AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ ZARZĄDZANIA

Praca dyplomowa

Klasyfikacja odręcznie pisanych cyfr za pomocą sieci neuronowych

Autor: *Piotr Zawal*
Kierunek studiów: Metody Statystycznej Analizy Danych
Opiekun pracy: *dr Artur Machno*

Kraków, 2020

Wstęp

Zbiór MNIST

Zbiór MNIST (ang. Modified National Institute of Standards and Technology) jest dużym zbiorem zawierającym odręcznie pisane cyfry. Składa się z przetworzonych zdjęć cyfr, napisanych przez studentów liceów i pracowników agencji United States Census Bureau. Dane te są często wykorzystywane w uczeniu maszynowym lub jako benchmark, szczególnie dla prostszych modeli. Oryginalny zbiór składa się z 60 000 cyfr w zbiorze treningowym i 10 000 w zbiorze testowym. Każdy obraz ma rozmiar 28x28 pikseli i jest w trybie monochromatycznym (skali szarości).

Sztuczne sieci neuronowe

Działanie sztucznych sieci neuronowych jest luźno inspirowane działaniem ludzkiego układu nerwowego. Zbudowany jest on z neuronów połączonych przez synapsy, a przetwarzanie informacji odbywa się w nim poprzez zmianę siły połączenia synaptycznego. W sztucznej sieci neuronowej (ang. *artificial neural network*, ANN) sztuczny neuron składa się z wielu wejść. Informacje z tych wejść są następnie sumowane z odpowiednimi wagami (analogia do synaps) i następnie poddawane działaniu funkcji aktywacji (analogia do różnych reguł uczenia w neuronach). Sygnał wyjściowy może być podany do warstwy wyjściowej bądź - jeśli jest to głęboka sieć neuronowa - do kolejnych warstw.

Konwolucyjne sieci neuronowe

Konwolucyjne sieci neuronowe (ang. *convolutional neural networks*, CNNs) potrafią filtrować poszczególne części danych i wyostrzać w nich pewne cechy. Pozwala to na lepszą dyskryminację w procesach rozpoznawania i klasyfikacji wzorców. CNN zbudowane są z jednej lub wielu warstw konwolucyjnych, których informacja wyjściowa podawana jest na wejście klasycznej sztucznej sieci neuronowej. Ekstrakcja cech z obrazu odbywa się poprzez przesuwaniu wzdłuż obrazu filtru, dokonującego operacji splotu (konwolucji) pomiędzy analizowanym fragmentem a filtrem, tworząc "mapę cech". Następnie, w celu ograniczenia liczby parametrów, przeprowadza się redukcję mapy. Tak przetworzone dane są dalej poddawane spłaszczeniu, czyli redukcji wymiarowości macierzy (np. z macierzy 2x2 na jednowymiarowy wektor wymagany przez klasyczną sieć neuronową). Fakt wykorzystania wielowymiarowych filtrów sprawia, że CNN dobrze nadają się do analizy obrazów (zarówno 2D jak i 3D), ponieważ zachowane są korelacje przestrzenne pomiędzy poszczególnymi pikselami. Pozwala to na ekstrakcję cech niemożliwą do osiągnięcia w przypadku jednokrokowej redukcji wymiarowości obrazu. Z tego powodu ten rodzaj sieci jest często wykorzystywany w rozpoznawaniu wzorców na obrazach, np. detekcji twarzy w aparatach fotograficznych.

Kaggle

Kaggle jest platformą pozwalającą na udział w konkursach dotyczących analizy danych. Część konkursów jest oferowana przez komercyjne firmy, oferujące nagrody pieniężne za opracowanie modeli do predykcji danych. Celem innych konkursów jest przede wszystkim wymiana pomysłów i umiejętności osób ze społeczności analityków danych oraz edukacja. Jednym z konkursów edukacyjnych, pozwalających na zapoznanie się z technikami uczenia maszynowego, jest przedstawiony w tej pracy konkurs polegający na opracowaniu modelu rozpoznającego odręcznie pisane cyfry.

Konkurs Digit Recognizer

Celem konkursu jest walidacja modelu rozpoznającego cyfry ze zbioru MNIST. Konkurs jest traktowany jako edukacyjny - obecnie głębokie sieci neuronowe zdolne są do rozpoznawania znacznie bardziej skomplikowanych

obrazów (np. baza ImageNet) niż te w zbiorze MNIST, dlatego konkurs traktuje się jako przygotowujący do bardziej skomplikowanych zadań. Stanowi on jednak dobry punkt wyjścia do rozpoczęcia pracy z sieciami neuronowymi.

Analiza danych

Analizę danych wykonano przy użycia języka R z wykorzystaniem poniższych bibliotek:

```
library(ggplot2) # rysowanie wykresów
library(readr) # czytanie plików CSV
library(dplyr) # porządkowanie danych
library(tidyr) # porządkowanie danych
library(keras) # API do tworzenia sieci neuronowych
library(knitr) # tabele

# ustal ziarno dla powtarzalności wyników
set.seed(123)

# ustaw styl wykresów
theme_set(theme_light())
```

Analiza eksploracyjna

Dane pobrane z Kaggle są podzielone na zbiór treningowy i testowy i zapisane w plikach *.csv.

```
# wczytaj dane
train <-  read.csv("./train.csv")
test <- read.csv("./test.csv")
```

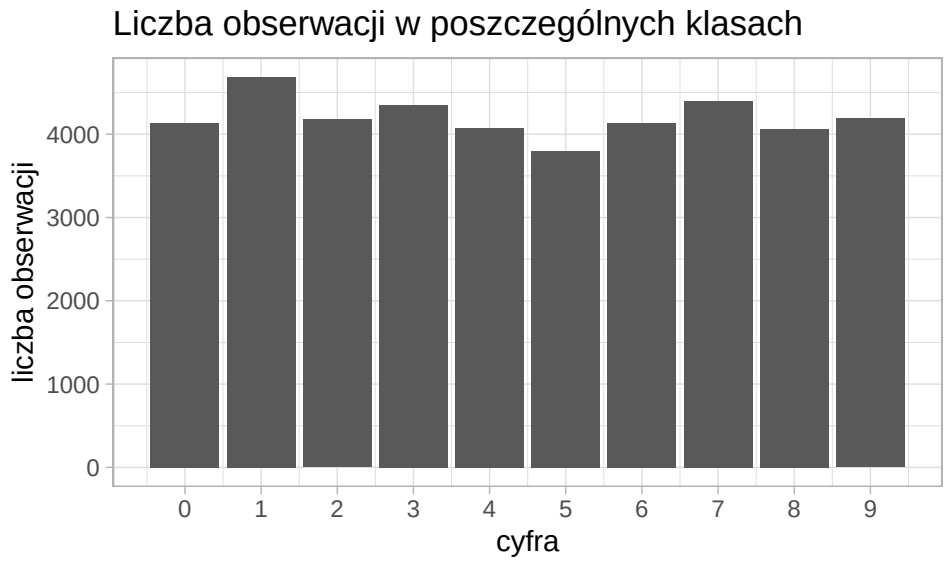
Zbiór treningowy zawiera 42 000 obserwacji, a testowy 28 000.

```
## Zbiór treningowy:  42000 785
```

```
## Zbiór testowy:  28000 784
```

W zbiorze treningowym pojawia się kolumna *label*, zawierająca klasę każdej cyfry. Kolumna ta nie pojawia się w zbiorze testowym, który ma 784 kolumny. Nie jest to więc klasyczny zbiór testowy służący ewaluacji model, ponieważ taki zawiera prawdziwe etykiety, które pozwalają na walidację predykcji. W kolumnach zapisano wartośc liczbową w zakresie [0, 255], odpowiadającą kolorowi piksela w skali szarości. Taka struktura danych jest jednowymiarową reprezentacją macierzy 28x28 ($\sqrt{784} = 28$), której wykreślenie w dalszej części pracy pozwoli na wizualizację cyfr. Powodem takiej struktury danych jest to, że jest ona już przygotowana do implementacji w algorytmach uczenia maszynowego.

```
# rozkład liczby obserwacji (cyfr) w zbiorze treningowym i testowym
ggplot(train, aes(x = label)) +
    geom_bar() +
    scale_x_continuous(breaks = 0:10) +
    ggtitle("Liczba obserwacji w poszczególnych klasach") +
    xlab("cyfra") +
    ylab("liczba obserwacji")
```

