VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ STUDIJŲ PROGRAMA

Vaizdų klasifikavimas naudojant konvoliucinius neuroninius tinklus

Laboratorinis darbas

Atliko: 4 kurso 1 grupės studentas
Paulius Minajevas (SN: 2110599)

Vilnius – 2024

Turinys

Įvadas

Užduoties tikslas – apmokyti konvoliucinį neuroninį tinklą vaizdams klasifikuoti, atlikti tyrimą.

1. Klasifikavimo duomenys ir metodika

1.1. Klasifikavimo duomenys

Pasirinkta viena iš nurodytų duomenų aibių – **Keksiukų ir šuniukų vaizdai** (`https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification/data`)

Šis duomenų rinkinys susideda iš dviejų klasių: **keksiukų ir šuniukų (čihuahua)**. Duomenys susideda iš realaus pasaulio nuotraukų, todėl vaizdai gali būti nevienodai kokybiški, su įvairiais fono elementais, kurie gali turėti įtakos klasifikavimo tikslumui.

Iš duomenų rinkinio išrinkta **1000** vaizdų, po 500 iš kiekvienos klasės, kurie vėliau padalinti į mokymo, validavimo ir testavimo rinkinius atitinkamai santykiu **80:10:10**.

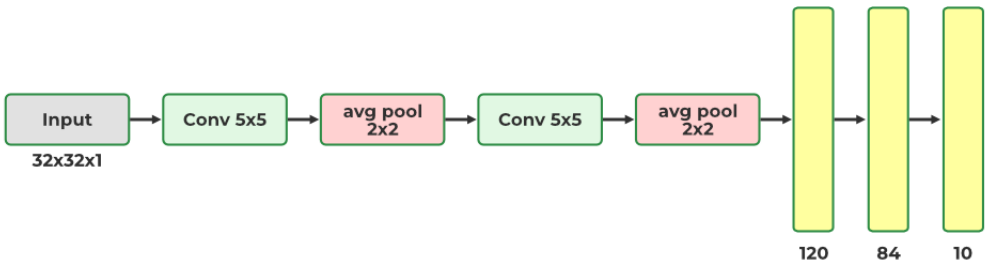
Kadangi duomenų rinkinys susideda iš dviejų labai panašių klasių (keksiukai ir čihuahua), tai suteikia išsūkį modelio mokymui, nes reikia gerai atskirti smulkius vaizdo elementus. Šis duomenų rinkinys yra idealus eksperimentuoti su konvoliucinio neuroninio tinklo gebėjimu išmokti sudėtingus vizualinius bruožus ir efektyviai juos klasifikuoti.

1.2. Metodika

1.2.1. Modelių architektūros

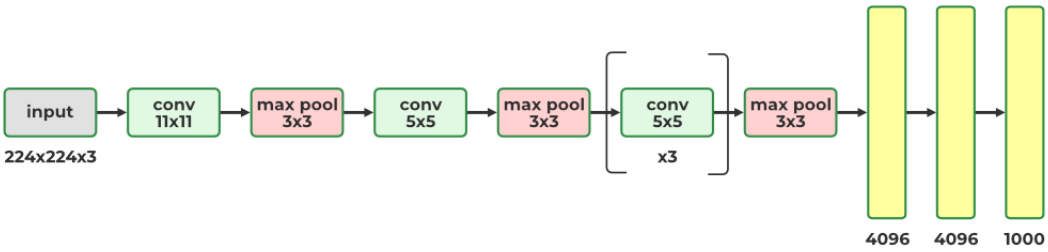
Šiame laboratoriniame darbe bus konstruojami trijų skirtingų architektūrų konvoliuciniai neuroniniai tinklai:

1. **LeNet-5**



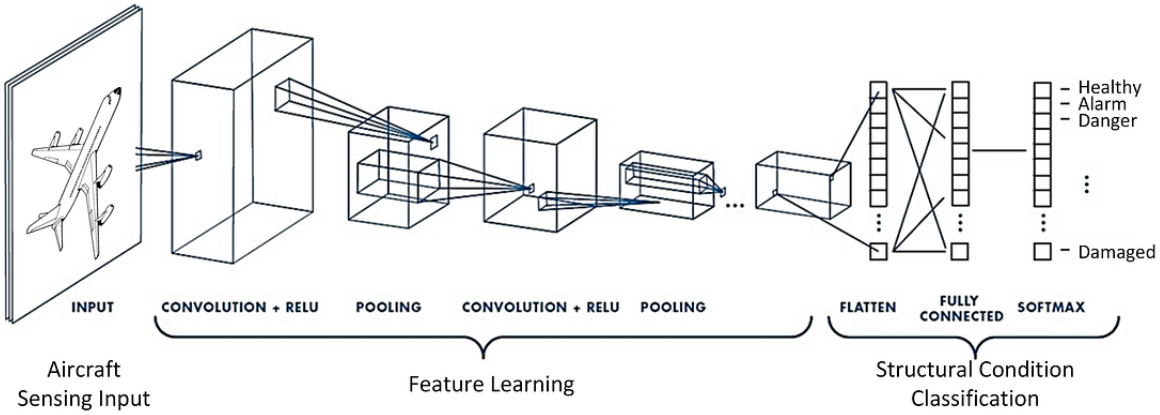
1 pav. LeNet-5 architektūra

2. **AlexNNet**



2 pav. AlexNNet architektūra

3. **SimpleCNN**



3 pav. SimpleCNN architektūra

1.2.2. Duomenų paruošimas

Kadangi duomenys susideda iš nuotraukų, tai, kad su šiais duomenimis būtų galima dirbti ant modelių juos reikia paversti į tensorius. Duomenų rinkinyje nuotraukos yra spalvotos, tai jos susideda iš trijų kanalų (RGB), taigi galutinės tensoriaus dimensijos atrodo taip: (3, x, y). Čia x – priklausomai nuo modelio nuotraukos plotis, y – priklausomai nuo modelio nuotraukos ilgis.

