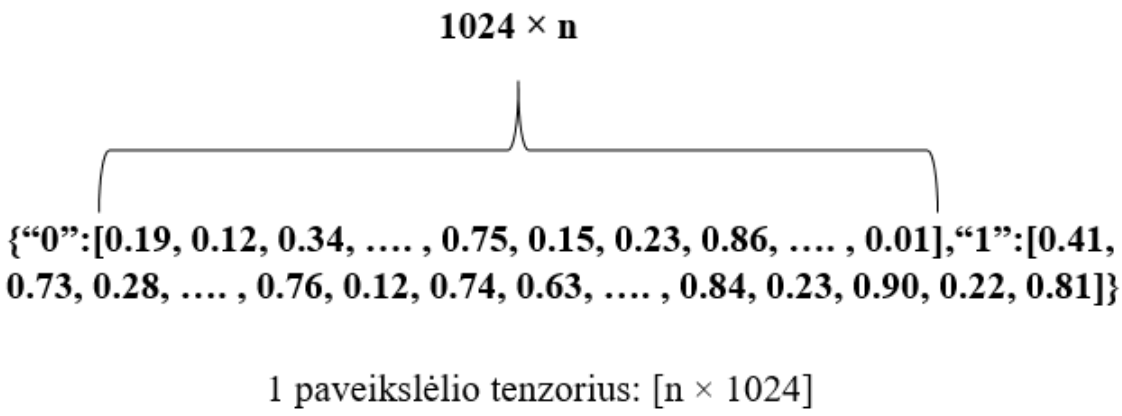
Tenzoriai yra saugomi pavidalo [n, 1024], kur n yra vieno objekto paveikslėlių kiekis (Pav.38).



38 pav. Paveikslėlių aktyvavimų tenzoriai.

Po išsaugojimo *Json* (*JavaScript Object Notation*) formatu, kiekvieno vaizdo tenzorius pasikeitė į pavidalą [1, n $\times$ 1024] (Pav.39).

Kad įkeltume išsaugotas paveikslėlių aktyvacijas į modelį, reikėjo kiekvieno objekto tenzorių padalinti iš 1024, kad gautume kiekvieno objekto paveikslėlių kiekį. Toliau panaudojant *TensorFlow.js* tensor funkciją, sukurti kiekvieno paveikslėlio tenzorių.



39 pav. Paveikslėlių aktyvavimų tenzoriai išsaugoti *Json* formatu.

3.4. *LayerAPI*

Sluoksniai yra pagrindinis statybinis blokas gilaus mokymo modelio konstravimui. Kiekvienas jų paprastai atliks tam tikrą skaičiavimą, kad transformuotų savo įvestį į savo *produkciją*. Po gaubtu kiekvienas sluoksnis naudoja *Tensorflow.js CoreAPI*.

Sluoksniai automatiškai pasirūpins, kad būtų sukurti ir inicijuoti įvairūs vidiniai kintamieji/svoriai. Taigi, iš esmės jie palengvina gyvenimą, didindami abstrakcijos lygį.[**12**]

3.5. *Handtrack.js*

Darbui atlikti naudojome *Handtrack.js* biblioteką, kuri, kaip galima suprasti iš pavadinimo, leidžia stebėti ir surasti rankų poziciją ant paveikslėlio. Ši biblioteka suranda ranką ant paveikslėlio ir apibrėžia keturkampį aplink ją. Mūsų atveju reikėjo surasti ranką vaizde, kuris buvo filmuojamas realiu laiku. *Handtrack.js* dalino vaizdo įrašą į atskirus kadrus ir surasdavo rankos poziciją kiekviename iš jų (Pav.40).

40 pav. Rankos atpažinimas realiu laiku.

Handtrack.js gali būti naudojamas daugelyje atvejuose. Kai kurie iš jų[**11**]:

- Kai pelės judėjimas gali būti susietas su rankos judesiu kontrolės tikslais.
- Kai rankų judesiai gali rodyti reikšmingus signalus.
- Scenarijai, kuriuose žmogaus rankų judėjimas gali būti naudojamas veiklos atpažinimui.

