ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ №3
З ДИСЦИПЛІНИ «СТАТИСТИЧНІ АЛГОРИТМИ НАВЧАННЯ»
ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ

Виконав студент 2к. маг. Ломако Олександр

Варіант 9 => 3

В третій лабораторній роботі матимемо справу з MNIST database (*Modified National Institute of Standards and Technology database*) – датасетом, що містить зразки рукописного написання цифр.

Використовуватимемо датасет mnist з бібліотеки keras. Оскільки на моєму комп'ютері присутня сучасна відеокарта (Radeon RX 570), використовуватимемо в тому числі і її потужності.

```
> install_keras(tensorflow = 'gpu')
> # підключаємо бібліотеки
> library(keras)
> library(tensorflow)
```

Спершу, завантажимо датасет (розмір приблизно 220 МБ).

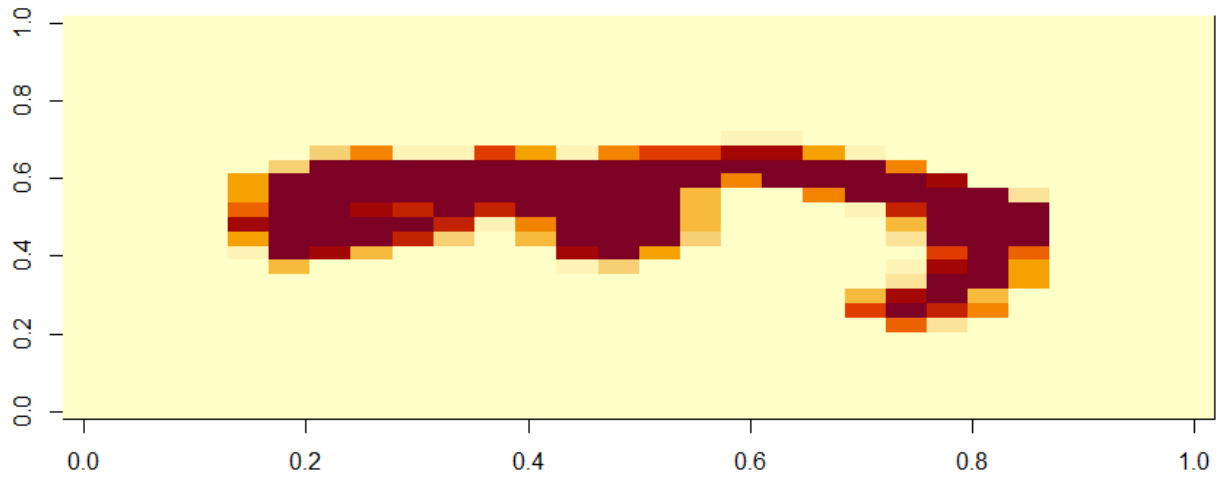
```
> # завантажуємо датасет
> data <- dataset_mnist()
```

Даний датасет вже розбитий на тренувальну та тестову частини, і в ньому всі зображення є закодованими. Проте, на даний момент всі ці вибірки знаходяться в змінній типу list, тому витягнемо їх звідти.

```
> c(c(x_train, y_train), c(x_test, y_test)) %<-% data # розбиваємо на тренува
льну і тестову частини
```

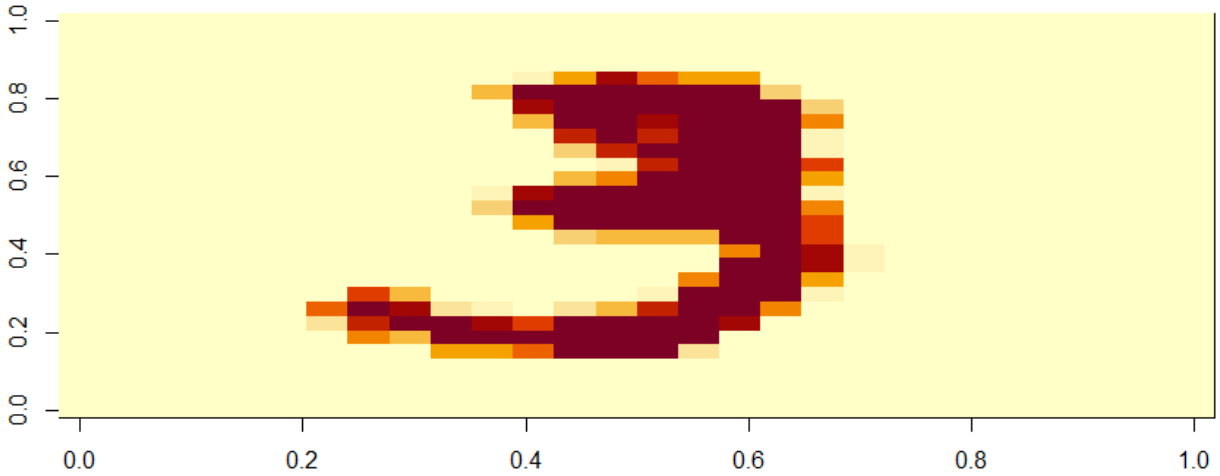
Спробуємо поглянути на який-небудь приклад.

```
> # виведемо для прикладу яке-небудь зображення
> image(x_train[11,,])
```



Здається, це перевернута трійка. Реалізуємо функцію, що «розвертатиме» зображення (в нашому випадку транспонуватиме матриці закодованих зображень).

```
> # функція, що розвертатиме зображення
> plotImage = function(im){
+   image(t(apply(im, 2, rev)))
+ }
> plotImage(x_train[11,,])
```



Так, тепер виглядає більш правдоподібно.

