VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Skaitmeninis intelektas ir sprendimų priėmimas

IV užduotis: Vaizdų klasifikavimas naudojant konvoliucinius neuroninius tinklus

Tomas Giedraitis
VU MIF Informatika
3 kursas 3 grupė

Vilnius
2021

1. Darbo tikslas

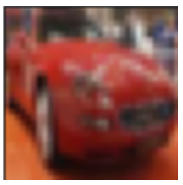
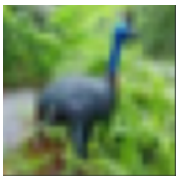
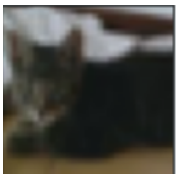
Konvoliucinio neuroninio tinklo vaizdams klasifikuoti apmokymas, bei tyrimo atlikmas. Tyrimą sudaro analizė, kaip neuroninio tinklo rezultatai priklauso nuo tinklo architektūros ir nuo hyperparametrų reikšmių. Rezultatai vertinami klasifikavimo tikslumo mato prasme testavimo duomenims.

2. Darbo eiga

2.1. Duomenys

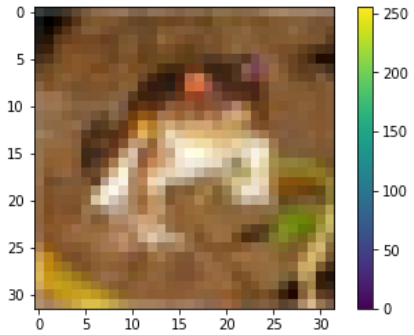
Pasirinkti CIFAR-10 vaizdų su žinomomis klasėmis duomenys[1]. Juos sudaro 60 000 spalvotų paveiksliukų, priklausančių vienai iš 10-ies klasių (po 6 000 paveiksliukų vienai klasei). Klasės vaizduoja gana skirtingus objektus, kitaip tariant, mažai persipina tarpusavyje (1 lentelė).

1 lentelė. CIFAR-10 paveiksliukų klasės

Klasės numeris	Klasės pavadinimas	Pavyzdys	Klasės numeris	Klasės pavadinimas	Pavyzdys
0	Lėktuvas		5	Šuo	
1	Automobilis		6	Varlė	
2	Paukštis		7	Arklys	
3	Katė		8	Laivas	
4	Elnias		9	Sunkvežimis	

[1] https://www.cs.toronto.edu/~kriz/cifar.html

Šiuos paveiksliukus sudaro 32×32 dydžio pikselių tinklelis, kur kiekvieno pikselio spalvinė reikšmė kinta nuo 0 iki 255. 1 pav. matome pavyzdinio paveiksliuko pikselių reikšmių diagramą (heatmap).



1 pav. Paveiksliuko pikselių reikšmių diagrama

2.1.2. Duomenų šaltinis

Duomenys buvo paimti iš *Keras* modulio:

```
(train_images,train_labels), (test_images, test_labels) =
tensorflow.keras.datasets.cifar10.load_data()
```

Šiuo kreipiniu jau gauname tvarkingus mokymo ir testavimo paveiksliukų masyvus su jų informacija (pikselių reikšmėmis) bei juos atitinkančius žymių sąrašus (t.y. jų tikrųjų klasių sąrašus).

2.1.3. Duomenų padalijimas

Standartiškai CIFAR-10 duomenys padalinti į 50 000 dydžio mokymo aibę ir 10 000 dydžio testavimo aibę. Šis padalijimo santykis ir buvo naudotas atliekant tyrimą. Tačiau programoje taip pat yra galimybė sudėti visus duomenis į vieną aibę, permaišyti ją atsitiktinai, jeigu norime to, ir tuomet laisvai pasirinkti bet kokį norimą padalijimo santykį.

2.1.4. Duomenų paruošimas

Prieš apmokant neuroninį tinklą, duomenys buvo normalizuojami, kad pikselių reikšmės – sveiki skaičiai iš intervalo [0; 255] – būtų transformuotos į realius skaičius iš intervalo [0; 1]. Tam dalinome reikšmes iš 255:

```
train_images, test_images = train_images / 255.0, test_images / 255.0
```

Svarbu, kad mokymo ir testavimo duomenys būtų apdorotos tokiu pat būdu.

2.2. Neuroninio tinklo architektūra

Šiame darbe buvo naudojamas konvoliucinis neuroninis tinklas (KNT). Iš programavimo perspektyvos jis buvo gana nesudėtingas, buvo naudojama *TensorFlow* platforma kartu su *Keras* moduliu – aukšto lygio aplikacijų programavimo sąsaja neuroninio tinklo modeliui sukurti ir apmokyti.

KNT architektūros kūrimas susideda iš neuroninio tinklo modelio sluoksnių (lygmenų) sukūrimo ir konfigūracijos, bei modelio kompiliavimo, kuomet papildomi jo parametrai yra nustatomi.

2.2.1. Modelio sukūrimas ir konfigūracija

KNT sluoksniai – tai pagrindinės sudedamosios KNT dalys. Jie išgauna duomenų reprezentaciją iš jiems paduodamų duomenų, ir yra tikimasi, kad tos reprezentacijos yra prasmingos sprendžiant esamą uždavinį. Sluoksniai sujungiami tarpusavyje į grandinę, ir dauguma sluoksnių turi parametrus, kurie yra nustatomi (arba keičiami, t.y. derinami) tinklo mokymo metu.

Modelis buvo kuriamas naudojant gana paprastą *Keras Sequential API* interfeisą, tam nereikėjo sudėtingesnio objektinio programavimo ir vaikinių šios *API* klasių įgyvendinimo, užteko paprastesnių procedūrinių operacijų.

Modelio inicijavimas:
```
model = tf.keras.Sequential()
```

Buvo nagrinėtos kelios neuroninio tinklo architektūros, tačiau pagrindinė architektūra, kuriai esant pastoviai, buvo keičiamos ir tiriamos hyperparametrų reikšmės, yra gana paprasta, susidedanti iš trijų konvoliucijos sluoksnių, ir yra apibrėžta šiuo programiniu kodu:

```
model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
 input_shape=(32, 32, 3)))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(10))
```

Conv2D – konvoliucinis sluoksnis. Jo išreikštinai nurodyti argumentai (pirmasis sluoksnis):

- išvesties filtrų skaičius konvoliucijoje (išvesties erdvės dimensiškumas): 32.
- branduolio, arba konvoliucijos lango, dydis: (3, 3), t.y. 3×3 matmenų kvadratas.
- aktyvacijos funkcija – ReLU („lygintuvo funkcija“).
- įvesties dimensijos: (32, 32, 3), t.y. matmenys – 32×32, kanalo dydis – 3 (specifikuoja RGB duomenis).

MaxPooling2D – sujungimo sluoksnis. Jo specifikuotas argumentas – langa dydis (*pool size*).
Flatten – ištiesinimo sluoksnis.
Dense – pilnai, arba tankiai, sujungti sluoksniai (antrasis (paskutinis) iš jų grąžina 10 elementų (t.y. tiek, kiek yra klasių) masyvą, sudarytą iš logitų – vektorių iš nenormalizuotų tikimybių, kurie po to yra perduodami aktyvacijos funkcijai. Parametrai - *Dense* sluoksnių dimensijos.

Šio modelio konfigūracijos išvestis, iškviečiant *model.summary()* funkciją:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 64)                65600
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

Keičiama architektūra:
Tyrime buvo pratestuotos kelios neuroninio tinklo architektūros, įskaitant aukščiau aprašytą. Vienoje iš jų dar prisidėjo *Dropout* ir *BatchNormalization* sluoksniai, kurie detaliau aptarti eksperimentinėje dalyje.

Keičiami hyperparametrai
Šioje aprašytoje modelio sukūrimo ir konfigūracijos dalyje eksperimento metu buvo keičiama aktyvacijos funkcija. Skirtingos testuotos aktyvacijos funkcijos aprašytos prie eksperimento.

2.2.2. Modelio kompiliavimas

Modelio buvo sukompiliuotas šioje programinio kodo dayje:

```
model.compile(
      optimizer='adam',
      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      metrics=['accuracy'])
```

Čia *optimizer* – optimizavimo funkcija, kuria remiantis modelis yra koreguojamas realiu laiku, besimokant, pagal gaunamus duomenis ir nuostolių funkciją.
loss – nuostolių funkcija. Buvo naudoja `SparseCategoricalCrossentropy` funkcija, kadangi klasių buvo ne dvi, o 10.
metrics – klasifikavimo matas. Neuroninio tinklo darbo rezultatui matuoti tyrime buvo naudojamas bendras klasifikavimo tikslumas (*accuracy*).

Keičiami hyperparametrai
Modelio kompiliavimo dalyje tyrime buvo naudojami skirtingi optimizavimo algoritmai bei nuostolių funkcija. Plačiau apie visus tirtus variantus – tyrimo rezultatų aprašyme.

2.2.3. Modelio apmokymas

Modelio apmokymą realizuoja šis kodas:

```
history = model.fit(
      train_images,
      train_labels,
      batch_size=32,
      epochs=epoch_count,
      validation_data=(test_images, test_labels))
```

Argumentų aprašymas*:*
batch_size – paketo dydis, kiek duomenų vienos iteracijos metu buvo apdorojama neuroninio tinklo.
epochs – modelio treniravimo epochų skaičius. Viena epocha – kai visi duomenys, padalinti į paketus, praleidžiami pro neuroninį tinklą.
validation_data – tai parametro pavadinimas, kuriuo perduodami testiniai duomenys. Nors parametras vadinasi „validavimo duomenys“, validavimo duomenų atskirai tyrime nebuvo naudota, ir per šį parametrą paduoti testavimo duomenys suteikė galimybę po kiekvienos iteracijos modeliui išspausdinti jo šiuo metu esamą klasifikavimo tikslumą su testavimo duomenimis. Tai buvo panaudota tyrime.

Keičiami hyperparametrai
Galutinėje – modelio apmokymo – dalyje buvo keičiamas paketo dydis bei epochų skaičius.

Pavyzdinė modelio apmokymo išvestis, kartu su klasifikavimo tikslumu mokymo (*accuracy*) ir testavimo duomenims (*val_accuracy*):

```
Epoch 1/20
1407/1407 [==============================] - 9s 4ms/step - loss: 1.7697 - accuracy: 0.3446 - val_loss: 1.2934 - val_accuracy: 0.5382
Epoch 2/20
1407/1407 [==============================] - 6s 4ms/step - loss: 1.2300 - accuracy: 0.5614 - val_loss: 1.1802 - val_accuracy: 0.5828
Epoch 3/20
1407/1407 [==============================] - 6s 4ms/step - loss: 1.0579 - accuracy: 0.6254 - val_loss: 1.0168 - val_accuracy: 0.6402
Epoch 4/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.9364 - accuracy: 0.6719 - val_loss: 0.9734 - val_accuracy: 0.6540
Epoch 5/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.8438 - accuracy: 0.7038 - val_loss: 0.9274 - val_accuracy: 0.6688
Epoch 6/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.7920 - accuracy: 0.7195 - val_loss: 0.8817 - val_accuracy: 0.6888
Epoch 7/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.7419 - accuracy: 0.7383 - val_loss: 0.8764 - val_accuracy: 0.7012
Epoch 8/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.6757 - accuracy: 0.7646 - val_loss: 0.8940 - val_accuracy: 0.7002
Epoch 9/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.6499 - accuracy: 0.7748 - val_loss: 0.8658 - val_accuracy: 0.7010
Epoch 10/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.6045 - accuracy: 0.7855 - val_loss: 0.9099 - val_accuracy: 0.6882
Epoch 11/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.5711 - accuracy: 0.8024 - val_loss: 0.9041 - val_accuracy: 0.7006
Epoch 12/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.5346 - accuracy: 0.8111 - val_loss: 0.8800 - val_accuracy: 0.7052
Epoch 13/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.4888 - accuracy: 0.8274 - val_loss: 0.9669 - val_accuracy: 0.6928
Epoch 14/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.4738 - accuracy: 0.8307 - val_loss: 0.9234 - val_accuracy: 0.7102
Epoch 15/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.4392 - accuracy: 0.8466 - val_loss: 0.9736 - val_accuracy: 0.6974
Epoch 16/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.4098 - accuracy: 0.8552 - val_loss: 0.9979 - val_accuracy: 0.7042
Epoch 17/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.3800 - accuracy: 0.8650 - val_loss: 1.0754 - val_accuracy: 0.7038
Epoch 18/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.3580 - accuracy: 0.8710 - val_loss: 1.0797 - val_accuracy: 0.6942
Epoch 19/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.3331 - accuracy: 0.8805 - val_loss: 1.1744 - val_accuracy: 0.6850
Epoch 20/20
1407/1407 [==============================] - 6s 4ms/step - loss: 0.3109 - accuracy: 0.8887 - val_loss: 1.1478 - val_accuracy: 0.6974
```

Buvo sukurtas atskiras failas (priedas A1.2), kuriame vienoje vietoje kintamiesiems buvo priskiriamos vis keičiamos hyperparametrų reikšmės:

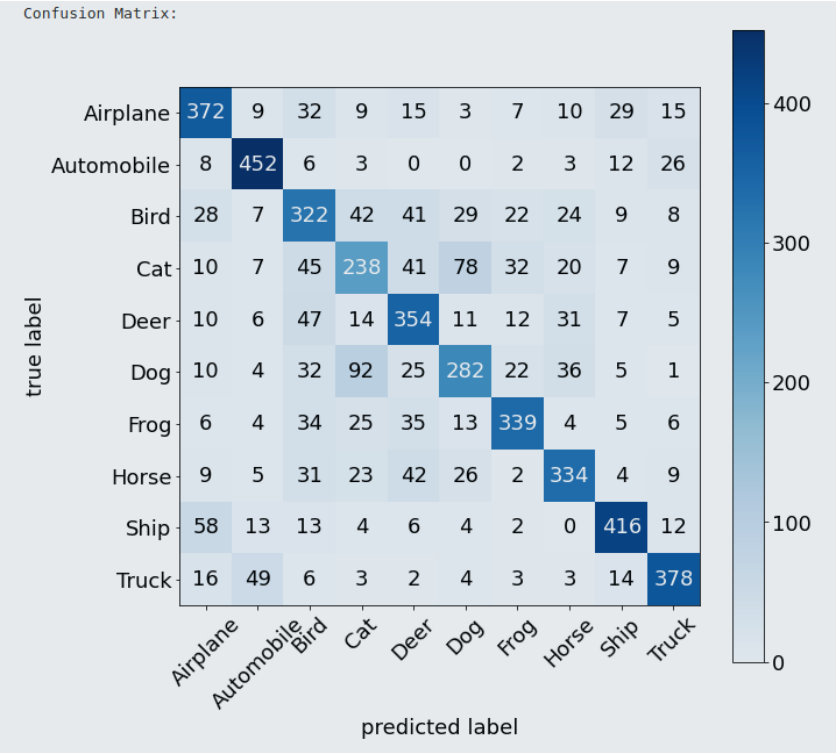
Set up parameters

In [21]:
```
param_epoch_count = 20
param_batch_size = 32
param_act_fn = 'relu'
param_optimizer = 'adam'
param_loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

2.2.4. Rezultatų išvestis

Galiausiai buvo nubrėžiami klasifikavimo tikslumo grafikai mokymo ir testavimo duomenims kas kiekvieną epochą, taip pat apskaičiuojama klasifikavimo matrica (*confusion matrix*). Grafikai pateikti tyrimo dalyje.

Klasifikavimo matricos pavyzdys:



2.3. Skaičiavimo resursai

Neuroninio tinklo apmokymui ir su juo atliekamais tyrimais buvo naudotas debesijos sprendimas *Gradient Paperspace* platformoje (https://gradient.paperspace.com/), kurioje buvo pasirinkta virtuali mašina, naudojanti vaizdo plokštę skaičiavimams.

Fizinės virtualios mašinos charakteristikos:

- Nvidia Quadro P4000 vaizdo plokštė.
- 8 branduolių Intel procesorius.
- 30 GB operatyviosios atminties.

3. Tyrimo rezultatai

Hyperparametrų tyrimas, kaip ir minėta, atliktas su anksčiau aprašyta architektūra. Jos parametrai fiksuoti, keičiant kurį nors vieną hyperparametrą, kiti išlieka tokie patys, kokie jie buvo pateikti toje architektūroje. Dar kartą atspausdiname jos parametrus:

Aktyvacijos funkcija - *ReLU*
Paketo dydis - *32*
Optimizavimo algoritmas - *Adam*
Nuostolių funkcija - *Sparse Categorical Crossentropy*

Iš pradžių tyrimai atlikti su 30 epochų, tačiau pasimatė, kad jau ties 6-7 epocha tinklas pradeda persimokyti. Taigi vėliau tiriant epochų skaičius buvo sumažintas iki 20, nes buvo verta

pasižiūrėti, kaip mokymosi tikslumas auga, nors ir persimokant. Epochų skaičius matosi iš grafikų.

Žemiau pateikti keičiami hyperparametrai ir jiems esant nustatytiems sugeneruoti grafikai.

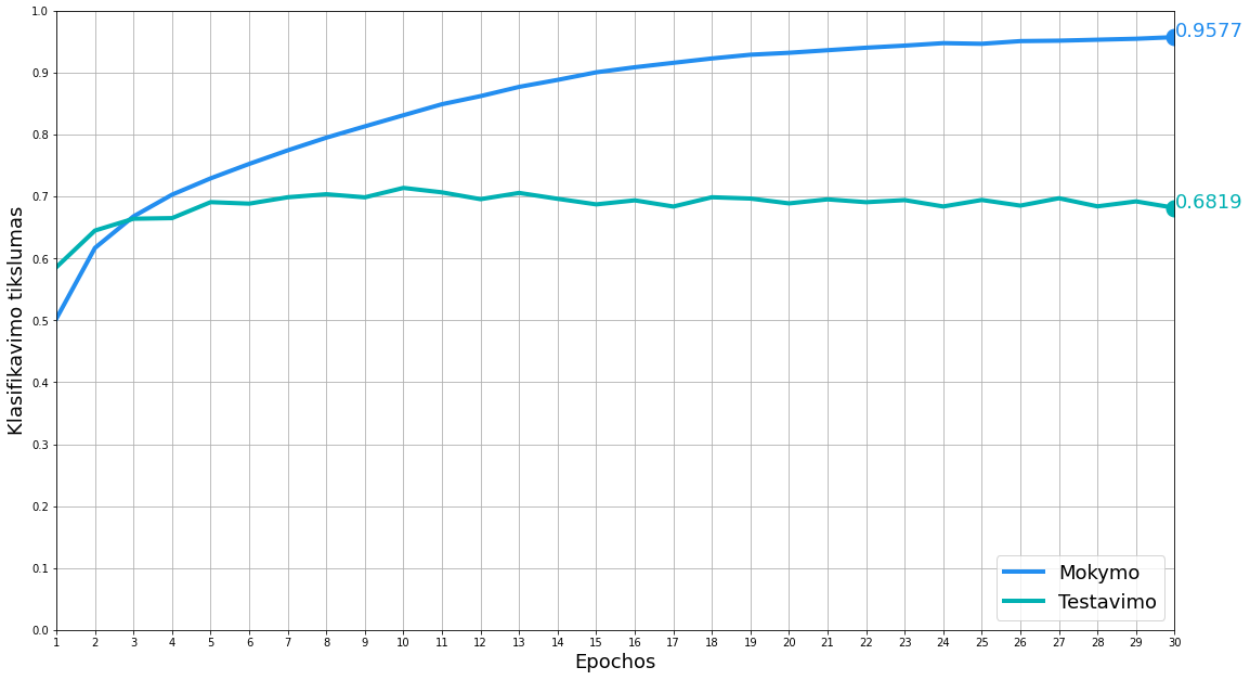