3.5.1. *Handtrack.js* API

Pateikiami keli metodai. Du pagrindiniai metodai yra *load()*, įkeliantis rankų aptikimo modelį ir *detect()* metodas, skirtas gauti prognozes[**11**]:

- *load()* – priima pasirinktinius modelio parametrus, kurie leidžia valdyti modelio veikimą. Šis metodas įkelia iš anksto paruoštą rankų aptikimo modelį (Pav.41).
- *detect()* – priima įvedimo šaltinio parametrą (*html* vaizdo objektą) ir grąžina apribojimo dėžutės, kurį nurodo rankos poziciją vaizde, dydį ir poziciją (Pav.42).

```
const img = document.getElementById('img');

handTrack.load(modelParams).then(model => {
    model.detect(img).then(predictions => {
    console.log('Predictions: ', predictions);
  });
});
```

41 pav. *Handtrack.js load()* ir *detect()* funkcijos.

```
[{
  bbox: [x, y, width, height],
  class: "hand",
  score: 0.8380282521247864
}, {
  bbox: [x, y, width, height],
  class: "hand",
  score: 0.74644153267145157
}]
```

42 pav. Prognozavimo rezultatai.

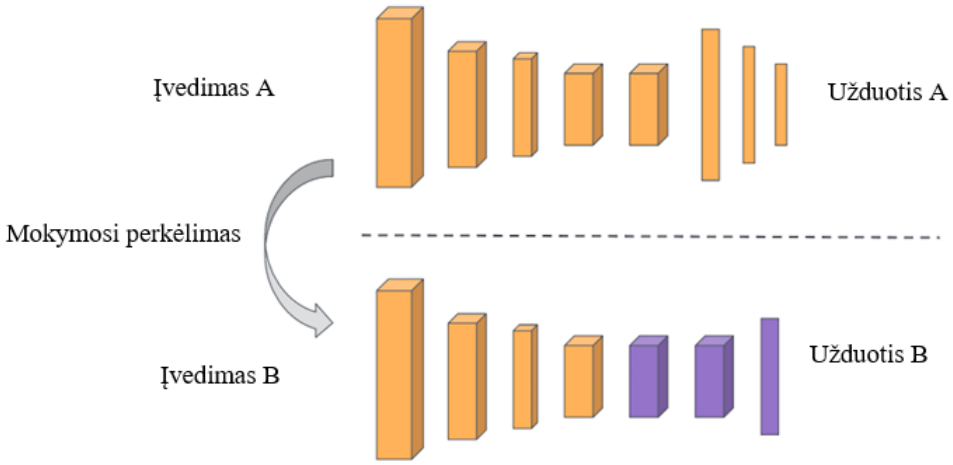
3.5.2. Duomenų surinkimas ir apribojimai

Apmokyti *Handtrack.js* buvo naudojami 4800 rankų vaizdų skirtinguose aplinkose (patalpose, lauke). Duomenys buvo padalinti, skiriant 80% apmokymui ir 20% testavimui.[**11**]

Pastebėjome, kad kartais prognozės yra neteisingos (pvz. kartais veidas aptinkamas kaip ranka). Manaome, kad tai atsitinka dėl skirtingų kamerų ir skirtingo apšvietimo. Tai galima būtu patobulinti papildomais duomenimis.

3.6. Mokymosi perkėlimas (*Transfer learning*)

Sudėtingi gilaus mokymo modeliai gali turėti milijonus parametrų, o pats mokymas dažnai reikalauja didelių skaičiavimo išteklių, papildomai mokymas gali užimti daug laiko. Mokymosi perkėlimas yra technika, kuri sutrumpina visą tai, paimdama modelį, kuris jau yra apmokytas susijusioje užduotyje, ir pakartotinai panaudojant tą modelį. Mes galime pritaikyti esamas žinias iš anksto parengtame modelyje, kad nustatytumėme savo vaizdo klases, naudojant daug mažiau mokymo duomenų, nei reikalautų originalus modelis. Tai naudinga sparčiai plėtojant naujus modelius, taip pat pritaikant modelius išteklių ribotoje aplinkoje, pavyzdžiui, naršyklėse ir mobiliuosiuose įrenginiuose. Dažniausiai, atliekant perkėlimą, nekoreguojame pradinių modelio svorių. Vietoj to mes pašaliname galutinį sluoksnį ir apmokame naują modelį (Pav.43). Savo darbui atlikti naudojome mokymosi perkėlimą, panaudojant *MobileNet* modelį, kuris jau yra apmokytas atpažinti tūkstančius skirtingų objektų tipų iš paveikslėlių.[**9**]



43 pav. Mokymosi perkėlimas (*Transfer learning*).

