VILNIAUS UNIVERSITETAS

MATEMATIKOS IR INFORMATIKOS FAKULTETAS

DUOMENŲ MOKSLO STUDIJŲ PROGRAMA

Laboratorinis darbas

Konvoliuciniai neuroniniai tinklai

Ieva Prevelytė

Vilnius
2025

Iliustracijų sąrašas

Lentelių sąrašas

Turinys

Žymėjimai

Šiame laboratoriniame darbe buvo naudojami šie žymėjimai:

- a - neurono sužadinimo reikšmė;
- $f(a)$ - ReLu aktyvacijos funkcija;
- $h(a)$ - nesandaraus ReLu aktyvacijos funkcija;
- $g(a)$ - sigmoidinė aktyvacijos funkcija;
- y_c - Softmax aktyvacijos funkcijos rezultatas, tikimybė, kad nuotrauka priklauso klasei c;
- C - klasių skaičius;
- z_c - neuroninio tinklo išėjimo reikšmė, prieš pritaikant SoftMax funckiją;
- $L(Y)$ - bendra paklaida visam duomenų rinkiniui;
- t_{ic} - tikrojo reikšmė;
- γ - mastelio koeficientas transformacijos metu;
- β - poslinkio koeficientas transformacijos metu;

Įvadas

Dirbtinis intelektas yra vis plačiau minimas ne tik informatikos srityse, bet ir bendroje erdvėje. Ši technologija naudojama automatiniam vertimui, sintetinio turinio generavimui, veido atpažinimo funkcijoms ir panašiai. Vienas iš labiausiai nagrinėjamų šios srities tipų yra neuroniniai tinklai. Neuroniniai tinklai gali būti įvairūs: tiesioginio sklidimo *(ang. feedforward neural networks)*, konvoliuciniai *(angl. convolutional neural networks)*, rekurentiniai *(angl. recurrent neural networks)* ir pan.. Šio laboratorinio darbo metu bus nagrinėjami konvoluciniai neuroniniai tinklai. Šių tinklų paskirtis yra atpažinti vaizdo ar kitų struktūrizuotų duomenų bruožus. Jie naudojami ir ankščiau minėtose veido atpažinimo sistemose. Todėl norint suprasti šias sistemas, reikia išnagrinėti bazines sudėtines dalis. Svarbu atkreipti dėmesį į konvoliucinių neuronų architektūrą, hiperparametrų reikšmes ir bendrą veikimo procesą.

Darbo tikslas: apmokyti konvoliucinius neuroninius tinklus vaizdams klasifikuoti, atlikti tyrimą.
Darbo uždaviniai:

- Pasirinkti vaizdinius duomenis, juos paruošti klasifikavimui.
- Sukurti konvoliucinio neuroninio tinklo struktūrą ir jį apmokyti naudojant mažų paketų gradientinio nusileidimo principą;
- Nustatyti hiperparametrus: paketo dydį, branduolio dydį, branduolių kiekį, filtro dydį sujungimo sluoksniuose, išmetimo sluoksnio tikimybę, optimizavimo algoritmą ir t. t.;
- Atlikti tyrimą, kaip klasifikavimo tikslumas priklauso nuo neuroninio tinklo architektūros ir hiperparametrų reikšmių;
- Nustatyti, kada neuroninio tinklo tikslumas yra didžiausias;
- Rasti hiperparametrų rinkinį, kuris garantuotų didžiausią modelio tikslumą.

1. Duomenys

Konvoliuciniai neuroniniai tinklai dažniausiai naudojami vaizdų klasifikavimui, atpažinimui, todėl buvo pasirinkta žaidimo ”Akmuo-popierius-žirklės” vaizdai iš Kaggle platformos[Bru]. Duomenys turi 2188 skirtingus vaizdus, suskirstytus į 3 skirtingas aibes: popierius (710 vaizdai), žirklės (752 vaizdai) ir akmuo (726 vaizdai). Visose nuotraukose matoma ranka, kuri vaizduoja vieną iš trijų žaidimo komponentų.

Nuotraukų parametrai: 300×200.

1.1. Duomenų pradinis apdorojimas

Pirmiausia visi vaizdai buvo įkeliami iš GitHub platformos. Kadangi buvo trys skirtingos klasės: popierius, akmuo ir žirklės, duomenys buvo įkeliami atskirai ir pagal tai, kuri duomenų aibė yra įkeliama priskiriama klasė. Buvo priskiriamos skaitinės klasės: akmuo – 0, popierius – 1 ir žirklės – 2.

Kiekviena nuotrauka buvo atidaryta, užtikrinant RGB režimą. Kiekvienas nuotraukos pikselis buvo apibrėžtas pagal tris spalvų kanalus: raudoną, žalią ir mėlyną. Vėliau nuotraukos transformuojamos. Kadangi jų pradinis dydis yra 300×200, mes jas sumažiname iki 128×128 ir užtikriname, kad visos nuotraukos turėtų vienodą dydį, tinkamą klasifikavimo modeliui. Nuotraukos mažinamos, kad mokymas būtų greitesnis ir reikalautų mažiau atminties. Vaizdams suteikiame kvadratinį pavidalą, kad visi vaizdai būtų vienodo dydžio. Tai padeda modeliams lengviau apdoroti vaizdus ir išvengti objektų deformacijų. Įgyvendinimą galima pamatyti 1 paveikslėlyje.

Galiausiai visos nuotraukos sudedamos į bendrą duomenų aibę ir sumaišomos, prieš duomenų padalijimą į mokymosi, validavimo ir testavimo aibes. Jos bus aptartos vėliau.

```
data_dir = "cnnData/rps"
categories = ["rock", "paper", "scissors"]
valid_ext = (".jpg", ".jpeg", ".png")

transform = transforms.Compose([
    transforms.Pad(padding = (0, 0, 100, 0), fill = 0),
    transforms.Resize((128,128)),
    transforms.ToTensor()
])

label_map = {"rock": 0, "paper": 1, "scissors": 2}
file_paths, labels = [], []
for category in categories:
    folder = os.path.join(data_dir, category)
    for img_name in os.listdir(folder):
        if img_name.lower().endswith(valid_ext):
            file_paths.append(os.path.join(folder, img_name))
            labels.append(label_map[category])
```

1 pav. *Nuotraukų įkėlimas*

Svarbu pritaikyti duomenis darbui su neuroniniu tinklu. Buvo naudojamas PyTorch modelis, todėl duomenų aibė yra paverčiama į PyTorch duomenų rinkinį. Tam buvo sukurta atskira klasė. At-

skiriamos nuotraukos (x) ir jų etiketės (y), taip pat sukuriamos funkcijos. Viena gražina duomenų rinkinio dydį, kita gražina po vieną pavyzdį: paveikslėlį ir jo etiketę.

Galiausiai kiekviena duomenų aibė (mokymosi, validavimo ir testavimo) paverčiama į PyTorch duomenų rinkinį. Tada naudojama DataLoader funkcija, kuri suskaido duomenis į paketus, pagal pasirinktą dydį (žr. 2 pav.).

```
class ImageDataset(Dataset):
    def __init__(self, paths, labels, transform=None):
        self.paths = paths
        self.labels = labels
        self.transform = transform

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

    def __getitem__(self, idx):
        img = Image.open(self.paths[idx]).convert("RGB")
        if self.transform:
            img = self.transform(img)
        return img, torch.tensor(self.labels[idx], dtype=torch.long)

g = torch.Generator()
g.manual_seed(myseed)

train_set = ImageDataset(X_train, y_train, transform=transform)
val_set   = ImageDataset(X_val, y_val, transform=transform)
test_set  = ImageDataset(X_test, y_test, transform=transform)

train_loader = DataLoader(train_set, batch_size=32, shuffle=True, generator=g)
val_loader   = DataLoader(val_set, batch_size=32)
test_loader  = DataLoader(test_set, batch_size=32)
```

2 pav. *Duomenų pritaikymas darbui su PyTorch*

1.1.1. Mokymosi, validavimo ir testavimo aibės

Kaip buvo minėta anksčiau, duomenų aibė buvo padalinta į tris mažesnes dalis: mokymosi, validavimo ir testavimo.