3.1. Aktyvacijos funkcijos

Buvo tirtos šios aktyvacijos funkcijos:

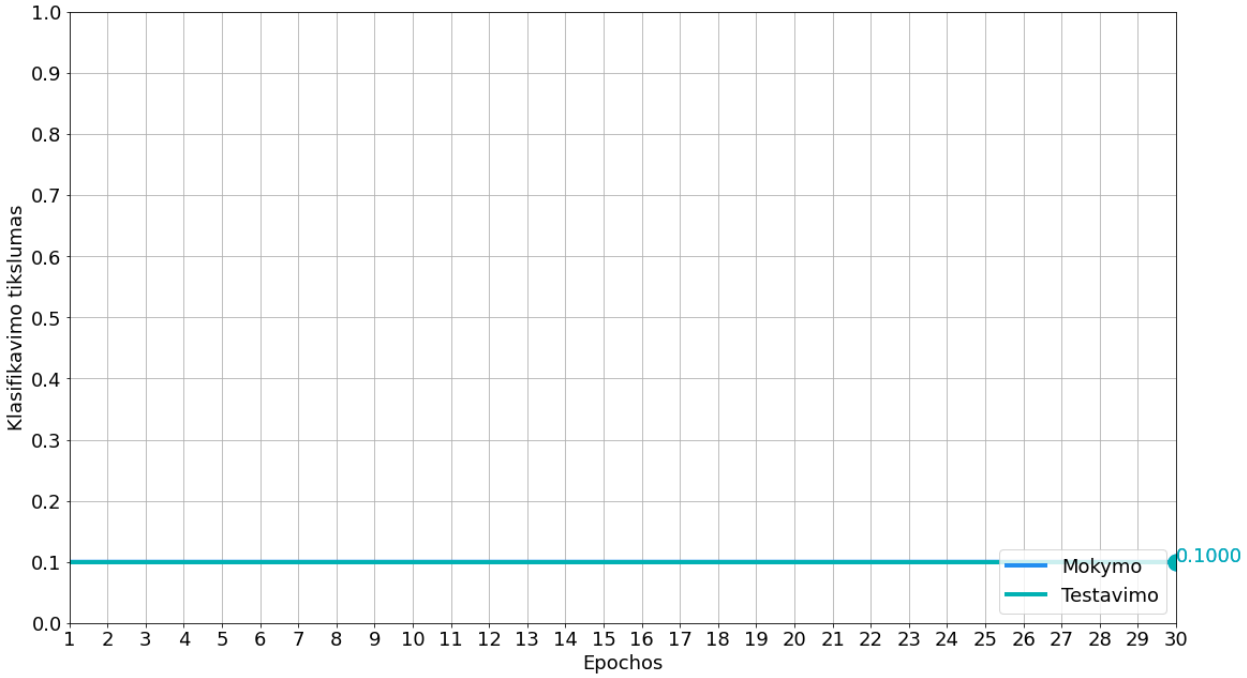
1. *elu*
2. *exponential*
3. *gelu*
4. *hard_sigmoid*
5. *linear*
6. *relu*
7. *selu*
8. *sigmoid*
9. *softplus*
10. *softsign*
11. *swish*
12. *tanh*

Rezultatai:

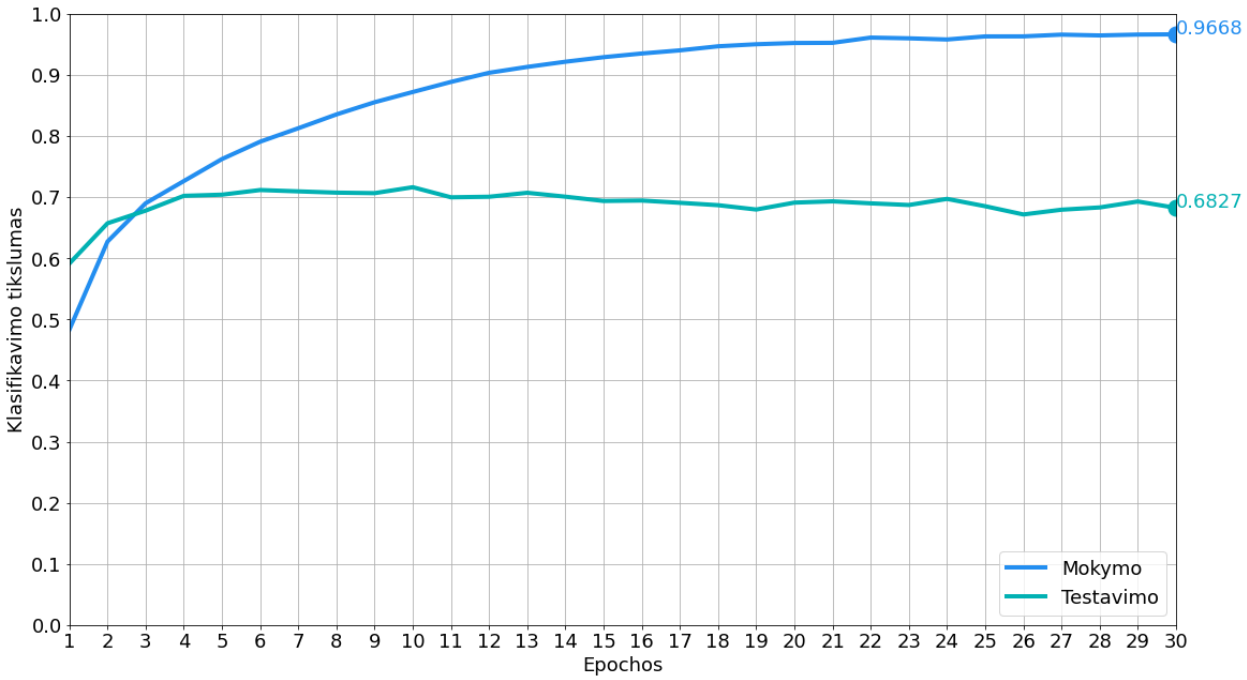
1. *elu*



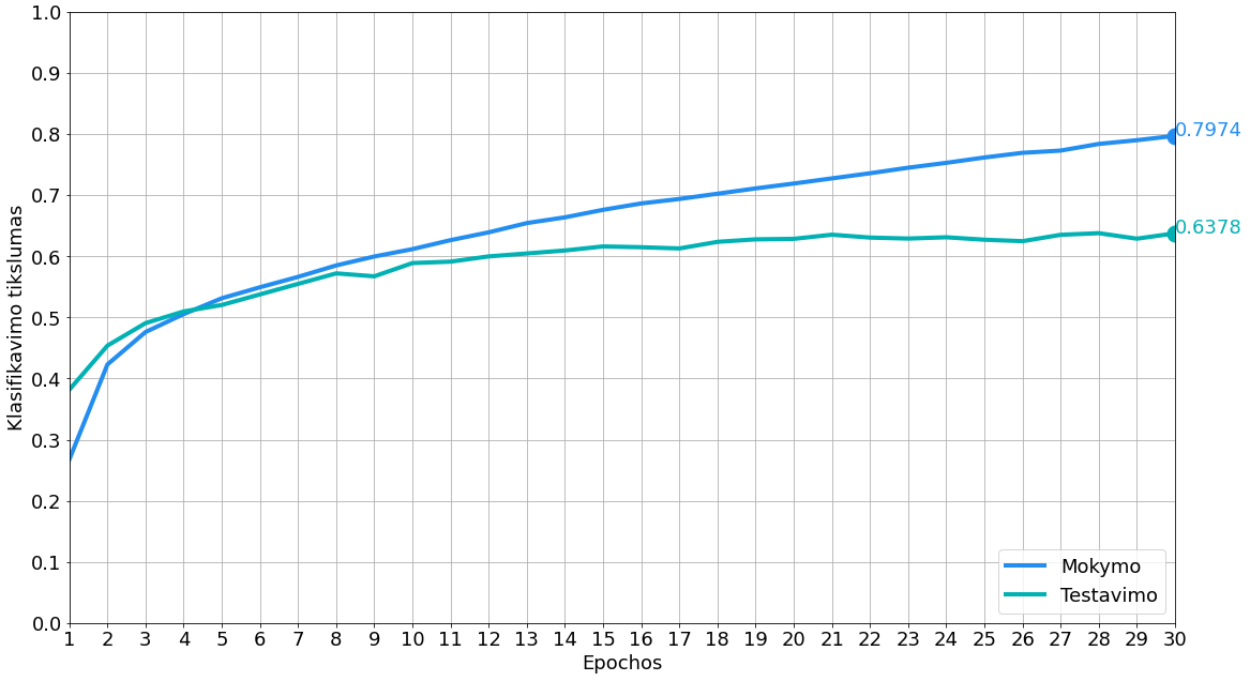
2. *exponential*



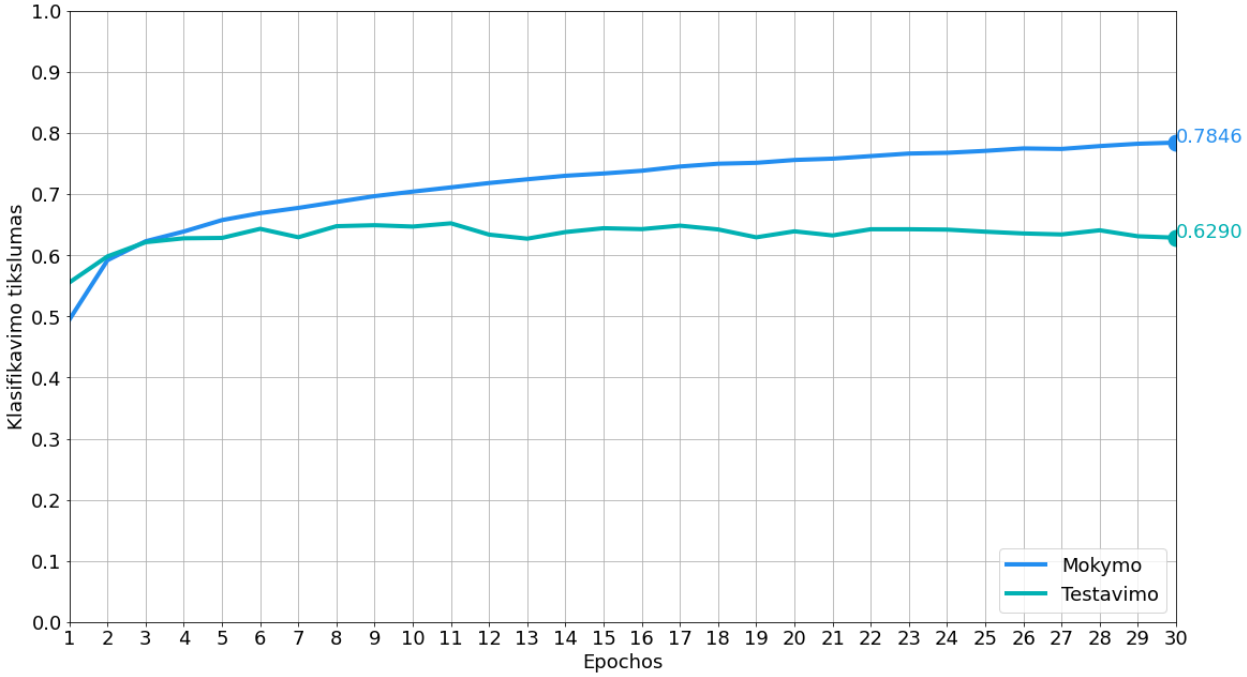
3. *gelu*



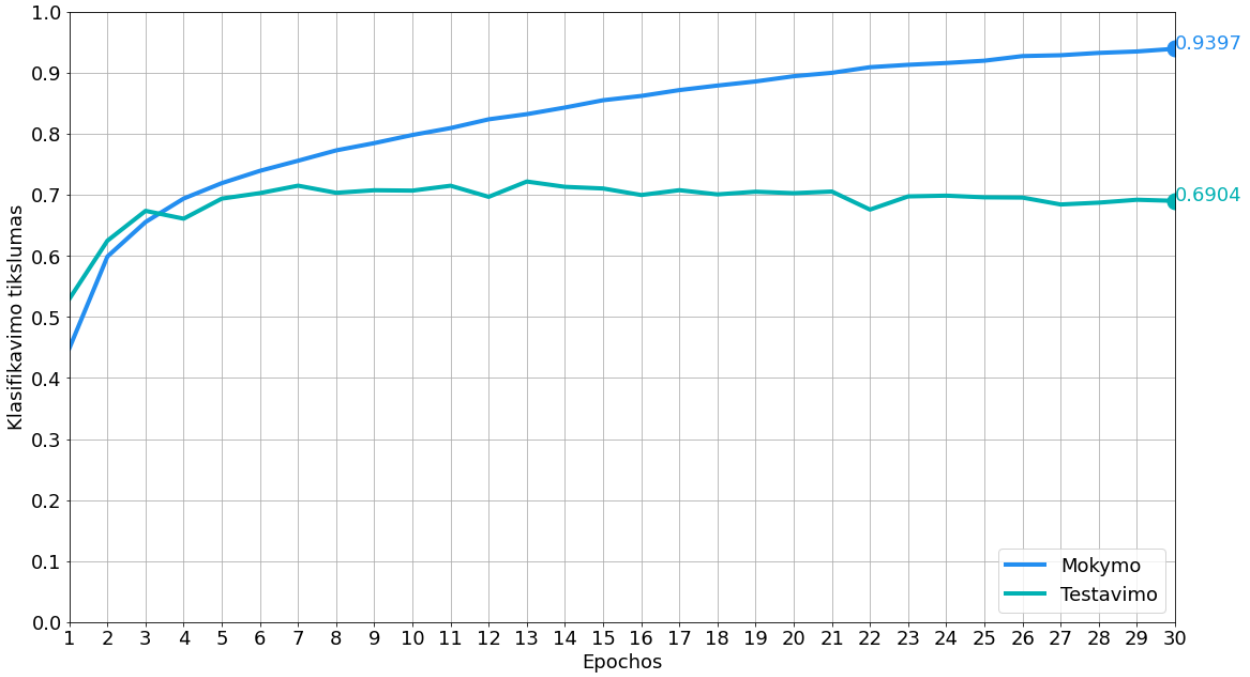
4. *hard_sigmoid*



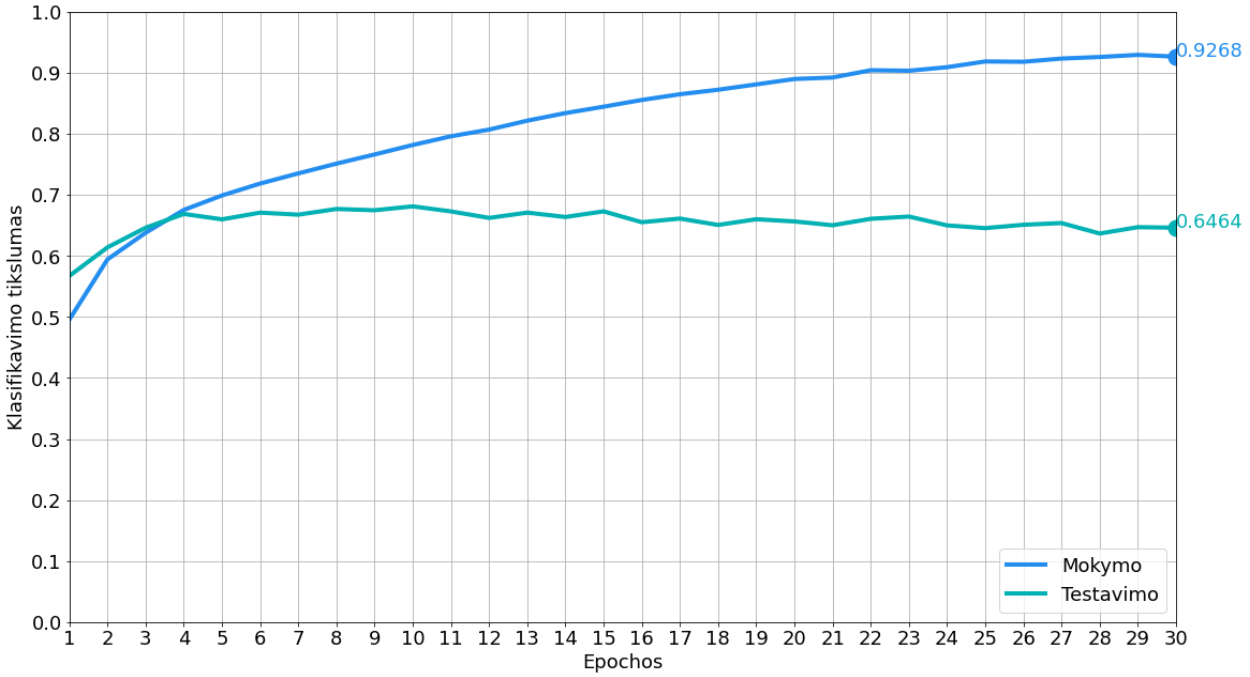
5. *linear*



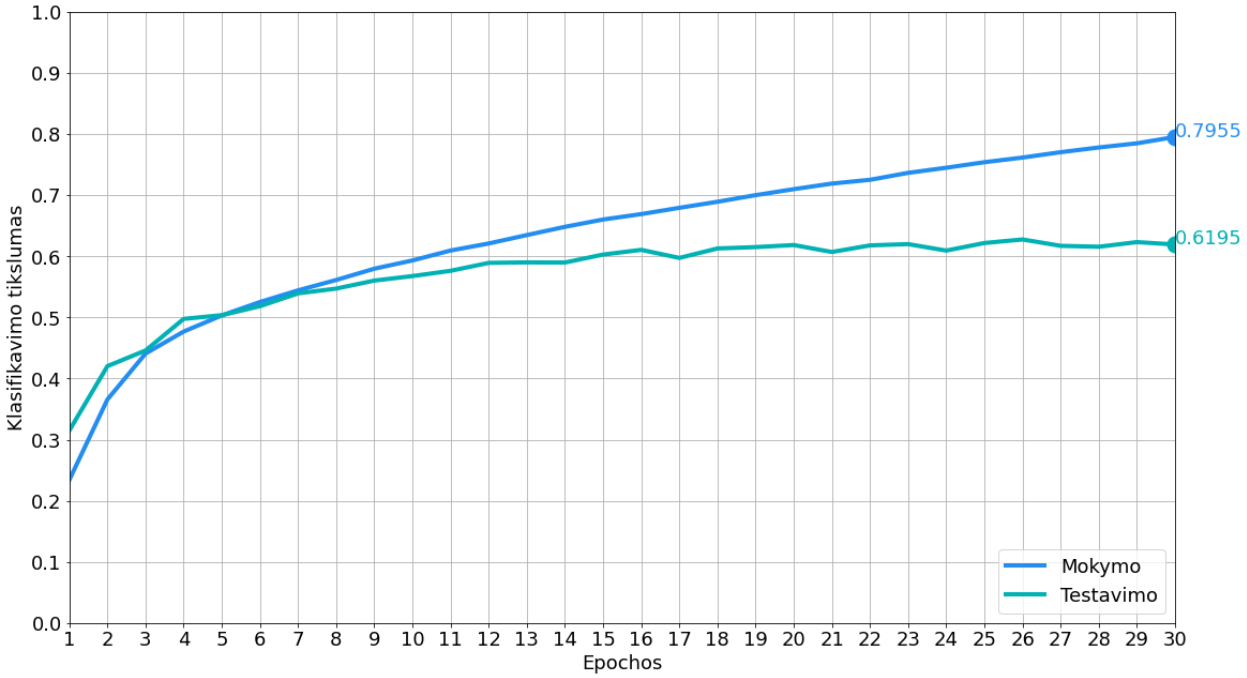
6. *relu*


Klasifikavimo tikslumas
Epochos
Mokymo
Testavimo
0.9397
0.6904

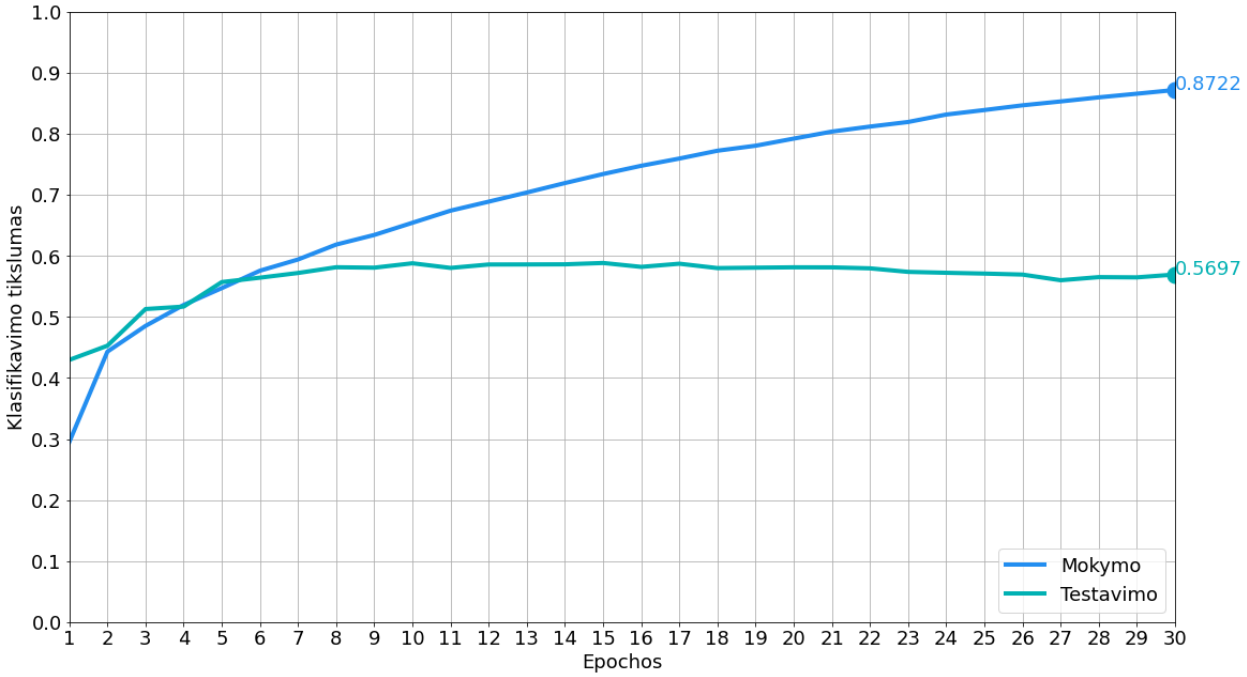
7. *selu*


Klasifikavimo tikslumas
Epochos
Mokymo
Testavimo
0.9268
0.6464

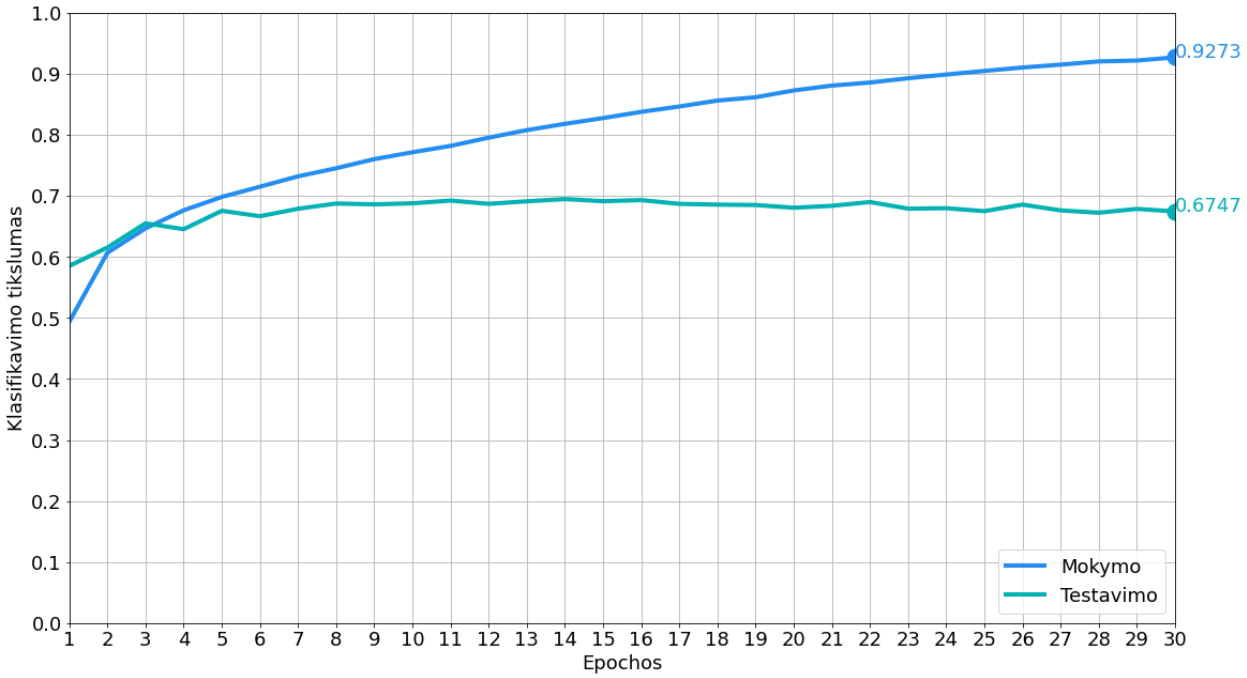
8. *sigmoid*



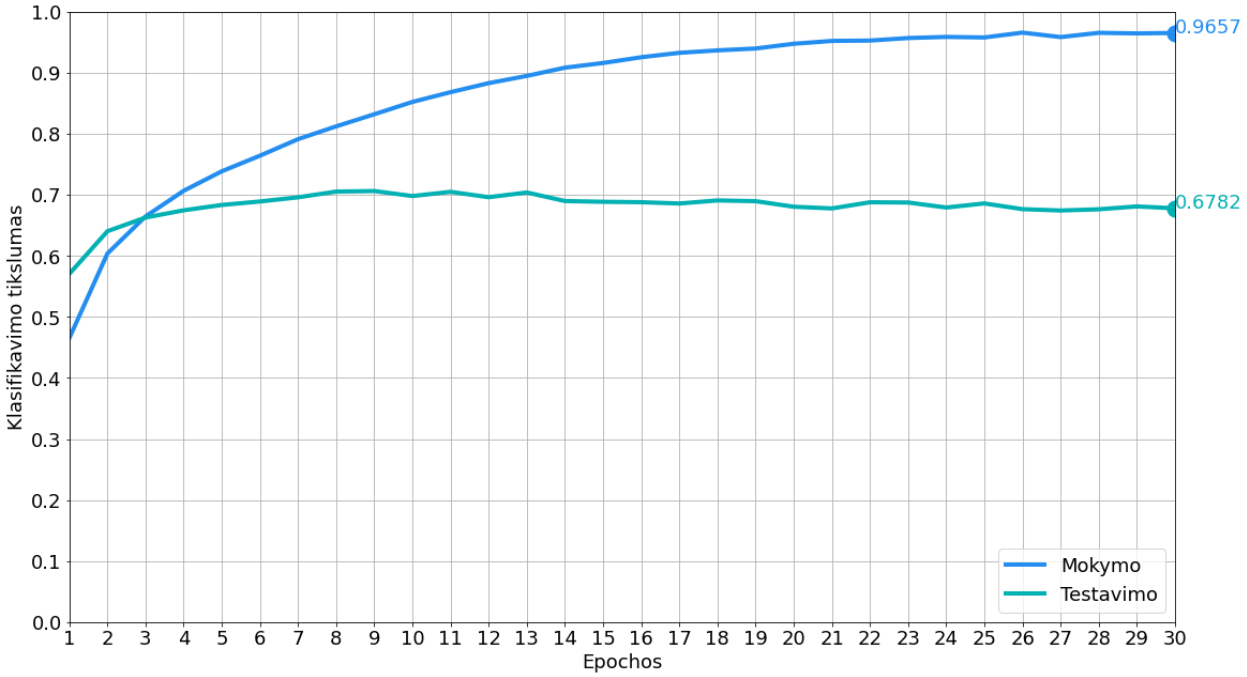
9. *softplus*



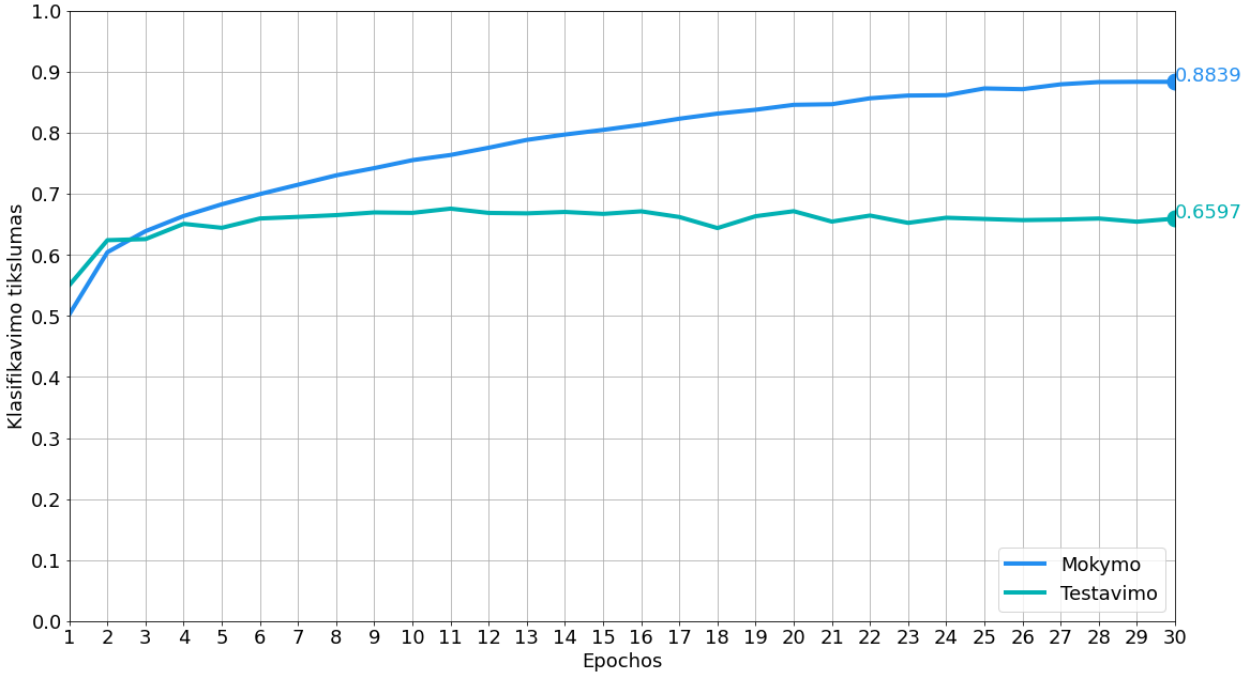
10. softsign



11. swish



12. tanh



Interpretacija:

Visų pirma, matome, kad neuroninis tinklas pradeda persimokyti jau ties 5-6 epocha, tad 30 šiam epochų tyrimui galimai yra per daug, tačiau galime matyti, kad mokymosi tikslumas vis auga, ir nuo 20 iki 30 epochų jis vis tiek pakyla visais atvejais 0.2–0.8 dydžiu. Kitiems tyrimams jau apsiribosime 20-čia epochų.

Tuo tarpu testavimo tikslumas pasiekia stabilią reikšmę ties 5-8 epocha, ir toliau per daug nesikeičia, jo reikšmė tai padidėja šiek tiek, tiek sumažėja.

Matome, kad *hard_sigmoid*, *sigmoid* ir *linear* mokymosi tikslumas yra mažesnis, mažesnis ir testavimo tikslumas, tačiau jis vis tiek išlieka gana panašus į kitus atvejus.

Su *exponential* funkcija tinklas visai neapsimokė, ir matyt tai buvo netinkamas jos panaudojimas šiame uždavinyje.