1.2.3. Naudoti skaičiavimo resursai

Konvoliuciniams neuroniniams tinklams apmokyti reikalingas didesnis resursų skaičius negu prieš tai atliktuose darbuose naudoti duomenys. Reikalingas resursų kiekis priklauso ir nuo naudojamo modelio, kadangi šie modeliai pakankamai maži ir duomenų skaičius yra tik 1000 nuotraukų, tai kaip resursų šaltinis panaudotas debesijos sprendimas – „Google Colab". Šis sprendimas suteikė nemokamą prieigą dirbti su „Nvidia T4" vaizdo plokšte ir ant jos atlikti reikalingus skaičiavimus.

2. Programinis įgyvendinimas

Užduotis atlikta naudojantis „Python" kalba. Tolesni poskyriai išskirti pagal esmines kodo realizacijos grupes.

2.1. LeNet-5 realizacija

```
class LeNet5(nn.Module):
    def __init__(self, in_shape, num_classes):
        super().__init__()
        self.num_classes = num_classes

        self.conv1 = nn.Conv2d(in_channels=in_shape, out_channels
            =6, kernel_size=5, stride=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16,
            kernel_size=5, stride=1)
        self.fc1 = nn.Linear(in_features=16 * 5 * 5, out_features
            =120)
        self.fc2 = nn.Linear(in_features=120, out_features=84)
        self.fc3 = nn.Linear(in_features=84, out_features=
            num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x
```

LeNet5 modelio architektūra ir realizacija su PyTorch biblioteka paimta iš – `https://www.geeksforgeeks.org/convolutional-neural-network-cnn-architectures/`. Ši klasė aprašo sekvencinio tipo modelį LeNet5.

2.2. AlexNet realizacija

```
class AlexNet(nn.Module):
    def __init__(self, in_shape, num_classes):
        super().__init__()
        self.num_classes = num_classes

        self.conv1 = nn.Conv2d(in_channels=in_shape, out_channels
            =96, kernel_size=11, stride=4, padding=2)
        self.pool = nn.MaxPool2d(kernel_size=3, stride=2)
        self.conv2 = nn.Conv2d(in_channels=96, out_channels=256,
            kernel_size=5, padding=2)
        self.conv3 = nn.Conv2d(in_channels=256, out_channels=384,
             kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(in_channels=384, out_channels=384,
             kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(in_channels=384, out_channels=256,
             kernel_size=3, padding=1)
        self.fc1 = nn.Linear(in_features=256 * 6 * 6,
            out_features=4096)
        self.fc2 = nn.Linear(in_features=4096, out_features=4096)
        self.fc3 = nn.Linear(in_features=4096, out_features=
            num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(F.relu(self.conv5(x)))
        x = x.view(-1, 256 * 6 * 6)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, 0.5)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x
```

AlexNet modelio architektūra ir realizacija su PyTorch biblioteka paimta iš – `https://www.geeksforgeeks.org/convolutional-neural-network-cnn-architectures/`. Ši klasė aprašo sekvencinio tipo modelį AlexNet.

2.3. SimpleCNN realizacija

```
class CustomCNN(nn.Module):
    def __init__(self, in_shape, num_classes):
        super().__init__()
        self.num_classes = num_classes

        self.conv1 = nn.Conv2d(in_channels=in_shape, out_channels
            =16, kernel_size=3, stride=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
            kernel_size=3, stride=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64,
            kernel_size=3, stride=1)
        self.flatten = nn.Flatten()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(in_features=(64 * 17 * 17),
            out_features=512)
        self.fc2 = nn.Linear(in_features=512, out_features=
            num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

SimpleCNN modelio architektūra ir realizacija su Tensorflow biblioteka paimta iš – `https://medium.com/@chenycy/a-simple-convolutional-neural-network-cnn-classifier-based-on-`
Rastas modelis buvo perrašytas su PyTorch biblioteka. Ši klasė aprašo sekvencinio tipo modelį SimpleCNN.

2.4. Vienos epochos treniravimo realizacija

```
def train_epoch(optimizer, loss_func, model, loader):
    classes = model.num_classes
    model.train()
    loss_acum = np.array([], dtype = np.float32)

    for images, labels in loader:
      optimizer.zero_grad()
      images = images.to(device)
      labels = torch.nn.functional.one_hot(labels, classes).float
          ().to(device)

      pred = model(images)
      loss = loss_func(pred, labels)
      loss_acum = np.append(loss_acum, loss.cpu().detach().numpy
          ())

      loss.backward()
      optimizer.step()

    return np.mean(loss_acum)
```

Šis metodas „train_epoch” atlieka vieną mokymo epochą neuroniniam tinklui. Jis naudoja nurodytą optimizatorių ir nuostolių funkciją, kad atnaujintų modelio parametrus, apdorodamas kiekvieną mini paketo duomenų rinkinį iš pateikto duomenų kroviklio (loader). Galiausiai jis grąžina vidutinę nuostolio (loss) vertę visai epochai.

2.5. Vienos epochos validavimo realizacija

```
def evaluate(model, loader):
  model.eval()

  correct_predictions = 0
  total_predictions = 0

  labels_acum = []
  predictions_acum = []

  for images, labels in loader:
    images = images.to(device)
    labels = labels.to(device)

    with torch.no_grad():
      pred = torch.sigmoid(model(images))

    label_pred = torch.argmax(pred, axis = 1)

    labels_acum = np.append(labels_acum, labels.cpu().detach().
       numpy())
    predictions_acum = np.append(predictions_acum, label_pred.cpu
       ().detach().numpy())

    correct_predictions += torch.sum(labels == label_pred)
    total_predictions += images.shape[0]