Mokymosi duomenys yra pagrindiniai duomenys, kurie bus naudojami modelio pradiniam apmokymui. Tačiau dar yra validavimo ir testavimo aibės. Atlikus dirbtinio neurono apmokymą, validavimo aibė yra kontrolinė priemonė. Sprendžiant pagal šiuos duomenis, galima nuspręsti, ar pasirinkti hiperparametrai yra tinkami. Testavimo duomenys yra paskutinis žingsnis jau po neuroninio tinklo apmokymo. Ši duomenų aibė leidžia suprasti, kaip gerai modelis geba klasifikuoti nematytus duomenis ir kokį tikslumą galima tikėtis gauti realiose situacijose. Šio laboratorinio darbo metu duomenys buvo buvo paskirstyti 80:10:10 santykiu (žr. 3 pav.).

```
X_train_val, X_test, y_train_val, y_test = train_test_split(file_paths, labels, test_size=0.1, random_state=myseed)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.111, random_state=myseed)
```

3 pav. *Duomenų paskirstymas į mokymosi, validavimo ir testavimo aibes*

2. Konvoliucinio neuroninio tinklo implementacija

Laboratorinio darbo metu buvo sukurta konvoliucinio neuroninio tinklo struktūra. Tam buvo naudojama **PyTorch** biblioteka. Konvoliucinis modelis apdoroja vaizdus per kelis konvoliucinius sluoksnius, kurie automatiškai išmoksta išskirti reikšmingus bruožus, o pilnai sujungti sluoksniai šiuos bruožus naudoja klasifikacijai. Sukurta konvoliuvinio neuroninio tinklo klasė. Joje nustatyta galimybė keisti:

- tinklo architektūrą (sluoksnių skaičių ir jų tipą);
- aktyvacijos funkciją (ReLU, nesandarus ReLu, sigmoidinė);
- išmetimo sluoksnių skaičių ir tikimybę;
- paketų normalizacija;
- optimizavimo algoritmą (Adam, SGD, RMSprop);
- mokymosi greitį ir epochų skaičių.

Lankstus modelio struktūros įgyvendinimas leidžia atlikti testus tolimesniuose laboratorinio darbo etapuose. Klasės įgyvendinimą galima pamatyti 4 paveikslėlyje.

```python
class Classifier(nn.Module):
    def __init__(self, architecture = 'Seklus', activation = 'relu', drop_out = 0.5, drop_out_layers = 1, batchNorm = True):
        super(Classifier, self).__init__()

        if activation == 'relu':
            self.act = nn.ReLU()
        elif activation == 'leakyrelu':
            self.act = nn.LeakyReLU()
        else:
            self.act = nn.Sigmoid()

        def architecture_block(in_, out_, pool = True):
            layers = [nn.Conv2d(in_, out_, kernel_size = 3, padding=1)]
            if batchNorm:
                layers.append(nn.BatchNorm2d(out_))
            layers.append(self.act)
            if pool:
                layers.append(nn.MaxPool2d(2,2))
            return layers
```

4 pav. *Sukurta konvoliucinio neuroninio tinklo klasė*

Konvoliucinių neuroninių tinklų architektūra dažniausia sudaro skirtingi sluoksniai:

- Konvoliucinis sluoksnis - naudoja mažus filtrus, kurie eina per vaizdą ir ieško tam tikrų raštų. Kiekvienas filtras mokosi atpažinti skirtingą bruožą. Šiame darbe kiekviename konvoliuciniame sluoksnyje naudotas 3×3 branduolys su 1 pikselio užpildu.
- Aktyvacijos arba netiesiškumo sluoksnis - įveda nelinijiškumą, kad tinklas galėtų atpažinti ne tik tiesinius, bet ir netiesinius ryšius.
- Sujungimo sluoksnis - sumažina duomenų matmenis, išlaikant svarbiausią informaciją
- Normalizacijos sluoksnis - normalizuoja duomenis po kiekvieno sluoksnio.
- Išmetimo sluoksnis - apmokymo metu atsitiktinai išjungia dalį neuronų.
- Pilnai sujungtas sluoksnis - sujungia visus neuronus iš ankstesnių sluoksnių kartu. Šio sluoksnio paskirtis paversti visą gautą informaciją į galutinį sprendimą.

2.1. Skirtingos architektūros

Kaip buvo minėta ankščiau konvoliucinio tinklo struktūra suteikia galimybę išbandyti skirtingas tinklo architektūras, suprasti kaip jos veikia ir kaip nuo architektūros priklauso modelio tikslumas.

Buvo sukurti trys pagrindiniai variantai, kurie skiriasi konvoliucinių sluoksnių kiekiu, filtrų skaičiumi ir pilnai sujungtų sluoksnių struktūra (žr. 5 pav.).

```
        if architecture == 'Paprastas':
            conv_layers = architecture_block(3, 32) + architecture_block(32, 64)
            fc_input = 64*32*32
        elif architecture == 'Vidutinis':
            conv_layers = architecture_block(3, 32) + architecture_block(32,64) + architecture_block(64, 128)
            fc_input = 128*16*16
        else:
            conv_layers = (
                architecture_block(3, 32)
                + architecture_block(32, 64)
                + architecture_block(64, 128)
                + architecture_block(128, 256)
            )
            fc_input = 256*8*8

        self.cnn = nn.Sequential(*conv_layers)
        fc_layers = [nn.Flatten(), nn.Linear(fc_input, 512), self.act]
        for _ in range(drop_out_layers):
            fc_layers.append(nn.Dropout(drop_out))
            fc_layers.append(nn.Linear(512, 512))
            fc_layers.append(self.act)
        fc_layers.append(nn.Linear(512, 3))

        self.fc = nn.Sequential(*fc_layers)

    def forward(self, x):
        return self.fc(self.cnn(x))
```

5 pav. *Skirtingos galimos konvoliucinio neuroninio tinklo architektūros*

2.1.1. Seklus modelis

Seklus modelis sudarytas iš dviejų konvoliucinių sluoksnių, po kurių seka sujungimo sluoksniai (*angl. MaxPooling*) ir vienas pilnai sujungtas sluoksnis. Šis tinklas apdoroja pagrindinius vaizdo bruožus: kraštus, kontūrus, pagrindines formas. Tai paprasčiausia architektūra, dažniausiai naudojama kaip bazinė palyginimo versija, siekiant nustatyti papildomų sluoksnių naudą.

2.1.2. Vidutinis modelis

Vidutinis modelis turi tris konvoliucinius sluoksnius, kuriuose filtrų skaičius palaipsniui didėja. Tai leidžia tinklui mokytis sudėtingesnius bruožus. Taip pat yra du sujungti sluoksniai, kas padidina tinklo gebėjimą tiksliau klasifikuoti.

2.1.3. Gilusis modelis

Gilusis modelis sudarytas iš keturių konvoliucinių sluoksnių, kuriuose filtrų skaičius palaipsniui didėja nuo 32 iki 256. Po kiekvieno konvoliucinio sluoksnio taikomas sujungimo sluoksnis. Galiausiai seka vienas pagrindinis ir keli papildomi pilnai sujungti sluoksniai, tarp kurių įterpiami išmetimo (*angl. Dropout*) sluoksniai.

3. Neuroninio tinklo apmokymas

Neuroninio tinklo apmokymas buvo atliktas remiantis atgalinio sklidimo (*angl. Backpropagation*) principu. Jis leidžia koreguoti svorius pagal apskaičiuotą gradientą. Pagrindinis tikslas yra sumažinti skirtumą, tarp prognozuotų ir tikrųjų reikšmių. Modelio apmokymui taikytas mažų paketų gradientinis nusileidimas. Duomenys buvo padalinti į mažus paketus ir svoriai atnaujinami po kiekvieno paketo apdorojimo. Naudojamos paketinio ir stochastinio gradientinio nusileidimo idėjos. Mažų paketų gradientinis nusileidimas leidžia tinklui mokytis greičiau ir išnaudoti kompiuterio resursus (šio laboratorinio darbo metu GPU resursus).

Kiekvienam stebėjimui pakete yra apskaičiuojamas nuostolis ir gradientas pagal kryžminės entropijos nuostolių funkciją. Galiausiai paskaičiuojamas gradiento vidurkis ir pagal jį po kiekvieno paketo atnaujinami svoriai. Svoriams atnaujinti naudojami optimizavimo algoritmai (žr. 6 pav.).

```
while(train_loss > e_min and epoch < epochs):
        model.train()
        train_loss, train_acc = 0, 0

        for imgs, labels in train_loader:
            imgs, labels = imgs.to(device), labels.to(device)
            optimizer_choice.zero_grad()
            outputs = model(imgs)
            loss = criterion(outputs, labels) # paklaidos funkcija
            loss.backward()
            optimizer_choice.step()

            train_loss += loss.item()
            train_acc += (outputs.argmax(1) == labels).float().mean().item()

        train_loss /= len(train_loader)
        train_acc /= len(train_loader)

        model.eval()
        val_loss, val_acc = 0, 0
        with torch.no_grad():
            for imgs, labels in val_loader:
                imgs, labels = imgs.to(device), labels.to(device)
                outputs = model(imgs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                val_acc += (outputs.argmax(1) == labels).float().mean().item()

        val_loss /= len(val_loader)
        val_acc /= len(val_loader)
```

6 pav. *Neuroninio tinklo apmokymas*

Pirmiausia apmokyme buvo naudojami mokymosi duomenų aibė, po to tie patys veiksmai kaip buvo aptarta ankščiau atlikti ir su validavimo duomenimis. Galutiniam modelio patikrinimui buvo naudojami testavimo duomenys. Tai modeliui nematyti duomenys, kurių dėka gaunamas galutinis tikslumas (žr. 7 pav.).

```
model.eval()
test_loss, test_acc = 0, 0
all_preds, all_labels = [], []
with torch.no_grad():
    for imgs, labels in test_loader:
        imgs, labels = imgs.to(device), labels.to(device)
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        test_loss += loss.item()
        test_acc += (outputs.argmax(1) == labels).float().mean().item()

        all_preds.extend(outputs.argmax(1).cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

test_loss /= len(test_loader)
test_acc /= len(test_loader)
```

7 pav. *Neuroninio tinklo testavimas*

Neuroninio tinklo apmokymui buvo naudojama platforma **Google Colab**, kadangi ji suteikia prieigą prie grafinių procesorių (GPU). GPU leidžia vienu metu atlikti daug lygiagrečių skaičiavimų, todėl gerokai pagreitina apmokymo procesą. Prieš pradedant mokymą buvo patikrinamos GPU naudojimo galimybės, jeigu GPU nepasiekiamas, buvo naudotas centrinis procesorius (CPU). Tačiau CPU naudojimas sulėtina skaičiavimus net keletą kartų.