1. Для подальшої роботи з даними треба перевести кодування (пікселі) в значення діапазону $(0; 1)$. Зробити це легко, врахувавши, що при кодуванні кожен піксель здобував значення від 0 до 255.

```
> # переведемо закодовані пікселі в діапазон від 0 до 1
> x_train = array_reshape(x_train, c(60000, 28, 28, 1))
> x_train = x_train / 255
>
> x_test  = array_reshape(x_test, c(10000, 28, 28, 1))
> x_test  = x_test / 255
```

Також для входу в нейронну мережу маємо підготувати і відгук шляхом його кодування у one-hot вектори.

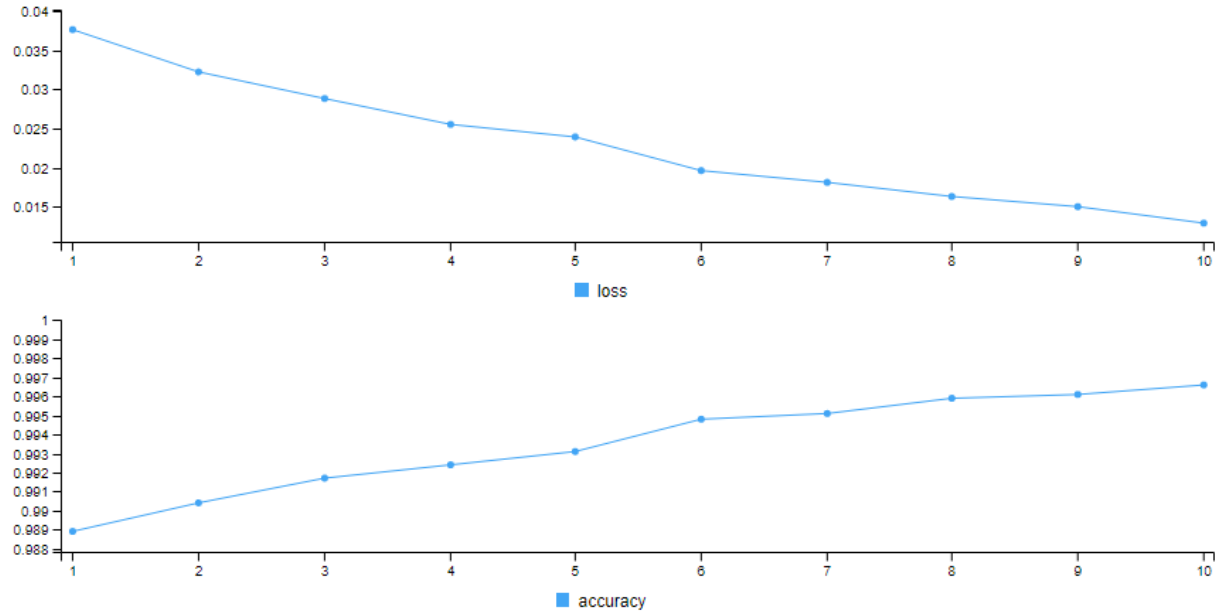
```
> # закодуємо відгук у one-hot вектори
> y_train = to_categorical(y_train)
> y_test  = to_categorical(y_test)
```

2. Спершу побудуємо звичайну щільну нейронну мережу. Проте цього разу, використовуватимемо вбудований функціонал бібліотеки keras.

Архітектура даної нейронної мережі буде наступною: 3 шари, на вхідному 784 нейрони, 1 прихований шар із 64 нейронами, і останній шар відгуків – 10 нейронів (як, власне кажучи, і цифр).

```
> # спершу реалізуємо щільну нейронну мережу
>
> denseModel = keras_model_sequential() %>%
+   layer_flatten(input_shape = c(28,28,1)) %>%
+   layer_dense(units = 64, activation = "relu") %>%
+   layer_dense(units = 10, activation = "softmax")
>
>
> denseModel %>% compile(
+   optimizer = "adam",
+   loss = "categorical_crossentropy",
+   metrics = c("accuracy")
```

```
+ )
>
> denseModel %>% fit(
+   x_train, y_train,
+   epochs = 10, batch_size=64
+ )
Epoch 1/10
938/938 [==============================] - 2s 2ms/step - loss: 0.3462 - accur
acy: 0.9046
Epoch 2/10
938/938 [==============================] - 1s 1ms/step - loss: 0.1733 - accur
acy: 0.9502
Epoch 3/10
938/938 [==============================] - 1s 1ms/step - loss: 0.1292 - accur
acy: 0.9622
Epoch 4/10
938/938 [==============================] - 1s 1ms/step - loss: 0.1042 - accur
acy: 0.9700
Epoch 5/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0866 - accur
acy: 0.9747
Epoch 6/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0729 - accur
acy: 0.9785
Epoch 7/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0628 - accur
acy: 0.9814
Epoch 8/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0539 - accur
acy: 0.9843
Epoch 9/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0481 - accur
acy: 0.9857
Epoch 10/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0422 - accur
acy: 0.9876
>
> fit(denseModel, x_train, y_train, epochs=10, batch_size=64)
Epoch 1/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0376 - accur
acy: 0.9889
Epoch 2/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0322 - accur
acy: 0.9904
Epoch 3/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0288 - accur
acy: 0.9917
Epoch 4/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0255 - accur
acy: 0.9924
Epoch 5/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0239 - accur
acy: 0.9931
Epoch 6/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0196 - accur
acy: 0.9948
Epoch 7/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0181 - accur
acy: 0.9951
Epoch 8/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0163 - accur
acy: 0.9959
Epoch 9/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0150 - accur
acy: 0.9961
Epoch 10/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0129 - accur
acy: 0.9966
```



Бачимо, що на тренувальних даних вдалося досягнути точності класифікації в 99.66% (тобто, частота помилки 0.34%). Спробуємо тепер застосувати цю мережу на тестових даних.