  accuracy = correct_predictions / total_predictions

  return accuracy, labels_acum, predictions_acum
```

Šis metodas „evaluate" įvertina modelio našumą, naudodamas pateiktą duomenų kroviklį (loader). Jis nustato modelį į vertinimo rėžimą (eval()), kad išjungtų gradientų skaičiavimą ir taip nenaujintų svorių tikrinimo metu. Kiekvienam mini paketui apskaičiuojamos prognozės ir lyginamos su tikromis etiketėmis, kad būtų galima įvertinti bendrą tikslumą. Galiausiai metodas grąžina modelio tikslumą, tikrąsias etiketes ir modelio prognozes visiems pateiktiems duomenims.

2.6. Modelio treniravimo algoritmas

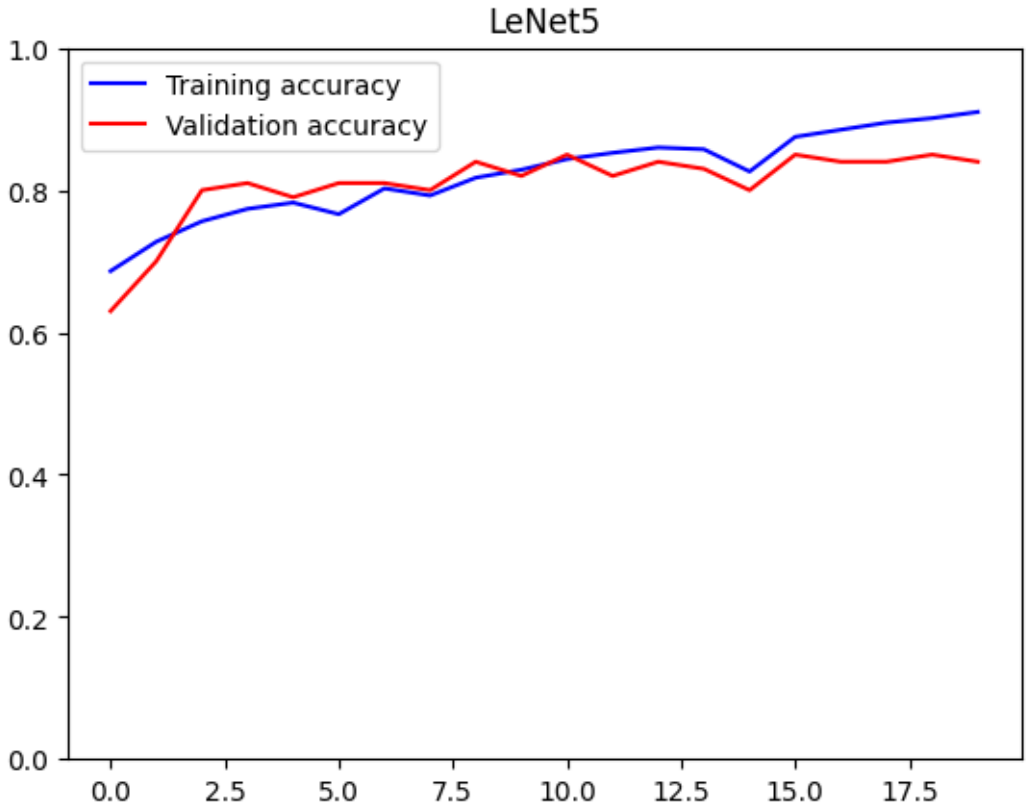
```
def train_and_eval(model, loader_train, loader_valid, epoch_count
    , lr):
  loss_func = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr = lr)

  start_time = datetime.now()

  train_accuracy_acum = []
  test_accuracy_acum = []
  for epoch in range(epoch_count):
    loss = train_epoch(optimizer, loss_func, model, loader_train)

    train_accuracy, _, _ = evaluate(model, loader_train)
    train_accuracy_acum.append(train_accuracy.cpu().numpy())
    test_accuracy, _, _ = evaluate(model, loader_valid)
    test_accuracy_acum.append(test_accuracy.cpu().numpy())

    elapsed = timedelta(seconds=((datetime.now() - start_time).
       total_seconds()))
    print(f'Epoch: {epoch}, Time: {elapsed}, Training loss: {loss
       }')
    print(f'Training accuracy: {torch.round(train_accuracy * 100)
       }, Validation accuracy: {torch.round(test_accuracy * 100)
       }')

  _, labels, predictions = evaluate(model, loader_valid)