4. Hiperparametrai

Neuroninio tinklo apmokymo metu svarbu nustatyti tam tikrus parametrus: aktyvacijos funkciją (*angl. activation function*), mokymosi greitį (*angl. learning rate*), paketo dydį (*angl. batch size*), išmetimo sluoksnius (*angl. dropout layers*) ir kitus vėliau minimus hiperparametrus.

```
activation_functions = ['relu', 'leakyrelu', 'sigmoid'] # aktyvacijos funkcijos
optimizer_name = ['adam', 'sgd', 'RMSprop'] # optimizavimo algorimtai
drop_outs = [0.3, 0.5] # išmetimo tikimybės
drop_out_layers = [1, 2] # išmetimo sluoksnių kiekis
learning_rate = 0.001 # mokymosi greitis
batchNorms = [True, False] # paketų normalizavimas
n_epochs = 5 # epochų skaičius
e_min = 0.0001 # minimali paklaida
```

8 pav. *Nustatyti hiperparametrai*

Kaip matome 8 paveikslėlyje, buvo nustatyti skirtingi hiperparametrų variantai. Tai buvo atlikta norint nustatyti, kaip skirtingi hiperparametrai veikia modelio tikslumą ir galiausiai atrinkti modelį su geriausiu deriniu. Toliau svarbu panagrinėti, kam reikalingi skirtingi parametrai ir kuo skiriasi pasirinkti variantai.

4.1. Aktyvacijos funkcijos

Skirtingos aktyvacijos funkcijos naudojamos rasti išėjimo reikšmes kiekvienam neuronui. Tačiau jos yra labai skirtingos. Dvejetainio klasifikavimo atveju, paprasčiausios aktyvacijos funkcijos yra slenkstinė ir sigmoidinė. Tačiau šio laboratorinio darbo metu buvo nagrinėtos ir sudetingesnės: ReLu, nesandarus ReLu. Taip pat tikslumo palyginimui buvo paimta ir sigmoidinė aktyvacijos funkcija.

4.1.1. ReLu aktyvacijos funkcija

ReLu aktyvacijos funkcija leidžia neuroniniu tinklui išmokti sudėtingus, ne tiesinius ryšius tarp duomenų. Visos teigiamos išėjimo reikšmės lieka nepakitusios, tačiau neigiamos reikšmės yra paverčiamos į nulį (žr. 1 funkciją). Tai pagreitina neuroninių tinklų darbą, taip pat išsprendžiama nykstančių gradientų problemą, gradientas tampa per daug mažas ir nebegeba atnaujinti svorių. Šios aktyvacijos funkcijos atveju gradientas visoms teigiamoms reikšmėms yra 1, o neigiamoms 0. Tai užtikrina stabilų neuroninių tinklų apmokymą. Tačiau gali sukelti mirusių neuronų problemą, kadangi neigiamų sužadinimų gradientas lygus 0, ir neurono svoriai nebebus atnaujinami.

$$f(a) = max(0, a) \tag{1}$$

Ši funkcija leidžia tinklui efektyviai mokytis, suprasti ne tik tiesinius, bet ir netiesinius ryšius tarp duomenų, todėl yra viena dažniausia naudojamų funkcijų konvoliuciniuose neuroniniuose tinkluose.

4.1.2. Nesandarus ReLu(*angl. Leaky ReLu*) aktyvacijos funkcija

Nesandarus ReLu padeda išlaikyti nedidelį gradientą neigiamoms įėjimo reikšmėms. Ši funkcija veikia labai panašiai, kaip prieš tai aptarta ReLu funkcija, tačiau bandoma ištaisyti mirštančių neuronų problema. Neigiamos reikšmės yra sumažinamos, todėl gradientas, nors ir mažas, gali būti perduodamas toliau ir neuronas sėkmingai gali mokytis.

$$h(a) = \begin{cases} a, \text{ jeigu } a > 0 \\ 0{,}01a, \text{ jeigu } a \leq 0 \end{cases} \tag{2}$$

Kaip matome 2 funkcijoje, nesandari ReLu aktyvacijos funkcija teigiamoms reikšmėms suteiks gradiento reikšmę 1, o neigiamoms 0,001. Taip išvengiama ir nykstančio gradiento, ir mirusių neuronų problemos.

4.1.3. Sigmoidinė aktyvacijos funkcija

Sigmoidinė aktyvacijos funkcija yra viena geriausiai žinomų ir dažniausiai naudojamų funkcijų. Visos sužadinimo reikšmės įgyja reikšmes nuo 0 iki 1 (žr. 3 funkc.). Tačiau ji susiduria su nykstančio gradiento problema. Jeigu įėjimo reikšmės yra labai didelės arba mažas, gradientas tampa beveik nulis, svoriai atnaujinami labai lėtai arba nustoja mokytis.

$$g(a) = \frac{1}{(1 + e^{-a})}, e \approx 2{,}718 \tag{3}$$

4.2. Mokymosi greitis

Mokymosi greitis yra hiperparametras, kuris reguliuoja svorių atnaujinimo žingsnio dydį neurono apmokymo metu. Per mažas greitis sulėtiną apmokymą, tuo tarpu per didelis mokymosi greitis gali sukelti modelio svyravimus, paklaida šokinės ir neras galimai neras minimumo. Šio laboratorinio darbo metu buvo pasirinktas optimali **0,001** mokymosi greičio reikšmė.

4.3. Paketo dydis

Šio laboratorinio darbo metu apmokymui buvau naudojamas mažų paketų gradientinis nusileidimas. Todėl būtina nustatyti paketų dydį. Tai reprezentuoja, kiek duomenų vienu metu yra naudojama gradientui apskaičiuoti ir svoriams atnaujinti. Darbo metu buvo naudojami **32** duomenų paketai. Šis skaičius buvo pasirinktas, nes jis optimizuotas darbui su GPU.

4.4. Išmetimo sluoksniai ir tikimybės

Išmetimo sluoksniai ir tikimybės yra neuroninio tinklo reguliavimo metodas. Apmokymo metu atsitiktinai išjungiama dalis neuronų. Naudojant šį metodą, sumažinama persimokymo tikimybė. Neuroniniai tinklai geriau apibendrina duomenis, todėl validavimo ir testavimo duomenų tikslumas dažnai būna aukštesnis. Išmetimo sluoksniai yra įterpiami tarp pilnai sujungtų sluoksnių, o išmetimo

tikimybė nurodo, kokia dalis neuronų vieno mokymosi žingsnio metu bus laikinai išjungiami. Šis metodas taikomas tik mokymo metu. Validavimo ir testavimo duomenims jis netaikomas ir visi neuronai yra aktyvūs.

Laboratorinio darbo metu buvo testuojama, kurios išmetimo sluoksnių ir išmetimų tikimybių kombinacijos tinka šių duomenų klasifikavimui. Buvo naudojami **1** arba **2** išmetimo sluoksniai ir išmetimo tikimybės: **0,3** ir **0,5**.

Išmetimo tikimybės pasirinkta reikšmė yra labai svarbi. Jeigu reikšmė pasirenkama per didelė, gali būti sumažinamas modelio tikslumas, kadangi prarandama daug informacijos. Todėl buvo pasirinktos dvi reikšmės.