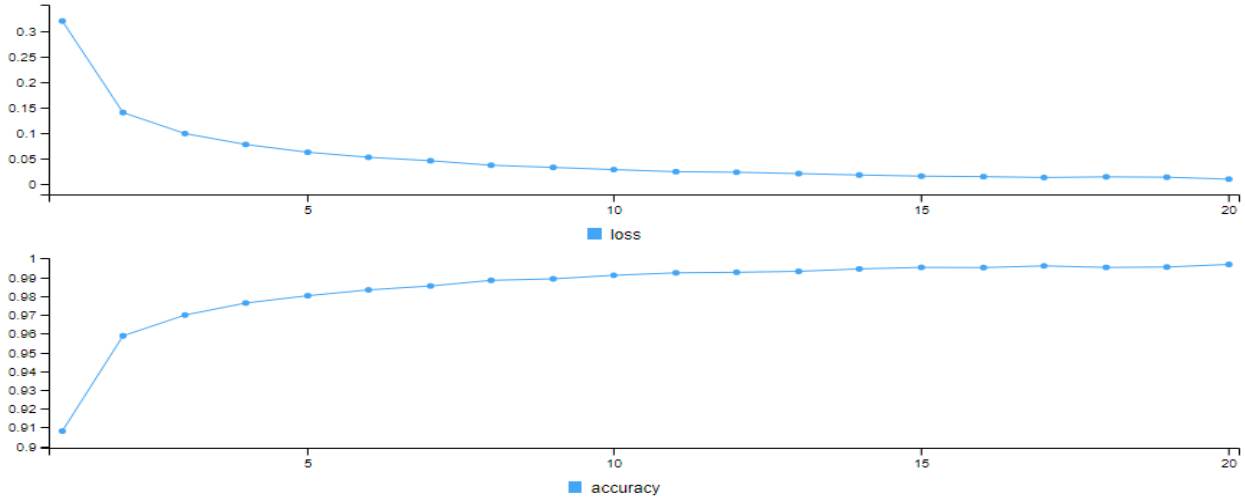
```
> # перевіримо на тестових даних
> result = evaluate(denseModel, x_test, y_test)
313/313 [==============================] - 0s 467us/step - loss: 0.0997 - acc
uracy: 0.9760
> result
      loss   accuracy
0.09974429 0.97600001
```

Можемо бачити, що на тестових даних тепер точність класифікації становить лише 97.6%, що на 2% менше, ніж на тренувальних. Існує припущення, що, можливо, тут мало місце бути перенавчання.

Спробуємо покращити модель: додамо ще один прихований шар із 64 нейронами, і натренуємо модель на тренувальних даних.

```
> # додамо до нейронної мережі ще один прихований шар
>
> denseModel = keras_model_sequential() %>%
+   layer_flatten(input_shape = c(28,28,1)) %>%
+   layer_dense(units = 64, activation = "relu") %>%
+   layer_dense(units = 64, activation = "relu") %>%
+   layer_dense(units = 10, activation = "softmax")
>
>
> compile(denseModel, optimizer="adam", loss="categorical_crossentropy", metr
ics=c("accuracy"))
>
> fit(denseModel,
+     x_train, y_train,
+     epochs=20, batch_size=64)
Epoch 1/20
938/938 [==============================] - 1s 1ms/step - loss: 0.3189 - accur
acy: 0.9080
Epoch 2/20
938/938 [==============================] - 1s 1ms/step - loss: 0.1396 - accur
acy: 0.9588
Epoch 3/20
```

```
938/938 [==============================] - 1s 1ms/step - loss: 0.0984 - accur
acy: 0.9699
Epoch 4/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0770 - accur
acy: 0.9763
Epoch 5/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0618 - accur
acy: 0.9802
Epoch 6/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0519 - accur
acy: 0.9833
Epoch 7/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0452 - accur
acy: 0.9854
Epoch 8/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0363 - accur
acy: 0.9884
Epoch 9/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0321 - accur
acy: 0.9892
Epoch 10/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0278 - accur
acy: 0.9911
Epoch 11/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0237 - accur
acy: 0.9924
Epoch 12/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0226 - accur
acy: 0.9927
Epoch 13/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0200 - accur
acy: 0.9932
Epoch 14/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0172 - accur
acy: 0.9945
Epoch 15/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0150 - accur
acy: 0.9953
Epoch 16/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0140 - accur
acy: 0.9952
Epoch 17/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0123 - accur
acy: 0.9961
Epoch 18/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0136 - accur
acy: 0.9953
Epoch 19/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0129 - accur
acy: 0.9955
Epoch 20/20
938/938 [==============================] - 1s 1ms/step - loss: 0.0091 - accur
acy: 0.9969
```



Можемо бачити, що на тренувальних даних суттєвого покращення ми не досягнули: наразі точність становить 99.69%.

Застосуємо покращену модель до тестових даних.

```
> # перевіримо дану модель на тестових даних
> result = evaluate(denseModel, x_test, y_test)
313/313 [==============================] - 0s 436us/step - loss: 0.1146 - acc
uracy: 0.9759
> result
     loss  accuracy
0.1146068 0.9759000
```

Тут маємо навпаки гірший результат, ніж в мережі з 1 прихованим шаром: тепер точність класифікації 97.5%, що менше, ніж для минулої щільної нейронної мережі.