Softplus atveju testavimo tikslumas gavosi mažiausias.

Tuo tarpu *elu*, *gelu*, *relu*, *selu*, ir *swish* atvejais tiek mokymo, tiek testavimo tikslumo rezultatai buvo geresni ir ganėtinai panašūs tarpusavyje, *tanh* mokymo tikslumas gavosi kiek mažesnis.

Didžiausias mokymosi tikslumas gavosi su *gelu* funkcija (0.9668), o testavimo - su *relu* (0.6904).

Bendrais bruožais pastebime, kad linijinių elementų turinčios funkcijos *(linear unit)* mūsų tyrime davė geriausius rezultatus bei tarpusavyje jų rezultatai buvo gana panašūs.

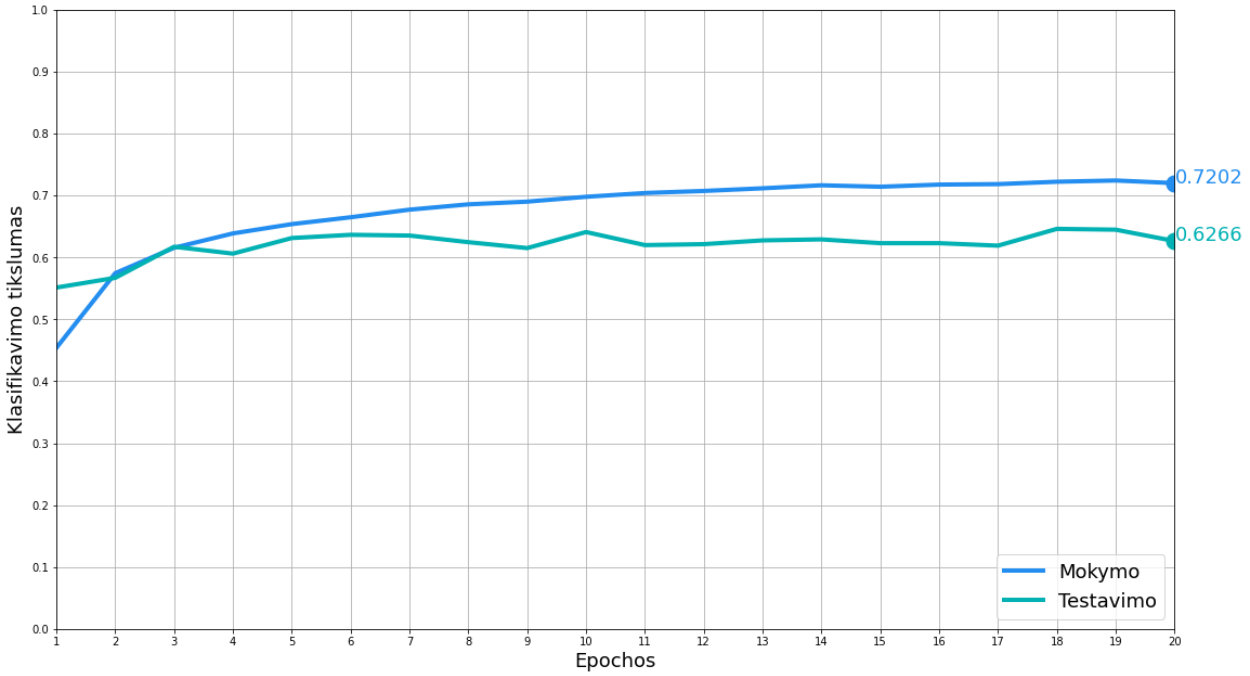
3.2. Paketo dydžiai

Tirti paketo dydžiai:

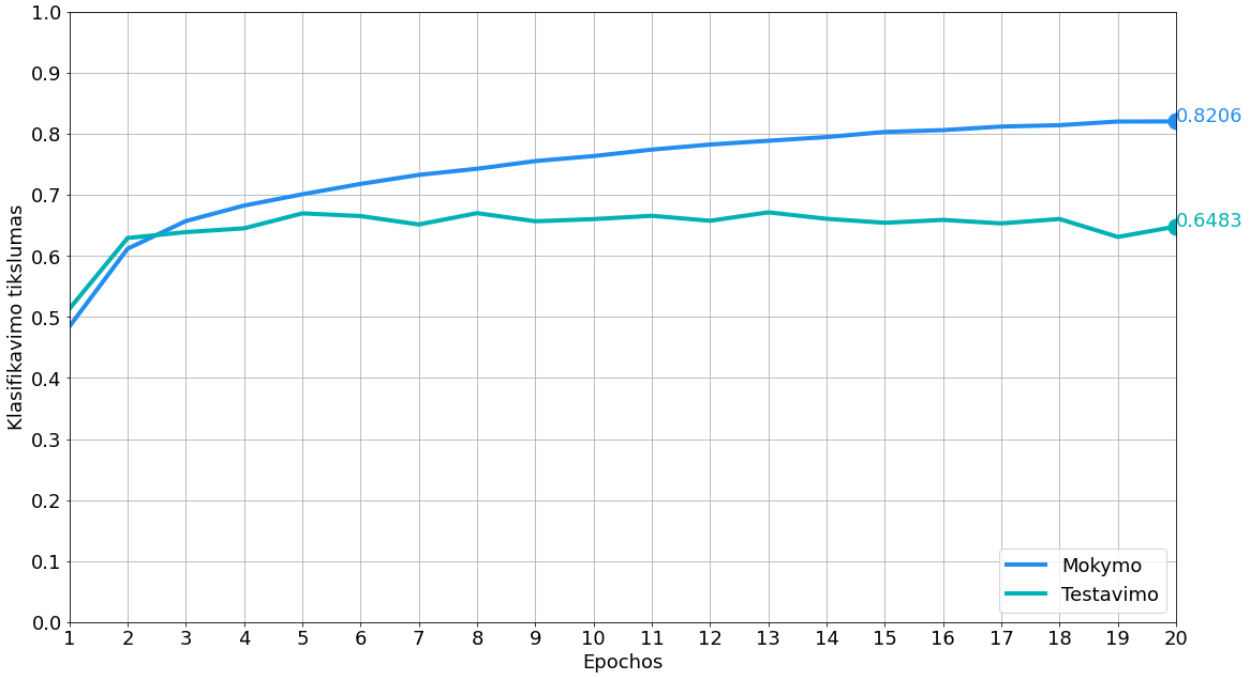
2, 3, 5, 6, 9, 10, 15, 20, 24, 32, 64, 128, 512, 1024, 2048, 4096, 8192, 10000, 15000.

Rezultatai:

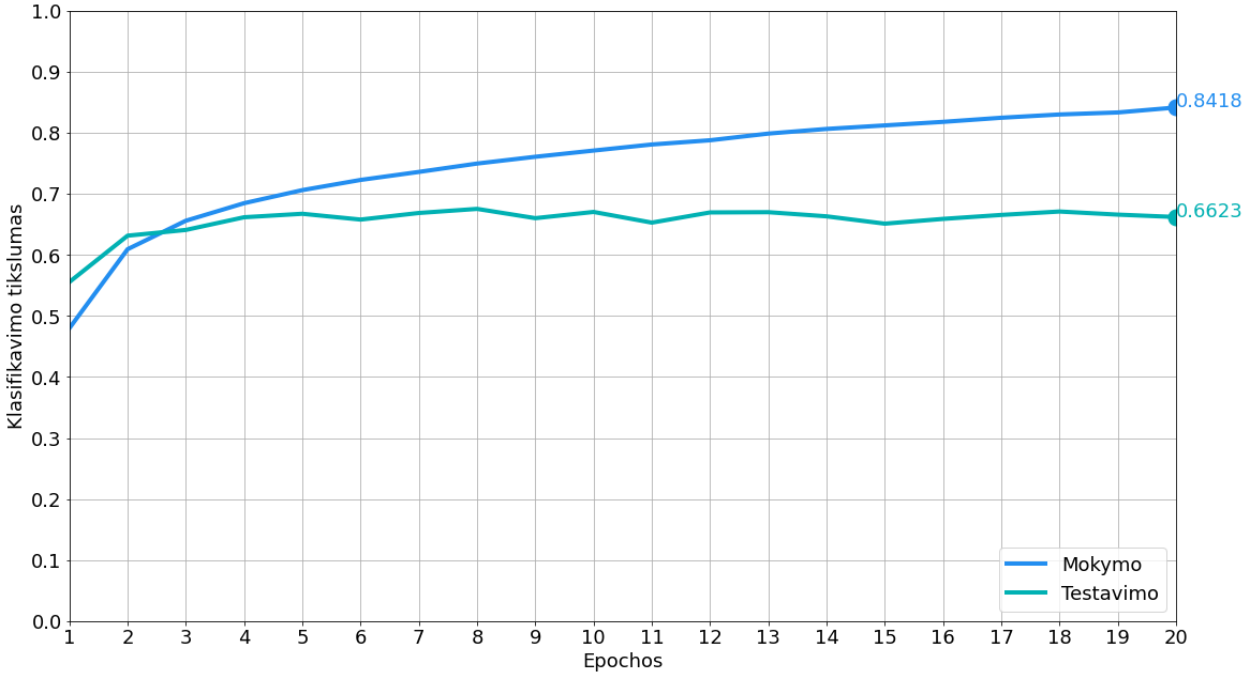
Paketo dydis = 2



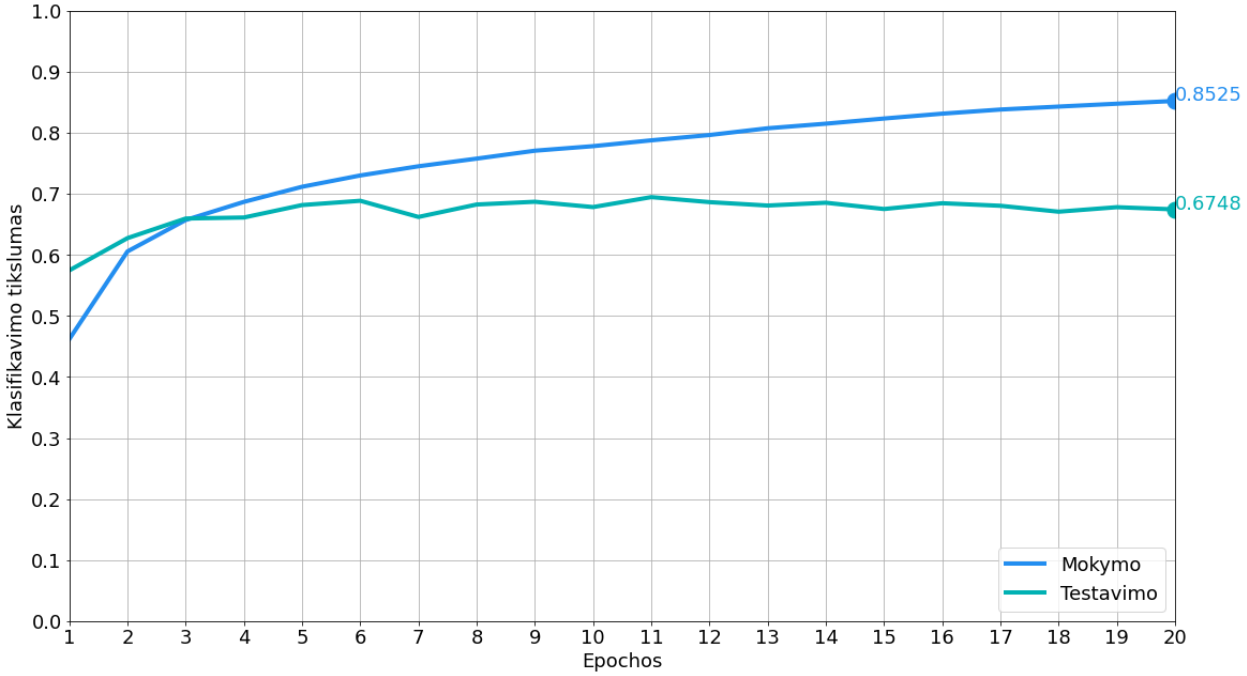
Paketo dydis = 3



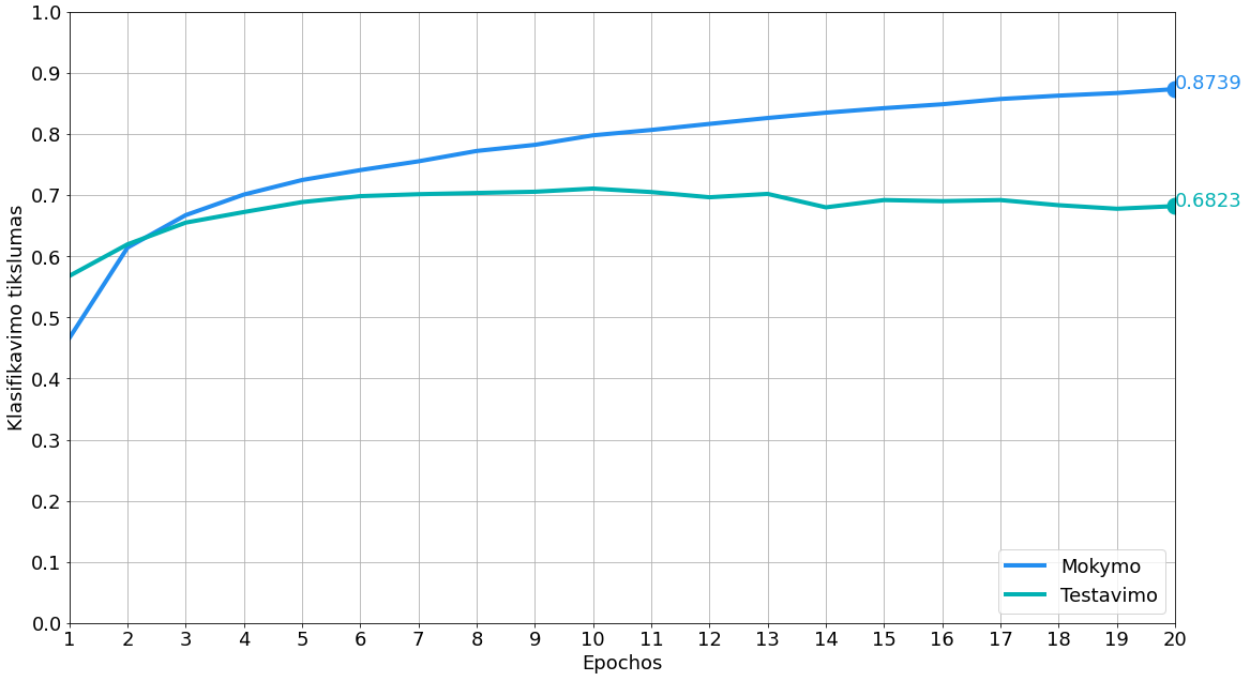
Paketo dydis = 5



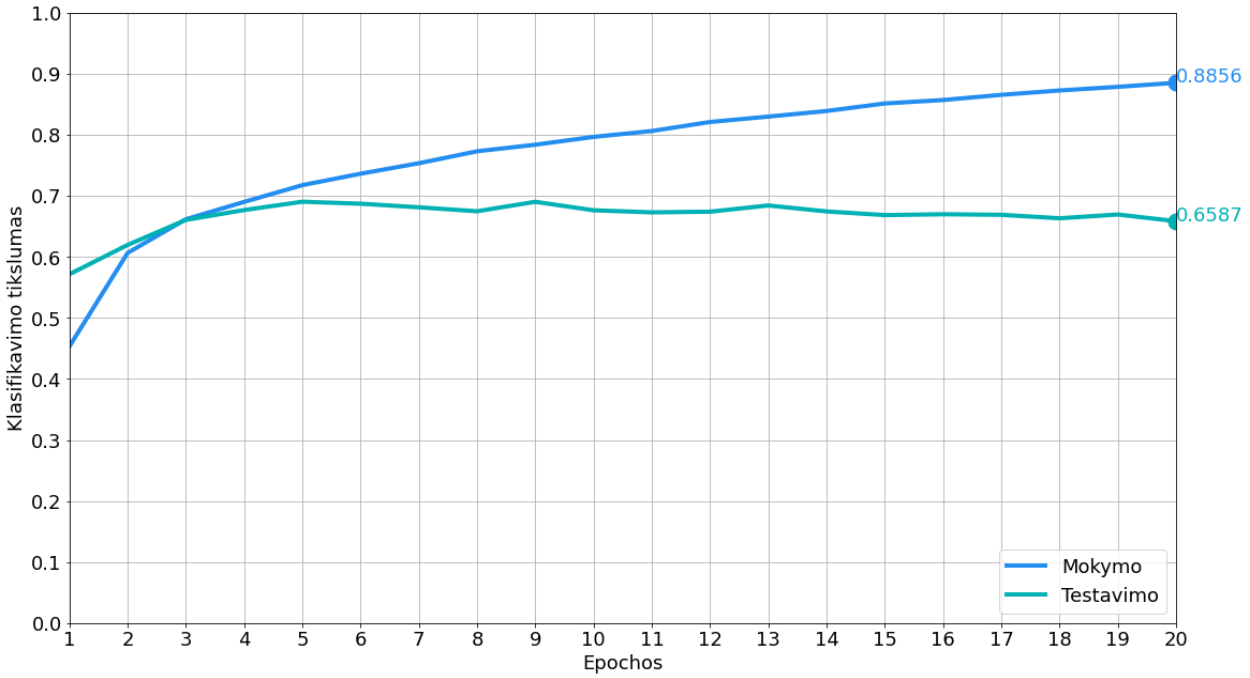
Paketo dydis = 6



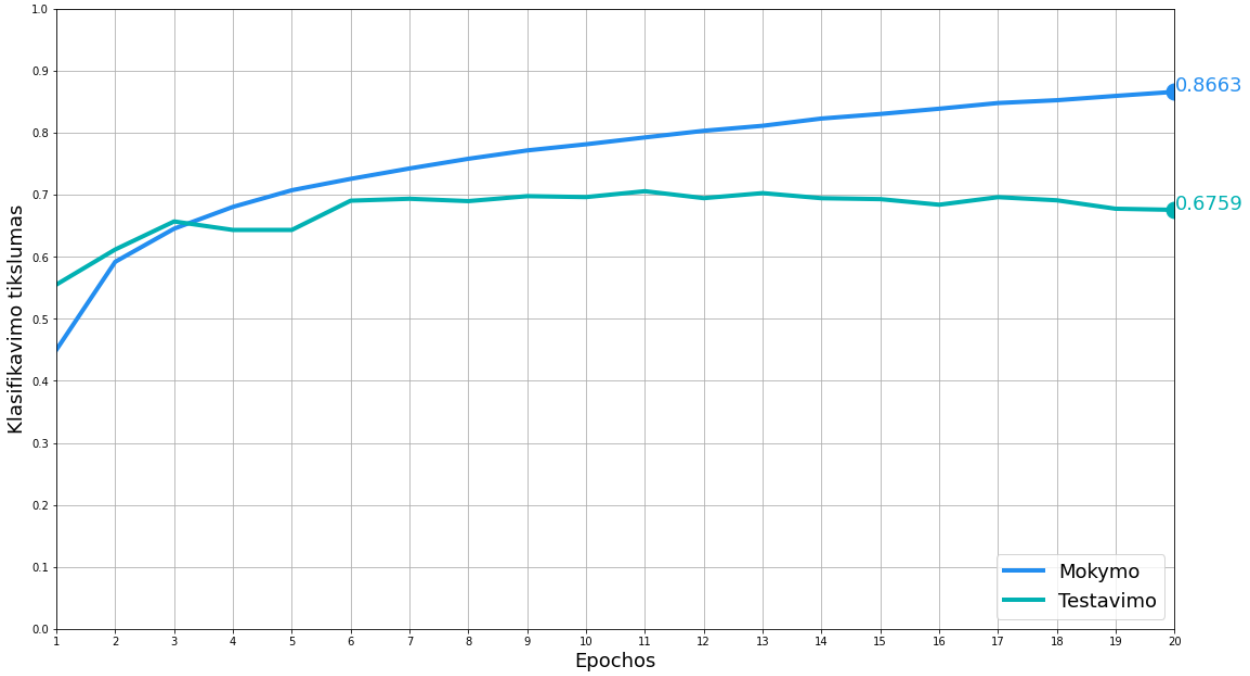
Paketo dydis = 9



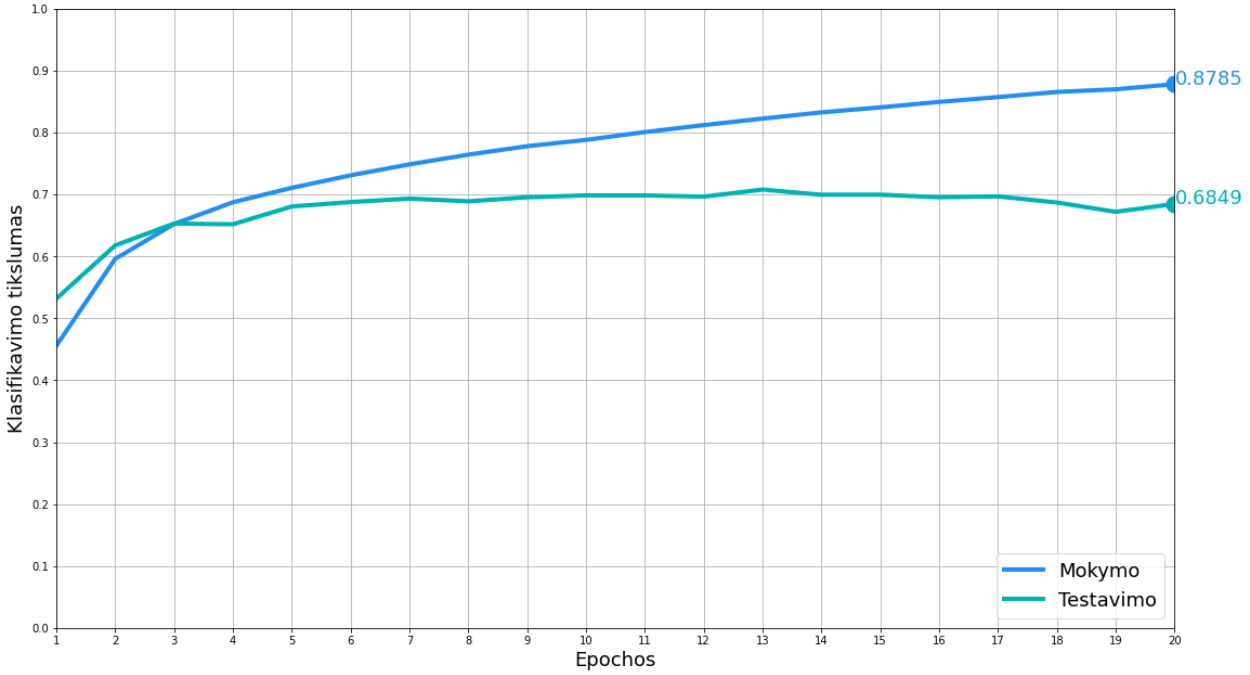
Paketo dydis = 10



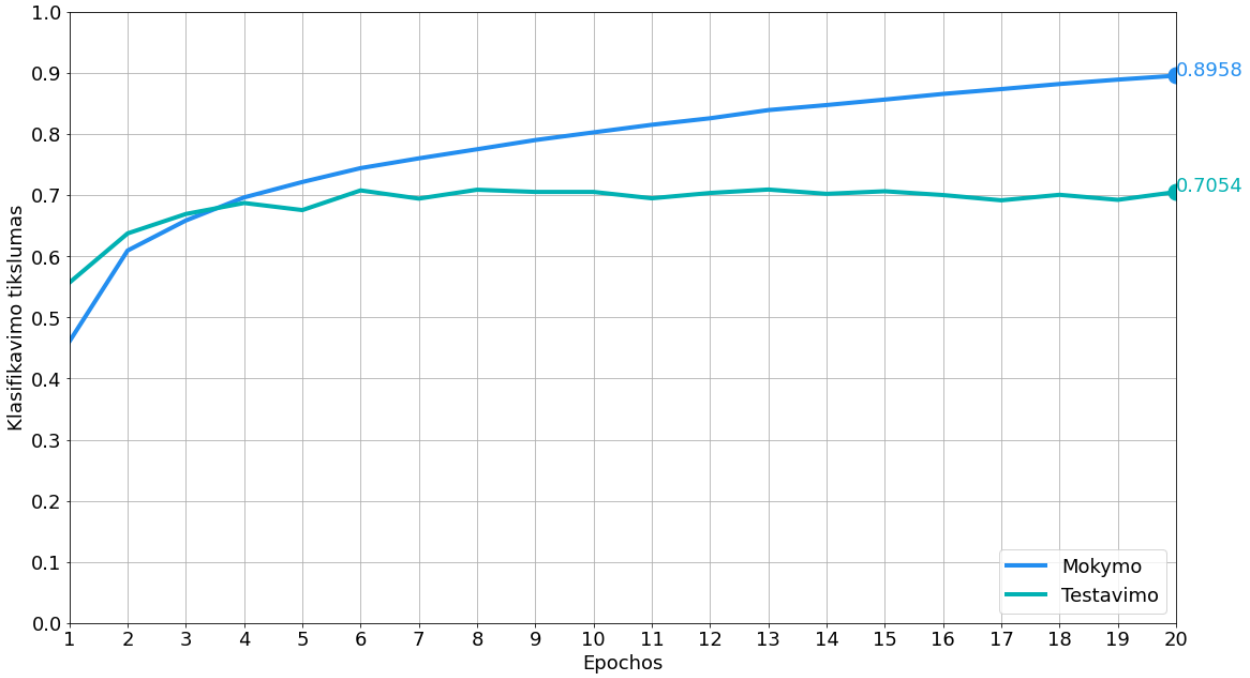
Paketo dydis = 15



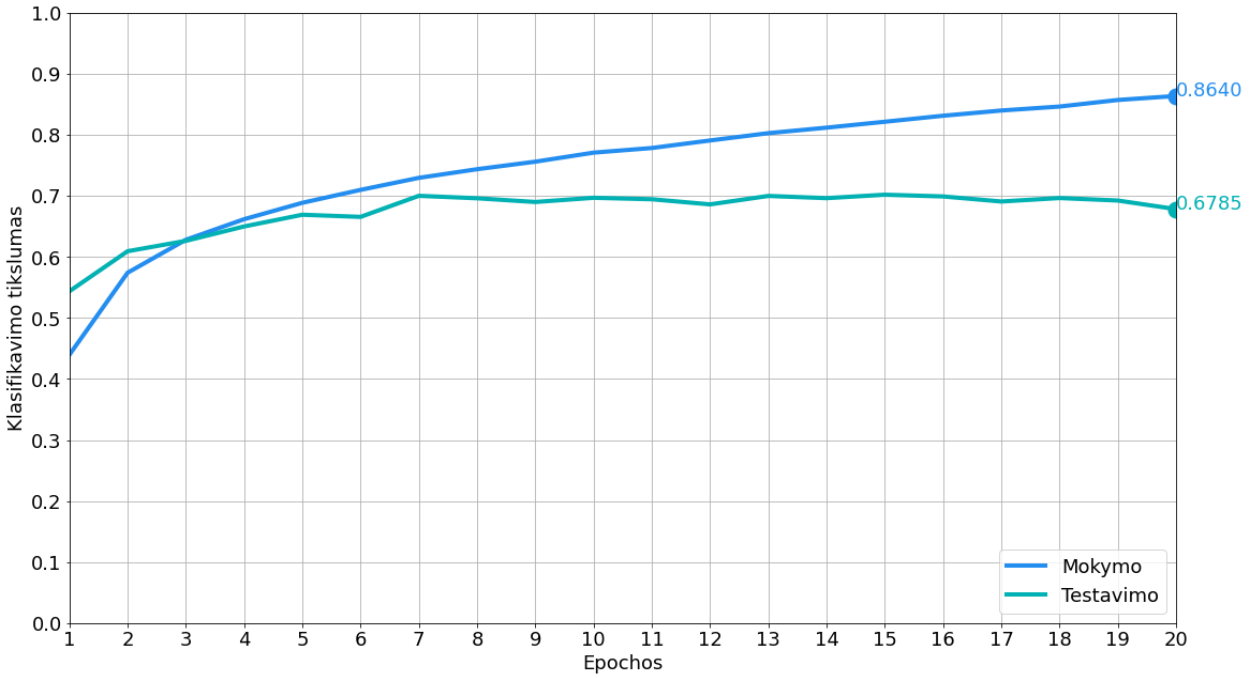
Paketo dydis = 20



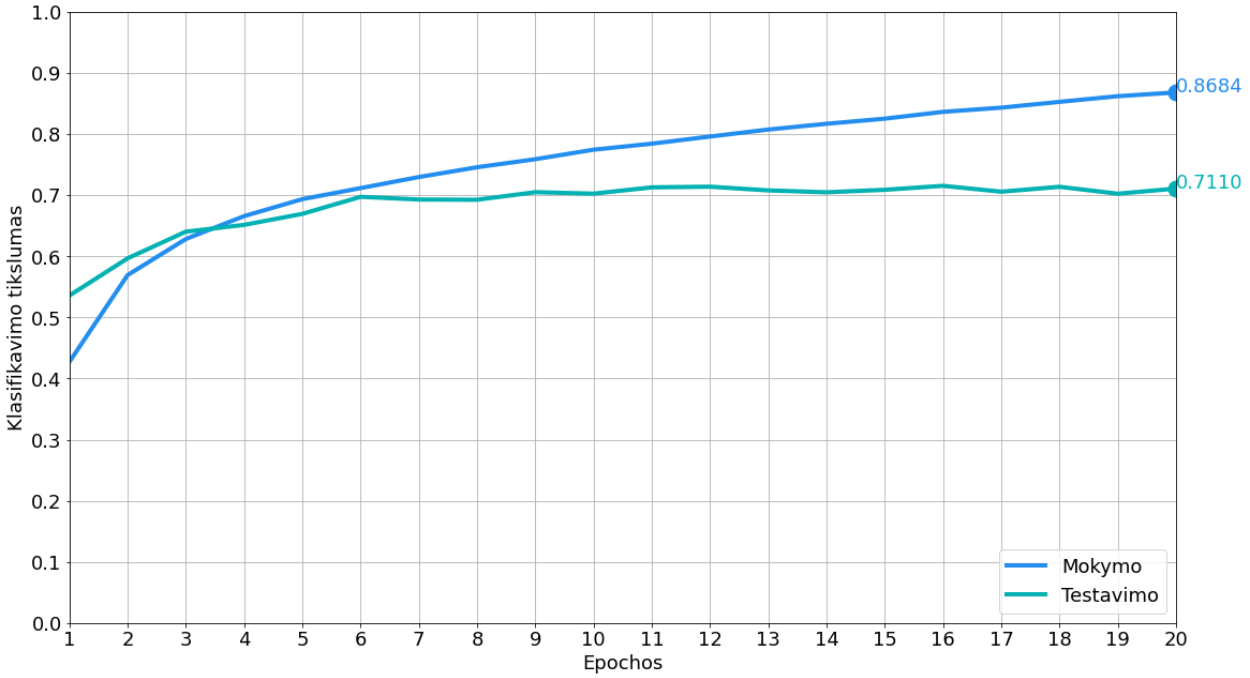
Paketo dydis = 24



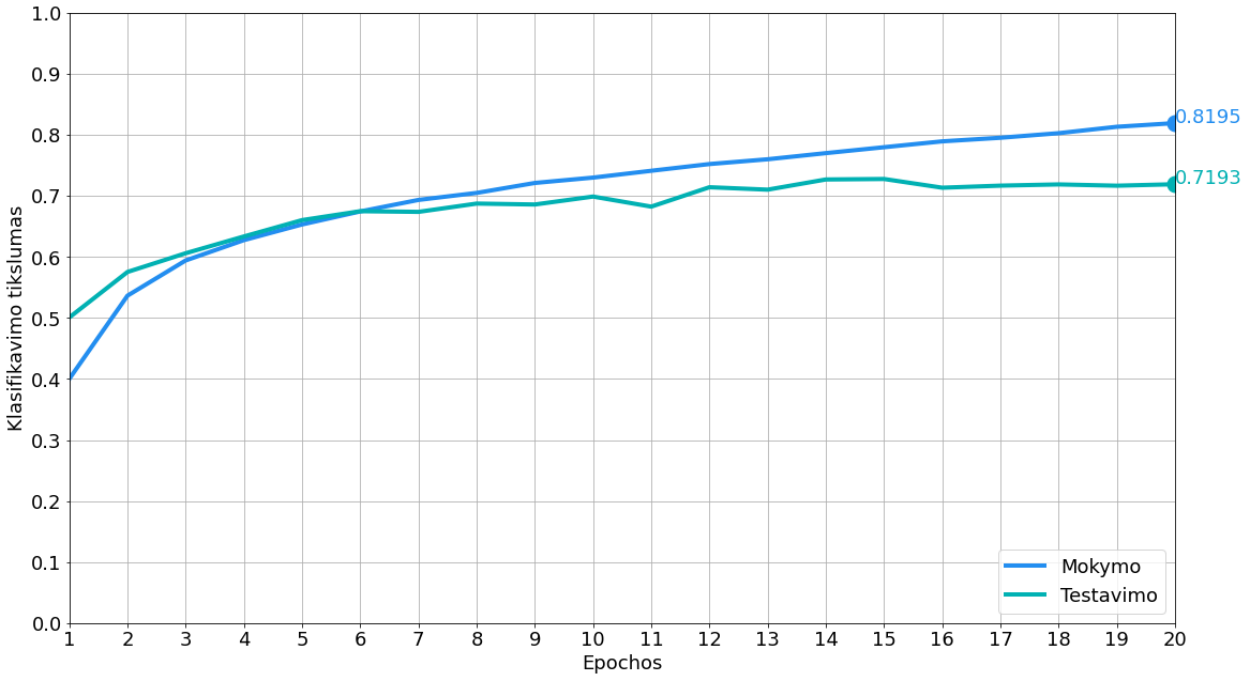
Paketo dydis = 32



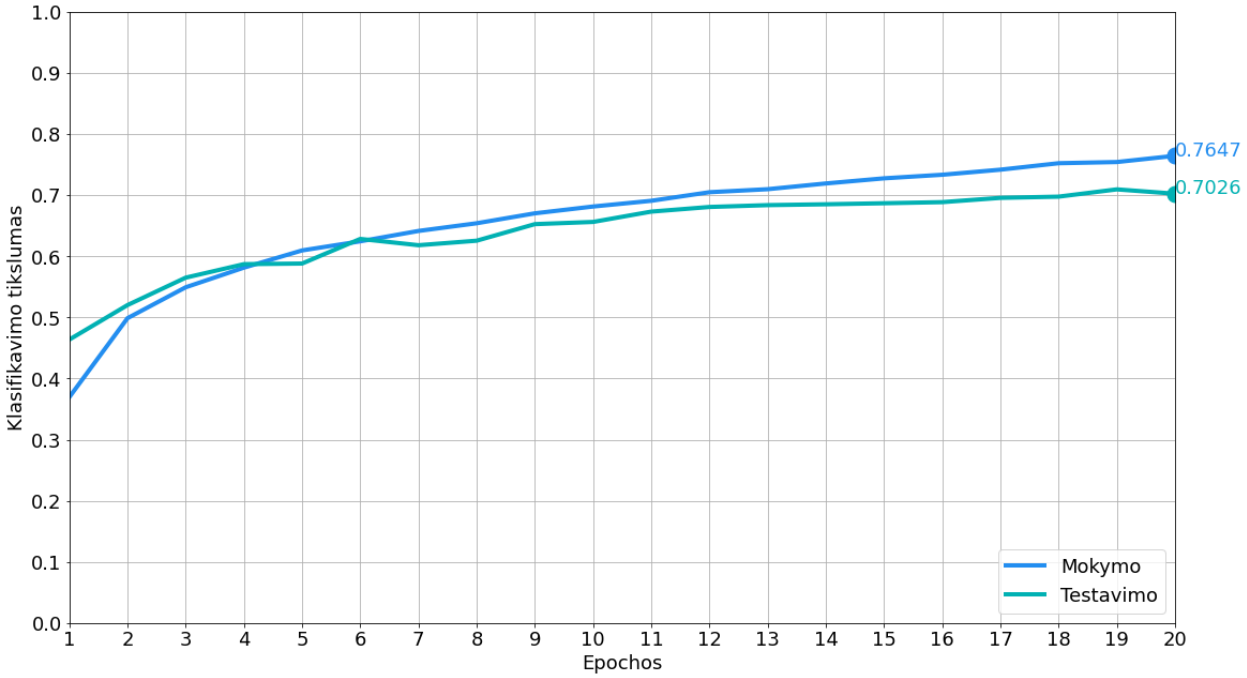
Paketo dydis = 64



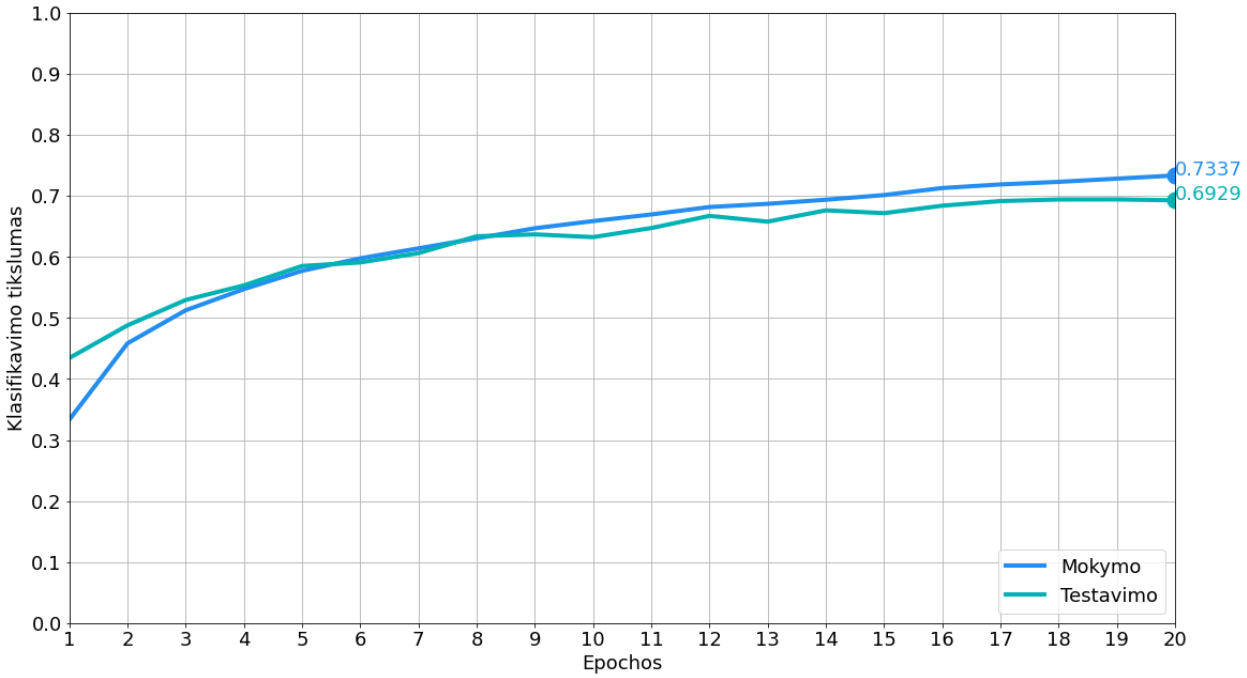
Paketo dydis = 128



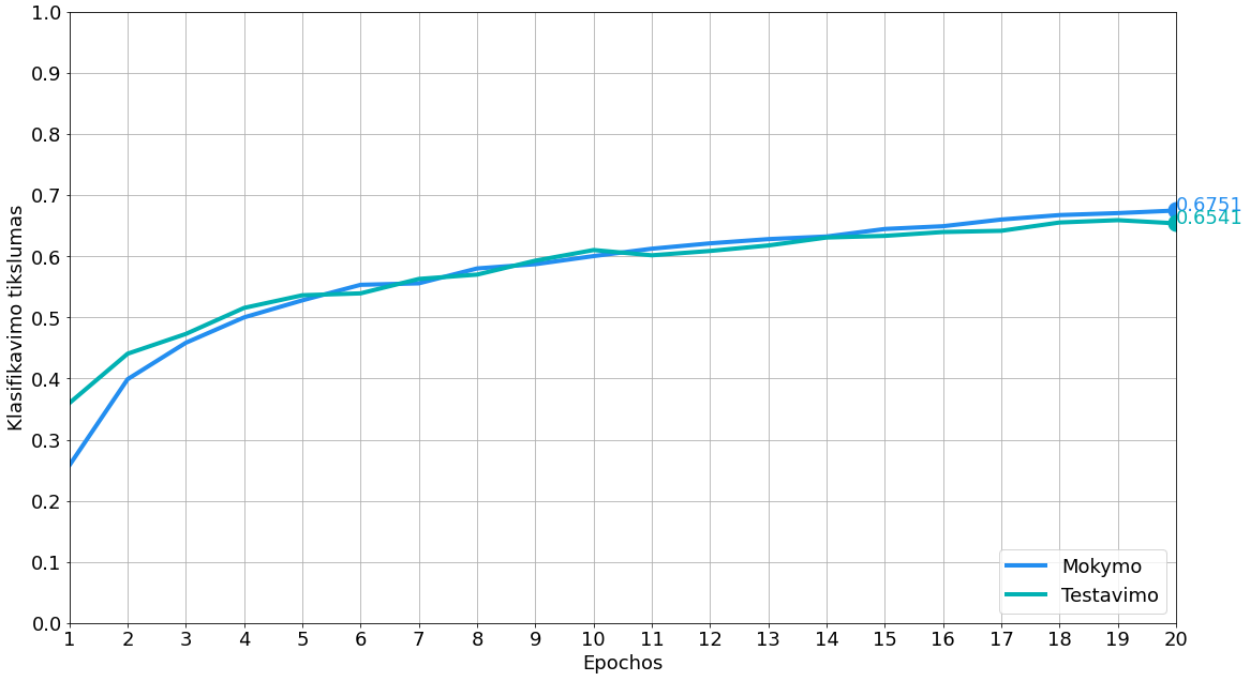
Paketo dydis = 256



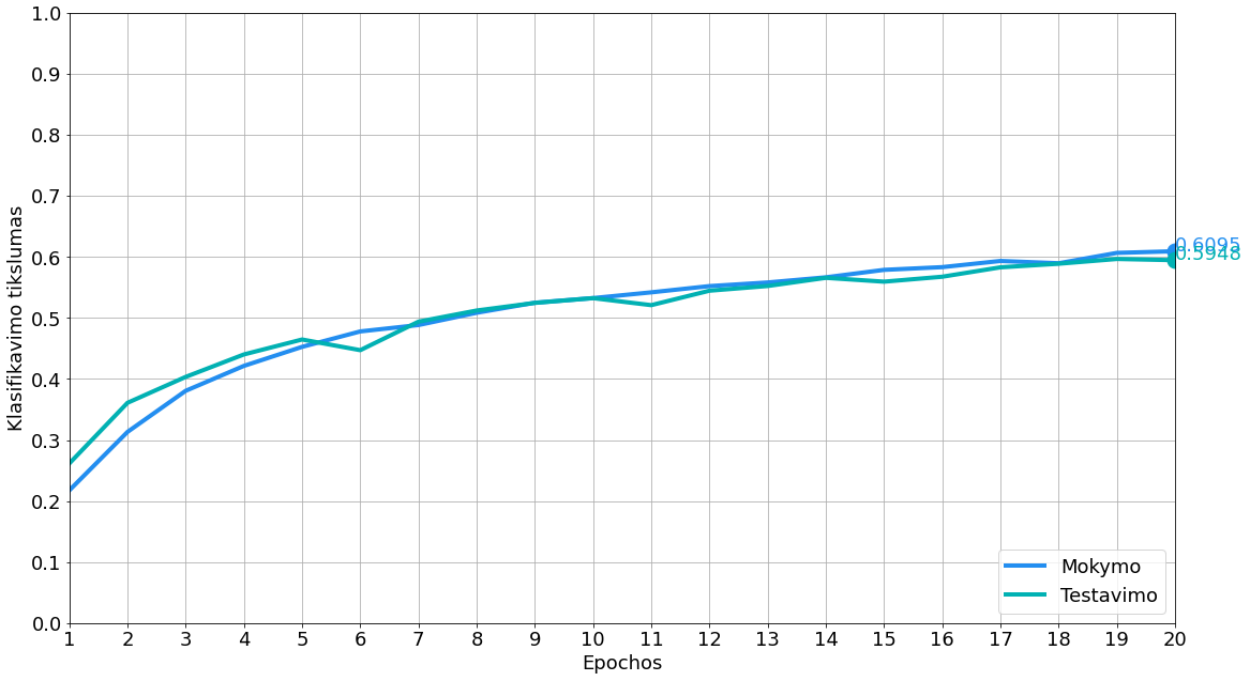
Paketo dydis = 512



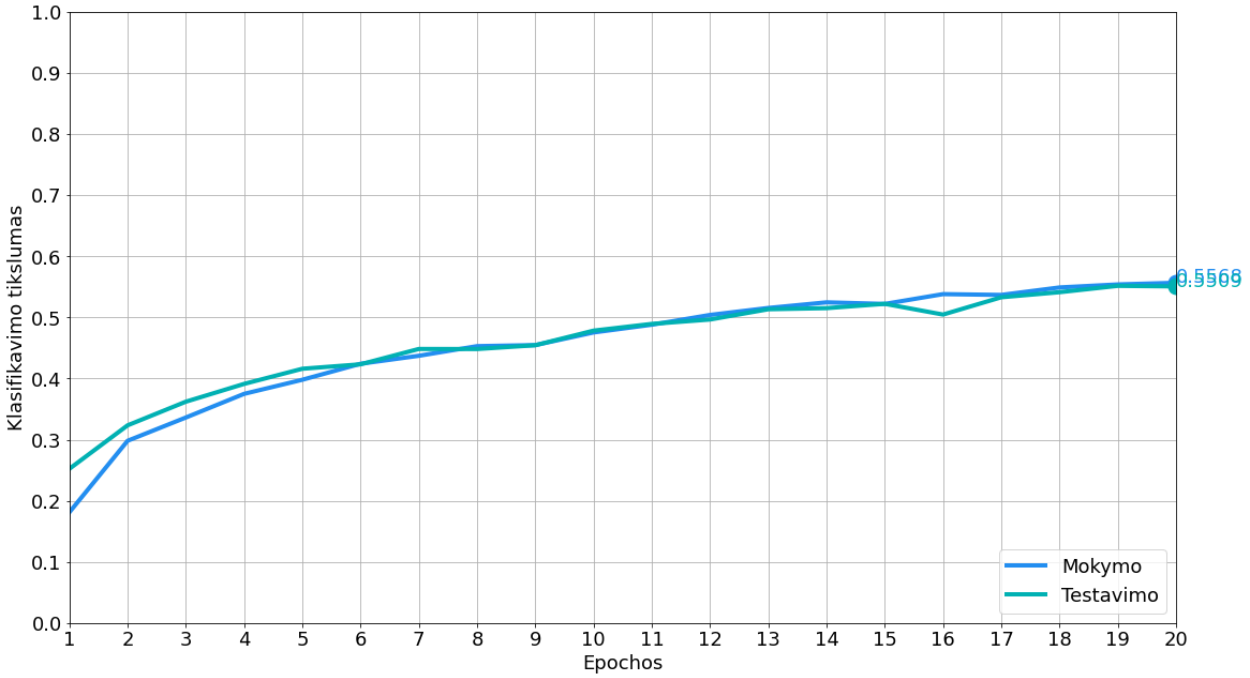
Paketo dydis = 1024



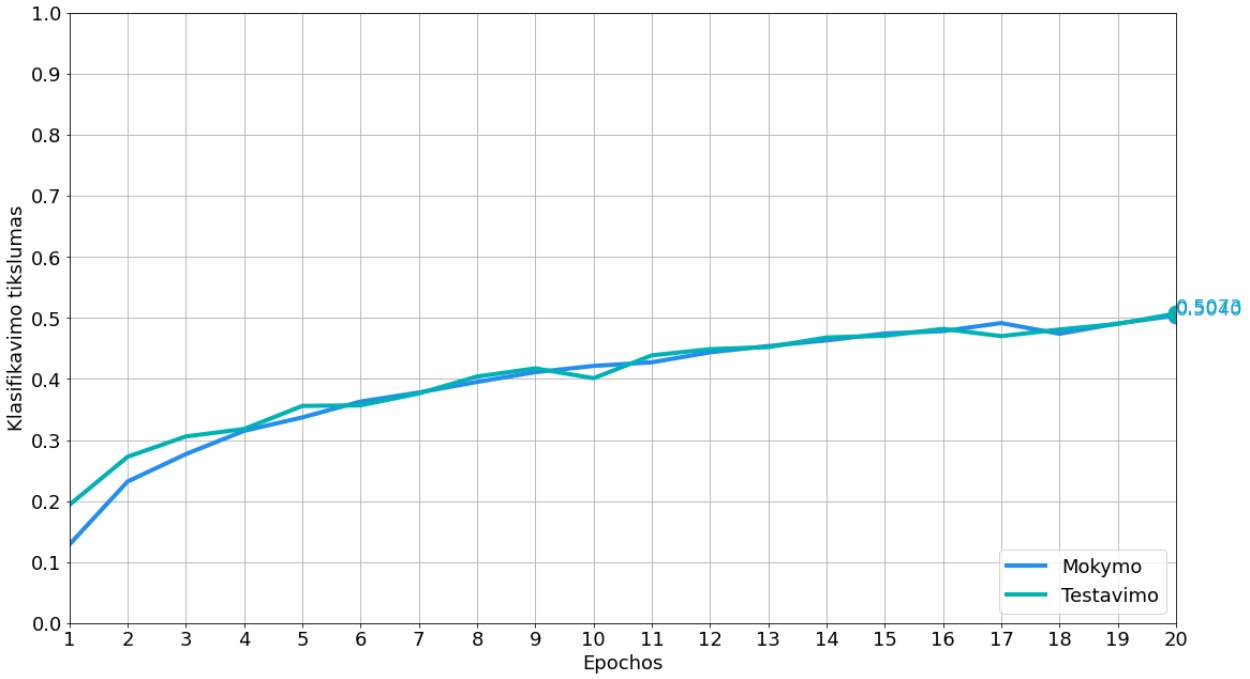
Paketo dydis = 2048