4.5. Optimizavimo algoritmas

Optimizavimo algoritmai yra naudojami svorių atnaujinimo metu. Jo metu yra nusprendžiama, kaip greitai ir kokia kryptimi turi būti keičiami svoriai, kad modelis pasiektų kuo didesnį tikslumą. Šio laboratorinio darbo metu buvo daromas testas ir tikrintas trijų optimizavimo algoritmų veiksmingumas. Laboratorinio darbo metu buvo sukurta funkcija trims algoritmams gauti (žr. 9 pav.).

```
def get_optimizer(optimizer_name, model, learning_rate):
    if optimizer_name == 'adam':
        return torch.optim.Adam(model.parameters(), lr = learning_rate)
    elif optimizer_name == 'sgd':
        return torch.optim.SGD(model.parameters(), lr = learning_rate, momentum=0.9)
    else:
        return torch.optim.RMSprop(model.parameters(), lr = learning_rate)
```

9 pav. *Optimizavimo algoritmo nustatymas*

4.5.1. Adam optimizavimo algoritmas

Adam (*angl. Adaptive Moment Estimation*) optimizavimo algoritmas yra vienas dažniausiai naudojamų neuroniniuose tinkluose. Jis apjungia dviejų metodų privalumus: AdaGrad (*angl. Adaptive Gradient Algorithm*) ir RMSProp (*angl. Root Mean Square Propagation*). Adam algoritmas apskaičiuoja kiekvieno parametro atnaujinimą pagal dabartinio ir ankstesnių gradientų vidurkius. Tokiu būdu jis mokosi, kaip keičiasi gradientų reikšmės. Nors laboratorinio darbo metu šiam algoritmui nustatomas bazinis mokymosi greitis, tačiau optimizavimo metu ši reikšmė kiekvienam parametrui koreguojama. Algoritmas remiasi bazine greičio reikšme, tačiau ją pakeičia priklausomai nuo gradientų pokyčių. Adam optimizavimo algoritmas nėra jautrus hiperparametrų pasirinkimui. Todėl dažnai pasirenkamas ypač tokio tipo užduotyse.

4.5.2. Stochastinio gradientinio nusileidimo optimizavimo algoritmas

Stochastinio gradientinio nusileidimo optimizavimo algoritmo metu svoriai atnaujinami po kiekvieno duomenų paketo. Apskaičiuojami gradientai kiekvienam stebėjimui pakete ir galiausiai svoriai atnaujinami naudojant gradientų vidurkį ir nustatytą mokymosi greitį. Kadangi svoriai yra

atnaujinami po paketo, o ne po kiekvieno stebėjimo atskirai, mokymosi procesas vyksta greičiau. Šio algoritmo metu taip pat naudojamas momentumas. Vietoje to, kad svoriai būtų keičiami atsižvelgiant tik į dabartinį gradientą, jie atnaujinami atsižvelgiant ir į ankstesnių gradientų kryptį. Taip mokymas tampa stabilesnis ir greitesnis. Tačiau Stochastinio gradiento nusileidimo optimizavimo algoritmas turi trūkumų. Jis yra labai jautrus pasirinkto mokymosi greičio reikšmei. Todėl svarbu pasirinkti tinkamą dydį.

Šio laboratorinio darbo metu jam buvo nustatyta **0,01** mokymosi greičio reikšmė. Taip pat **0,9** momentumo koeficientas, kuris reiškia, kad algoritmas prisimena 90% ankstesnių gradientų krypties.

4.5.3. RMSprop optimizavimo algoritmas

RMSprop (*angl. Root Mean Square Propagation*) veikia panašiai, kaip SGD optimizavimo algoritmas. Jis taip pat atnaujina svorius remdamasis gradientais po kiekvieno duomenų paketo. Tačiau pagrindinis skirtumas, kad RMSprop prisitaiko prie kiekvieno parametro mokymosi greičio pagal tai, kaip laikui bėgant kinta jo gradientai. Šis algoritmas seka gradientų kvadratus ir skaičiuoja jų slankųjį vidurkį, kad nuspręstų, kokio dydžio žingsnį daryti. Priešingai negu Adam algoritmas, RMSprop nenaudoja momentumo koeficiento. RMSprop reguliuoja tik žingsnio dydį.

4.6. Nuostolių funkcija

Nuostolių funkcija yra naudojama paskutiniame konvoliucinių neuroninių tinklų sluoksnyje, apskaičiuojant skirtumą tarp tikrųjų reikšmių ir tinklo išėjimo reikšmių. Šio laboratorinio darbo metu buvo naudojama kryžminės entropijos nuostolių funkcija (*angl. cross Entropy Loss*). Ši funkcija yra dažniausia naudojama, kai atliekamas kelių klasių klasifikavimas. Šios funkcija įvertina, kiek modelio prognozuotos tikimybės sutampa su tikrosiomis klasėmis. Šiuo atveju tikimybės yra apskaičiuojamos SoftMax funkciją (žr. 4 funkc.). Ši funckcija paima z_c išėjimo reikšmes ir paverčia jas į tikimybes y_c. Kitaip tariant, kiekvienai klasei priskiriama tikimybė, kad įvestis priklauso būtent tai klasei. Kuo didesnė z_c reikšmė, tuo didesnė tikimybė, kad pavyzdys priklauso tam tikrai klasei.

$$y_c = \frac{e^{z_c}}{\sum_{d=1}^{C} e^{z_d}}, \text{kur} c = 1,...,C \tag{4}$$

Panaudojus SoftMax funkciją, skaičiuojamas bendras duomenų nuostolis. Ji apskaičiuojama pagal 5 funkciją. Apskaičiuojamas kryžminės entropijos nuostolis visiems n duomenims ir visoms C klasėms. Tikroji reikšmė t_{ic} įgyja 1 ir 0 reikšmes, priklausomai nuo to ar stebėjimas priklauso pasirinktai klasei c ar ne. Suskaičiuojama SoftMax apskaičiuotos tikimybės logaritmas, kadangi ši reikšmė bus neigiama, dauginame iš -1. Visos tikrosios reikšmės yra dauginamos su tikimybių logaritmais, taip pasilieka tik teisingai priskirtos reikšmės. Galiausiai jos susumuojamos ir gaunamas bendras modelio nuostolis.

$$L(Y) = -\sum_{i=1}^{n}\sum_{c=1}^{C}(t_{ic} \times log(y_{ic})) \tag{5}$$

4.7. Paketų normalizavimas

Paketų normalizavimas yra metodas naudojamas pagreitinti modelio apmokymą ir stabilizuoti tinklo veikimą. Neuroniniai tinklai yra jautrūs, jeigu reikšmių skalės skiriasi. Dėl šios priežasties sukuriamas paketo normalizavimo sluoksnis, kuriam perduodamos aktyvacijos funkcijos reikšmės.

Kiekvienam paketui atskirai apskaičiuojamas kiekvieno požymio vidurkis ir standartinis nuokrypis, o vėliau duomenys standartizuojami. Tai reiškia, kad duomenų vidurkis bus 0, o standartinis nuokrypis 1. Po to atliekama duomenų transformacija, kurios metu pakeičiamas vidurkis ir standartinis nuokrypis, pagal koeficientus (γ ir β), apskaičiuotus apmokymo metu (žr. 6 funkc.).

$$y_i = \gamma \hat{x}_i + \beta \tag{6}$$

y_i yra galutinė normalizuotų duomenų elemento reikšmė po transformacijos. γ yra mastelio koeficientas, kuris leidžia keisti duomenų sklaidą. β yra poslinkio koeficientas, leidžia keisti duomenų vidurkį po normalizavimo. Šie koeficientai yra apmokomi kartu su neuroninių tinklų svoriais, todėl modelis juos pats prisitaiko prie duomenų pasiskirstymo. Paketų normalizavimas padeda sumažinti nykstančių gradientų problemą.

5. Geriausių hiperparametrų derinio radimas

Šio laboratorinio darbo metu buvo atlikta daug testų, padedančių įvertinti, kaip skirtingi hiperparametrai veikia modelio tikslumą. Tačiau svarbu nustatyta, kuris hiperparametru rinkinys užtikrina didžiausią tikslumą ir geriausiai klasifikuoja.

Modelio hiperparametrų paieškai buvo naudotas **„Optuna"** optimizacijos įrankis. Šis metodas leidžia rasti tinkamiausius hiperparametrų rinkinius. „Optuna" remiasi Bajeso optimizacijos metodu, kuris įvertina ryšį tarp hiperparametrų ir gauto rezultato (šiuo atveju validavimo tikslumo). Vietoje atsitiktinės paieškos naudojama ankstesnių bandymų informacija ir parenka naujus hiperparametrų rinkinius, pagal tikimybę pasiektį aukštesnį tikslumo rezultatą.