3. Тепер побудуємо згорткову нейронну мережу з вхідним шаром, трьома шарами згортки і двома шарами пулінгу. На першому згортковому шарі застосуємо в якості активаційної функції SELU (Scaled Exponential Linear Unit), на решті двох шарах – ReLU (Rectified Linear Unit).

```
> # побудуємо тепер згорткову мережу. Скроситаємось functional API
>
> inputs = layer_input(shape=c(28,28,1))
> z = layer_conv_2d(inputs, filters=32, kernel_size=c(3,3), activation="selu"
)
> z = layer_max_pooling_2d(z, pool_size = c(2, 2))
> z = layer_conv_2d(z, filters = 64, kernel_size = c(3, 3), activation = "rel
u")
> z = layer_max_pooling_2d(z, pool_size = c(2, 2))
> z = layer_conv_2d(z, filters = 64, kernel_size = c(3, 3), activation = "rel
u")
> outputs = z
> model = keras_model(inputs, outputs)
```

Додамо далі останній шар для, власне кажучи, класифікації.

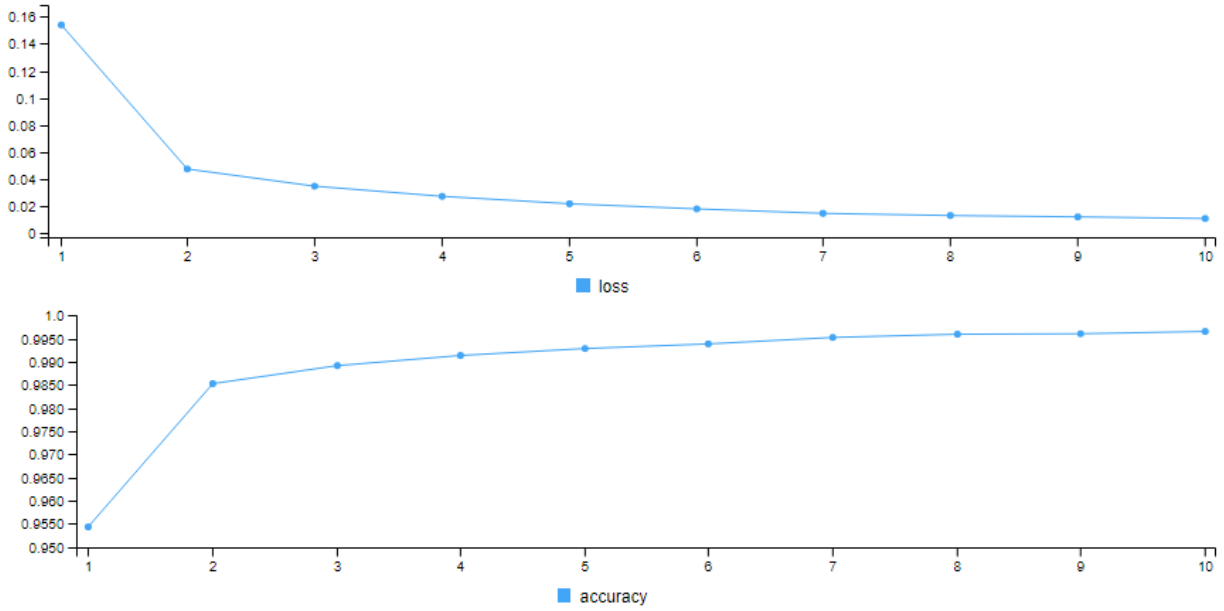
```
> ## Додамо останній шар для класифікації
> z = layer_flatten(z)
> z = layer_dense(z, units = 64, activation = "selu")
> outputs = layer_dense(z, units = 10, activation = "softmax")
> model = keras_model(inputs, outputs)
```

Активаційною функцією останнього шару буде softmax, оскільки маємо справу з багатокласовою класифікацією.

Тепер проведемо процес навчання. Задамо кількість епох рівною 10, а параметр batch size = 64.

```
> compile(model, optimizer="adam", loss="categorical_crossentropy", metrics=c
("accuracy"))
> fit(model, x_train, y_train, epochs=10, batch_size=64)
Epoch 1/10
938/938 [==============================] - 19s 20ms/step - loss: 0.1539 - acc
uracy: 0.9543
Epoch 2/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0474 - acc
uracy: 0.9853
Epoch 3/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0347 - acc
uracy: 0.9892
Epoch 4/10
```

```
938/938 [==============================] - 19s 21ms/step - loss: 0.0272 - acc
uracy: 0.9914
Epoch 5/10
938/938 [==============================] - 21s 22ms/step - loss: 0.0217 - acc
uracy: 0.9929
Epoch 6/10
938/938 [==============================] - 20s 21ms/step - loss: 0.0179 - acc
uracy: 0.9939
Epoch 7/10
938/938 [==============================] - 22s 23ms/step - loss: 0.0146 - acc
uracy: 0.9953
Epoch 8/10
938/938 [==============================] - 22s 23ms/step - loss: 0.0130 - acc
uracy: 0.9960
Epoch 9/10
938/938 [==============================] - 22s 24ms/step - loss: 0.0120 - acc
uracy: 0.9961
Epoch 10/10
938/938 [==============================] - 23s 24ms/step - loss: 0.0108 - acc
uracy: 0.9966
```



Маємо на тренувальних даних точність в більше, ніж 99.5%. Перевіримо на тестових даних.