  return train_accuracy_acum, test_accuracy_acum, labels,
     predictions
```

Metodas „train_and_eval” atlieka modelio mokymą ir vertinimą per nurodytą epochų skaičių. Jis naudoja konvoliucinį tinklą, CrossEntropyLoss nuostolių funkciją ir Adam optimizatorių. Kiekvienai epochai modelis mokomas naudojant train_epoch metodą, po to įvertinamas mokymo (loader_train) ir validavimo (loader_valid) rinkinių tikslumas. Rezultatai išvedami konsolėje kiekvienos epochos pabaigoje, nurodant mokymo nuostolį, tikslumą ir praėjusį laiką. Galiausiai metodas grąžina sukauptas tikslumo reikšmes ir validavimo rinkinio etiketes bei prognozes.

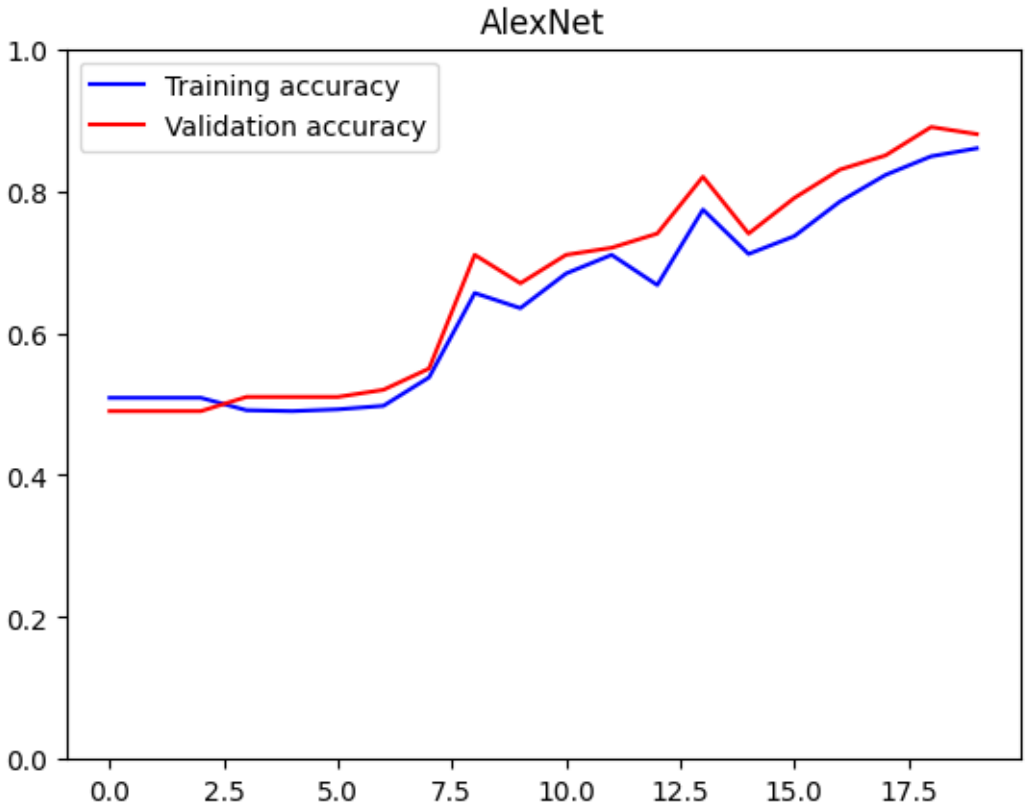
3. Tyrimas

3.1. Architektūrų palyginimas

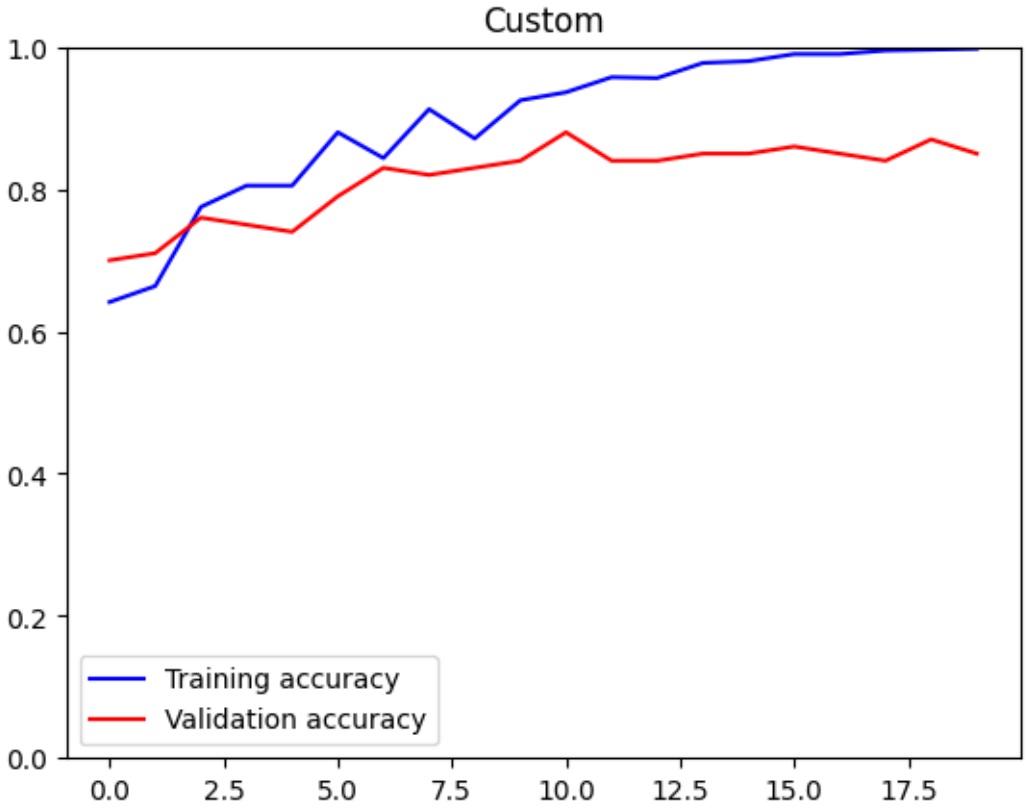
Kad atrinkti tinkamiausią architektūrą buvo apmokytos visos prieš tai aprašytos architektūros ir gautos metrikos po apmokymų palygintos. Visi trys modeliai buvo apmokyti naudojant Sigmoid aktyvacijos funkciją, CrossEntropyLoss nuostolio funkciją, Adam optimizatorių, 20 epochų bei 0,001 mokymo žingsnį. Žemiau paveikslėliuose pateikiami mokymo ir validavimo rezultatai kiekvienai architektūrai.



4 pav. Lenet-5 mokymo rezultatai



5 pav. AlexNNet mokymo rezultatai



6 pav. CustomCNN mokymo rezultatai

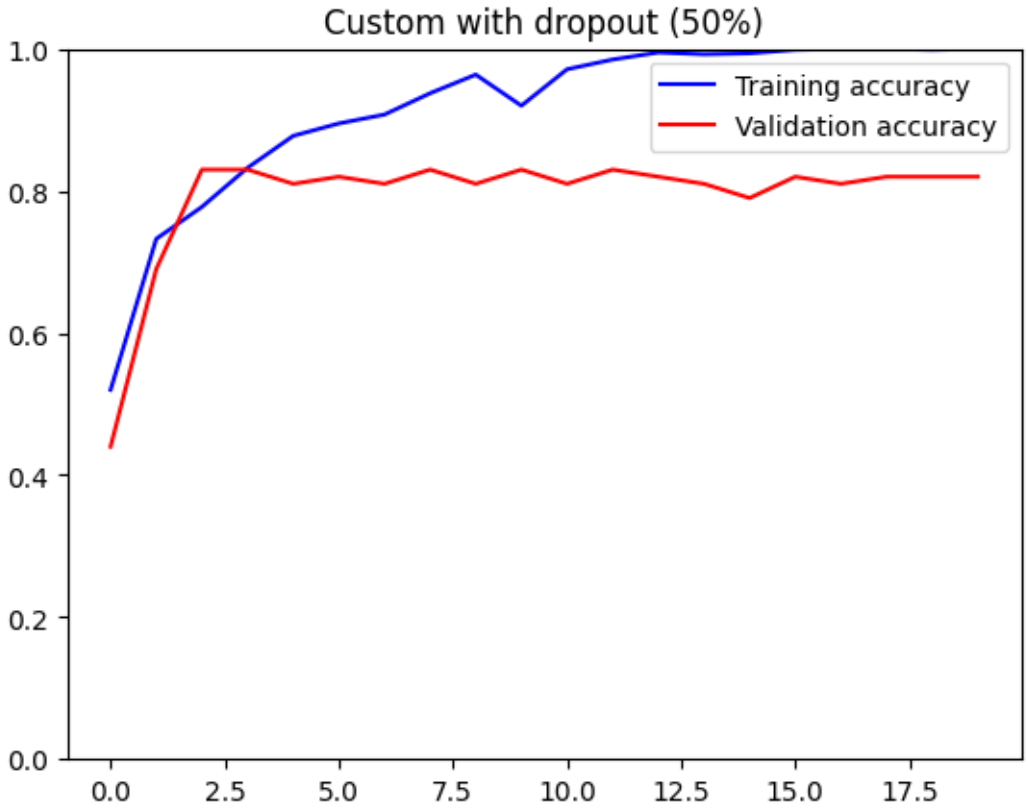
Pagal gautus grafikus matome, kad CustomCNN pasiekė aukščiausią tikslumą 100% mokymo