Kiekvieno bandymo metu „Optuna" parenka skirtingas hiperparametrų kombinacijas. Modelis apmokomas, skaičiuojamas modelio tikslumas. Pasibaigus visiems bandymams pateikiamas hiperparametrų rinkinys, kuris užtikrina didžiausią modelio tikslumą. Tokiu būdu Bajeso optimizacija padeda greičiau rasti tinkamiausius hiperparametrus. Optimizavimo įgyvendinimas matomas 10 paveikslėlyje.

```
def objective(trial):
    architecture = trial.suggest_categorical("architecture", architectures)
    activation = trial.suggest_categorical("activation", activation_functions)
    drop_out = trial.suggest_categorical("drop_out", [0.3, 0.5])
    drop_out_layer = trial.suggest_int("drop_out_layer", 1, 2)
    batchNorm = trial.suggest_categorical("batchNorm", [True, False])
    opt_name = trial.suggest_categorical("opt_name", optimizer_name)

    model, train_accs, train_losses, val_accs, val_losses, test_acc, test_loss, y_true, y_pred = train_and_validate(
        epochs=3, e_min=e_min,
        architecture=architecture,
        activation=activation,
        drop_out=drop_out,
        drop_out_layers=drop_out_layer,
        batchNorm=batchNorm,
        opt_name=opt_name,
        learning_rate=learning_rate
    )

    return val_accs[-1]

study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=20)
print("Geriausi parametrai:", study.best_params)
```

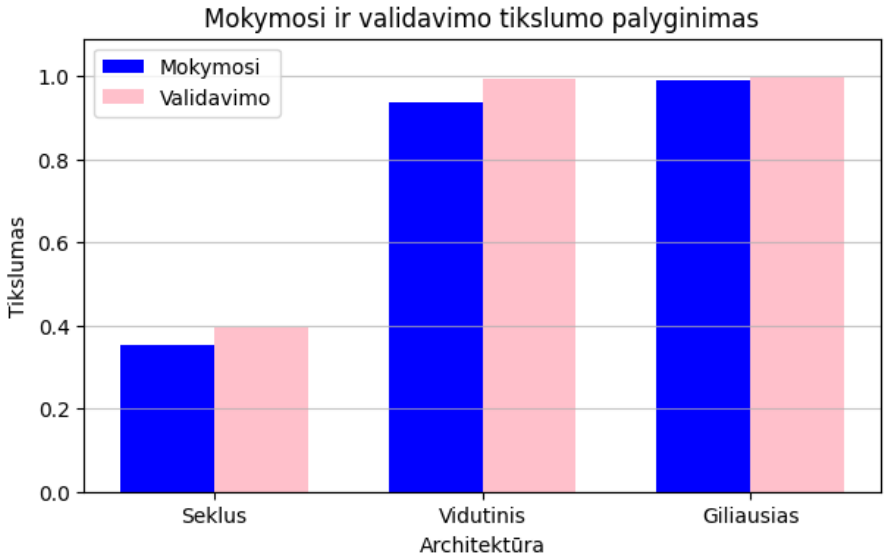
10 pav. *Geriausių hiperparametrų rinkinių nustatymas*

6. Rezultatai

Laboratorinio darbo metu buvo atlikti testai su skirtingais hiperparametrais. Buvo siekta išsiaiškinti, kaip skirtingi pasirinkimai gali įtakoti modelio tikslumą ir kurie parametrai geriausi. Galiausiai rastas geriausias hiperparametrų rinkinys nagrinėtam duomenų rinkiniui klasifikuoti.

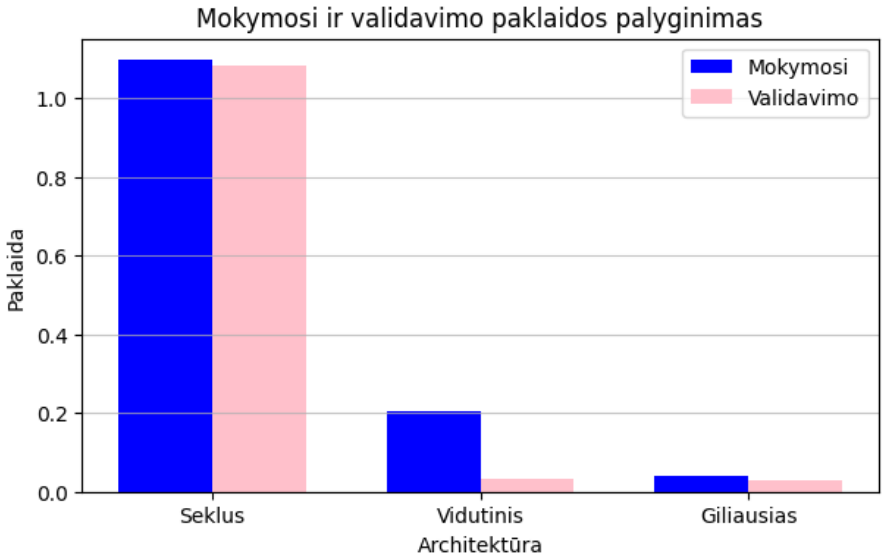
6.1. Modelio tikslumo priklausomybė nuo tinklo architektūros

Buvo išbandytos trys skirtingos architektūros: sekli, vidutinė ir giliausia. Kaip matyti 11 paveikslėlyje, didžiausią tikslumą pasiekė giliausias ir vidutinis modelis, tačiau seklio architektūros modelis nepasiekia net 0,5 tikslumo ribos. Tai rodo, kad modelis neišmoksta atpažinti reikšmingų raštų, bruožų. Šio tinklo tipas turi tik du konvoliucinius sluoksniu, todėl nesugeba aptikti sudėtingesnių detalių.



11 pav. *Mokymosi ir validavimo tikslumas skirtingoms architektūroms*

Šiuos rezultatus patvirtina ir 12 paveikslėlis. Galima pastebėti, kad seklios architektūros paklaida perkopia vieneto ribą. Tuo tarpu giliausio paklaidos yra labai mažos. Iš mokymosi ir validavimo tikslumo ir paklaidos palyginimų, nežvelgiamas persimokymas, todėl toliau galima pereiti prie testavimo.



12 pav. *Mokymosi ir validavimo paklaida skirtingoms architektūroms*

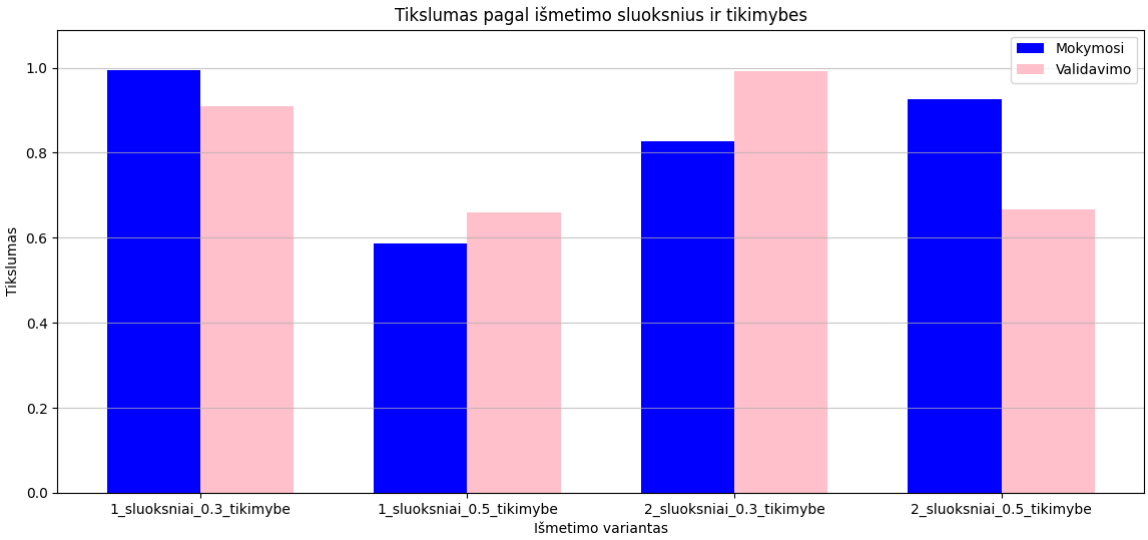
1 lentelėje matomi testavimo duomenų tikslumai ir paklaidos. Kaip ir ankščiau matytuose grafikuose didžiausią tikslumą įgyja giliausia architektūra. Ši struktūra pasiekė net 0,988 tikslumą, nes turi daugiau filtrų gebančių išgauti sudėtingesnius bruožus. Seklios architektūros tikslumas yra labai mažas, todėl labiausiai tikėtina, kad geriausiame hiperparametrų rinkinyje bus pasirinkta vidutinis arba gilusis modelis.