3.6.1. Modelio konstravimas

Pradžioje reikia įkelti *MobileNet* modelį. Po to nustatyti kamerą, per kurią bus gaunamas vaizdas, pagal kurį bus atliekamas atpažinimas.

```
async function app() {
  console.log('Loading mobilenet..');
  // Modelio įkelimas
  net = await mobilenet.load();
  console.log('Sucessfully loaded model');
  await setupWebcam();
  while (true) {
    // Prognozes sudarymas per atvaizdavimo modelį
    const result = await net.classify(webcamElement);
    document.getElementById('console').innerText = `
      prediction: ${result[0].className}\n
      probability: ${result[0].probability}
    `;
    await tf.nextFrame();
  }
}
```

44 pav. *MobileNet* įkelimas.

Dabar padarysime šį tinklą naudingesnį. Prie modelio pridėsime 3 klasių objekto klasifikatorių, su kurio pagalba galima bus tinklą apmokyti atpažinti 3 skirtingus objektus. Naudosiu modelį vadinamą „K artimiausiu kaimynų klasifikatoriumi“ (*K-Nearest Neighbors Classifier*), kuris leidžia kameros vaizdus (iš tikrųjų, jų *MobileNet* aktyvavimus) sudėti į skirtingas kategorijas (arba klases), o kai vartotojas prašo daryti prognozę, mes paprasčiausiai pasirenkame klasę, turinčią labiausiai panašią aktyvavimą į tą, kurio atpažinimą darome.

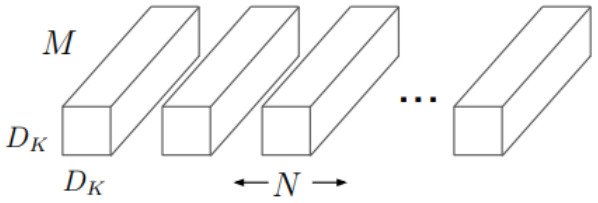
```
async function app() {
  console.log('Loading mobilenet..');
  // Modelio įkelimas
  net = await mobilenet.load();
  console.log('Sucessfully loaded model');
  await setupWebcam();
  // Nuskaito vaizdą iš kameros ir asocijuoja jį su speifinės klasės indeksu
  const addExample = classId => {
    // Gauti tarpinį MobileNet 'conv_preds' aktyvavimą ir perduoti jį į
    // KNN klasifikatorių
    const activation = net.infer(webcamElement, 'conv_preds');
    classifier.addExample(activation, classId);
  };
  // Kada paspaudžiamas mygtukas, pridėti pavyzdį nurodytai klasei
  document.getElementById('class-a').addEventListener('click', () => addExample(0));
  document.getElementById('class-b').addEventListener('click', () => addExample(1));
  document.getElementById('class-c').addEventListener('click', () => addExample(2));
  while (true) {
    if (classifier.getNumClasses() > 0) {
      // Gauti aktyvavimą iš vaizdo iš kameros
      const activation = net.infer(webcamElement, 'conv_preds');
      // Pasirinkti tinkamiausią klasę iš klasifikatoriaus modulio
      const result = await classifier.predictClass(activation);
      const classes = ['A', 'B', 'C'];
      document.getElementById('console').innerText = `
        prediction: ${classes[result.classIndex]}\n
        probability: ${result.confidences[result.classIndex]}
      `;
    }
    await tf.nextFrame();
  }
}
```

45 pav. Modelio konstravimas.