duomenims tačiau AlexNNET pasiekė aukščiausią tikslumą 88% validavimo duomenims, tolimesniems tyrimams naudosime CustomCNN, kad sumažintume per didelį prisitaikymą mokymo duomenims ir tuo pačiu pamatysime ar standartinės praktikos tokioje situacijoje padeda.

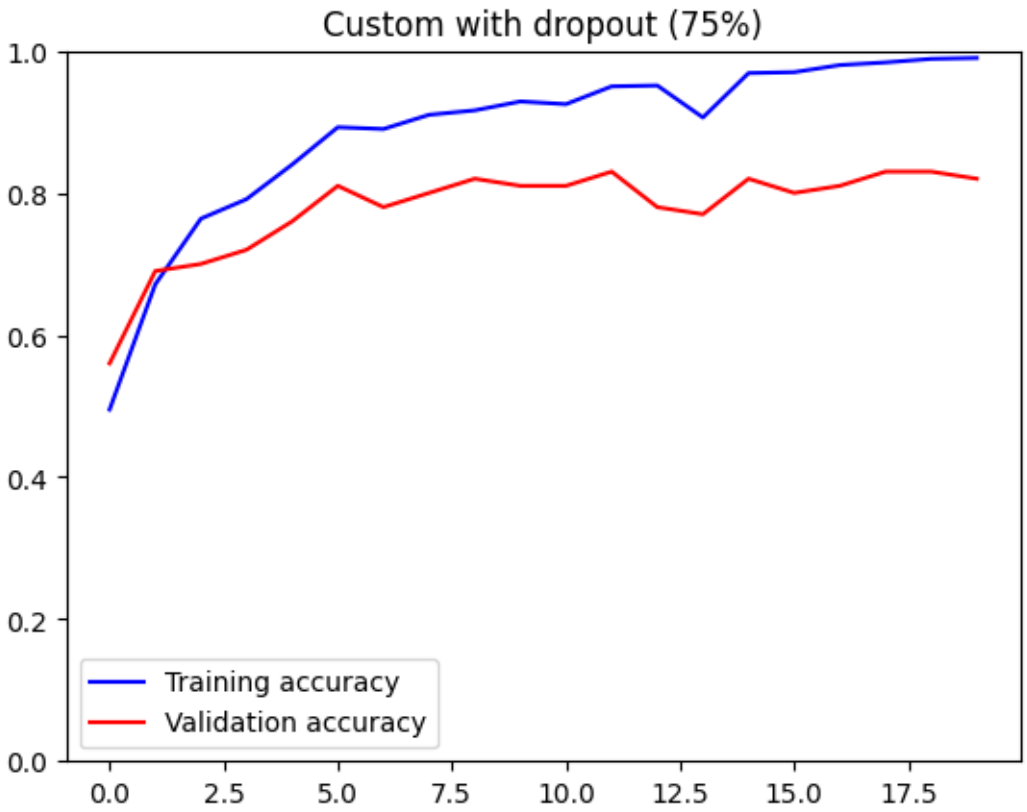
3.2. Hiperparametrų palyginimas

3.2.1. Išmėtimo sluoksniai

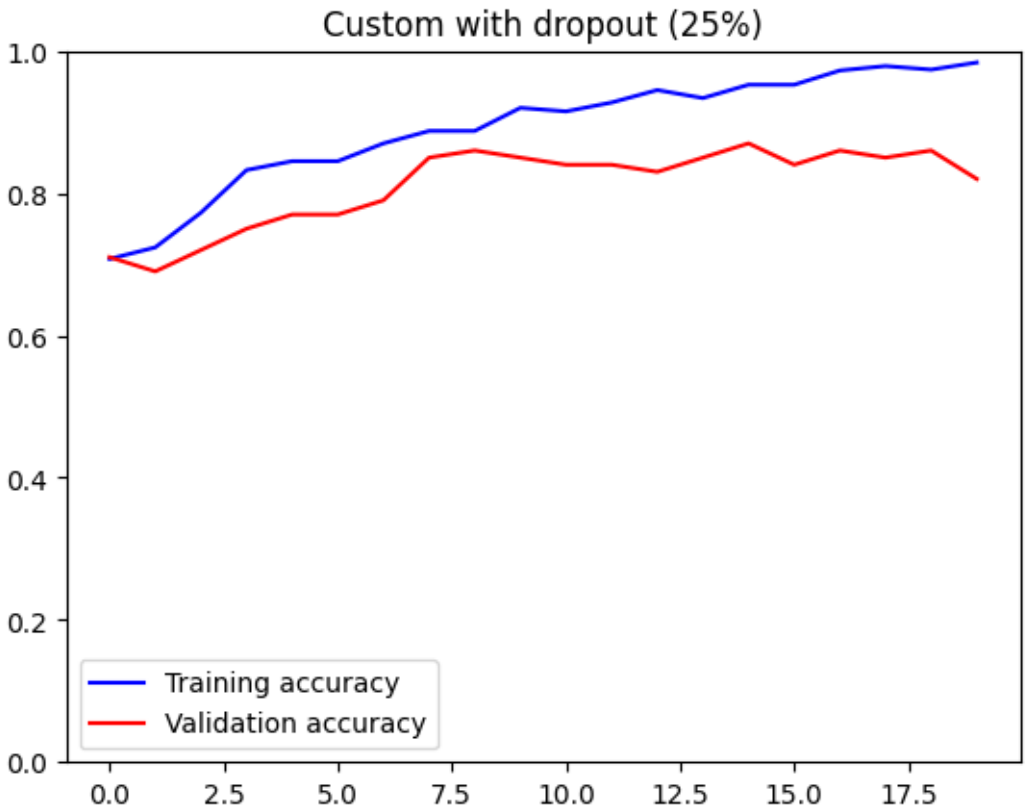
Išmėtimo sluoksniai (angl. Dropout layers) naudojami neuroniniuose tinkluose siekiant sumažinti per dideli prisitaikymą prie mokymo duomenų (angl. overfitting). Jie atsitiktinai išjungia tam tikrą dalį neuronų kiekvieno mokymo etapo metu, todėl tinklas tampa mažiau priklausomas nuo konkrečių neuronų ir išmoksta labiau bendrinti informaciją. Tai padeda padidinti modelio gebėjimą gerai veikti su naujais, nematytais duomenimis. Atliktas tyrimas ar tai pagerina šiuo atveju tikslumą ar ne. Ši metodika pritaikyta geriausiai pasirodžiusiam modeliui praeitame poskyryje, žemiau pateikiami mokymo rezultatai.



7 pav. CustomCNN su išmėtimo sluoksniu (50%) mokymo rezultatai



8 pav. CustomCNN su išmėtimo sluoksniu (75%) mokymo rezultatai

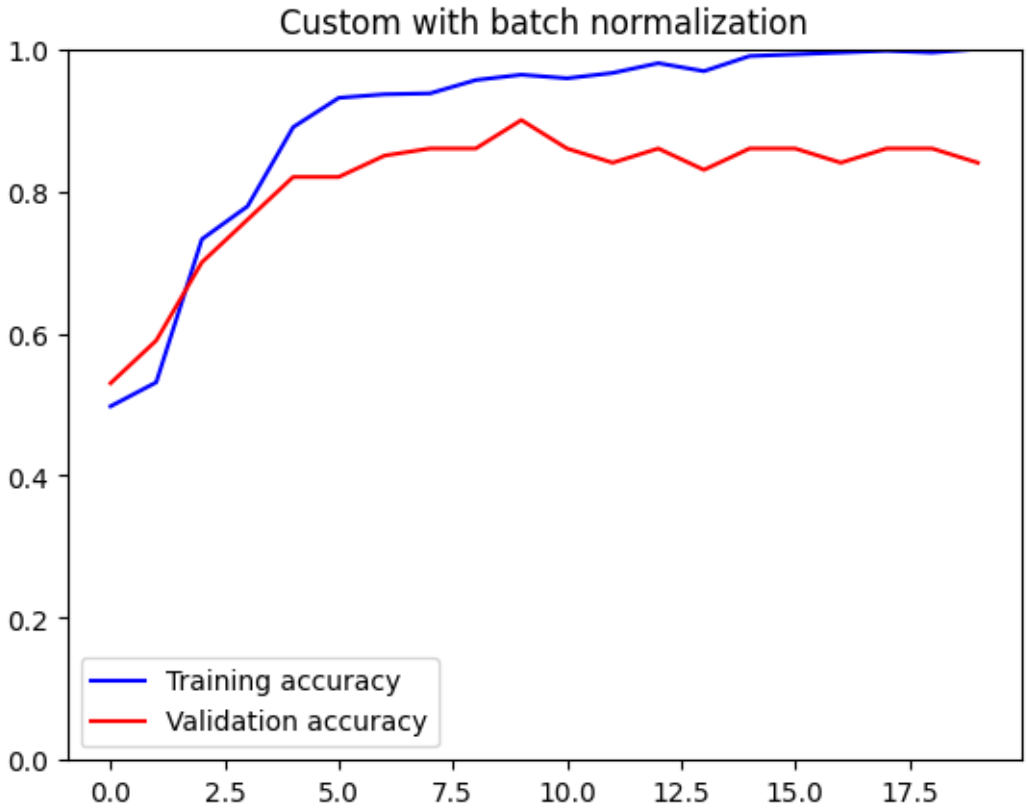


9 pav. CustomCNN su išmėtimo sluoksniu (25%) mokymo rezultatai

Deja, bet nei vienas testas negavo geresnių rezultatų negu naudota pradinė architektūra, todėl papildomas išmėtimo sluoksnis nebus šiuo atveju naudojamas.