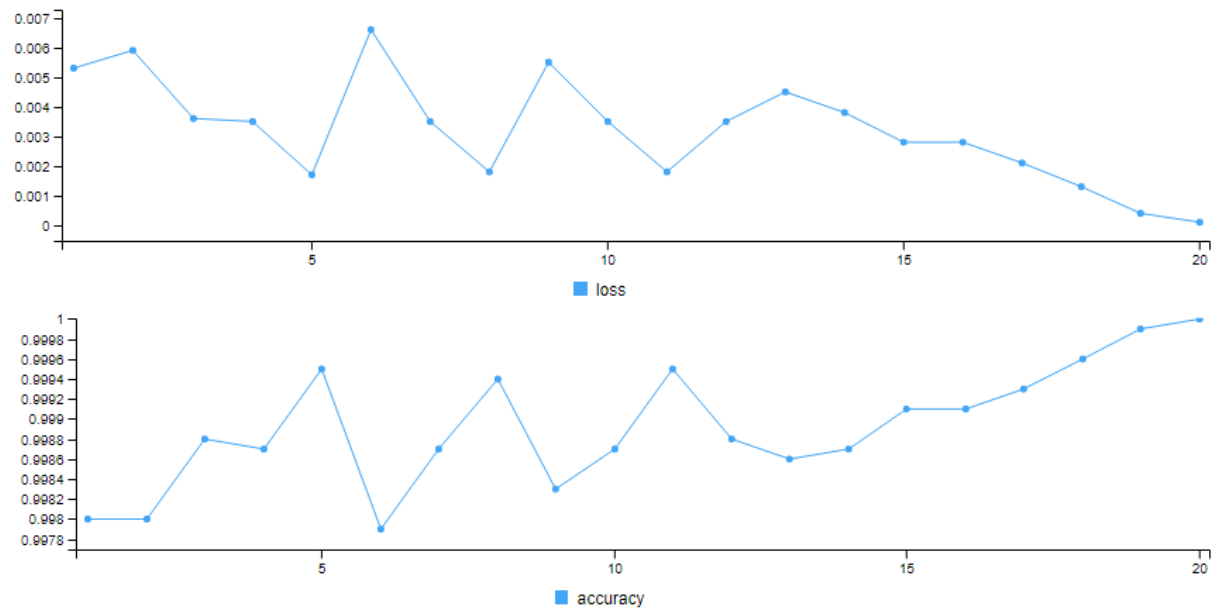
```
> results = model %>% evaluate(x_test, y_test)
313/313 [==============================] - 1s 3ms/step - loss: 0.0332 - accur
acy: 0.9920
> results
      loss   accuracy
0.03320665 0.99199998
```

Отже, маємо точність роботи класифікатора на рівні 99.2%, що на відміну від щільної нейронної мережі, не так значно відрізняється від отриманої точності на тренувальних даних, тому в даному випадку перенавчання не мало місця бути.

4. Спробуємо тепер покращити отриманий результат, погравшись з параметрами. Спершу, наприклад, для тієї ж самої мережі, виконаємо процес навчання на 20 епохах з параметром batch_size = 128.

```
> compile(model, optimizer="adam", loss="categorical_crossentropy", metrics=c
("accuracy"))
```

```
> fit(model, x_train, y_train, epochs=20, batch_size=128)
Epoch 1/20
469/469 [==============================] - 19s 40ms/step - loss: 0.0053 - acc
uracy: 0.9980
Epoch 2/20
469/469 [==============================] - 19s 40ms/step - loss: 0.0059 - acc
uracy: 0.9980
Epoch 3/20
469/469 [==============================] - 19s 40ms/step - loss: 0.0036 - acc
uracy: 0.9988
Epoch 4/20
469/469 [==============================] - 19s 40ms/step - loss: 0.0035 - acc
uracy: 0.9987
Epoch 5/20
469/469 [==============================] - 19s 40ms/step - loss: 0.0017 - acc
uracy: 0.9995
Epoch 6/20
469/469 [==============================] - 20s 42ms/step - loss: 0.0066 - acc
uracy: 0.9979
Epoch 7/20
469/469 [==============================] - 20s 42ms/step - loss: 0.0035 - acc
uracy: 0.9987
Epoch 8/20
469/469 [==============================] - 21s 44ms/step - loss: 0.0018 - acc
uracy: 0.9994
Epoch 9/20
469/469 [==============================] - 19s 41ms/step - loss: 0.0055 - acc
uracy: 0.9983
Epoch 10/20
469/469 [==============================] - 19s 40ms/step - loss: 0.0035 - acc
uracy: 0.9987
Epoch 11/20
469/469 [==============================] - 19s 41ms/step - loss: 0.0018 - acc
uracy: 0.9995
Epoch 12/20
469/469 [==============================] - 19s 41ms/step - loss: 0.0035 - acc
uracy: 0.9988
Epoch 13/20
469/469 [==============================] - 19s 41ms/step - loss: 0.0045 - acc
uracy: 0.9986
Epoch 14/20
469/469 [==============================] - 19s 40ms/step - loss: 0.0038 - acc
uracy: 0.9987
Epoch 15/20
469/469 [==============================] - 19s 41ms/step - loss: 0.0028 - acc
uracy: 0.9991
Epoch 16/20
469/469 [==============================] - 19s 41ms/step - loss: 0.0028 - acc
uracy: 0.9991
Epoch 17/20
469/469 [==============================] - 19s 41ms/step - loss: 0.0021 - acc
uracy: 0.9993
Epoch 18/20
469/469 [==============================] - 19s 40ms/step - loss: 0.0013 - acc
uracy: 0.9996
Epoch 19/20
469/469 [==============================] - 19s 39ms/step - loss: 3.6384e-04 -
accuracy: 0.9999
Epoch 20/20
469/469 [==============================] - 18s 39ms/step - loss: 5.1950e-05 -
accuracy: 1.0000
```



Цікавою виявилась поведінка точності – можемо бачити, що на 6, 9 і 12-13 кроках точність навпаки падала (відповідно, функція втрат зростала). Але все одно на таких параметрах на тренувальних даних вдалося досягнути точності майже в 100%.