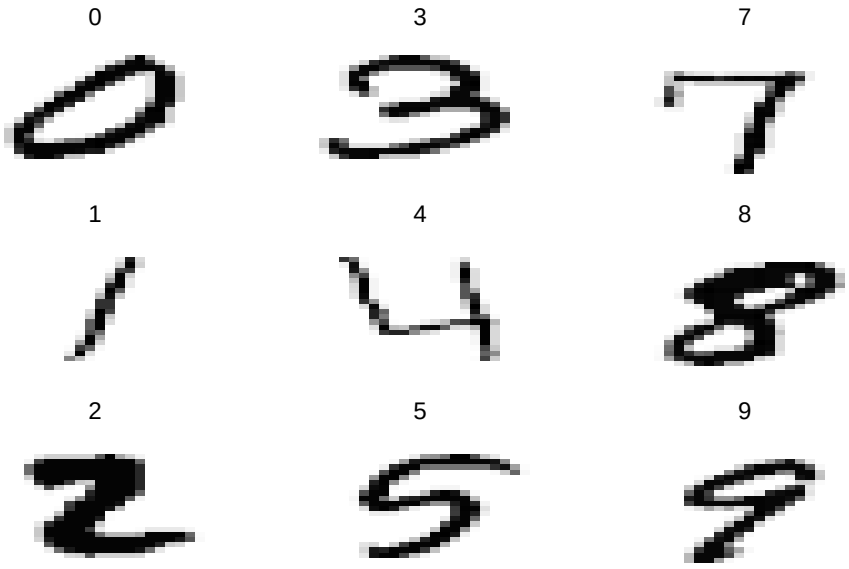


Wizualizacja zbioru danych

W celu wizualizacji cyfr, dane przekształcono do postaci dwuwymiarowej macierzy 28x28:

```
train_2dim <- train %>%
                head(1000) %>%
                # dodaj dodatkowy rząd z numerem wiersza
                mutate(instance = row_number()) %>%
                # zmień format tabeli (wide data -> long data)
                gather(pixel, value, -label, -instance) %>%
                # ekstrakcja numeru piksela z kolumny "pixel" za pomocą wyrażeń regularnych
                tidyr::extract(pixel, "pixel", "(\\d+)", convert = TRUE) %>%
                # dodaj kolumnę z koordynatami piksela
                mutate(x = pixel %% 28,
                       y = 28 - pixel %/% 28)
```

```
train_2dim %>%
    filter(instance <= 18) %>%
    ggplot(aes(x, y, fill = value)) +
    geom_tile(show.legend = FALSE) +
    facet_wrap(facets = ~ label,
               dir = "v") +
    scale_fill_gradient2(low = "white",
                         high = "black",
                         mid = "gray",
                         midpoint = 127.5) +
    xlab("") +
    ylab("") +
    theme_void()
```

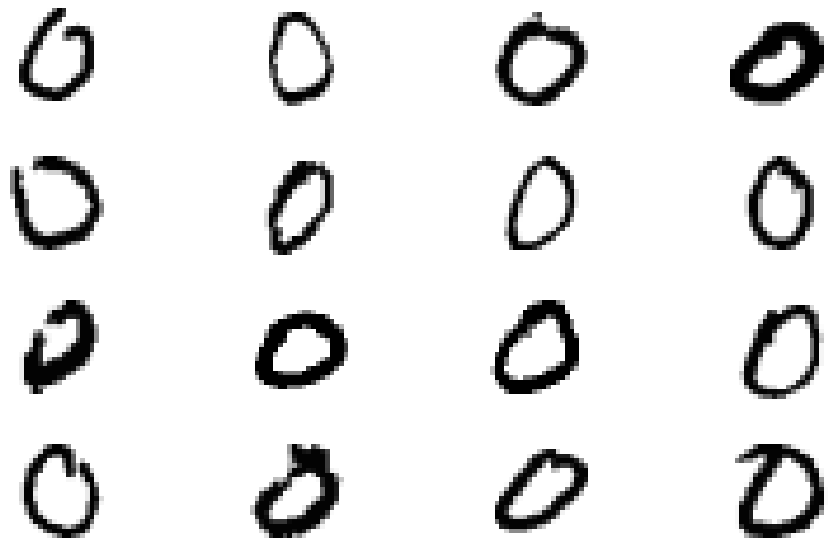


W celu obrazowania danych w postaci dwuwymiarowej zdefiniowano funkcję printDigits(). Argumentami funkcji jest cyfra, która będzie wykreślona (*digit*), liczba paneli (domyślna wartość argumentu *numOfDigits* to 16) oraz argument logiczny *plotRandom* (domyślna wartość TRUE) określający, czy dobór liczb do wizualizacji ma odbywać się w sposób losowy.

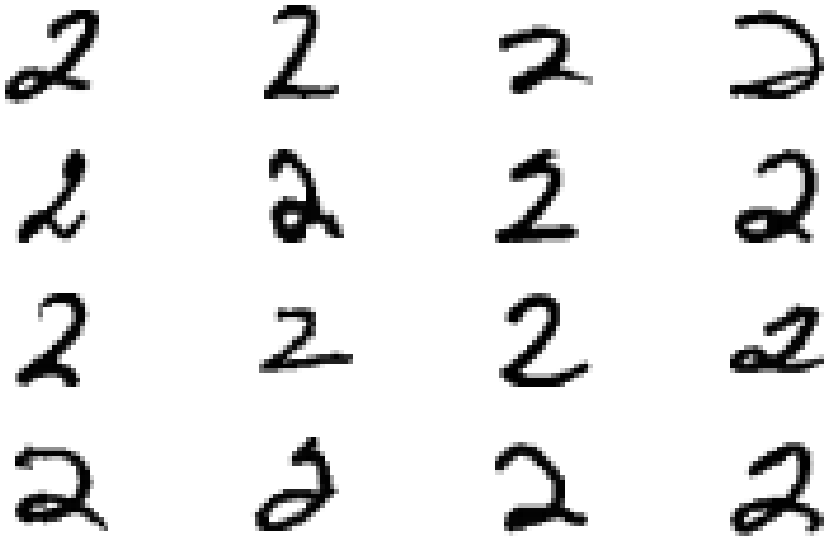
```
printDigits <- function(dataset, digit, numOfDigits = 16, plotRandom = TRUE) {
    if (!require("gridExtra")) install.packages("gridExtra")

    plots <- list()

    for (i in 1:numOfDigits) {

        if (plotRandom) {
            digits_data <- dataset[dataset$label == digit, ] %>% sample_n(1)
        } else {
            digits_data <- dataset[dataset$label == digit, ][i, ]
        }


        plots [[i]] <- digits_data %>%
            gather() %>%
            filter( key != "label") %>%
            mutate(row = row_number() - 1) %>%
            mutate(col = row %% 28, row = row %/% 28) %>%
            ggplot() +
            geom_tile(aes(col, 28 - row, fill = value), show.legend = FALSE) +
            scale_fill_gradient(low = "white", high = "black") +
            coord_equal() +
            theme_void()
    }

do.call("grid.arrange", c(plots, ncol = 4, nrow = ceiling(numOfDigits / 4)))
}
```

Z pomocą tej funkcji łatwo będzie można wykreślić losowo wybrane cyfry, np:

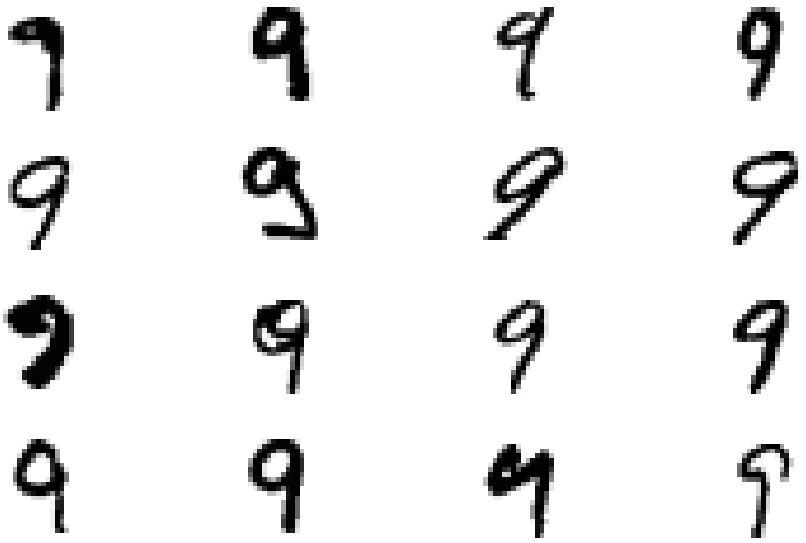
```
printDigits(train, 0)
```



```
printDigits(train, 2)
```



```
printDigits(train, 9)
```



Docelowo dane powinny być przekształcone ze skali szarości do skali zawierającej tylko dwa skrajne kolory: czarny i biały. Sprawdzono, jak wygląda rozkład wartości zapisanych w pikselach, gdzie 0 odpowiada kolorowi białemu a 255 czarnemu; wartości pośrednie reprezentują różne odcienie szarości.

```
ggplot(train_2dim, aes(value)) +
    geom_histogram(binwidth = 10) +
    xlab("wartość piksela") +
    ylab("suma obserwacji") +
    ggtitle("Histogram wartości zapisanych w pikselu")
```

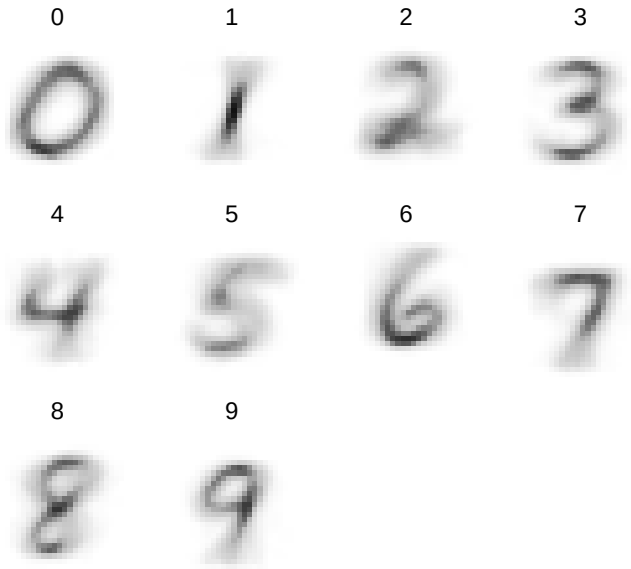
Największa suma obserwacji jest dla wartości "0" - na każdym obrazie cyfry dominuje kolor biały, stanowiący tło. Histogram z wyłączeniem koloru białego - wynika z niego, że dominującą wartością jest 255 (kolor czarny), cyfry powinny mieć więc dobrze zdefiniowane kontury.

```
train_2dim %>%
    filter(value != 0) %>%
    ggplot(aes(value)) +
    geom_histogram(binwidth = 10) +
```

```
    xlab("wartość piksela") +
    ylab("suma obserwacji") +
    ggtitle("Histogram wartości zapisanych w pikselu \nz pominięciem wartości 0")
```