3.2.2. Paketų normalizavimas

Paketų normalizavimas (angl. Batch Normalization) yra technika, naudojama neuroniniuose tinkluose siekiant pagreitinti mokymo procesą ir stabilizuoti modelio mokymą. Ji normalizuoja kiekvieno sluoksnio išvestis atsižvelgiant į mini paketo statistiką (vidurkį ir standartinį nuokrypį), kad sumažintų kintančių įvesties skirstinių problemą (angl. internal covariate shift). Tai leidžia efektyviau mokyti modelį ir padeda sumažinti drastiškus svorio pakeitimus dėl išskirtinių duomenų. Atliktas tyrimas ar tai pagerina šiuo atveju tikslumą ar ne. Ši metodika pritaikyta geriausiai pasirodžiusiam modeliui praeitame poskyryje, žemiau pateikiami mokymo rezultatai.



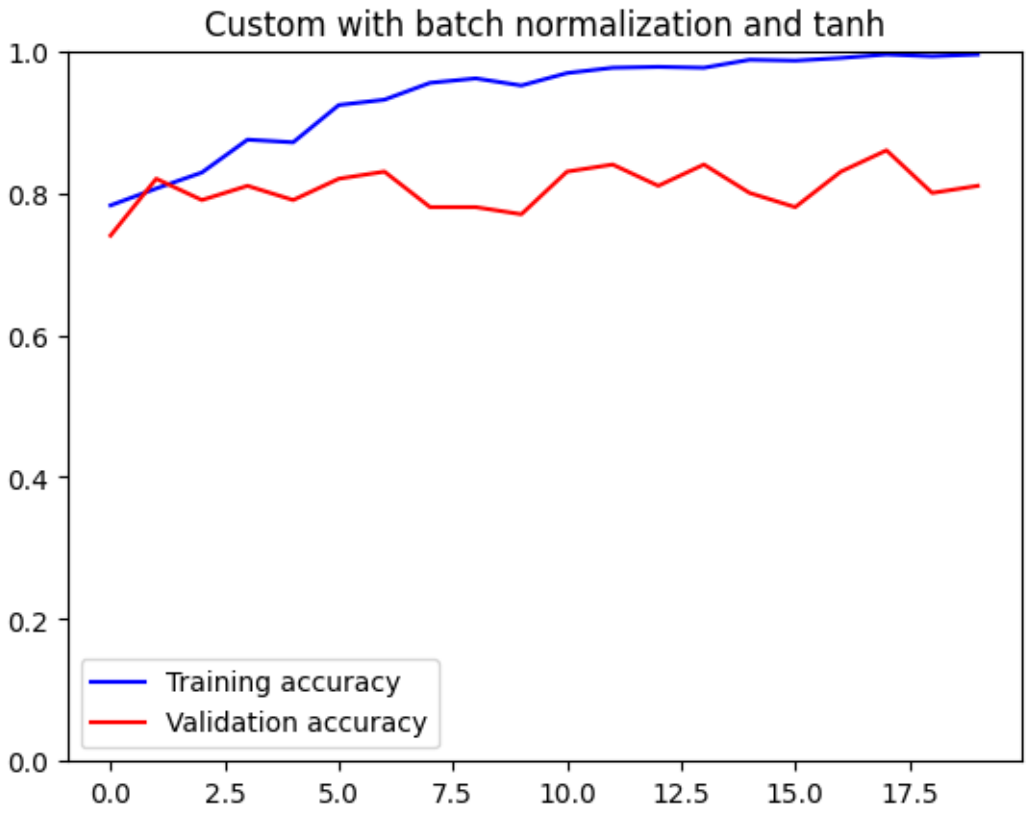
10 pav. CustomCNN su išmėtimo sluoksniu mokymo rezultatai

Mokymosi eigoje modelis pasiekė 90% tikslumą validavimo duomenims ir 96% mokymo duomenims, taigi paketų normalizavimas padėjo padidinti modelio tikslumą ir ši architektūra bus naudojama tolimesniems tyrimams.

3.2.3. Aktyvacijos funkcija

Aktyvacijos funkcija neuroniniame tinkle lemia, kaip kiekvieno neurono įvesties reikšmės paverčiamos jo išvestimi. Ji prideda nelinearišKumą, leidžiantį tinklui išmokti sudėtingus duomenų ryšius. Populiarios aktyvacijos funkcijos yra ReLU (tiesinė vieneto funkcija), kuri dažnai naudojama dėl efektyvumo ir paprastumo, sigmoid, kuri tinka dvejetainiams rezultatams, ir tanh, kuri