1 lentelė. *Testavimo tikslumas ir paklaida skirtingoms architektūroms*

Architektūra	Testavimo tikslumas	Testavimo paklaida
Seklus	0,371	1,087
Vidutinis	0,976	0,049
Gilusis	0,988	0,053

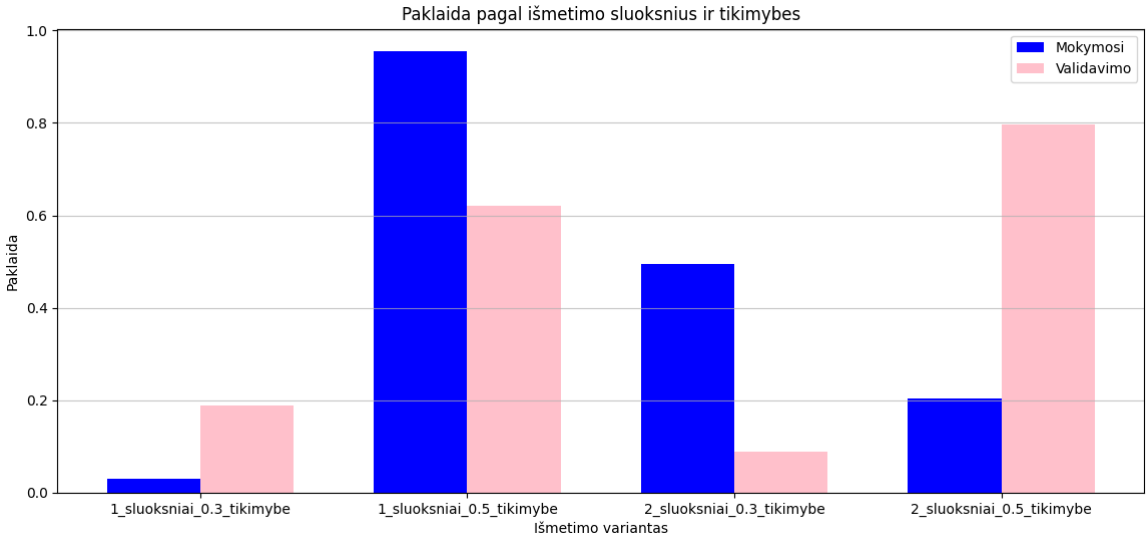
6.2. Modelio tikslumo priklausomybė nuo išmetimo sluoksnių ir išmetimo tikimybės

Svarbu išsiaiškinti, kiek išmetimų sluoksnių ir su kokiomis tikimybėm reikėtų, kad pasiekti didžiausią tikslumą. Pagal 13 galima pastebėti, kad didžiausius mokymosi ir validavimo tikslumus pasiekė modeliai su 1 išmetimo sluoksniu su 0,3 tikimybe ir 2 išmetimo sluoksniai su 0,3 tikimybe. Tikėtina, kad 0,5 išmetimo tikimybė yra per didelė šiems duomenims, ir taip panaikiname per daug reikalingos informacijos apmokymo metu.



13 pav. *Mokymosi ir validavimo tikslumas pagal skirtingus išmetimo sluoksnių kiekius ir tikimybes*

Paklaidų grafikas taip pat parodo tapačią tendenciją. 14 paveikslėlyje matome, kad 1 sluoksnis su 0,5 tikimybe turi beveik dvigubai didesnę paklaidą, negu visi kiti modeliai. Tačiau taip pat galima pastebėti, kad modelis su 2 sluoksniais ir 0,5 išmetimo tikimybe, galimai persimoko. Matomas didelis skirtumas tarp validavimo duomenų paklaidos ir mokymosi paklaidos. Kadangi mokymosi duomenų aibės paklaida gerokai mažesnė, atsiranda įtarimas, kad neuroninis tinklas per daug prisitaiko prie mokymosi duomenų.



14 pav. *Mokymosi ir validavimo paklaida pagal skirtingus išmetimo sluoksnių kiekius ir tikimybes*

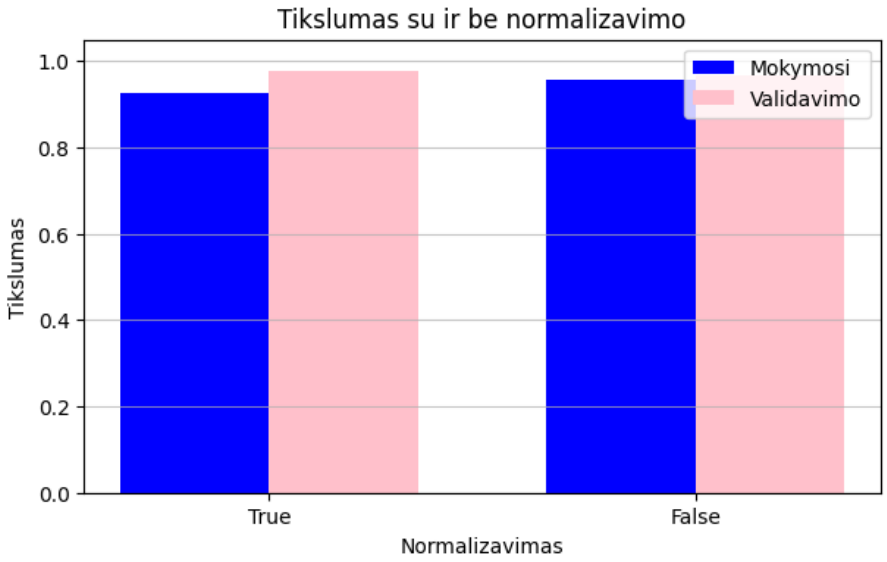
Galiausiai visos prielaidos pasitvirtina pažvelgus į testavimo tikslumo ir paklaidos lentelę (žr. 2 lent.). Galima pastebėti, kad 2 išmetimo sluoksnių modelio su 0,3 tikimybe tikslumas pernoko 1 modelį. Todėl optimaliausias variantas šių duomenų klasifikavimui yra 2 sluoksniai su 0,3 išmetimo tikimybe, nes išvengiama persimokymo.

2 lentelė. *Testavimo tikslumas ir paklaida pagal skirtingus išmetimo sluoksnių kiekius ir tikimybes*

Išmetimo sluoksnių kiekis ir tikimybė	Testavimo tikslumas	Testavimo paklaida
1 sluoksnis, 0,3 tikimybė	0,898	0,261
1 sluoksnis, 0,5 tikimybė	0,653	0,629
2 sluoksniai, 0.3 tikimybė	0,984	0,077
2 sluoksniai, 0,5 tikimybė	0,677	0,839

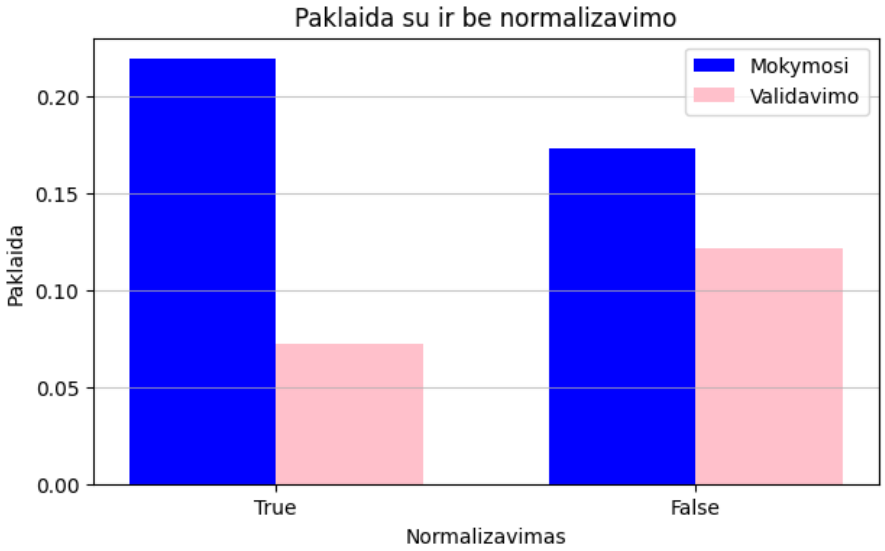
6.3. Modelio tikslumo priklausomybė nuo paketų normalizavimo

Laboratorinio darbo metu taip pat buvo tiriama, kokią įtaką paketų normalizavimas turi modelio tikslumui. Kaip matoma 15 paveikslėlyje, skirtumas tarp tikslumų yra labai mažas. Abudu modeliai, nesvarbu ar normalizavimas pritaikytas ar ne, pasiekia gan aukštą tikslumą. Tai reiškia, kad duomenų rinkinys yra subalansuotas. Kaip buvo aptarta prieš tai visos klasės turi panašų duomenų kiekį, taipogi nuotraukos yra aiškios ir kontrastingos.



15 pav. *Mokymosi ir validavimo tikslumas pagal normalizavimo pritaikymą*

Pažvelgus į 16 grafiką, taip pat matoma, kad paklaidos labai nesiskiria.