Також, від себе зазначу, виявилось цікавим навантаження на процесор (шестиядерний AMD Ryzen 5 3600) в >70%, в той час як навантаження на відеокарту становило порядку 3%.

Тепер використаємо її для класифікації на тестових даних.

```
> results = model %>% evaluate(x_test, y_test)
313/313 [==============================] - 1s 3ms/step - loss: 0.0501 - accur
acy: 0.9921
> results
      loss   accuracy
0.05008209 0.99210000
```

Отже, незважаючи на подвоєння кількості епох і параметра batch_size (хоча його і варто брати степенем двійки), точність на тестових даних вдалося підвищити всього-навсього на 0.0002%. Воно того не варте.

Спробуємо додати в нашу мережу шар Batch Normalization. Тренуватимемо її на 10 епохах з параметром batch_size = 64 (підвищення даних параметрів, як можемо бачити вище, того не варте).

Спершу задамо нашу модифіковану мережу.

```
> inputs = layer_input(shape=c(28,28,1))
> z = layer_conv_2d(inputs, filters=32, kernel_size=c(3,3), activation="selu"
)
> z = layer_max_pooling_2d(z, pool_size = c(2, 2))
> z = layer_conv_2d(z, filters = 64, kernel_size = c(3, 3), activation = "rel
u")
> z = layer_max_pooling_2d(z, pool_size = c(2, 2))
> z = layer_batch_normalization(z)
> z = layer_conv_2d(z, filters = 64, kernel_size = c(3, 3), activation = "rel
u")
> outputs = z
```

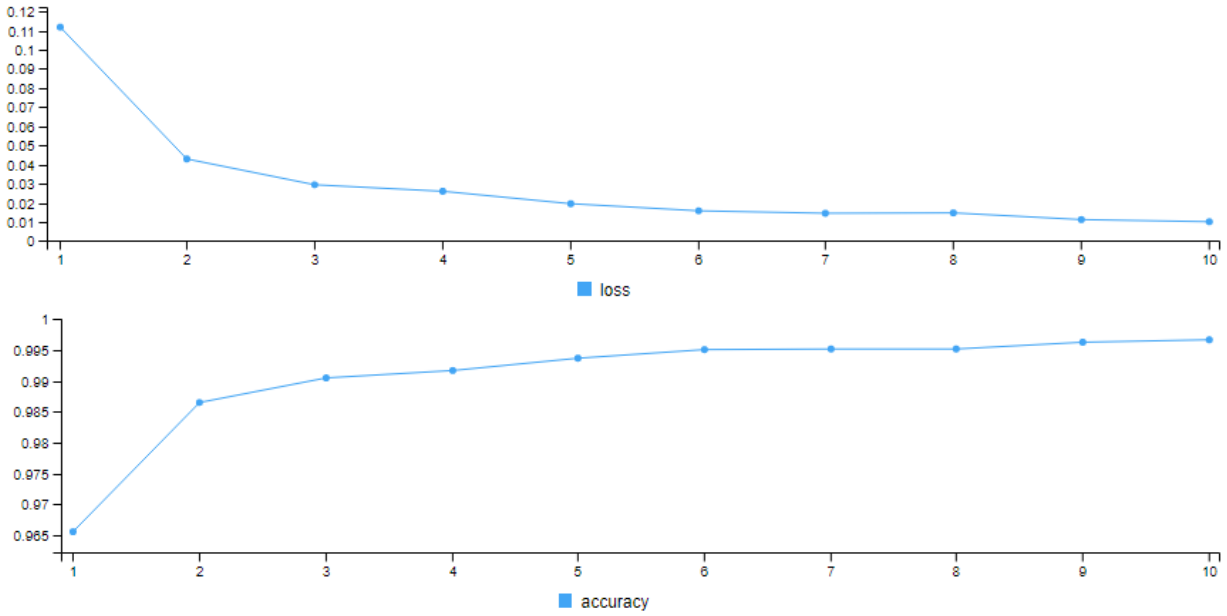
```
> model = keras_model(inputs, outputs)
```

Знову додамо останній шар для класифікації.

```
> z = layer_flatten(z)
> z = layer_dense(z, units = 64, activation = "selu")
> outputs = layer_dense(z, units = 10, activation = "softmax")
> model = keras_model(inputs, outputs)
```