normalizuoja išvestį į intervalą nuo -1 iki 1. Atliktas tyrimas ar aktyvacijos pakeitimas iš ReLU į tanh tarp sluoksnių pagerina šiuo atveju tikslumą ar ne. Ši metodika pritaikyta geriausiai pasirodžiusiam modeliui praeitame poskyryje, žemiau pateikiami mokymo rezultatai.

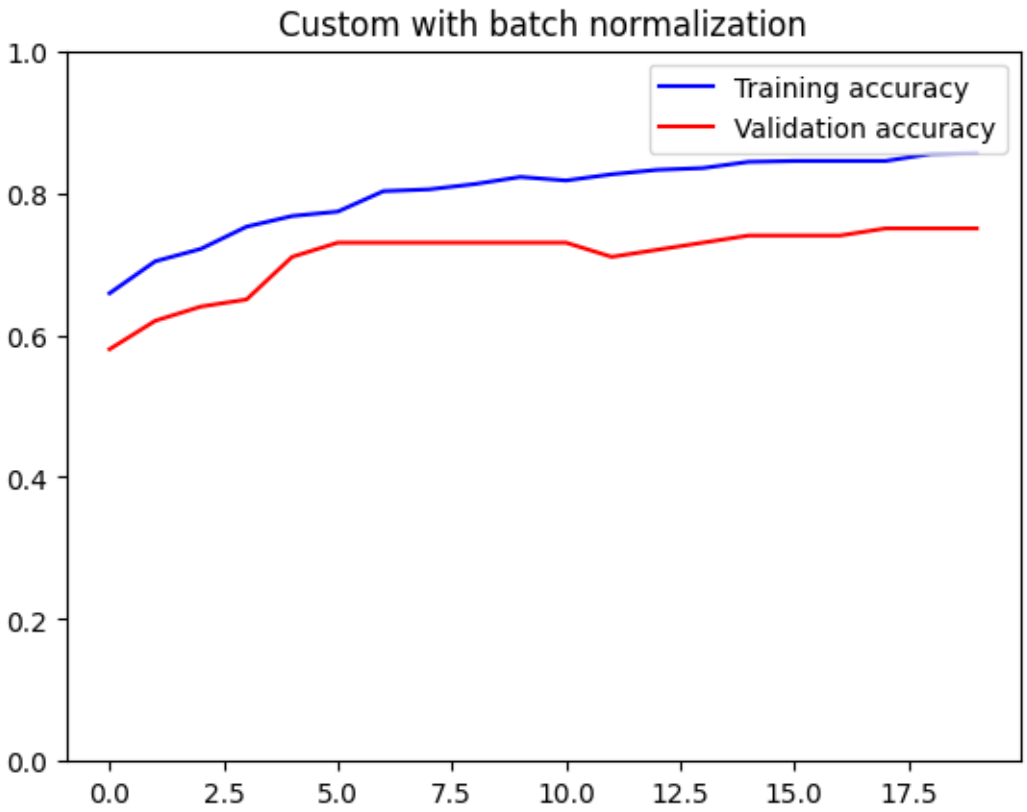


11 pav. CustomCNN su išmėtimo sluoksniu ir tanh aktyvacija mokymo rezultatai

Didesnių pagerėjimų nepastebėta, todėl aktyvacijos funkcija nekeičiama.

3.2.4. Optimizavimo algoritmas

Optimizacijos algoritmas yra naudojamas neuroninių tinklų svorių atnaujinimui taip, kad sumažėtų modelio nuostoliai ir būtų pasiekti geresni rezultatai. Populiarūs optimizacijos algoritmai – Stochastic Gradient Descent (SGD), kuris atnaujina parametrus pagal gradientus, ir Adam, kuris yra dažnai naudojamas dėl greitesnio konvergavimo bei gebėjimo pritaikyti mokymosi greitį kiekvienam parametrui atskirai. Atliktas tyrimas ar optimizacijos algoritmo pakeitimas iš Adam į SGD pagerina šiuo atveju tikslumą ar ne. Ši metodika pritaikyta geriausiai pasirodžiusiam modeliui praeitame poskyryje, žemiau pateikiami mokymo rezultatai.



12 pav. CustomCNN su išmėtimo sluoksniu ir SGD optimizavimu mokymo rezultatai

Didesnių pagerėjimų nepastebėta, todėl optimizavimo algoritmas nekeičiamas.

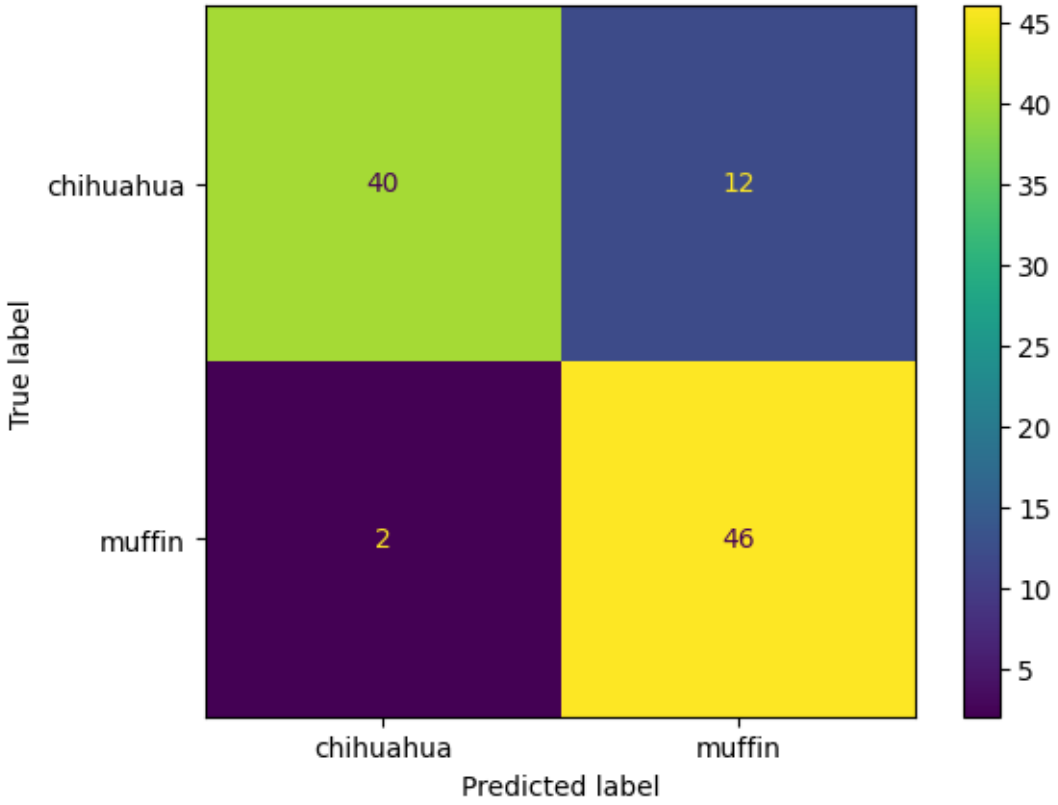
3.3. Tyrimo rezultatai