16 pav. *Mokymosi ir validavimo paklaida pagal normalizavimo pritaikymą*

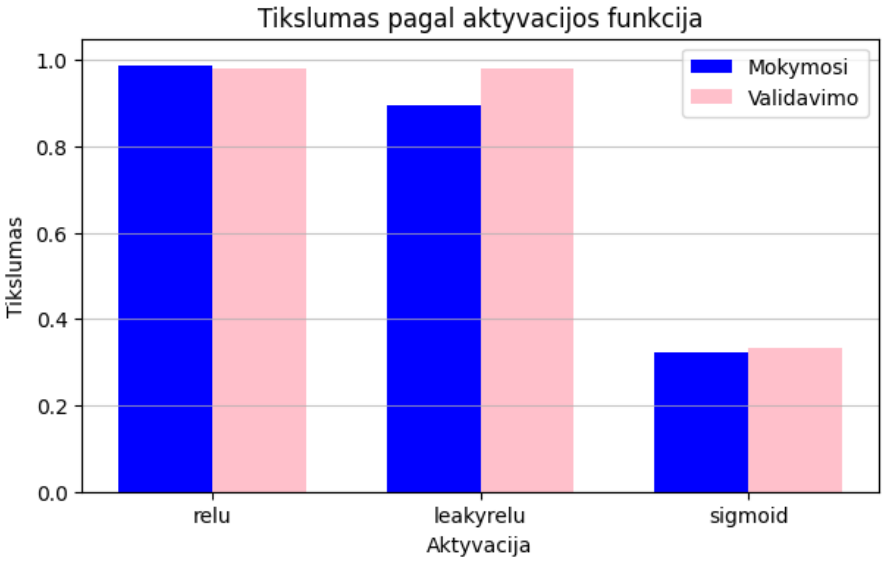
Galima daryti išvada, kad nesvarbu ar normalizavimas taikomas ar ne, tiek apmokymo metu, tiek testavimo metu bus pasiektas didelis modelio tikslumas (žr. 3 lent.).

3 lentelė. *Testavimo tikslumas ir paklaida pagal normalizavimą*

Normalizavimo taikymas	**Testavimo tikslumas**	**Testavimo paklaida**
Su normalizavimu	0,979	0,050
Be normalizavimo	0,981	0,017

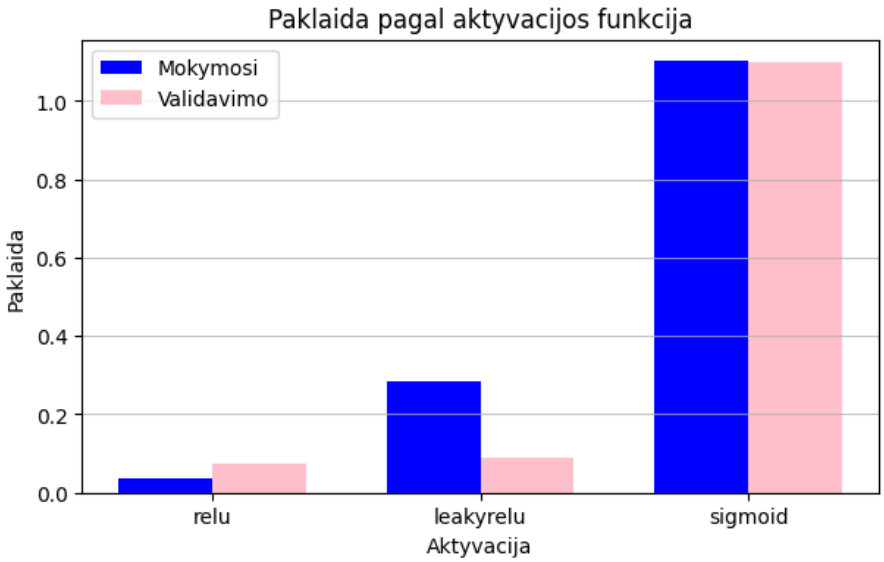
6.4. Modelio tikslumo priklausomybė nuo aktyvacijos funkcijos

Buvo naudotos trys skirtingos aktyvacijos funkcijos: ReLu, nesandarus ReLu ir sigmoidinė. Žvelgiant į 17 paveikslėlį, galima pamatyti, kad didžiausią tikslumas buvo pasiektas naudojant ReLu aktyvacijos funkciją. Tačiau dvigubai prastesnį tikslumą pasiekė sigmoidinė funkcija. Galima daryti prielaidą, kad taip atsitinka dėl nykstančių gradientų problemos.



17 pav. *Mokymosi ir validavimo tikslumas pagal aktyvacijos funkcijas*

Labai didelę sigmoidinės funkcijos paklaidą pastebime ir 18 paveikslėlyje. Ji perkopia net 1 riba, tiek mokymosi, tiek validavimo duomenims. ReLu ir nesandarus ReLU turi panašią validacijos duomenų paklaidą, todėl testavimo tikslumas tikėtinai taip pat panašus.



18 pav. *Mokymosi ir validavimo paklaida pagal aktyvacijos funkcijas*

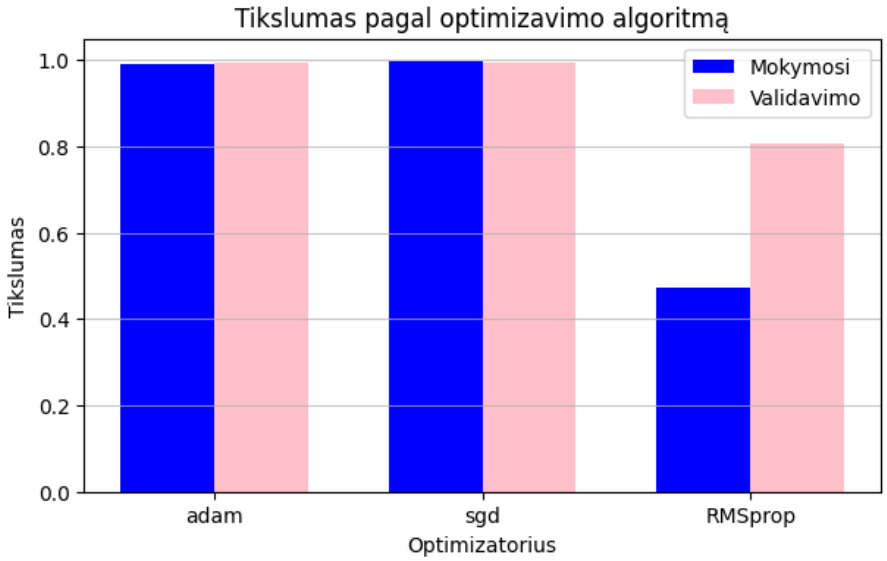
4 lentelėje, matome, kad prielaida dėl testavimo tikslumo panašumo pasitvirtino. ReLu ir nesandarus Relu išlaiko labai panašius tikslumus ir paklaidas, panaikindami nykstamojo gradiento problema. Tuo tarpu sigmoidinė aktyvacijos funkcija turi labai mažą tikslumą, todėl jos naudoti nerekomenduojama. Sigmoidinė funkcija tinkama paprastesniems tinklams, tuo tarpu ReLu funkcija patartina naudoti konvoliuciniams neuroniniams tinklams.

4 lentelė. *Testavimo tikslumas ir paklaida pagal aktyvacijos funkcijas*

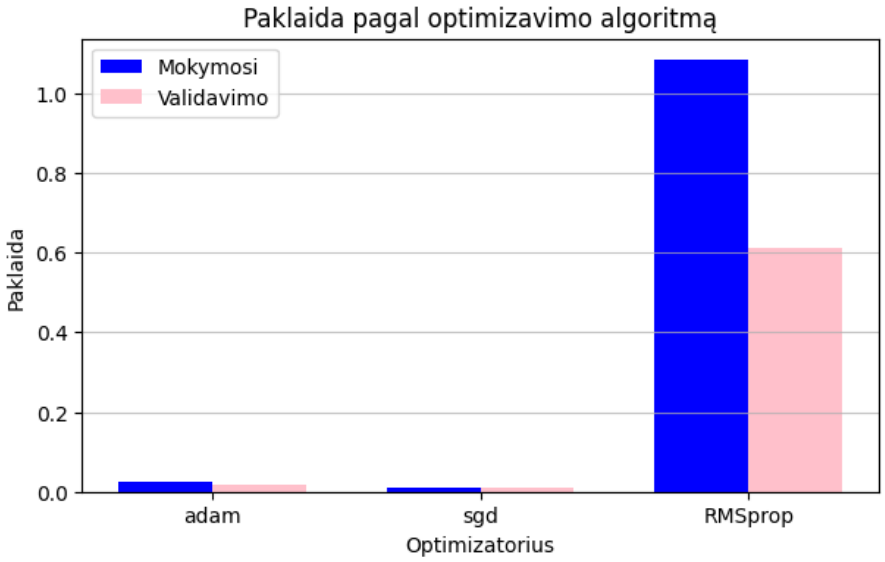
Aktyvacijos funkcija	Testavimo tikslumas	Testavimo paklaida
ReLu	0,954	0,128
Nesandarus ReLu	0,948	0,138
Sigmoidinė	0,342	1,099

6.5. Modelio tikslumo priklausomybė nuo optimizavimo algoritmo

Nuo skirtingų optimizavimo algoritmų priklauso, kaip bus atnaujinami svoriai. Todėl laboratorinio darbo metu buvo nagrinėti trys skirtingi algoritmai: adam, SGD, RMSprop. Jie buvo aprašyti ankščiau. Pasižiūrėjus į 19 ir 20 paveikslėlius, galime pamatyti, kad adam ir SGD tikslumas yra labai panašus, taip pat ir paklaida. Tačiau pastebimas skirtumas su RMSprop optimizacijos tikslumu, jis žymiai mažesnis. Tai tikriausiai atsitinka, nes RMSprop nenaudoja momentumo ir blogiau stabilizuoja svorius.