Obliczono średnią dla każdego piksela, pozwalając na wizualizację cyfr uśrednionych po wszystkich obserwacjach:

```
# oblicz jak wygląda uśredniona cyfra
average_digit <- train_2dim %>%
    group_by(x, y, label) %>%
    summarize(mean_pixel_value = mean(value)) %>%
    ungroup()

# average_digit

average_digit %>%
    ggplot(aes(x, y, fill = mean_pixel_value)) +
        geom_tile(show.legend = FALSE) +
        facet_wrap(facets = ~ label) +
        scale_fill_gradient2(low = "white",
                             high = "black",
                             mid = "gray",
                             midpoint = 127.5) +
        xlab("") +
        ylab("") +
        coord_equal() +
        theme_void()
```



Wszystkie uśrednione cyfry są czytelne i mogą bez trudu być rozróżnione przez człowieka. W zbiorze na pewno znajdują się jednak cyfry, których kształt odbiega od uśrednionego kształtu. Aby ocenić skalę zjawiska sprawdzono, w której klasie jest największa wariancja. Jako miarę różnicy pomiędzy średnią i wybraną obserwacją, wybrano średnią odległość euklidesową dla każdej cyfry:

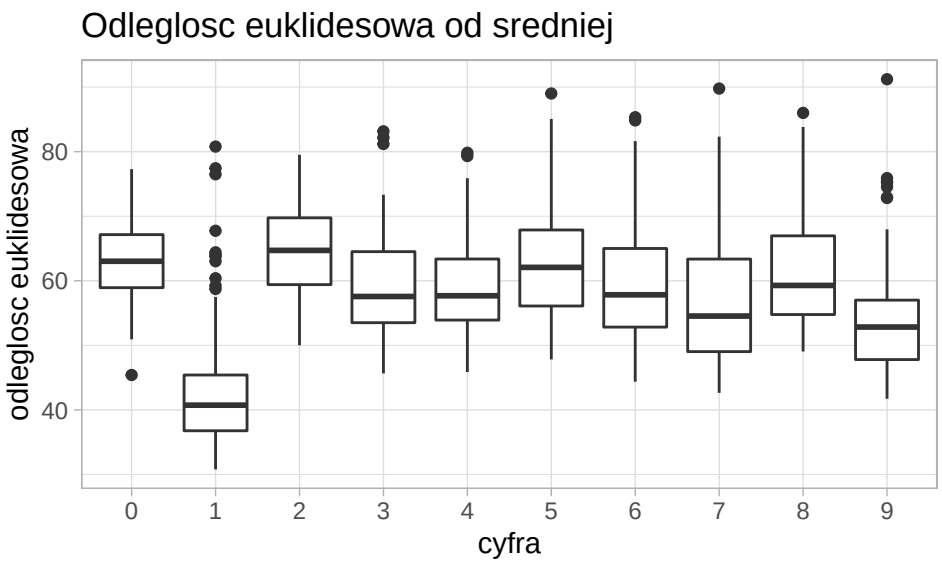
$$d_{Euklides} = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$$

gdzie: y_i to wartość i-tego piksela, $\hat{y}_i$ to wartość piksela uśredniona po wszystkich obserwacjach w danej klasie a n to liczba wszystkich obserwacji.

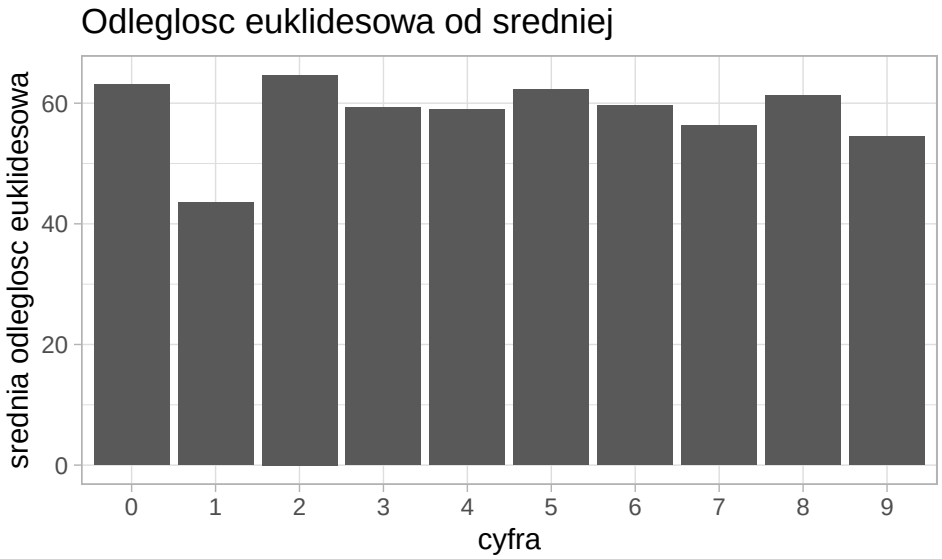
```
digits_joined <- inner_join(train_2dim, average_digit, by = c("label", "x", "y"))

digit_distances <- digits_joined %>%
    group_by(label, instance) %>%
    summarize(euclidean_distance = sqrt(mean((value - mean_pixel_value) ^ 2)))
```

Odległości euklidesowe zwizualizowano na poniższych wykresach:

```
ggplot(digit_distances, aes(x = factor(label), y = euclidean_distance)) +
    geom_boxplot() +
    xlab("cyfra") +
    ylab("odległość euklidesowa") +
    ggtitle("Odległość euklidesowa od średniej")
```




```
digit_distances %>%
    group_by(label) %>%
    summarise(mean_distance = mean(euclidean_distance)) %>%
    ggplot(aes(x = as.factor(label), y = mean_distance)) +
    geom_bar(stat = "identity") +
    ylab("średnia odległość euklidesowa") +
    xlab("cyfra") +
    ggtitle("Odległość euklidesowa od średniej")
```



Największa wariancja występuja dla cyfry “1” - ma to najpewniej związek z faktem, że może być narysowana pod innymi kątami oraz ze “stopką”. Z drugiej strony, jest to cyfra o najniższej średniej odległości, zatem większość jedynek jest do siebie podobna. Największe wartości średniej zaobserwowano dla cyfr “2”, “5” i “8”, nie są one jednak znacząco różne od pozostałych.

Obliczona odległość euklidesowa pozwala na uszeregowanie cyfr względem jej wartości:

```
digit_distances %>%
    arrange(desc(euclidean_distance)) %>%
```

```
    head(20) %>%
    kable()
```

label	instance	euclidean_distance
9	500	91.21559
7	680	89.76955
5	444	89.00802
8	682	86.00489
6	180	85.33374
5	20	85.06932
6	923	84.83836
8	68	83.81750
3	241	83.15105
7	133	82.32457
8	177	82.28932
3	904	82.17008
6	757	81.63603
8	863	81.43698
3	716	81.18354
6	991	81.14376
1	898	80.79027
8	198	80.58815
6	501	80.31876
5	916	80.15243

Na tej podstawie zwizualizowano te cyfry, które mają nawiększą wartość odległości euklidesowej, a więc najbardziej odbiegają od standardowego wyglądu:

```
highest_distances <- as.tbl(digit_distances) %>%
    # dla każdej klasy wybierz 8 obserwacji z największymi odległościami euklidesowymi
    top_n(euclidean_distance, n = 8) %>%
    # dodaj kolumnę szeregującą obserwacje wg. odległości euklidesowej
    mutate(number = rank(-euclidean_distance))


inner_join(train_2dim, highest_distances, by = c("label", "instance")) %>%
    ggplot(aes(x, y, fill = value)) +
    geom_tile(show.legend = FALSE) +
    scale_fill_gradient2(low = "white",
                         high = "black",
                         mid = "gray",
                         midpoint = 127.5) +
    facet_grid(label ~ number) +
    theme_void() +
    theme(strip.text = element_blank())
```

Rzeczywiście, duża wariancja wśród "1" wynika najprawdopodobniej z różnych kątów i jest ona także możliwą przyczyną wysokiej wartości odległości euklidesowej wśród "8". Wartości dla "2" i "5" wynikają z kolei przede wszyskim z charakteru pisma oraz grubości kreski.