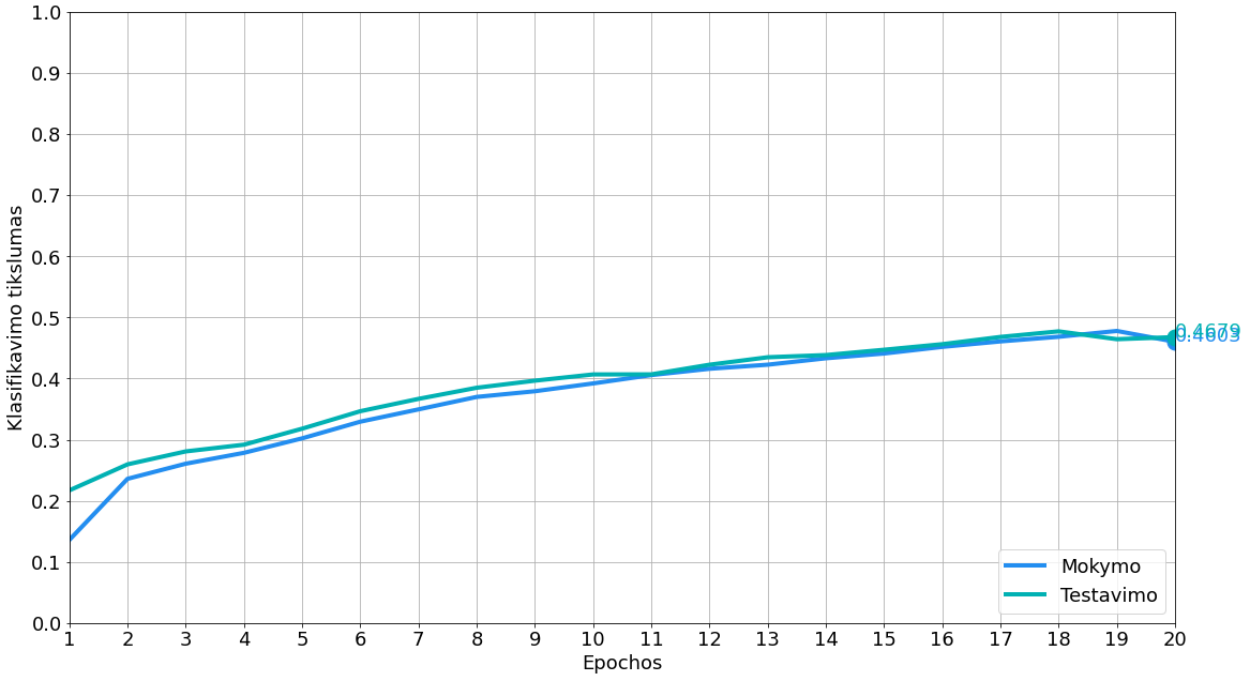
Paketo dydis = 4096



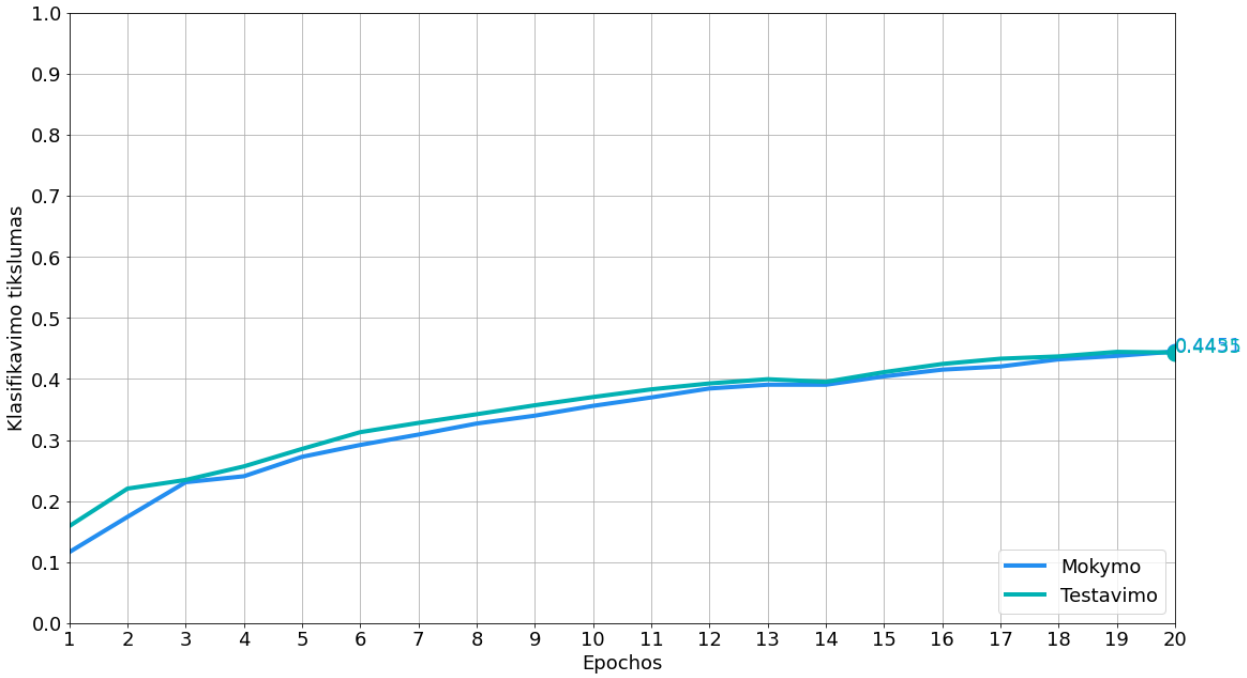
Paketo dydis = 8192



Paketo dydis = 10 000



Paketo dydis = 15 000



Interpretacija:

Šiuo atveju paketo dydis darė įtaką neuroninio tinklo mokymosi greičiui, ir paketo dydžiams esant mažiems (2, 3, 5, 6, 9, 10), tinklas apsimokydavo nuo kelių iki dešimties kartų ilgiau nei su šio tinklo kode numatytu standartiniu 32 dydžio paketu.

Tačiau maži paketo dydžiai šiame tyrime davė netgi prastesnius rezultatus. Tai galima paaiškinti tuo, kad tyrime naudojama neuroninio tinklo architektūra pritaikyta spartesniam apsimokymui ir geriau veikia su kiek didesniais paketų dydžiais.

Paketo dydį padidinus iki 9, 10, padidejo tikslumas ir jau rezultatai tapo stabilesni nuo šios vietos, tad toliau didinant paketų skaičių iki 15, 20, 24, 32, 64, 128, rezultatai buvo daugmaž panašūs. Geriausias mokymosi tikslumas gautas su dydžiu 24 (0.8958), o testavimo - su 128 (0.7193). Paketo dydžiai 64 ir 128 davė šiek tiek geresnius rezultatus nei 32, tad matome, kad eksperimentuojant su paketo dydžiais, galima nevengti bandyti ir didinti šį dydį kelis kartus.

Su paketo dydžiu 256 jau aikvaizdžiai matomas tikslumų sumažėjimas, ir jie toliau mažėja, didinant paketų dydį, o mokymosi tikslumas pradeda beveik sutapti su testavimo tikslumu. Tokiu atveju visiškai išvengta persimokymo per 20 epochų, tačiau klasifikavimo tikslumas gaunamas mažesnis.

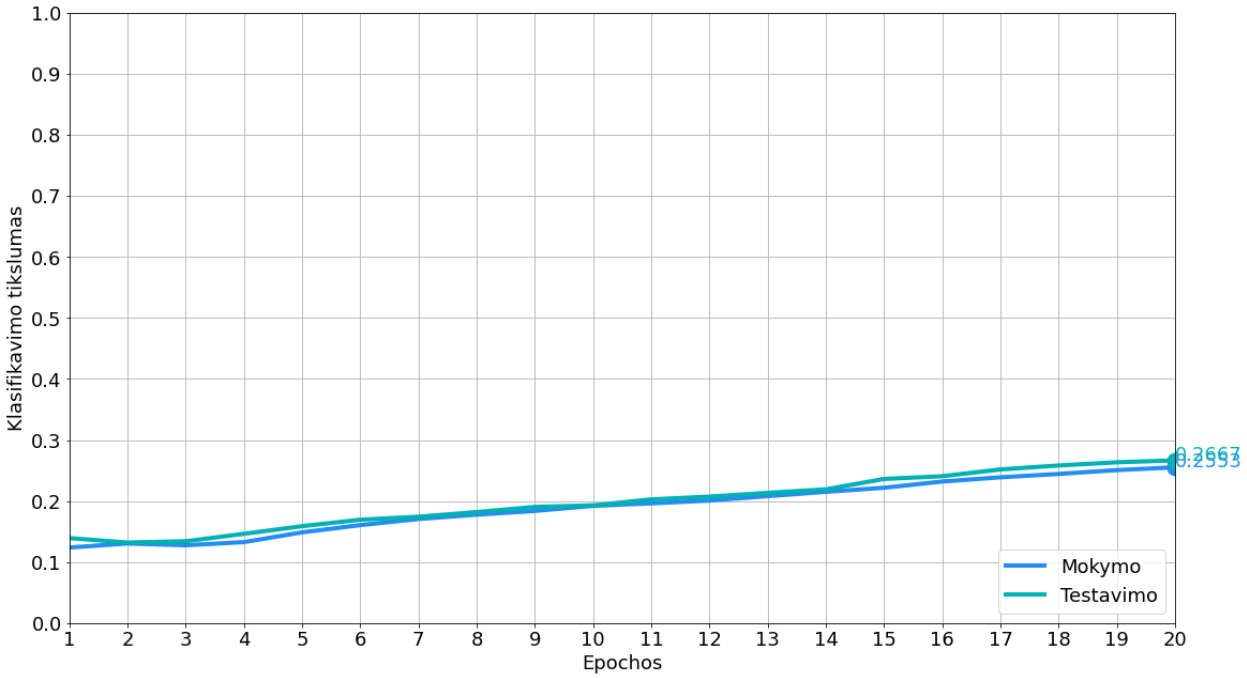
3.3. Optimizavimo algoritmai

Tirti šie algoritmai:

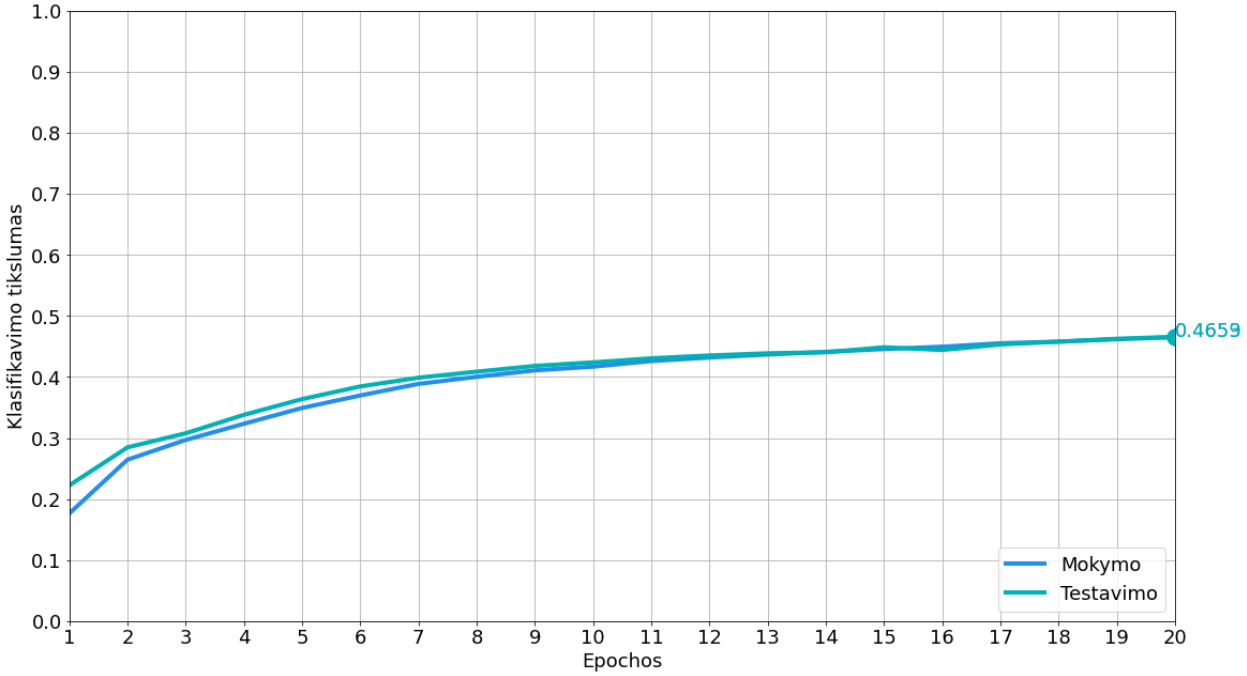
1. *adadelta*
2. *adagrad*
3. *adam*
4. *adamax*
5. *ftrl*
6. *nadam*
7. *rmsprop*
8. *sgd*

Rezultatai:

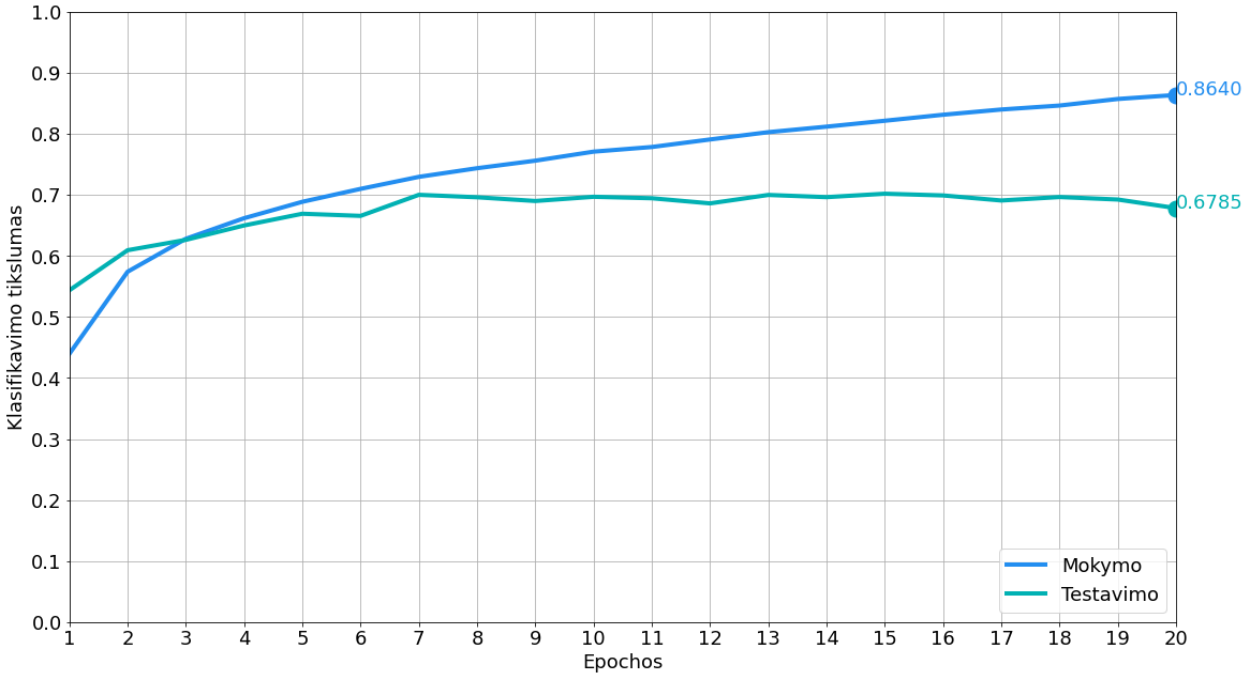
1. *adadelta*



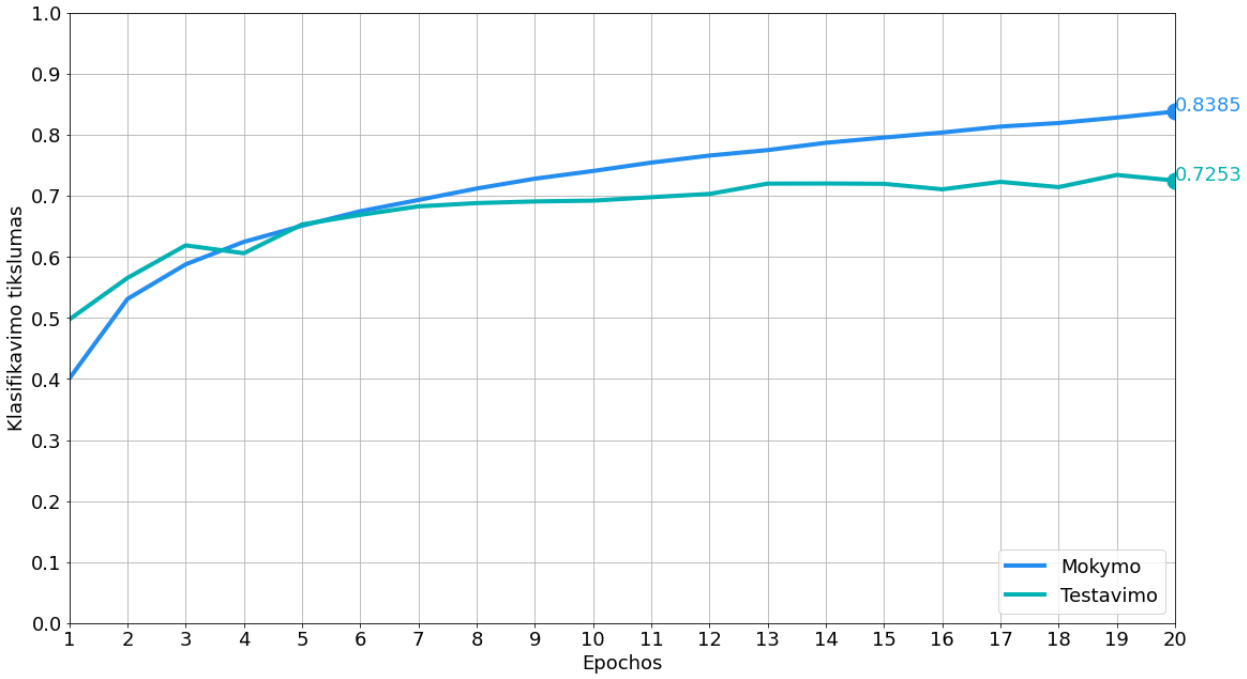