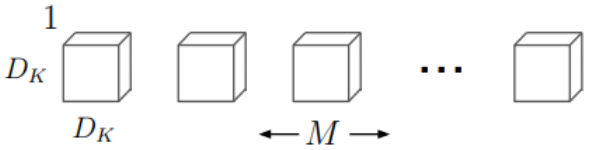
Toks tinklas atpažįsta rankoje laikomus didesnius, skirtingos spalvos objektus su tikimybe 79%, o vienodos spalvos – 74%. Mažesnius, skirtingos spalvos objektus atpažįsta su tikimybe 76%, o vienodos spalvos – 68%. Pridedant prie modelio papildomas klases atpažinti didesnį kiekį objektų, naršyklė negalėjo susitvarkyti su apmokytu tinklu.

3.7. *MobileNet* modelis

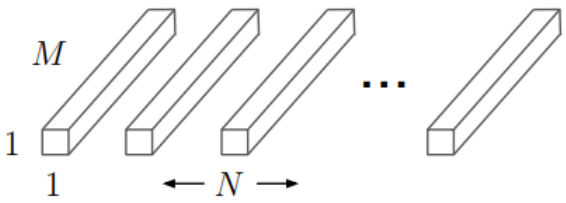
Mokymosi perkėlimui naudojome *MobileNet* modelį, nes jo architektūra yra pakankamai lengva. Jis naudoja giliai atskiriamas sąsūkas (*depthwise separable convolution*), kas iš esmės reiškia, kad kiekviename spalviniame kanale yra atliekama viena sąsūka, o ne visų trijų kanalų sujungimas ir išlyginimas (*flatten*). Tai lemia įvesties kanalų filtravimą. *MobileNet* gili sąsūka (*depthwise convolution*) priskiria vieną filtrą kiekvienam įvesties kanalui (Pav.47). Tada taškinė sąsūka (pointwise convolution) taiko 1x1 filtrą, kad sujungti išvedimą su gilią sąsūką (Pav.50). Standartinė sąsūka filtruoja ir sujungia įvedimus į naują išvesties rinkinį vienu žingsniu (Pav.46), o giliai atskiriama (depthwise separable convolution) tai padalija į du sluoksnius: atskirą filtravimo sluoksnį ir atskirą sluoksnį jungimui. Šis faktorizavimas labai efektyviai sumažina skaičiavimus ir modelio dydį.[**10**]



46 pav. Standartinė sąsūka (*standart convolution*).



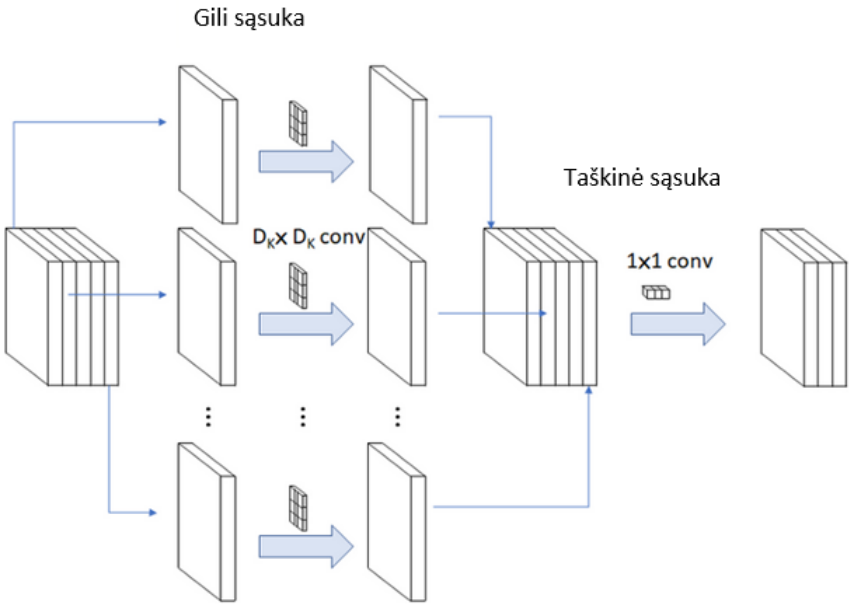
47 pav. Gili sąsūką (*depthwise convolution*).



48 pav. Giliai atskirta sąsūka (*depthwise separable convolution*).