І запустимо процес навчання.

```
> compile(model, optimizer="adam", loss="categorical_crossentropy", metrics=c
("accuracy"))
> fit(model, x_train, y_train, epochs=10, batch_size=64)
Epoch 1/10
938/938 [==============================] - 20s 21ms/step - loss: 0.1118 - acc
uracy: 0.9655
Epoch 2/10
938/938 [==============================] - 19s 21ms/step - loss: 0.0429 - acc
uracy: 0.9865
Epoch 3/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0294 - acc
uracy: 0.9905
Epoch 4/10
938/938 [==============================] - 19s 21ms/step - loss: 0.0260 - acc
uracy: 0.9917
Epoch 5/10
938/938 [==============================] - 20s 21ms/step - loss: 0.0195 - acc
uracy: 0.9937
Epoch 6/10
938/938 [==============================] - 20s 21ms/step - loss: 0.0158 - acc
uracy: 0.9951
Epoch 7/10
938/938 [==============================] - 20s 21ms/step - loss: 0.0145 - acc
uracy: 0.9952
Epoch 8/10
938/938 [==============================] - 20s 22ms/step - loss: 0.0147 - acc
uracy: 0.9952
Epoch 9/10
938/938 [==============================] - 20s 22ms/step - loss: 0.0112 - acc
uracy: 0.9963
Epoch 10/10
938/938 [==============================] - 20s 21ms/step - loss: 0.0101 - acc
uracy: 0.9967
```



На тренувальних даних точність 99.67%. Запустимо на тренувальних.

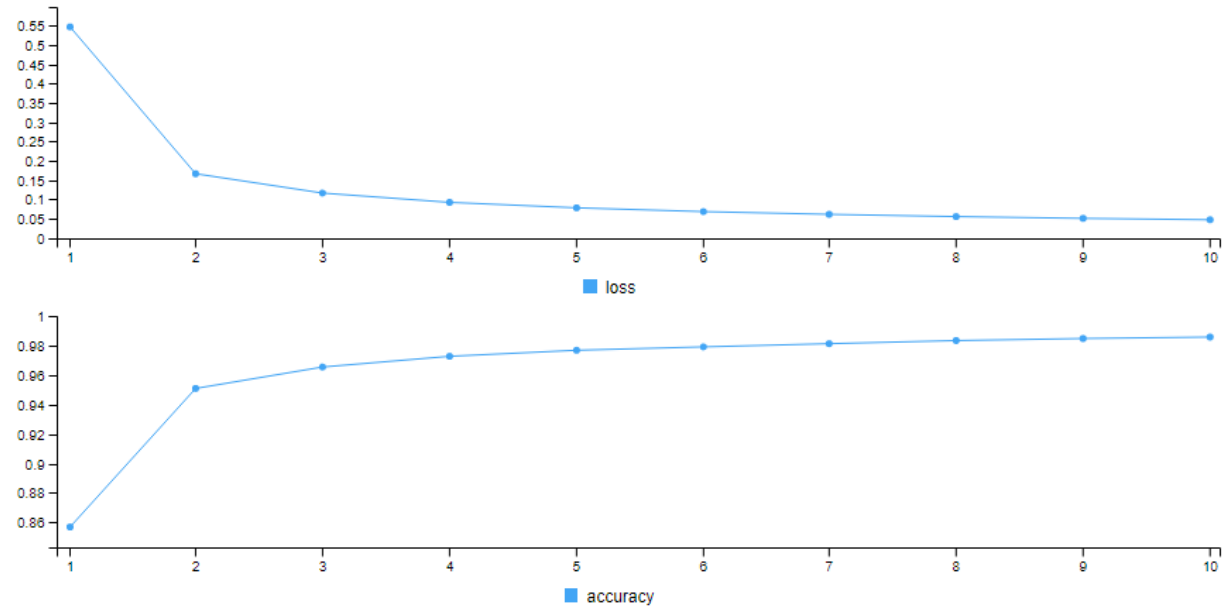
```
> results = model %>% evaluate(x_test, y_test)
313/313 [==============================] - 1s 3ms/step - loss: 0.0558 - accur
acy: 0.9887
> results
      loss   accuracy
0.05575252 0.98869997
```

Тут точність лише понизилась, у порівнянні з минулими випадками. Можливо, таке розміщення нового шару не було виправданим.

Для згорткової нейронної мережі без шару BatchNormalization спробуємо для тренування використати інший оптимізатор (наприклад, SGD).

Запустимо процес навчання для кількості епох = 10 і параметра batch_size = 64.

```
> compile(model, optimizer="sgd", loss="categorical_crossentropy", metrics=c(
"accuracy"))
> fit(model, x_train, y_train, epochs=10, batch_size=64)
Epoch 1/10
938/938 [==============================] - 18s 19ms/step - loss: 0.5467 - acc
uracy: 0.8568
Epoch 2/10
938/938 [==============================] - 18s 19ms/step - loss: 0.1659 - acc
uracy: 0.9511
Epoch 3/10
938/938 [==============================] - 18s 19ms/step - loss: 0.1158 - acc
uracy: 0.9657
Epoch 4/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0917 - acc
uracy: 0.9730
Epoch 5/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0775 - acc
uracy: 0.9771
Epoch 6/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0677 - acc
uracy: 0.9794
Epoch 7/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0606 - acc
uracy: 0.9816
Epoch 8/10
938/938 [==============================] - 19s 21ms/step - loss: 0.0548 - acc
uracy: 0.9837
Epoch 9/10
938/938 [==============================] - 18s 20ms/step - loss: 0.0502 - acc
uracy: 0.9851
Epoch 10/10
938/938 [==============================] - 18s 19ms/step - loss: 0.0466 - acc
uracy: 0.9861
```



Використання такого оптимізатора дає найнижчу точність навіть на тренувальних даних. Що ж, можливо це не той оптимізатор, який варто використовувати для задач розпізнавання образів…

Перевіримо на тестових даних.

```
> results = model %>% evaluate(x_test, y_test)
313/313 [==============================] - 1s 3ms/step - loss: 0.0513 - accur
acy: 0.9835
> results
      loss   accuracy
0.05129744 0.98350000
```