Sieci neuronowe

Głębokie sieci neuronowe

Przygotowanie danych do modelowania

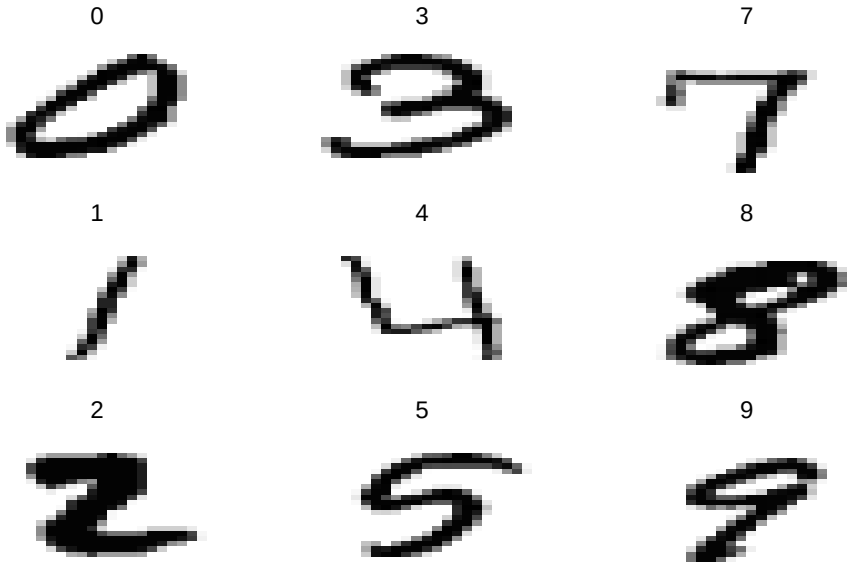
Przed treningiem sieci dane poddano normalizacji:

```
train_y <- train$label
train_y <- as.factor(train_y)
train_x <- train[, -1]

# normalizacja danych do zakresu [0, 1]
train_x <- ceiling(train_x / 255)
test_x <- ceiling(test / 255)
```

W celu łatwiejszej wizualizacji ustandaryzowanych danych posłużono się zmienną *train_2dim*, wobec której zastosowano taką samą technikę standaryzacji wartości pikseli.

```
train_2dim %>%
    filter(instance <= 18) %>%
```

```
    ggplot(aes(x, y, fill = value / 255)) +
    geom_tile(show.legend = FALSE) +
    facet_wrap(facets = ~ label,
               dir = "v") +
    scale_fill_gradient(low = "white", high = "black") +
    xlab("") +
    ylab("") +
    theme_void()
```



Implementację sieci neuronowych przeprowadzono z wykorzystaniem biblioteki *keras*:

```
# wczytaj bibliotekę do głębokich sieci neuronowych
library(keras)

# przygotowanie zbioru treningowego i testowego
train_images <- train[, -1]
train_labels <- train$label

test_images <- test

train_images <- train_images / 255

test_images <- test_images / 255
```

Wymagany przez Keras typ danych wejściowych to macierze, dlatego zbiorzy przekształcono z obecnego typu (lista):

```
train_images <- as.matrix(train_images)
test_image <- as.matrix(test_images)
```

Dodatkowo zmieniono typ danych zawierających klasy (cyfry). W przeciwnym wypadku byłyby traktowane jako wartości liczbowe, co prowadziłoby do nieprawidłowego wytrenowania modelu. Przykładowo, różnica między klasami "1" i "3" byłaby traktowana jako mniejsza niż pomiędzy "1" a "7". Funkcja *to_categorical()* pozwala zaimplementować technikę *one hot encoding*. W takiej reprezentacji klasa "1" będzie odpowiadała wektorowi [1, 0 , 0, . . .], klasa "2" [0, 1, 0, . . .] itd., co niweluje błąd wynikający z interpretacji poszczególnych klas jako liczb:

```
train_labels <- to_categorical(train_labels)
head(train_labels, 10)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    1    0    0    0    0    0    0    0     0
##  [2,]    1    0    0    0    0    0    0    0    0     0
##  [3,]    0    1    0    0    0    0    0    0    0     0
##  [4,]    0    0    0    0    1    0    0    0    0     0
##  [5,]    1    0    0    0    0    0    0    0    0     0
##  [6,]    1    0    0    0    0    0    0    0    0     0
##  [7,]    0    0    0    0    0    0    0    1    0     0
##  [8,]    0    0    0    1    0    0    0    0    0     0
##  [9,]    0    0    0    0    0    1    0    0    0     0
## [10,]    0    0    0    1    0    0    0    0    0     0
```

W kolejnym kroku zdefiniowano funkcję wpisującą obliczone predykcje klas do pliku, który następnie zostaje wysłany do Kaggle celem walidacji poprawności predykcji:

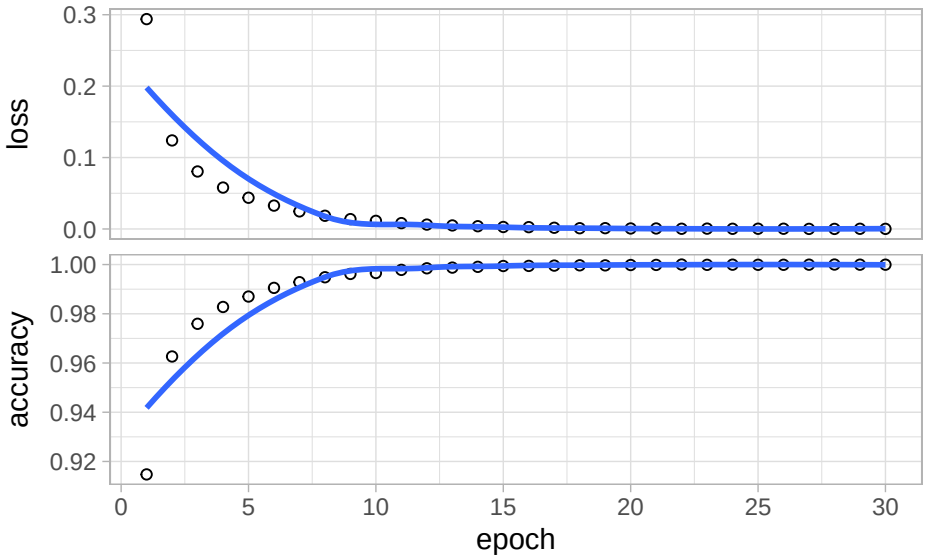
```
submitKaggle <- function(network) {
    predictions <- network %>%
                    predict_classes(as.matrix(test_images))
    submit_predictions <-  read.csv("./sample_submission.csv")
    submit_predictions$Label <- predictions
    write.csv(submit_predictions, "./sample_submission_keras.csv", row.names = FALSE)
}
```

Przygotowanie modelu sztucznej sieci neuronowej

Przy pomocy biblioteki *Keras* w czytelny sposób można zaprojektować architekturę sieci neuronowej. Na początek zdefiniowano model składający się z tylko jednej warstwy zbudowanej z 512 neuronów z funkcją aktywacji *ReLU*, której wyjście podawane jest na warstwę wyjściową o rozmiarze odpowiadającej liczbie klas (10) oraz z funkcją aktywacji *softmax*. W kompilacji wykorzystano optimizer RMSprop oraz kategoryczną entropię krzyżową jako funkcję straty wraz z dokładnością jako metryką uczenia sieci. Są to parametry startowe sugerowane przez twórców biblioteki *Keras*. Sieć trenowana jest przez 30 iteracji (*epochs*).

```
# keras - model 1
network <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
  layer_dense(units = 10, activation = "softmax")

network %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy"))

model_1 <- network %>% fit(train_images,
                           train_labels,
                           epochs = 30,
                           batch_size = 128)

plot(model_1)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

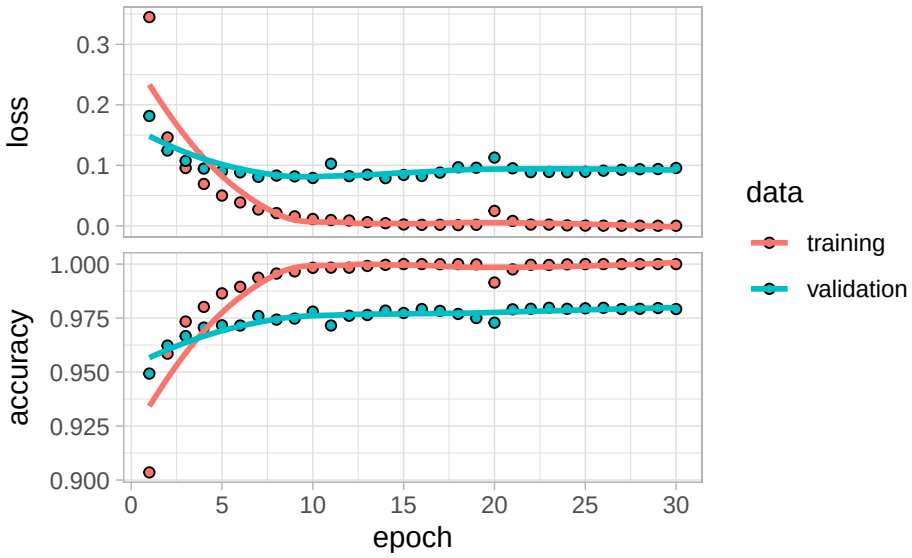



```
submitKaggle(network)
```

Już przy tak prostej sieci otrzymano bardzo zadowalający wynik rzędu 97.89% (przy 30 iteracjach; dla 10 iteracji wyniósł on 97.857%). Pomimo dobrej dokładności, dalsza optymalizacja tego modelu jest utrudniona ze względu na brak jakiejkolwiek kontroli na zbiorze treningowym. Przekłada się to na brak monitorowania i kontroli nad zjawiskiem overfittingu, a jedyną możliwością kontroli rzeczywistej dokładności modelu jest wysłanie wyników do Kaggle (limit 5 dziennie). Aby zoptymalizować model, podczas procedury treningu wydzielono losowo wybrane obserwacje stanowiące 20% zbioru treningowego (*validation_split=0.2*). Minusem tego rozwiązania jest zmniejszenie zbioru treningowego: zamiast 42 000 obserwacji do trenowania modelu wykorzystanych zostanie 33 600, co przełoży się na niższą finalną dokładność modelu. Z tego powodu, po zoptymalizowaniu architektury sieci, trenowanie modelu będzie przeprowadzone na pełnym zbiorze treningowym.