3.7.1. Giliai atskiriama sąsuka (*Depthwise Separable Convolution*)

Giliai atskiriama sąsūka yra gili (*depthwise*) sąsūka, po kurios seka taškinė (*pointwise*) sąsūka (Pav.49).



49 pav. Giliai atskiriama sąsūka.

- Gili sąsūka yra kanalo dydžio $D_K \times D_K$ erdvinė sąsūka.
- Taškinė sąsūka iš esmės yra 1×1 filtras, skirtas dimensijai pakeisti.

Taigi operacijos kaina yra (giliai atskiriamos sąsūkos kaina: gilios sąsūkos kaina + taškinės sąsūkos kaina):

$$D_K \times D_K \times M \times D_F \times D_F + M \times N \times D_F \times D_F$$

kur M: įvedimo kanalų skaičius, N: išvedimo kanalų skaičius, D_K: branduolio dydis, D_F: aktyvavimo žemėlapio (*activation map*) dydis.

Standartinės sąsūkos atveju:

$$D_K \times D_K \times M \times N \times D_F \times D_F$$

Taigi skaičiavimo sumažinimas yra:

$$\frac{D_K \times D_K \times M \times D_F \times D_F + M \times N \times D_F \times D_F}{D_K \times D_K \times D_K \times M \times N \times D_F \times D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

Kai $D_K \times D_K$ yra 3×3, galima pasiekti 8–9 kartus mažesnį skaičiavimų kiekį, tačiau su nedideliu tikslumo sumažinimu.

3.7.2. *MobileNet* modelio architektūra

Bendra *MobileNet* architektūra yra sudaryta iš 30 sluoksnių[**10**]:

1. Sąsūkos sluoksnis su poslinkiu 2.
2. Gilus (*depthwise*) sluoksnis.
3. Taškinis (*pointwise*) sluoksnis, padvigubinantis kanalų skaičių.
4. Gilus (*depthwise*) sluoksnis su poslinkiu 2.
5. Taškinis (*pointwise*) sluoksnis, padvigubinantis kanalų skaičių.
6. etc.

Tipas / Poslinkis	Filtro forma	Įvedimo dydis
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

50 pav. *MobileNet* tinklo pilna architektūra.

Išvados

Atlikus darbą paaiškėjo, kad yra skirtingi būdai realizuoti gilųjį mokymą. Vienas iš jų yra panaudojant *Keras* biblioteką, kuri yra *TensorFlow* apvalkalas. Šis būdas yra geriau žinomas ir dažniau naudojamas. *Keras* turi daug naudingų funkcijų, kurios leidžia prieiti prie giliojo mokymosi problemų iš skirtingų pusių. Taip pat *Keras* labai gerai tinka žmonėms, kurie nori pradėti naudoti gilųjį mokymą, kadangi *Keras* neturi tokio didelio abstrakcijos lygio, kaip *TensorFlow.js*, todėl žmogus turi suprasti ką naudoją ir iškviečia rašant programą su *Keras*. Bet tuo pačiu metu *Keras* yra pakankamai draugiškas vartotoju, palyginant su *TensorFlow*. Vienas iš *Keras* trūkumų palyginant jį su *TensorFlow.js* yra įrangos nustatymas, kuris gali užimti nemažai laiko. Taip pat norint efektyviai apmokyti tinklą, reikia turėti pakankamai geras *Python* žinias.

TensorFlow.js yra pakankamai naujas projektas, kol kas ne daug žmonių netgi žino apie jį. Dėl to, kad tai yra naujas projektas, jame trūksta pakankamai daug reikalingo funkcionalumo, pvz: vaizdų aktyvavimų išsaugojimo ir įkėlimo. Taip pat *TensorFlow.js* nėra labai gerai optimizuotas – apmokant tinklą panaudojant didesnį kiekį objektų, naršyklė negalėjo susitvarkyti su apmokytu tinklu. Nekreipiant dėmesio į trūkumus, *TensorFlow.js* turi daug pranašumų. Jis turi didelį abstrakcijos lygį, kuris leidžia žmogui, kuris yra susipažinęs su giliuoju mokymusi, greitai sukurti ir apmokyti tinklą naršyklėje. Taip pat *TensorFlow.js* turi pakankamai daug naudingo funkcionalumo. Vienas iš jų yra mokymosi perkėlimas, su kurio pagalba tinklą, kuri reikėtu mokyti porą valandų, galima apmokyti per pora minučių. Nekreipiant dėmesio į trūkumus, *TensorFlow.js* yra projektas turintis daug potencialo, kuris ateityje taps vienu iš pagrindinių įrankių giliojo mokymosi srityje.