Nustatyta, kad tinkamiausia architektūra iš trijų pradinių yra CustomCNN, papildomai pritaikius paketinį normalizavimą, žemiau poskyriuose pateikiami gauti rezultatai.

3.3.1. Testavimo metrikos

Klasifikavimo tikslumas testiniams duomenims – 86%.

Klasifikavimo matrica:



13 pav. Klasifikavimo matrica

3.3.2. Testavimo duomenų pavyzdžiai

1 lentelė. Tikrosios ir modelio spėtos klasės

Tikrojį klasė	Spėta klasė
1	1
0	1
1	1
1	1
0	1
0	1
0	0
0	0
0	0
0	0
1	1
0	0
1	1
0	0
0	0
1	1
0	0
1	1
0	0
0	1
1	1
1	1
1	1
0	0
0	0
0	0
1	1
0	0
1	1
1	1

Išvados

1. Atliekant tyrimus su trimis tinklo architektūromis (LeNet-5, AlexNet ir CustomCNN), nustatyta, kad geriausi rezultatai buvo pasiekti naudojant CustomCNN su paketiniu normalizavimu. Ši architektūra užtikrino 90% tikslumą validavimo duomenims ir 96% mokymo duomenims.
2. Išmėtimo sluoksniai (dropout) nesuteikė geresnių rezultatų ir buvo pašalinti.
3. Paketų normalizavimas pagerino modelio tikslumą.
4. Aktyvacijos funkcijos pakeitimas iš ReLU į tanh nepagerino tikslumo.
5. Optimizavimo algoritmas Adam buvo efektyvesnis lyginant su SGD.
6. Testavimo duomenų tikslumas pasiekė 86%, kas parodo modelio gebėjimą gerai klasifikuoti naujus duomenis.
7. Testavimo metu modelis užtikrino aukštą klasifikavimo kokybę, tačiau pastebėtas tam tikrų klaidų skaičius tarp vizualiai panašių klasių.
8. Tinkamai pritaikytos praktikos, tokios kaip paketinė normalizacija, gali pagerinti klasifikavimo tikslumą nematytiems duomenims