```
# keras - model 2
network2 <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
  layer_dense(units = 10, activation = "softmax")

network2 %>% compile(
  optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = c("accuracy"))

model_2 <- network2 %>% fit(train_images,
                            train_labels,
                            epochs = 30,
                            batch_size = 128,
                            validation_split = 0.2)

plot(model_2)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Wybór powyższej architektury został oparty na metrykach dla wielu różnych architektór sieci, które uzyskano zmieniając takie parametry jak: liczba neuronów w warstwie, typ optimizera (RMSprob i Adam) oraz różna wartość parametry *batch_size* (dla optimizera Adam).

Table 2: Optimizer: rmsprop

units	loss	accuracy	val_loss	val_accuracy
256	0.0001788	1.0000	0.1498	0.9763
512	0.0004071	0.9999	0.1437	0.9792
784	0.0001047	1.0000	0.1496	0.9808

Table 3: Optimizer: adam

units	loss	accuracy	val_loss	val_accuracy
256	0.0004730	1	0.1009	0.9770
512	0.0002839	1	0.0938	0.9802
784	0.0001135	1	0.0984	0.9814

Table 4: Batch sizes (dla units = 512, optimizer = adam)

batch_size	loss	accuracy	val_loss	val_accuracy
512	0.0033000	1.0000	0.0866	0.9760
256	0.0008838	1.0000	0.0871	0.9790
128	0.0002839	1.0000	0.0938	0.9802
32	0.0041000	0.9986	0.1683	0.9754

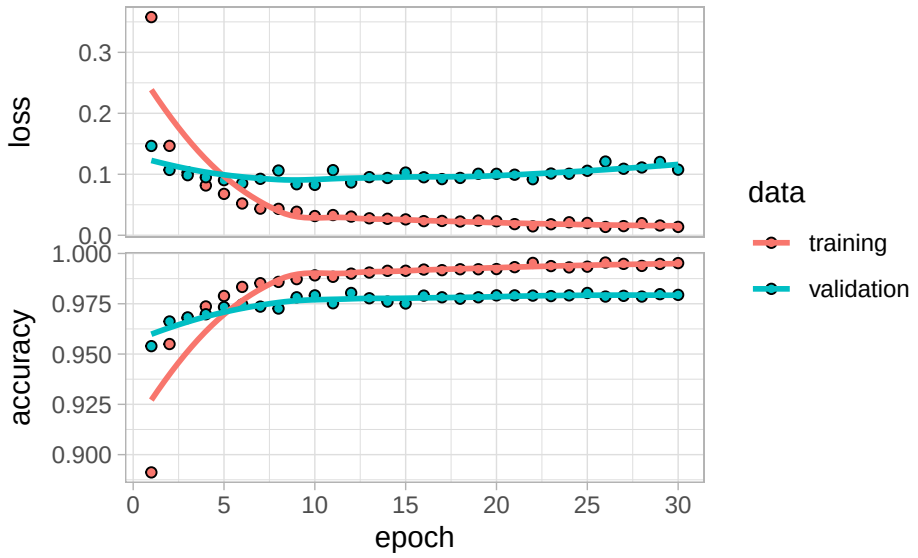
Na wykresie metryk od iteracji widać jednak jednak, że parametr val_loss wzrasta nieznacznie wraz z kolejnymi iteracjami. Jest to skutek overfittingu, tj. model "zapamiętuje" dane i optymalizuje się pod kątem zbioru treningowego zamiast coraz lepiej klasyfikować obserwacje w zbiorze testowym. Zjawisko to jest jeszcze lepiej widoczne po dodaniu kolejnej warstwy:

```
knitr::opts_chunk$set(cache = TRUE)
# model 3 - spróbujemy dodać więcej warstw
network3 <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
  layer_dense(units = 10, activation = "softmax")

network3 %>% compile(
  optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = c("accuracy"))

model_3 <- network3 %>% fit(train_images,
                            train_labels,
                            epochs = 30,
                            batch_size = 128,
                            validation_split = 0.2)

# loss: 0.0082 - accuracy: 0.9978 - val_loss: 0.1524 - val_accuracy: 0.9768
```

Finalnie owocuje to spadkiem dokładności predykcji do 97.68%. Aby zapobiegać overfittingowi do architektury sieci wprowadzono warstwy dropout (*layer_dropout()*). Ich zadaniem jest losowe przypisanie danym wejściowym wartości 0 (zerowanie wag), a parametrem regulującym liczbę wyzerowanych wartości jest parametrem *rate*. Przyjmując wartość 0.3 oznacza to, że 30% wag wejściowych zostanie przypisana wartość 0.

```
knitr::opts_chunk$set(cache = TRUE)
# model 4 - dodanie dropoutów
network4 <- keras_model_sequential() %>%
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
    layer_dense(units = 10, activation = "softmax")

network4 %>% compile(
  optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = c("accuracy"))


#load_model_weights_hdf5(network4, "./network4.h5")
model_4 <- network4 %>% fit(train_images,
                            train_labels,
                            epochs = 30,
                            batch_size = 128,
                            validation_split = 0.2)
plot(model_4)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Powyższy model wybrano na podstawie kilku konfiguracji parametru *rate* w obu warstwach typu dropout, a testowane kombinacje obejmowały:

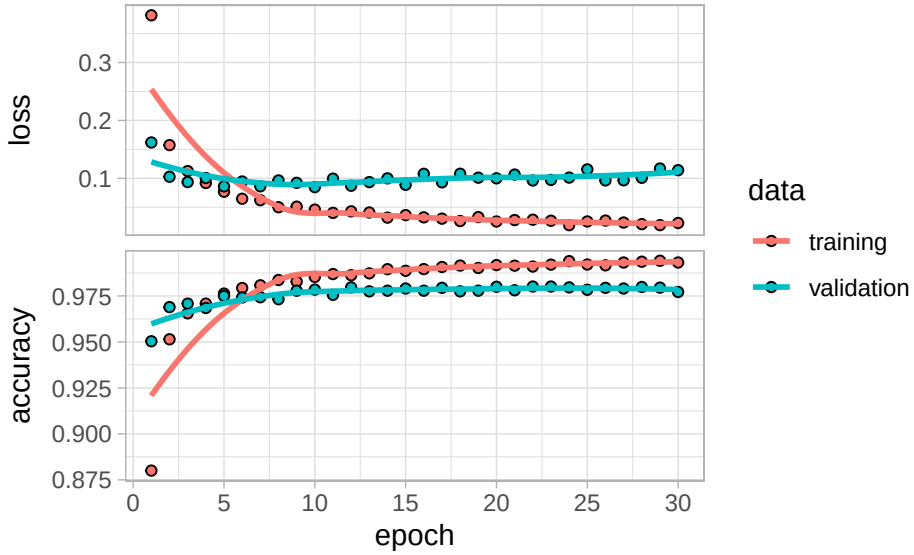
Table 5: Wpływ dropout_rate na

dropout_rate_1	dropout_rate_2	loss	accuracy	val_loss	
0.70	0.50	0.0801	0.9750	0.0764	
0.50	0.50	0.0742	0.9753	0.0791	
0.45	0.45	0.0294	0.9902	0.0976	
0.30	0.30	0.0156	0.9949	0.0948	
0.20	0.20	0.0126	0.9961	0.1122	
0.30	0.20	0.0133	0.9953	0.0963	
Na podstawie wyni	ków, jako optymal	ną wartoś	ć tego para	metru wybra	
Dodanie kolejnej	warstwy do modelu	spowodow	ało jednak	spadek dokł	adności, dlatego uznano, że optymaln

```
# model 4 - dodanie jeszcze jednej warstwy
network5 <- keras_model_sequential() %>%
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
    layer_dense(units = 10, activation = "softmax")

network5 %>% compile(
  optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = c("accuracy"))


#load_model_weights_hdf5(network4, "./network4.h5")
model_5 <- network5 %>% fit(train_images,
                            train_labels,
```

```
                          epochs = 30,
                          batch_size = 128,
                          validation_split = 0.2)
plot(model_5)
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
kable(rbind.data.frame(c(2, 0.0156, 0.9949, 0.0948, 0.9820),
          c(3, 0.0231, 0.9929, 0.1167, 0.9768)),
col.names = c("hidden_layers", "loss", "accuracy", "val_loss", "val_accuracy"),
caption = "Metryki sieci dla różnej liczby warstw")
```

Table 6: Metryki sieci dla różnej liczby warstw

hidden_layers	loss	accuracy	val_loss	val_accuracy
2	0.0156	0.9949	0.0948	0.9820
3	0.0231	0.9929	0.1167	0.9768

Jednym z najbardziej istotny parametrów w treningu sieci neuronowej jest szybkość uczenia sieci (ang. *learning rate*). Zgodnie z intuicją, wysoka wartość tego parametru poprawia szybkość uczenia sieci, ale można skutkować niższą dokładnością. Wybór zbyt niskiej szybkości uczenia może z kolei utrudnić znalezienie globalnego minimum jeśli w przestrzeni parametrów będą występować także minima lokalne.