Šiame darbe sukūrėme neuroninį tinklą, kuris atpažįsta *„Lego“* detales su tikimybe 77%, panaudojant *Keras* biblioteką. Su *ImageJ* pagalba panaudojom skirtingus būdus normalizuoti duomenų rinkinį. Po skirtingų normalizavimų būdų lyginome paveikslėlių histogramas. Toliau palyginome skirtingas neuroninio tinklo architektūras. Toliau sukonstravome neuroninį tinklą *TensorFlow.js* pagalba, kuris randa rankos poziciją vaizde ir atpažįsta rankoje laikomą objektą su tikimybe 75%. Tinklo konstravimui naudojome mokymosi perkėlimą (*Transfer learning*), kurio pagalba apmokyti tinklą atpažinti objektą reikia kur kas mažiau apmokymo duomenų ir laiko. Rankos atpažinimui panaudojome *HandTrack.js* biblioteką.

Santrauka

Šiame darbe konstravome neuroninius tinklus naudojant *Keras* ir *TensorFlow.js* bibliotekas. Konstruojant neuroninį tinklą, kuris galėtu atpažinti skirtingas *„Lego“* detales, su *Keras*, bandėme normalizuoti duomenų rinkinį ir palyginti skirtingas tinklo architektūras. Konstruojant tinklą, kurį galėtume apmokyti atpažinti skirtingus rankoje laikomus objektus, su *TensorFlow.js*, panaudojom mokymo perkėlimą, su kurio pagalba tinklo apmokymui prireikė kur kas mažiau laiko ir apmokymo duomenų.

Summary

In this work we have constructed neural networks using Keras and TensorFlow.js libraries. By constructing a neural network, that could recognize different Lego pieces, with Keras, we tried to normalize the data set and compare different network architectures. Using a TensorFlow.js library to construct a neural network, that could be trained to recognize different hand-held objects, we used a training transfer that made network training less time-consuming and also require less training data.

Literatūros sąrašas

[1] Francois Chollet. *Deep Learning with Python*. October 28, 2017.
https://www.manning.com/books/deep-learning-with-python

[2] *Keras documentation*.
https://keras.io/

[3] *Convolutional neural networks with keras*.
https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/

[4] *Keras Functional API for Deep Learning*.
https://machinelearningmastery.com/keras-functional-api-deep-learning/

[5] *Applications of Deep Learning*.
https://machinelearningmastery.com/inspirational-applications-deep-learning/

[6] *ImageJ API*.
https://imagej.nih.gov/ij/developer/api/index.html

[7] *Writing ImageJ Plugins-A Tutorial*.
https://media.ijm.fr/fileadmin/www.ijm.fr/MEDIA/imagerie/fichiers/tut_pluginwb.pdf

[8] *Image Data Pre-Processing for Neural Networks*.
https://becominghuman.ai/image-data-pre-processing-for-neural-networks-498289068258

[9] *TensorFlow.js documentation*.
https://www.tensorflow.org/js

[10] Andrew G. Howard. *MobileNets: Efficient Convolutional Neural Networks for Mobile VisionApplications*. April 17, 2017.
https://arxiv.org/pdf/1704.04861.pdf

[11] *Handtrack.js: Hand Tracking Interactions in the Browser using Tensorflow.js*.
https://hackernoon.com/handtrackjs-677c29c1d585

[12] *Get to know TensorFlow.js*.
https://medium.freecodecamp.org/get-to-know-tensorflow-js-in-7-minutes-afcd0dfd3d2f

[13] *JavaScript for Machine Learning using TensorFlow.js*.
https://blog.bitsrc.io/javascript-for-machine-learning-using-tensorflow-js-6411bcf2d5cd