19 pav. *Mokymosi ir validavimo tikslumas pagal optimizavimo algoritmą*



20 pav. *Mokymosi ir validavimo paklaida pagal optimizavimo algoritmą*

Adam ir SGD tikslumo panašumas pastebimas ir testavimo metu gautų tikslumo ir paklaidos lentelėje (žr. 5 lent.). Nors RMSprop turi gan gerą testavimo tikslumą, tačiau rekomenduojama remtis kitais dviem algoritmais, nuotraukų klasifikavimui.

5 lentelė. *Testavimo tikslumas ir paklaida pagal optimizavimo algoritmus*

Optimizavimo algoritmas	Testavimo tikslumas	Testavimo paklaida
Adam	1	0,007
SGD	0,996	0,016
RMSprop	0,783	0,639

6.6. Geriausias modelis

Laboratorinio darbo pabaigoje buvo atliktas bajeso optimizacijos metodas, geriausiam hiperparamterų rinkiniui nustatyti. Taip buvo atrinkti geriausi parametrai, su kuriais modelis pasiekia didžiausią tikslumą ir turi mažiausią paklaidą.

Geriausi parametrai:

- Architektūra: Giliausias
- Aktyvacijos funkcija: ReLU
- Išmetimo sluoksnių kiekis: 2
- Išmetimo tikimybė: 0,5
- Su normalizavimu
- Optimizavimo algoritmas: SGD

Geriausi parametrai atitinka hiperparametrų testavimo metu gautas išvadas. Naudojami didžiausius tikslumus pasiekę hiperparametrai.

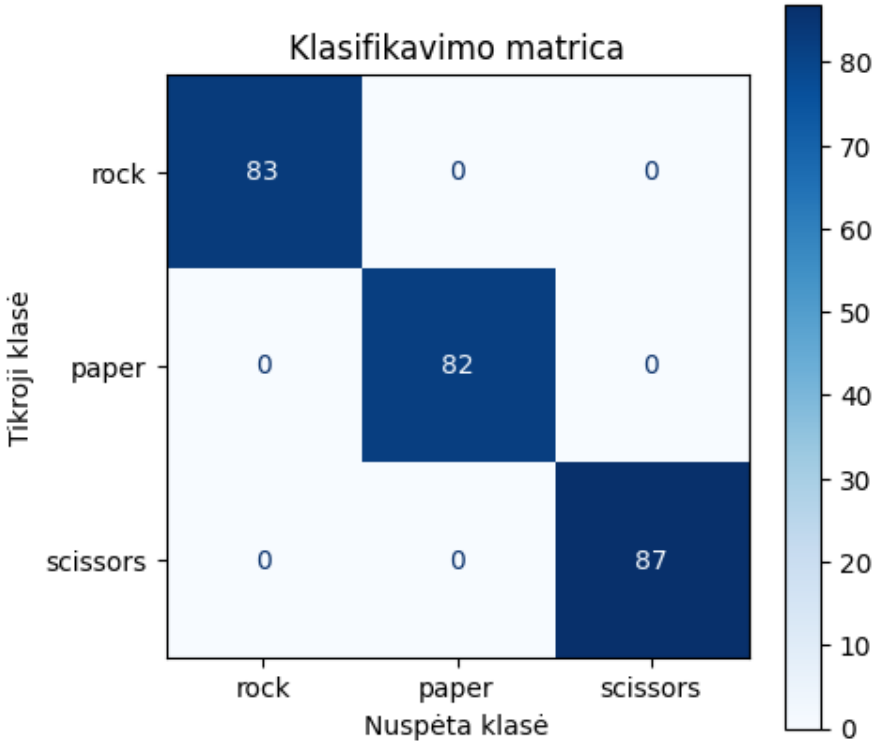
Apmokius galutinį modelį su parinktais hiperparametrais ir jį ištestavus, buvo gauti galutinio

tikslumo ir paklaidos duomenys (žr. 6 lent.) Galima pamatyti, kad pasiekiamas maksimalus tikslumas ir paklaida yra labai maža. Tai reiškia, kad visos reikšmės testuojant atitiko savo tikrąsias reikšmes.

6 lentelė. *Geriausio modelio testavimo tikslumas ir paklaida*

Testavimo tikslumas	Testavimo paklaida
1	0,005

Geriau įsivaizduoti modelio tikslumą galima pažvelgus į 21 paveikslėlį. Galima pamatyti, kad visos klasės buvo prognozuotos teisingai. Kiekviena prognozuota reikšmė atitinka tikrąją klasę.



21 pav. *Geriausio modelio klasifikavimo matrica*

Iš 7 lentelės, galime matyti, kad visos prognozuotos klasės yra teisingos. Todėl modelis klasifikuoja labai gerai. Patvirtinamas maximalus modelio tikslumas. Tačiau gali būti įžvelgiamas pavojus, kad modelis per daug prisitaikęs prie duomenų rinkinio.

7 lentelė. *30 įrašų palyginimas*

Nr.	Tikroji klasė	Prognozuota klasė
1	paper	paper
2	paper	paper
3	scissors	scissors
4	rock	rock
5	scissors	scissors
6	paper	paper
7	scissors	scissors
8	rock	rock
9	scissors	scissors
10	rock	rock
11	scissors	scissors
12	paper	paper
13	scissors	scissors
14	paper	paper
15	rock	rock
16	scissors	scissors
17	scissors	scissors
18	paper	paper
19	paper	paper
20	scissors	scissors
21	paper	paper
22	rock	rock
23	rock	rock
24	scissors	scissors
25	paper	paper
26	rock	rock
27	scissors	scissors
28	scissors	scissors
29	scissors	scissors
30	rock	rock

Išvados

Laboratorinio darbo metu buvo įgyvendinta konvoliucinio neuroninio tinklo struktūra, atlikti tyrimai su skirtingomis architektūromis ir hiperparametrais. Buvo gautos pagrindinės išvados:

- Analizuojant skirtingų architektūrų poveikį modelio tikslumui, nustatyta, kad gilus modelis pasiekė didžiausią 0,988 tikslumą, tuo tarpu sekli architektūra pasižymėjo 0,371. Galima to priežastis, kad sekli architektūra turi mažiau filtrų ir geba atskirti tik pagrindinius bruožus.
- Geriausiai tinkantis išmetimo sluoksnių ir tikimybės derinys yra 2 sluoksniai su 0,3 išmetimo tikimybe. Didesnė išmetimo tikimybė, tokia kaip 0,5, mažina modelio stabilumą, apmokymo metu pašalinama per daug naudingos informacijos.
- Normalizavimas turėjo minimalų poveikį, todėl galima daryti išvadą, kad duomenys buvo tvarkingi, nuotraukos didelio kontrasto ir gero kokybės.
- Aktyvacijos funkcijų palyginimas parodė, kad labiausiai tinkančios yra ReLu ir nesandarus ReLU. Jos išlaiko aukštą tikslumą, mažindamos nykstančio gradiento riziką.
- Optimizavimo algoritmai pasižymėjo panašiais rezultatais. Modelio tikslumas buvo geriausias naudojant Adam ir SGD optimizavimo algoritmus.
- Išrinktas geriausias modelis pasiekė maksimalų tikslumą ir 0,005 paklaidą, todėl puikiai tinka šių duomenų klasifikavimui. Tačiau įžvelgiama modelio prisitaikymo prie duomenų rinkinio problema.

Apibendrinant, galima teigti, kad visi hiperparametrai yra labai svarbūs ir gali stipriai paveikti neuroninio tinklo klasifikavimo tikslumą. Verta pabandyti šiuos testus įgyvendinti su didesne duomenų aibę, kuri turėtų įvairesnio tipo nuotraukas.

Literatūra ir šaltiniai

[Bru] J. de la Bruère-Terreault. *Rock-Paper-Scissors Images*. URL: https://www.kaggle.com/datasets/drgfreeman/rockpaperscissors/data (žiūrėta 2025-10-19).

[Ope25] OpenAI. *ChatGPT*. Buvo naudotas architektūrų pasirinkimo įgyvendinimui, Optuna optimizacijos įrankiui įgyvendinti, apmokymo proceso priderinimui darbui su tinklais, nuotraukų etikečių priskirimui. 2025. URL: https://www.openai.com/ (žiūrėta 2025-10-27).

[Pre25] I. Prevelyte. *Konvoliuvinių neuroninių tinklų programos kodas*. Laboratorinio darbo Python kodas. 2025. URL: https://colab.research.google.com/drive/1bLUikIghIsEQhmH82WeFq4qMYH4VDYvs?usp=sharing.