```
learning_rates = c(0.01, 0.005, 0.001, 0.0005, 0.0001)
accuracy_vector = c()
loss_vector = c()
val_accuracy_vector = c()
val_loss_vector = c()

for (rate in learning_rates) {

    network_lr <- keras_model_sequential() %>%
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
```

```
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
    layer_dense(units = 10, activation = "softmax")

    network_lr %>% compile(optimizer = "adam",
                           loss = "categorical_crossentropy",
                           metrics = c("accuracy"))

    model_lr <- network_lr %>% fit(train_images,
                            train_labels,
                            epochs = 30,
                            batch_size = 128,
                            validation_split = 0.2,
                            learning_rates = rate)
    plot(model_lr)

    accuracy_vector <- append(accuracy_vector,
                              tail(model_lr$metrics$accuracy, n = 1))
    loss_vector <- append(loss_vector,
                          tail(model_lr$metrics$loss, n = 1))
    val_accuracy_vector <- append(val_accuracy_vector,
                                  tail(model_lr$metrics$val_accuracy, n = 1))
    val_loss_vector <- append(val_loss_vector,
                              tail(model_lr$metrics$val_loss, n = 1))

}

# hidden layers = 2   loss: 0.0156 - accuracy: 0.9949 - val_loss: 0.0948 - val_accuracy: 0.9820 OVRF
# hidden layers = 3   loss: 0.0231 - accuracy: 0.9929 - val_loss: 0.1167 - val_accuracy: 0.9768
```

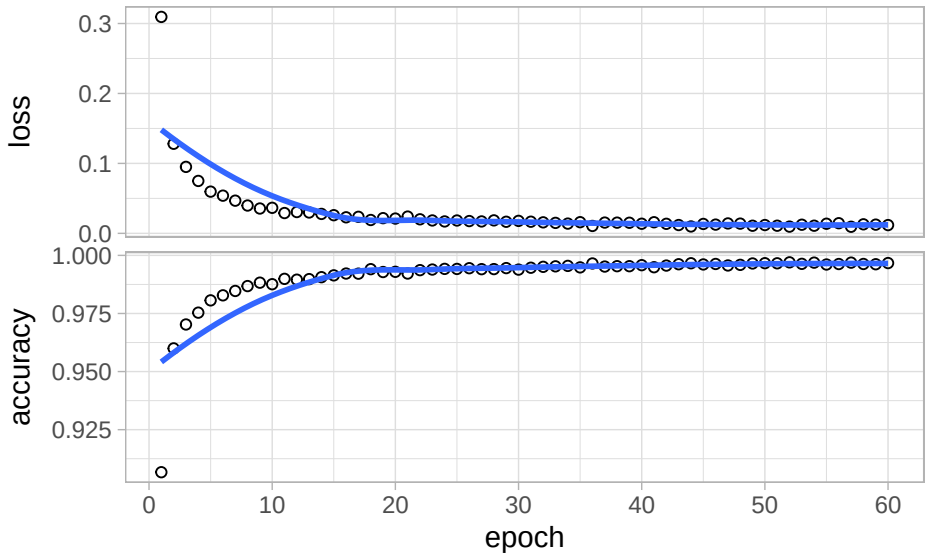
	0.01	0.005	0.001	5e-04	
accuracy_vector	0.9947917	0.9937202	0.9952083	0.9948512	
loss_vector	0.0163099	0.0197452	0.0146679	0.0156573	
val_accuracy_vector	0.9802381	0.9800000	0.9794047	0.9783334	
val_loss_vector	0.0986082	0.1033510	0.1008353	0.1181140	
Na podstawie otrzymany	ch wyników w	idać, że naj	lepsze metry	ki otrzymano	dla domyślnej wartości parametru

Na podstawie ustalonej powyżej architektury sieci, trenowanie przeprowadzono na całym (42 00 obserwacji) zbiorze treningowym:

```
mlp_network_final <- keras_model_sequential() %>%
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
    layer_dense(units = 512, activation = "relu", input_shape = c(784)) %>%
    layer_dropout(rate = 0.3) %>%
    layer_dense(units = 10, activation = "softmax")

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

```
#load_model_weights_hdf5(network4, "./network4.h5")
model_final <- mlp_network_final %>% fit(train_images,
                                         train_labels,
                                         epochs = 60,
                                         batch_size = 128,
                                         learning_rate = 0.001)
plot(model_final)
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
submitKaggle(mlp_network_final)

mlp_network_final_accuracy <- tail(model_final$metrics$accuracy, n = 1)
mlp_network_final_loss <- tail(model_final$metrics$loss, n = 1)
```

Dokładność sprawdzono dla różnej liczby iteracji, przy czym najlepszy wynik Kaggle uzyskano dla 60 (*epochs = 60*). Niższa dokładność przy 100 iteracjach mogła wynikać z niewielkiego overfittingu sieci, a przy 30 ze zbyt krótkiej procedury treningowej (funkcja nie osiągała globalnego minimum).

```
## accuracy:  0.9966905
```

```
## loss:  0.01181713
```

Table 8: Wyniki z Kaggle dla różnej liczby iteracji

epochs	Kaggle score
30	0.98028
60	0.98128
100	0.98057

Konwolucyjne sieci neuronowe (CNN)

Konwolucyjne sieci neuronowe są często wykorzystywane do klasyfikacji obrazów. Ich przewaga nad wielowarstwowymi sieciami polega na tym, że dane wejściowe nie muszą zostać przekształcone do postaci jednowymi-

arowego wektora. Obniżenie wymiarowości danych (w opisywanym przypadku z dwuwymiarowej macierzy do jendowymiarowego wektora) sprawia, że utracona zostaje informacja, które piksele ze sobą sąsiadują w oryginalnym obrazie.

Trening CNN

Konwolucyjne sieci neuronowe wymagają znacznie większej mocy obliczeniowej niż klasyczne perceptrony. Dla przykładu, trening opisanych wyżej sieci zajmował ok. 20 minut (przy 30 iteracjach), a nawet dla nieskomplikowanej CNN czas ten zwiększył się do ok. 80 minut. Z tego względu zdecydowano przenieść się obliczenia na platformę Google Cloud Platform, pozwalającą na użycie większej mocy obliczeniowej oraz trenowanie sieci na jednostkach GPU, skracając tym samym czas do ok. 20 minut. [1]

Implementacja CNN

Dane przekształcić można do postaci dwuwymiarowej z wykorzystaniem funkcji array_reshape() z pakietu keras:

```
library(keras)
train_data_path <- "./train.csv"
test_data_path <- "./test.csv"

train <- read_csv(train_data_path)
```

```
## Warning in guess_header_(datasource, tokenizer, locale): '.Random.seed[1]' nie
## jest poprawną liczbą całkowitą, więc został zignorowany
```

```
## Parsed with column specification:
## cols(
##   .default = col_double()
## )
```

```
## See spec(...) for full column specifications.
```

```
test <- read_csv(test_data_path)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double()
## )
## See spec(...) for full column specifications.
```

```
y_train <- to_categorical(train$label, num_classes = 10)
x_train <- train[, -1] %>% as.matrix()
x_test <- test %>% as.matrix

img_rows <- 28
img_cols <- 28

x_train <- x_train / 255
x_test <- x_test / 255
```

[1] Jednym z zakładanych celów pracy miało być przeprowadzenie optymalizacji hiperparametrów (ang. *hyperparameter tuning*) sieci CNN celem znalezienia najlepszej konfiguracji. Metoda ta polega na sprawdzeniu wielu różnych architektór różniących się tzw. hiperparametrami, czyli np. rozmiarem warstwy filtrującej, rozmiarem kernela, funkcji aktywacji, wyboru optimizera czy doboru szybkości uczenia. Niestety, rozwój biblioteki *cloudml* nie nadąża za zmianami na Google Cloud Platform (używa m. in. niewspieranego już Pythona 2.7 zamiast 3.5 oraz innej wersji runtime). W efekcie metryki modelu (dokładność) nie są poprawnie zwracane do API platformy, co uniemożliwa optymalizację hiperparametrów z wykorzystaniem tej biblioteki.

```
x_train <- array_reshape(x_train, c(nrow(x_train), img_rows, img_cols, 1))
x_test <- array_reshape(x_test, c(nrow(x_test), img_rows, img_cols, 1))

input_shape <- c(dim(x_train)[-1])
```

Architektura sieci

Ze względu na konieczność wykorzystania płatnej usługi Google Cloud Platform, zamiast iteracyjnie testować różne modele, zdecydowano się na implementację już opublikowanej architektury sieci oraz metodologii przygotowania danych: https://www.kaggle.com/couyang/easiest-cnn-with-99-6-accuracy-using-keras-top-5

Architektura sieci wygląda następująco:

```
model_cnn <- keras_model_sequential() %>%

  layer_conv_2d(filters = 128,
                kernel_size = c(5, 5),
                activation = 'relu',
                padding = 'same',
                input_shape = input_shape) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 64,
                kernel_size = c(5, 5),
                activation = 'relu',
                padding = 'same',
                input_shape = input_shape) %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.25) %>%

  layer_conv_2d(filters = 64,
                kernel_size = c(3, 3),
                activation = 'relu',
                padding = 'same') %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 64,
                kernel_size = c(3, 3),
                activation = 'relu',
                padding = 'same') %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2),
                       strides = c(2, 2)) %>%
  layer_dropout(rate = 0.25) %>%

  layer_conv_2d(filters = 64,
                kernel_size = c(3, 3),
                activation = 'relu',
                padding = 'same') %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.25) %>%

  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
```

```
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.25) %>%

  layer_dense(units = 10, activation = "softmax")

model_cnn %>% compile(
  optimizer = optimizer_rmsprop(epsilon = 1e-08),
  loss = "categorical_crossentropy",
  metrics = c("accuracy"))
```

Zmodyfikowano także funkcję wpisującą wyniki do pliku wysyłanego do Kaggle:

```
submitKaggleCNN <- function(network) {
    predictions <- network %>%
                    predict_classes(x_test)
    submit_predictions <-  read.csv("./sample_submission.csv")
    submit_predictions$Label <- predictions
    write.csv(submit_predictions, "./sample_submission_CNN_aug.csv", row.names = FALSE)
}

submitKaggleCNN(load_model_hdf5("model_aug_lr_reduction_epochs_50.h5"))
```

Wynik Kaggle osiągnięty przez tę sieć wyniósł 99.457%. Ocenia się jednak, że maksymalny wynik, jaki może zostać osiągnięty przez sieć CNN na zbiorze MNIST to ok. 99.7%. Aby jednak zbliżyć się do tego rezultatu, należy wprowadzić kilka poprawek do metodologii trenowania sieci.