2. *adagrad*



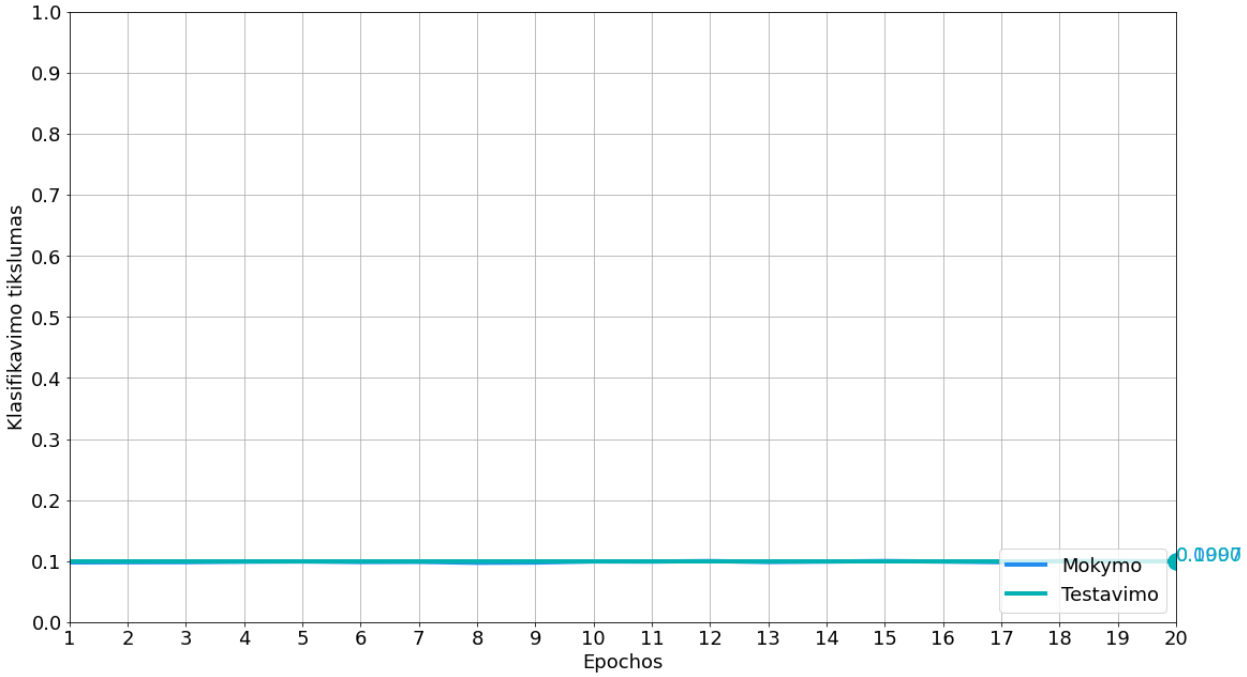
3. *adam*



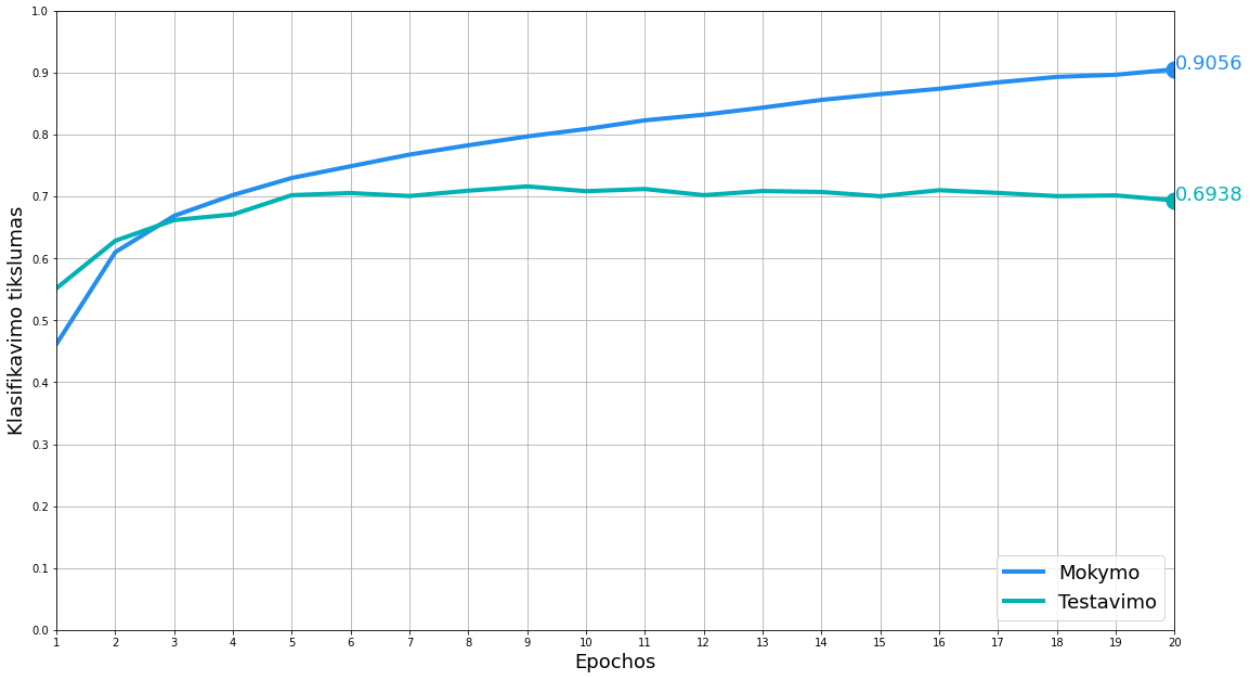
4. *adamax*



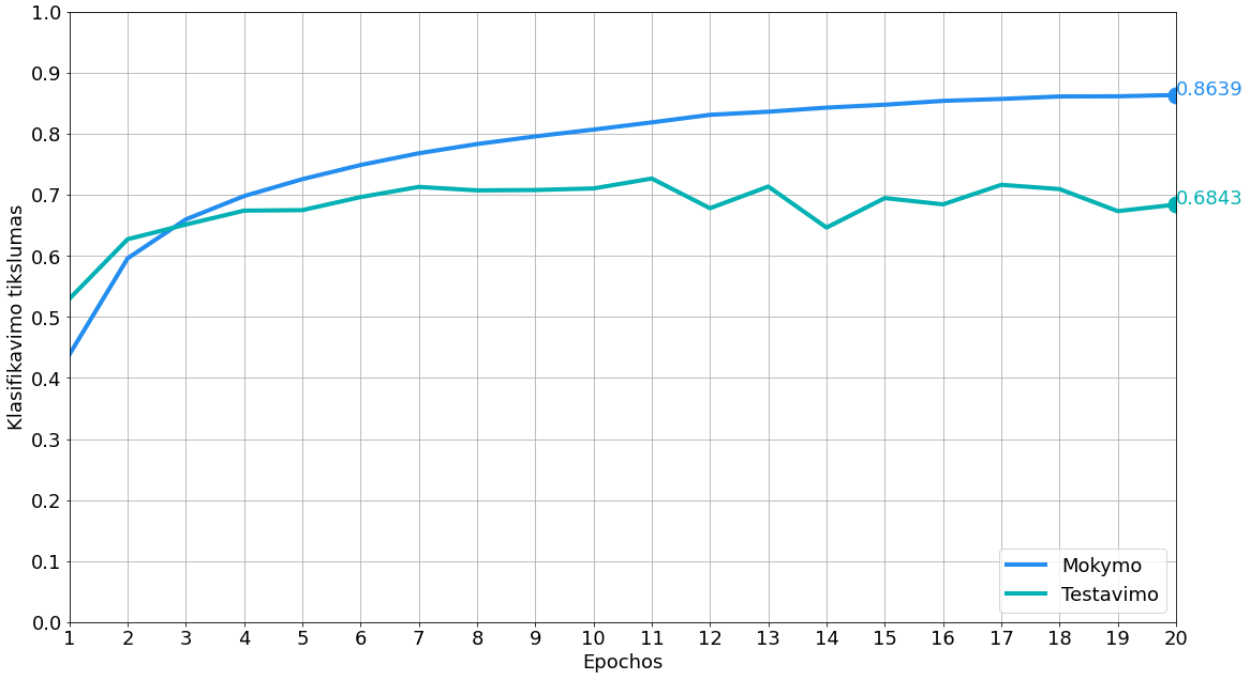
5. *ftrl*



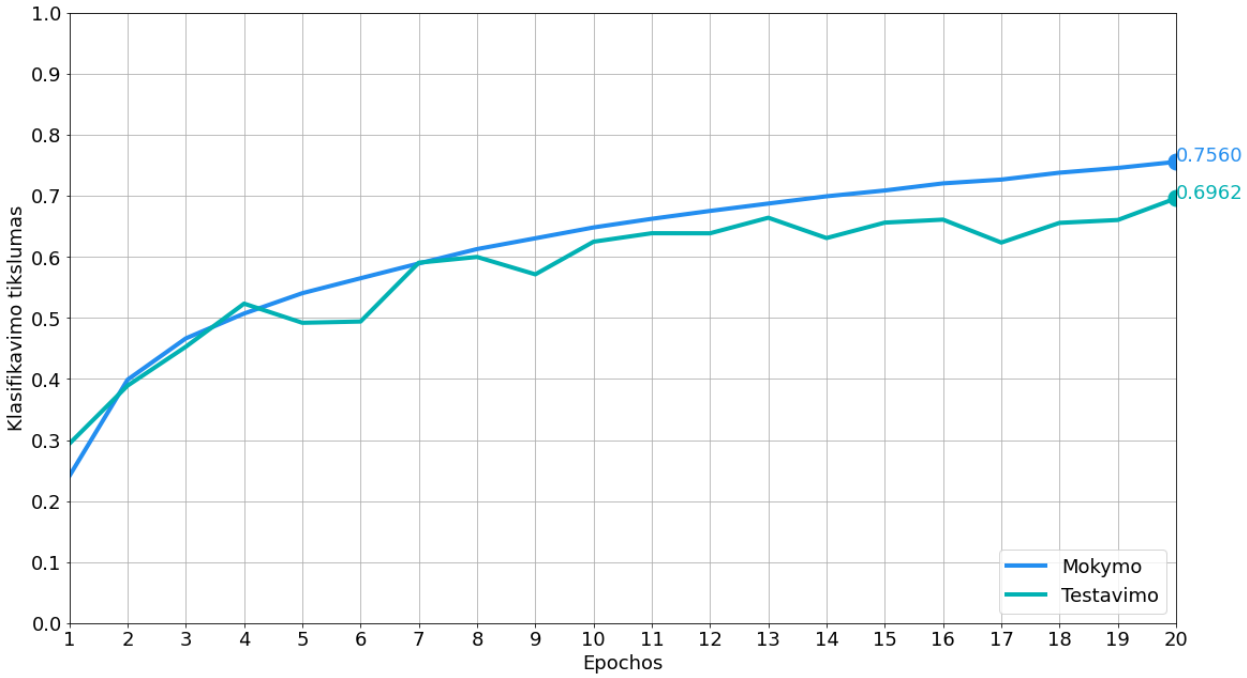
6. *nadam*



7. *rmsprop*



8. *sgd*



Interpretacija:

Mokymosi tikslumo prasme geriausiai suveikė *nadam* (0.9056), *adam* (0.8640), *rmsprop* (0.0839) ir *adamax* (0.8385) optimizavimo algoritmai, sgd davė kiek mažesnį tikslumą (0.7650).

Tuo tarpu testavimo tikslumas gavosi didžiausias *adamax* atveju (0.7253), tikslumą tarp 0.67–0.69 gavome su tais pačiais *nadam*, *adam* ir *rmsprop* algoritmais, bei prie jų dar prisidėjo *sgd*.

adadelta ir *adagrad* davė ryškiai mažesnius tikslumus, nors ir jų atveju neįvyko persimokymas per 20 epochų. Tokie palyginus maži tikslumai rodo, kad galbūt šie algoritmai ne visai teisingai pagal paskirtį buvo panaudoti šiame neuroniniame tinkle. Tuo tarpu *ftrl* atveju mokymasis visai nevyko.

Taip pat pastebime, kad *sgd* atveju algoritmas mažai persimokė, ir tai yra labai geras rezultatas. Dar labiau padidinus epochų skaičių galime tikėtis dar didesnio testavimo tikslumo, tad šiuo atveju mažesnis mokymo tikslumas palyginus su kitais minėtais algoritmais gali būti interpretuojamas kaip teigiamas dalykas.

adamax persimokymas, palyginus su *nadam*, *adam* ir *rmsprop* buvo ženkliai mažesnis.

3.4. Architektūros

Buvo tirtos kelios architektūros.

Jų hyperparametrų reikšmės buvo pastovios, tokios pat, kokios buvo ir anksčiau tyrime:

Aktyvacijos funkcija - *ReLU*
Paketo dydis - *32*
Optimizavimo algoritmas - *Adam*
Nuostolių funkcija - *Sparse Categorical Crossentropy*

Architektūra nr. 1
Tai - pirmoji, pagrindinė, jau pažįstama tyrime naudota architektūra, apibrėžta kodu:

```
#### Create the convolutional base
model.add(tf.keras.layers.Conv2D(32, (3, 3), activation=param_act_fn, input_shape=(32, 32, 3)))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Conv2D(64, (3, 3), activation=param_act_fn))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Conv2D(64, (3, 3), activation=param_act_fn))

#### Add Dense layers on top
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(64, activation=param_act_fn))
model.add(tf.keras.layers.Dense(10))
```

model.summary() išvestis:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 64)                65600
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
```

Architektūra nr. 2:

Šioje architektūroje vietoj trijų konvoliucijos sluoksnių buvo apibrėžtas tik vienas.

Programinis kodas:

```
: #### Create the convolutional base
  model.add(tf.keras.layers.Conv2D(32, (3, 3), activation=param_act_fn, input_shape=(32, 32, 3)))
  model.add(tf.keras.layers.MaxPooling2D((2, 2)))

  #### Add Dense layers on top
  model.add(tf.keras.layers.Flatten())
  model.add(tf.keras.layers.Dense(64, activation=param_act_fn))
  model.add(tf.keras.layers.Dense(10))
```

Tikėtina, kad ji duos prastesnius rezultatus.

Architektūra nr. 3:

Šioje architektūroje yra net 6 konvoliuciniai sluoksniai.

Taip pat, joje yra dar keli prieš tai nenaudoti sluoksniai:

BatchNormalization sluoksnis (paketo normalizacija), įterpiamas po kiekvienos konvoliucinio sluoksnio. Jis kiekvieną sykį normalizuoja konvoliucinio sluoksnio išvestį, siekiant palaikyti vidutinę reikšmę artimą 0, o standartinį nuokrypį artimą 1. Tai leidžia turėti didesnį mokymo greitį bei pagreitina neuroninio tinklo mokymąsi. Šioje architektūroje

Dropout sluoksnis, leidžiantis išvengti permokymo. Jo parametras - *Dropout* sluoksnio greitis (*rate*). Šis sluoksnis panaudojamas kiek vėliau - jis atsiranda du kartus prie pirmąjį ir prieš antrąjį tankųjį *Dense* sluoksnį. *Dropout* tam tikra prasme sluoksnis sumažina neuroninio tinklo dydį, padaro jį „plonesnį“.

Architektūros programinis kodas:

Set up architecture

In [17]:
```
y_train, y_test = train_labels.flatten(), test_labels.flatten()

# number of classes
K = len(set(y_train))
print("number of classes:", K)

from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, \
                                    Dropout, GlobalMaxPooling2D, MaxPooling2D, \
                                    BatchNormalization

#### Create the convolutional base
i = Input(shape=train_images[0].shape)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(i)
x = BatchNormalization()(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

#### Add Dense layers on top
x = Flatten()(x)
x = Dropout(0.2)(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(K, activation='softmax')(x)

model = tf.keras.models.Model(i, x)
```

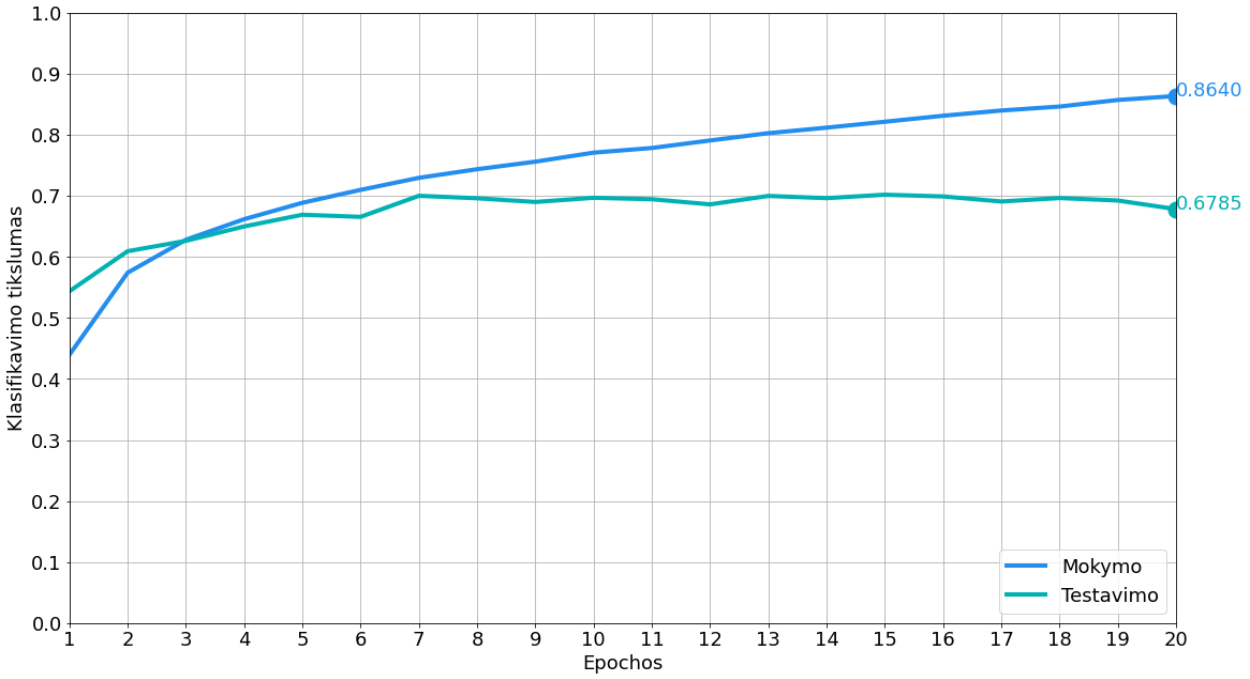
```
number of classes: 10
```

model.summary() išvestis:

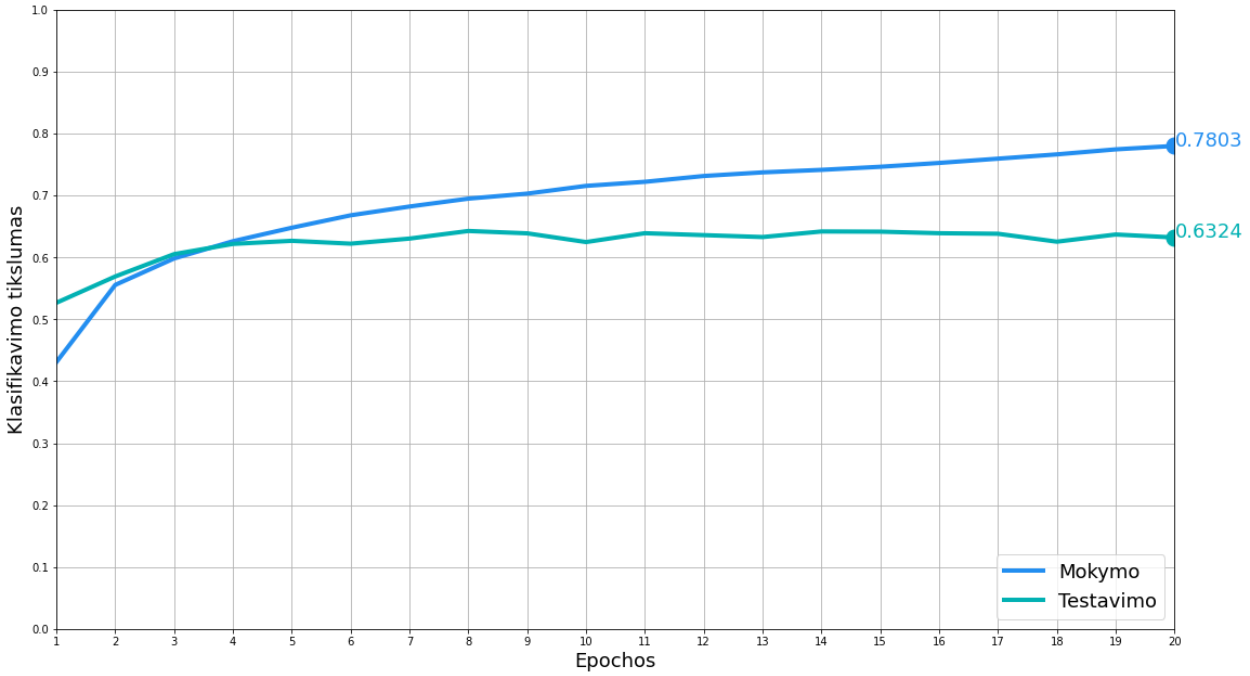
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
batch_normalization_4 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
batch_normalization_5 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dropout (Dropout)            (None, 2048)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              2098176
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                10250
=================================================================
Total params: 2,397,226
Trainable params: 2,396,330
Non-trainable params: 896
_________________________________________________________________
```

Rezultatai:

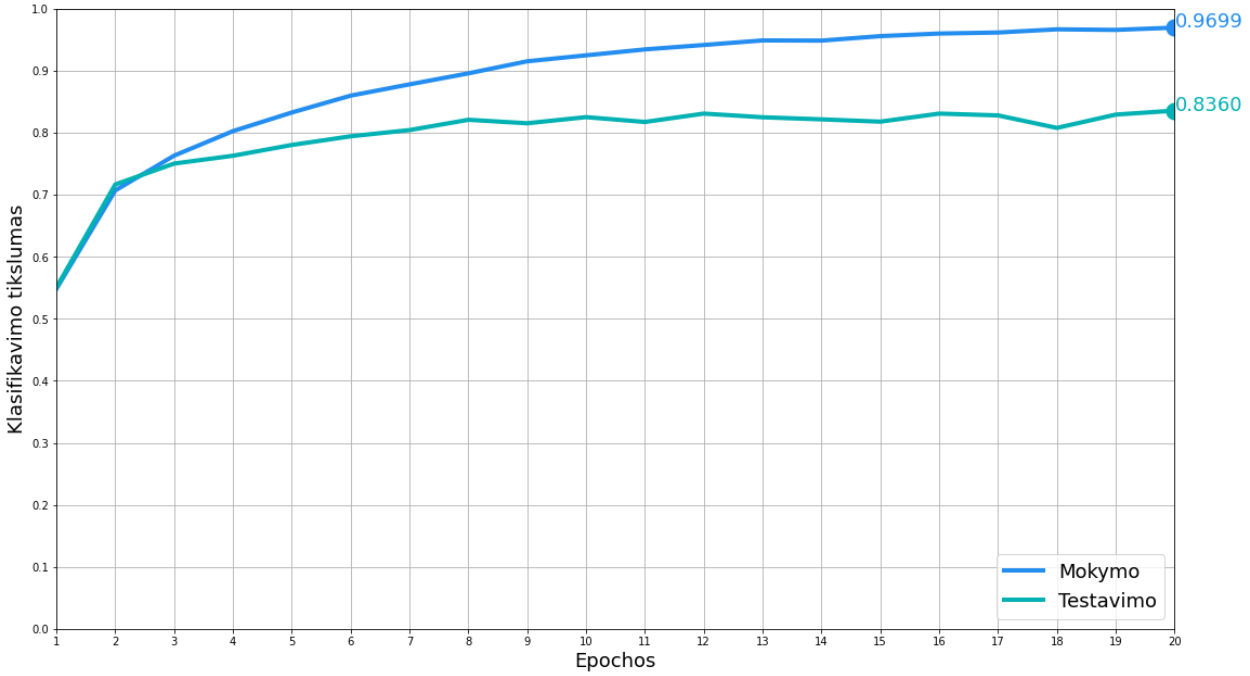
Architektūra nr. 1



Architektūra nr. 2



Architektūra nr. 3



Interpretacija:

Rezultatus gavome tokius, kokių ir galima bendrai buvo tikėtis, lyginant šias architektūras tarpusavyje. Konvoliucinis tinklas su vienu konvoliucijos sluoksniu (architektūra nr. 2) davė mažesnį tiek mokymosi, tiek testavimo tikslumą, nei trijų sluoksnių atveju (architektūra nr. 1). Tuo tarpu 6 konvoliucijos sluoksniai bei papildomi *BatchNormalization* ir *Dropout* sluoksniai (architektūra nr. 3) leido pasiekti žymiai didesnį mokymosi bei testavimo tikslumą, o tinklo persimokymas, matome, netgi yra kiek mažesnis nei 1-os architektūros atveju (lyginant mokymo tikslumo ir testavimo tikslumo skirtumą bei mokymosi kreivės statumą).

3.5. Geriausias atvejis

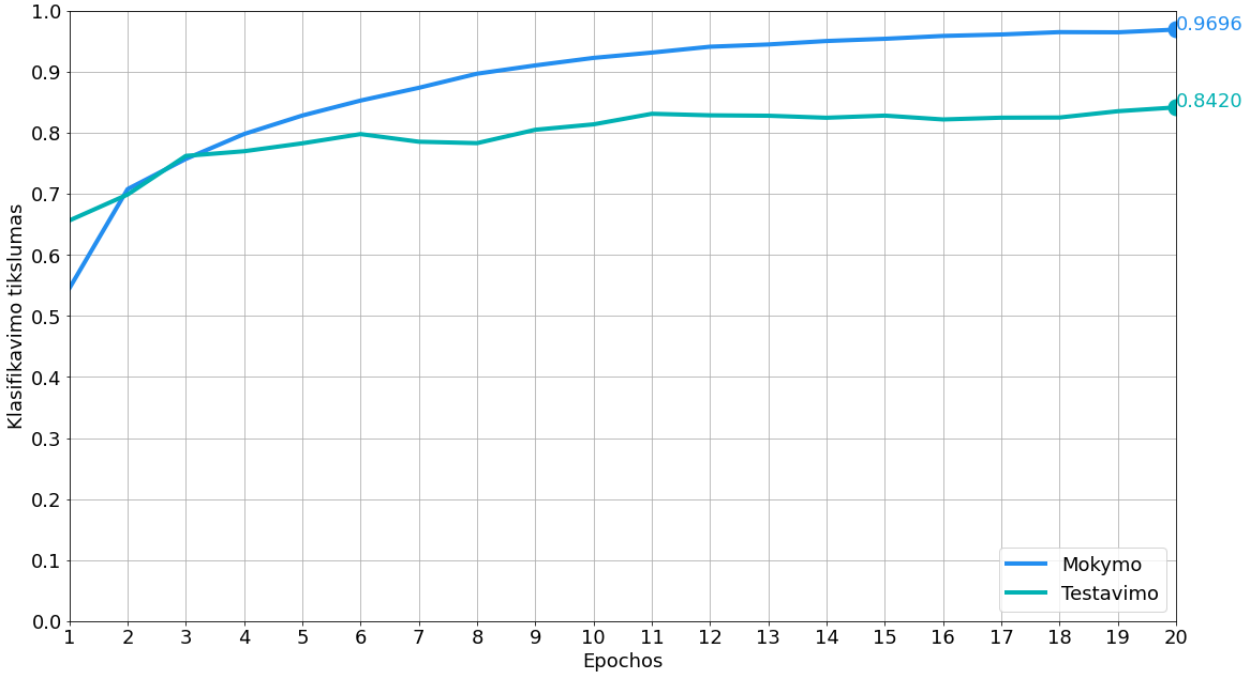
Taigi, geriausio atvejo architektūra yra 3-ioji. Jos programinis kodas ir santrauka jau aprašyta prieš tai.

Hyperparametrų keitimas būtent šiai architektūrai nėra pateiktas apraše. Hyperparametrai jai buvo keičiami, tačiau didelio skirtumo jie nedavė, o vienas iš geriausių atvejų ir buvo su jau pradžioje nustatytais parametrais, kuriems esant fiksuotiems buvo atlikti jau aprašyti tyrimai, tad šiuos parametrus ir naudosime kaip geriausio atveju parametrus, žinoma, su papildoma galimybe altikti nuodugnesnę hyperparametrų patikrą.

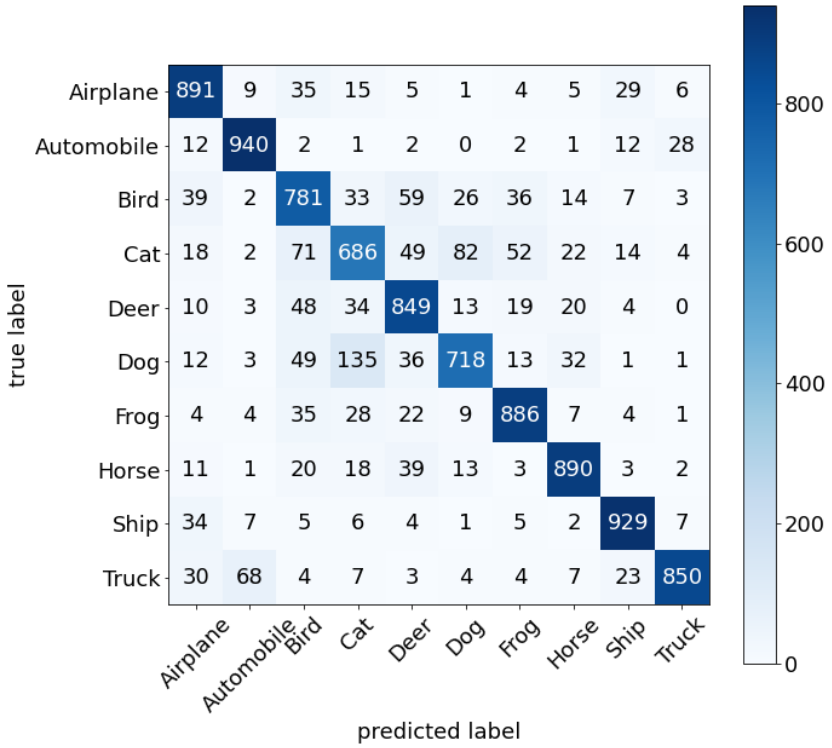
Aktyvacijos funkcija - *ReLU*
Paketo dydis - *32*
Optimizavimo algoritmas - *Adam*
Nuostolių funkcija - *Sparse Categorical Crossentropy*

Geriausias atvejis detaliai pateiktas A.3 priede.

Jo grafikas:



Klasifikavimo matrica testavimo duomenims:



Taip pat, iš testavimo aibės buvo parinkti 30 įrašų iš visų klasių, ir pateikti jų klasifikavimo rezultatai - kokias klases nustatė neuroninis tinklas, ir kokios klasės yra iš tikrųjų:


[16]
Correct class: Ship (#8)
Predicted class: Ship(#8)
[21]
Correct class: Horse (#7)
Predicted class: Horse(#7)
[26]
Correct class: Bird (#2)
Predicted class: Bird(#2)
[17]
Correct class: Dog (#5)
Predicted class: Dog(#5)
[22]
Correct class: Airplane (#0)
Predicted class: Airplane(#0)
[27]
Correct class: Deer (#4)
Predicted class: Deer(#4)
[18]
Correct class: Horse (#7)
Predicted class: Horse(#7)
[23]
Correct class: Deer (#4)
Predicted class: Deer(#4)
[28]
Correct class: Airplane (#0)
Predicted class: Airplane(#0)
[19]
Correct class: Ship (#8)
Predicted class: Ship(#8)
[24]
Correct class: Truck (#9)
Predicted class: Truck(#9)
[29]
Correct class: Truck (#9)
Predicted class: Truck(#9)
[20]
Correct class: Frog (#6)
Predicted class: Frog(#6)
[25]
Correct class: Dog (#5)
Predicted class: Deer(#4)
[30]
Correct class: Frog (#6)
Predicted class: Frog(#6)

Pateikiamos ir šių klasių klasifikavimo tikimybės (prognozės):

```
Output probabilities:

[1]:

[0.0853 0.0853 0.0853 0.232  0.0853 0.0853 0.0853 0.0853 0.0853 0.0853]


[2]:

[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853]


[3]:

[0.0854 0.0855 0.0854 0.0854 0.0854 0.0854 0.0854 0.0854 0.2317 0.0854]


[4]:

[0.2162 0.0892 0.0866 0.0868 0.0863 0.0863 0.0863 0.0863 0.0874 0.0886]


[5]:

[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853 0.0853 0.0853]


[6]:

[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853 0.0853 0.0853]


[7]:

[0.0854 0.2304 0.0854 0.0854 0.0854 0.0854 0.0854 0.0854 0.0854 0.0861]


[8]:

[0.0856 0.0856 0.087  0.0856 0.0856 0.0856 0.2285 0.0856 0.0856 0.0856]


[9]:

[0.0853 0.0853 0.0853 0.232  0.0853 0.0853 0.0853 0.0853 0.0853 0.0853]


[10]:

[0.0853 0.232  0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853]
```

```
[11]:
[0.1849 0.0879 0.0888 0.1073 0.0892 0.0883 0.0881 0.0879 0.0894 0.0882]

[12]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232 ]

[13]:
[0.086  0.086  0.0866 0.0872 0.0888 0.2215 0.086  0.086  0.086  0.086 ]

[14]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853 0.0853]

[15]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232 ]

[16]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853]

[17]:
[0.0855 0.0855 0.0855 0.0867 0.0855 0.2292 0.0855 0.0855 0.0855 0.0855]

[18]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853 0.0853]

[19]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853]

[20]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853 0.0853 0.0853]
```

```
[21]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853 0.0853]

[22]:
[0.1897 0.0876 0.1098 0.0876 0.0876 0.0876 0.0876 0.0876 0.0876 0.0876]

[23]:
[0.0886 0.0885 0.1371 0.0885 0.155  0.0885 0.0885 0.0885 0.0885 0.0885]

[24]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232 ]

[25]:
[0.0853 0.0853 0.0853 0.0853 0.232  0.0853 0.0853 0.0853 0.0853 0.0853]

[26]:
[0.0858 0.0858 0.2243 0.0858 0.0882 0.0858 0.0868 0.0858 0.0858 0.0858]

[27]:
[0.0865 0.0865 0.0865 0.0952 0.2125 0.0869 0.0865 0.0865 0.0865 0.0865]

[28]:
[0.232  0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853]

[29]:
[0.0853 0.0854 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.2319]

[30]:
[0.0853 0.0853 0.0853 0.0853 0.0853 0.0853 0.232  0.0853 0.0853 0.0853]
```

4. Išvados

Buvo sukurtas ir apmokytas konvoliucinis neuroninis tinklas vaizdams klasifikuoti, bei atliktas tyrimas, kaip neuroninio tinklo rezultatai priklauso nuo tinklo architektūros ir nuo hyperparametrų reikšmių.

Pamatėme, kad daugiau konvoliucijų sluoksnių duoda geresnį rezultatą, bent jau kol tas sluoksnių skaičius nėra per didelis, bent jau šiame tyrime sluoksnių kaita nuo 1 iki 3, ir nuo 3 iki 6 davė žymius pokyčius. Taip pat geriausius rezultatus davusi architektūra nuo kitų skyrėsi ir papildomais *BatchNormalization* ir *Dropout* sluoksniais. Taigi, matome, kad kuriant konvoliucinio neuroninio tinklo architekūrą, žinoma, neapsiribojančia vien konvoliucijos sluoksniais, reikalingi tam tikri jos projektavimo sprendimai, kurie parenkami tiek prieš pradedant mokyti tinklą, tiek koreguojami ir keičiami tinklo mokymosi metu.

Tiriant aktyvacijos funkcijas, pamatėme, kad *linear unit* funkcijų grupė (*elu, gelu, relu, selu*) davė gana panašų ir geriausią rezultatą mūsų uždaviniui. Tačiau taip pat verta eksperimentuoti ir su kitomis, pvz. *swish* taip pat davė gerą rezultatą, o dauguma kitų tirtų funkcijų irgi nežymiai atsilieka.

Testuojant skirtingus paketo dydžius, bent jau šiame uždavinyje pamatėme, kad vos po kelis duomenis viename pakete pateikus gaunami daug ilgesnį (nuo kelių iki dešimties kartų) tinklo apmokymo laiką, o ir rezultatai šio tyrimo atveju buvo netgi geresni su didesniais paketų dydžiais (24, 32, 64, 128). Taip pat, padidinus paketų dydį jau iki 512, kuomet 50 000 mokymo duomenų buvo padalinta į vos 100 paketų, rezultatas pastebimai pradėjo prastėti. Nors persimokymas tokiu atveju smarkiai sumažėja arba jo nebėra, testavimo tikslumas yra žymiai mažesnis.

Taip pat labai aiškiai galima buvo pastebėti, kaip tinklas persimoko, ir, kadangi buvo tiriama dažniausiai apmokius tinklą 20 epochų, didesnėje grafiko dalyje mokymo ir testavimo tikslumai labiau skyrėsi, nei sutapo, ir tas skirtumas didėjo sulig epochų skaičiumi. Tačiau keletas bandytų variantų vis tik davė smarkiai mažesnį persimokymą (*adamax* optimizavimo algoritmas), arba beveik jokio (*sgd* algoritmas). Testuojant kitus optimizavimo algoritmus, pamatėme, kad labiau persimokantys ir šiek tiek mažesnį testavimo tikslumą duodantys nadam ir nadam užtat duoda aukštą mokymo tikslumą.

Taigi, kuriant ir koreguojant neuroninį tinklą reikalinga ir jo konfigūracija eksperimento būdu, jo architektūros išsirinkimas, testavimas, hyperparametrų priderinimas. Šiame darbe buvo progos ir pasinaudoti debesijos sprendimais, padedančios sparčiau apmokyti neuroninį tinklą, bei TensorFlow ir Keras programavimo bibliotekomis, suteikiančiomis priemones greitai ir nesudėtingai atlikinėti pakeitimus kuriant ir testuojant neuroninį tinklą.

A Priedai

A.1.1 **`cnn_cifar10.ipynb`** – programos failas su išsamiais komentarais ir pilnomis išvestimis (įskaitant klasifikavimo matricą, prognozuotas tikimybes, dalį klasifikavimo rezultatų).

A.1.2 **`cnn_cifar10.html ir cnn_cifar10_files/`** – į *HTML* formatą konvertuotas A1.1 priedas.

A.1.3 **`code_authorship/`** – direktorija su A1.1 priedo paveiksliukais, kuriuose žalia spalva pažymėtas kodas ir komentarai nurodo tas vietas, kurios buvo mano pakeistos ar savarankiškai atliktos. Nepažymėti blokai nurodo nuoseklias programos dalis, panaudotas be didesnio pakeitimo, ir todėl tikėtina rasti nuoseklias ir šaltiniuose.

Naudoti programos šaltiniai:

https://www.tensorflow.org/tutorials/images/cnn
https://www.tensorflow.org/tutorials/quickstart/beginner
https://www.tensorflow.org/tutorials/keras/classification

A.1.4 **`cnn_cifar10_changeable_params.ipynb`**
(kartu su **`cnn_cifar10_changeable_params.html`**) – modifikuotas programos failas su galimybe keisti hyperparametrus.

A.1.5 **`cnn_cifar10_changeable_params_best_case.ipynb`** (kartu su **`cnn_cifar10_changeable_params_best_case.html`**) – programos failas, kuriame yra geriausias šio tyrimo atvejis, įskaitant neuroninio tinklo architektūra bei hyperparametrus, taip pat detaliai atvaizduotus rezultatus.

A.2.1 **`experiment_results.txt`** – Visų eksperimento rezultatų duomenys, pagal kuriuos buvo gauti tyrime aprašyti grafikai.

A.2.2. Aplankai su eksperimentų grafikais (sudėti į vieną aplanką **experiment_results/**):

activation_functions/
batch_sizes/
optimization_algorithms/
architectures/

A.3 best_case/ – aplankas su geriausio varianto kodu, jo klasifikavimo matrica, bei jo nustatytų 30-ies pasirinktų pavyzdinių paveiksliukių klasėmis ir jų tikimybėmis.