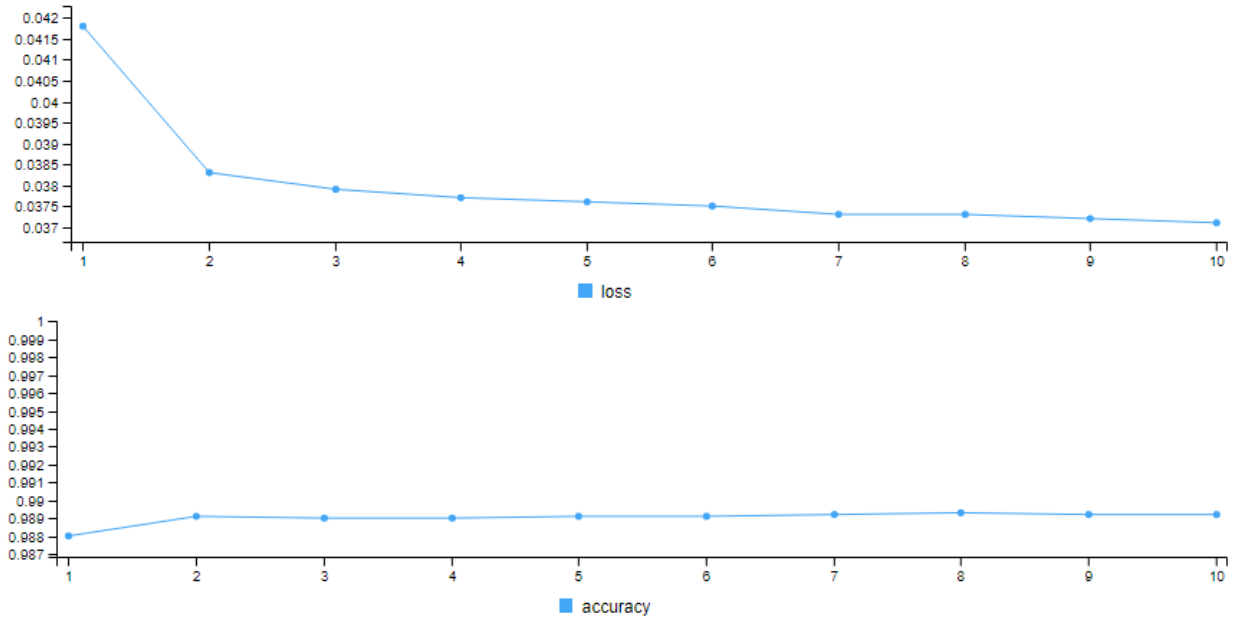
Іншого, власне кажучи, і не варто було очікувати – точність вийшла порядку 98.35%. Що ж, принаймні, не відбулося перенавчання.

А що якщо в якості оптимізатора обрати Adadelta? Запустимо процес навчання.

Я, зізнаюсь чесно, не дуже гарно знаюсь на різних параметрах, і як вони працюють, адже і цей оптимізатор виявився не дуже гарним…

```
> compile(model, optimizer="adadelta", loss="categorical_crossentropy", metri
cs=c("accuracy"))
> fit(model, x_train, y_train, epochs=10, batch_size=64)
Epoch 1/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0418 - acc
uracy: 0.9880
Epoch 2/10
938/938 [==============================] - 19s 21ms/step - loss: 0.0383 - acc
uracy: 0.9891
Epoch 3/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0379 - acc
uracy: 0.9890
Epoch 4/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0377 - acc
uracy: 0.9890
Epoch 5/10
938/938 [==============================] - 19s 20ms/step - loss: 0.0376 - acc
uracy: 0.9891
Epoch 6/10
```

```
938/938 [==============================] - 20s 21ms/step - loss: 0.0375 - acc
uracy: 0.9891
Epoch 7/10
938/938 [==============================] - 20s 21ms/step - loss: 0.0373 - acc
uracy: 0.9892
Epoch 8/10
938/938 [==============================] - 20s 21ms/step - loss: 0.0373 - acc
uracy: 0.9893
Epoch 9/10
938/938 [==============================] - 20s 21ms/step - loss: 0.0372 - acc
uracy: 0.9892
Epoch 10/10
938/938 [==============================] - 21s 22ms/step - loss: 0.0371 - acc
uracy: 0.9892
```



Тут вже на першій епосі точність на тренувальних даних становила порядку 98.8%, але навіть після 10 епох вона не піднялася навіть до 99%. Подібного результату, думаю, варто очікувати і від точності на тестових даних.

```
> results = model %>% evaluate(x_test, y_test)
313/313 [==============================] - 1s 3ms/step - loss: 0.0374 - accur
acy: 0.9875
> results
      loss   accuracy
0.03742085 0.98750001
```

Ну тут принаймні найближче значення точності на тренувальних з точністю на тестових (98.92% проти 98.75% відповідно).

Отже, виконавши дану лабораторну роботу, можна переконатися, що такий клас нейронних мереж як згорткові нейронні мережі є потужним інструментом для такого класу задач, як розпізнавання образів, в чому ми переконалися, порівнявши з результатом, отриманим звичайною щільною нейронною мережею. Використання оптимізаторів sgd та adadelta програли оптимізатору adam, але і в тих випадках все одно згорткова нейронна мережа переважала над щільною (в плані точності на тестових даних). Збільшувати кількість епох і параметр batch_size може виявитись неефективним, і призводити, в певній мірі, до перенавчання.