Poszerzanie zbioru danych

Poszerzanie zbioru danych (and. data augmentation) polega na wprowadzeniu niewielkich modyfikacji w danych treningowych i połączeniu ich z oryginalnym zbiorem. Dzięki temu otrzymywany jest zwiększony zbiór, w których nowymi obserwacjami są lekko zmienione wersje pierwotnych danych. Wprowadzane zmiany muszą być na tyle niewielkie, żeby model był w stanie poprawnie przyporządkować obserwacje do właściwych klas. Istnieje wiele parametrów pozwalających na modyfikacje obrazów, ale do poszerzenia zbioru zdecydowano się na: zmianę szerokości i wysokości $\pm$ 15%, powiększenie $\pm$ 15% , obrót o 15 stopni i odkształcenie (wyrażone miarę kąta w radianach).

```
datagen <- image_data_generator(featurewise_center = F,
                                samplewise_center=F,
                                featurewise_std_normalization = F,
                                samplewise_std_normalization=F,
                                zca_whitening=F,
                                horizontal_flip = F,
                                vertical_flip = F,
                                width_shift_range = 0.15,
                                height_shift_range = 0.15,
                                zoom_range = 0.15,
                                rotation_range = .15,
                                shear_range = 0.15
                                )

datagen %>% fit_image_data_generator(x_train)
```

Ze względu na zmodyfikowany zbiór treningowy, do trenowania modelu wykorzystano funkcję fit_generator i flow_images_from_data, która generuje paczki danych wymagane do trenowania. Szybkość uczenia

kontrolowana jest za pomocą parametru callbacks.

```
model %>% fit_generator(flow_images_from_data(x_train,
                                               y_train,
                                               datagen,
                                               batch_size = FLAGS$batch_size),
                        steps_per_epoch = nrow(x_train)/64,
                        epochs = 30,
                        verbose = 1,
                        callbacks = learning_rate_reduction)
```

Z uwagi na skomplikowaną architekturę sieci, wszystkim parametrom przypisano te same wartości. Walidację przeprowadzono dla wartości 0.15 i 0.1, uzyskując odpowiednio 99.557% i 99.428%.

Redukcja szybkości uczenia

Szybkość uczenia modelu jest uważana za hiperparametr, który najmnocniej wpływa na finalną dokładność modelu. Szybkie uczenie zmniejsza czas potrzebny na wytrenowanie modelu, ale może skutkować jego niższą dokładnością. Z drugiej strony, stosując bardzo niskie szybkości uczenia modelu, model może znaleźć się w minimum lokalnym. Dobrą praktyką jest stopniowe zmniejszanie szybkości uczenia, jeśli zmiany dokładności modelu pomiędzy kolejnymi iteracjami są bardzo małe. Keras posiada wbudowaną funkcję <tensorflow.python.keras.callbacks.ReduceLROnPlateau>, która pozwala zaimplementować zmiany szybkości uczenia:

```
learning_rate_reduction <- callback_reduce_lr_on_plateau(monitor = "acc",
                                                         patience = 3,
                                                         verbose = 1,
                                                         factor = 0.5,
                                                         min_lr = 0.00001)
```

Wprowadzenie tej poprawki pozwoliło na uzyskanie wyniku 99.642%, zbliżonego do limitu możliwości sieci konwolucyjnych. Zwiększenie liczby iteracji do 50 spowodowało z kolei spadek dokładności do 99.485%, dlatego za najlepszy model przyjęto ten uzyskany przy 30 iteracjach.

Walidacja modelu

Walidacja na podstawie pełnego zbioru treningowego MNIST

Walidację modelu zdecydowano się sprawdzić na pełnym zbiorze treningowym MNIST, zawierającym 60 000 obserwacji. Należy tutaj jednak zaznaczyć, że nie jest to w pełni prawidłowa walidacja. Obserwacje w zbiorze treningowym Kaggle pokrywają się z tymi w zbiorze MNIST, więc częściowo są to te same dane, na których wytrenowano model. Zdecydowano się przestawić ten krok, ponieważ zbiór Kaggle był mniejszy od zbioru MNIST (odpowiednio 42 000 i 60 000 obs.), a zatem część cyfr z pełnego zbioru nie była zawarta w zbiorze Kaggle.

```
# pełny zbiór treningowy MNIST
validate_set <- read_csv("./mnist_full.csv")

x_val_set <- validate_set[, -1]
y_val_set <- validate_set$label

x_val_set <- array_reshape(as.matrix(x_val_set), c(nrow(x_val_set), 28, 28, 1))
```

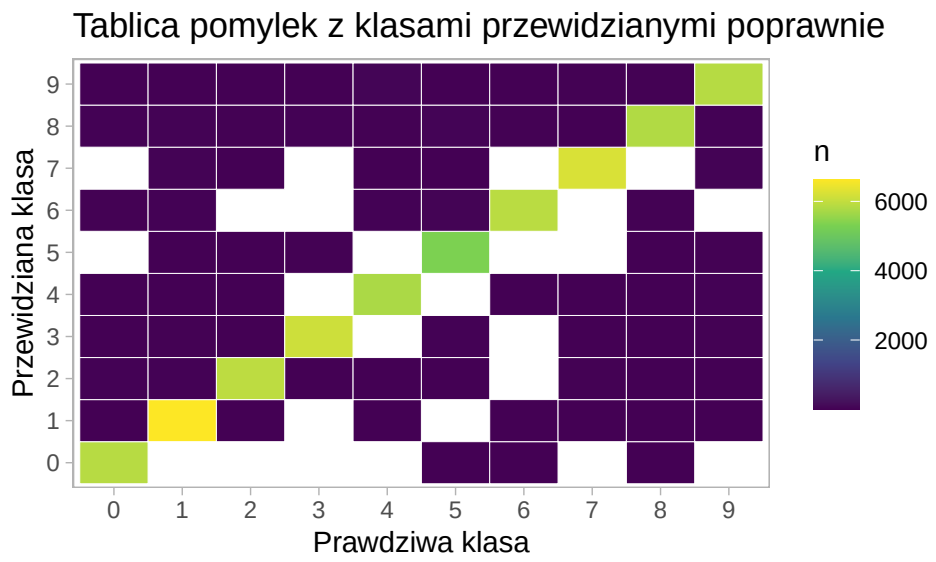
```
x_val_set <- ceiling(x_val_set / 255)
pred_val_set <- load_model_hdf5("model_aug_lr_reduction.h5") %>%
                  predict_classes(x_val_set)
```

Na podstawie tablicy pomyłek widać, że większość cyfr została sklasyfikowana poprawnie.

```
pred_df <- tibble(y_val_set, pred_val_set)

print(table(pred_df))
```

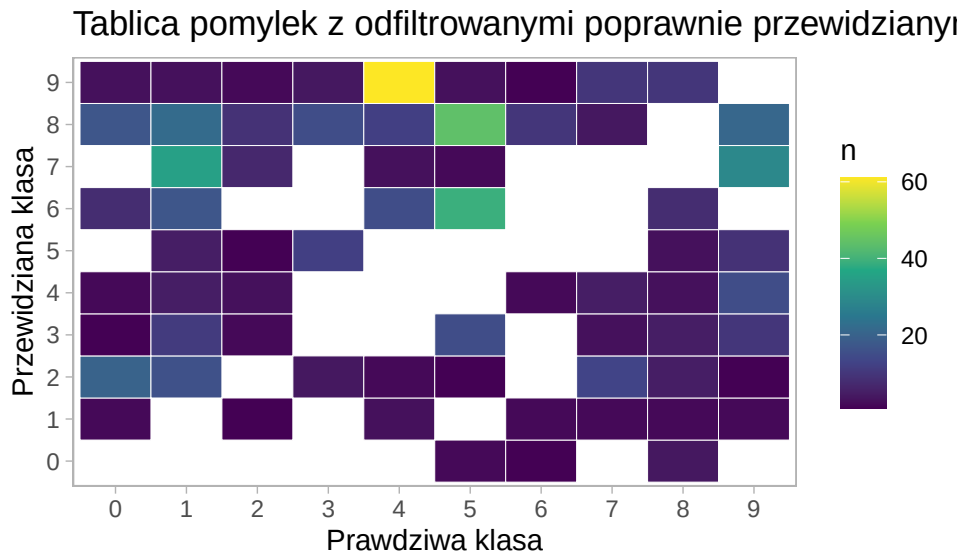
```
##           pred_val_set
## y_val_set    0    1    2    3    4    5    6    7    8    9
##         0 5870    2   20    1    2    0    8    0   17    3
##         1    0 6628   16   11    5    5   17   35   22    3
##         2    0    1 5933    2    3    1    0    7    9    2
##         3    0    0    4 6096    0   12    0    0   15    4
##         4    0    3    2    0 5746    0   15    3   12   61
##         5    2    0    1   15    0 5315   39    2   44    3
##         6    1    2    0    0    2    0 5902    0   10    1
##         7    0    2   13    3    5    0    0 6228    4   10
##         8    4    2    5    5    3    3    8    0 5811   10
##         9    0    2    1   10   15    9    0   29   21 5862
```

```
# tablica pomyłek z klasami przewidzianymi poprawnie
count(pred_df, y_val_set, pred_val_set) %>%
  ungroup() %>%
  ggplot(aes(x = as.factor(y_val_set), y = as.factor(pred_val_set))) +
  geom_tile(aes(fill = n), colour = "white") +
  scale_fill_viridis_c(na.value="#FFFFFF00") +
  theme(panel.grid.major = element_blank()) +
  ylab("Przewidziana klasa") +
  xlab("Prawdziwa klasa") +
  ggtitle("Tablica pomyłek z klasami przewidzianymi poprawnie")
```



Aby zapewnić lepszą przejrzystość, odfiltrowano poprawne obserwacje na diagonalnej. Dzięki temu skala lepiej:

```
count(pred_df, y_val_set, pred_val_set) %>%
  ungroup() %>%
  filter(y_val_set != pred_val_set) %>%
  ggplot(aes(x = as.factor(y_val_set), y = as.factor(pred_val_set))) +
  geom_tile(aes(fill = n), colour = "white") +
  scale_fill_viridis_c(na.value="#FFFFFF00") +
  theme(panel.grid.major = element_blank()) +
  ylab("Przewidziana klasa") +
  xlab("Prawdziwa klasa") +
  ggtitle("Tablica pomyłek z odfiltrowanymi poprawnie przewidzianymi klasami")
```



Największa liczba błędnych klasyfikacji przypada na 4 (prawdziwa klasa) i 9 (predykcja); model często mylił także 5 z 6, 5 z 8 oraz 1 z 7. W przypadku odręcznego pisma, niewielkie zmiany długości lub kształtu niektórych fragmentów tych cyfr łatwo mogą doprowadzić do błędnej klasyfikacji.

Walidacja na podstawie zdjęć pisma odręcznego

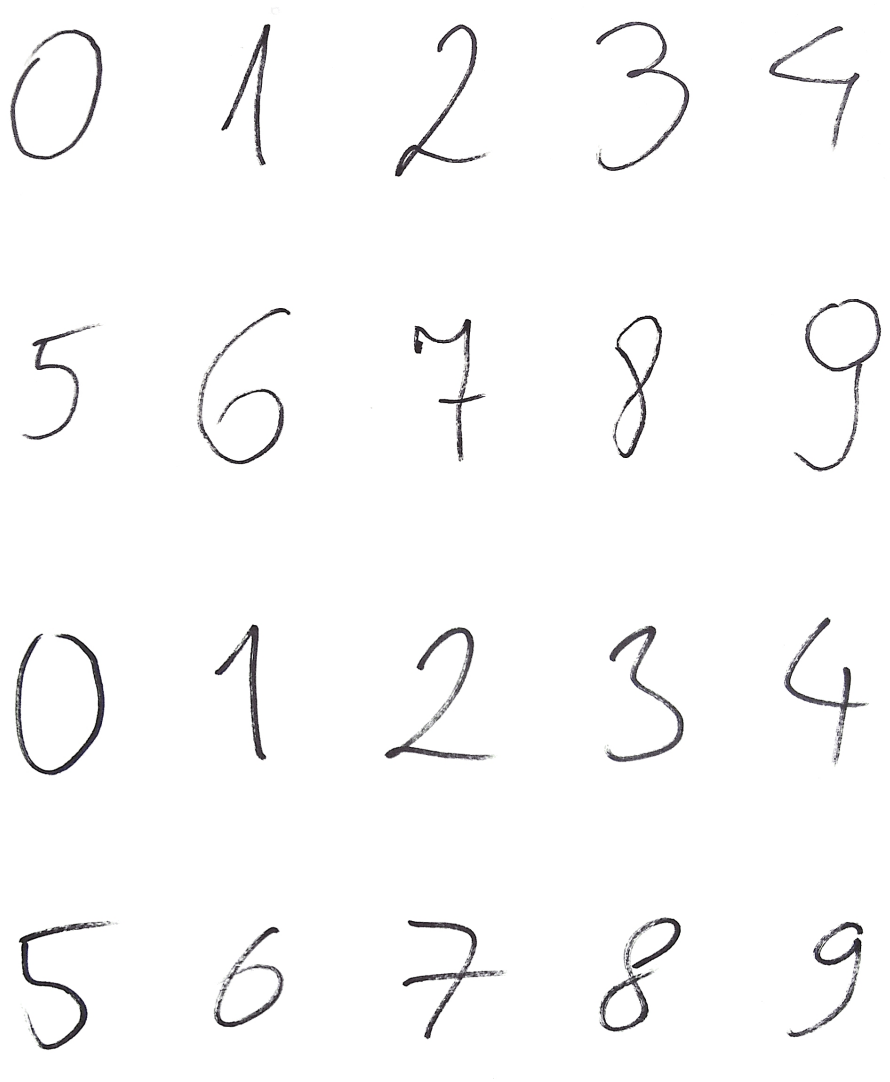
Sprawdzono, w jaki sposób model poradzi sobie z klasyfikacją cyfr niepochodzących ze zbioru MNIST, ale pobranego od dwóch osób. Cyfry zostały napisane odręcznie, a zdjęcia obrobione za pomocą biblioteki EBImage.

```
if (!require("EBImage")) {
  chooseCRANmirror(ind = 0)
  install.packages("BiocManager")
  BiocManager::install("EBImage")
} else {
  library(EBImage)
}
```

Zdjęcia przetworzono za pomocą oprogramowania IrfanView, aby obrazy z cyframi miały taki sam rozmiar.

```
file_list <- list.files(pattern = "*.jpg")

pic_list <- list()
```

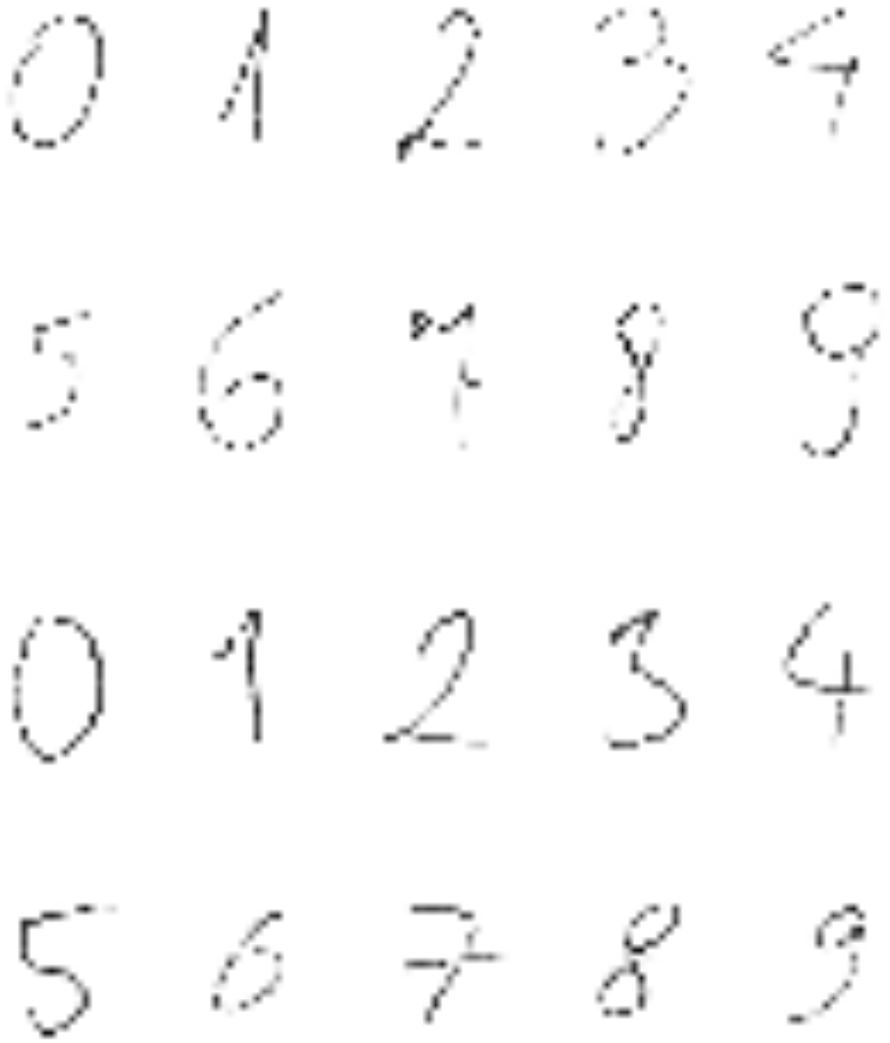
```
for (i in 1:length(file_list)) {
  pic_list[[i]] <- readImage(file_list[[i]])
}

par(mfrow = c(2, 5))
for (i in 1:length(pic_list)) {
  plot(pic_list[[i]])
}
```



Obróbka obrazów polegała na doprowadzeniu ich do takiej postaci, w jakiej znajdowały się dane treningowe. Przetwarzanie obrazów obejmowało po kolei: konwersję na skalę szarości, zmniejszenie rozmiaru do 28x28 pikseli, zmianę kodowania koloru (w pakiecie EBImage 0 jest kodowane jako czarny, 1 jako biały, czyli odwrotnie niż w przypadku poprzednich danych), zmianę kształtu wymaganego przez sieć konwolucyjną i wreszcie połączenie w jeden zbiór danych.

```
# konwersja na skalę szarości
for (i in 1:length(pic_list)) {
```

```
  colorMode(pic_list[[i]]) <- Grayscale
}

# zmiana rozmiarów
for (i in 1:length(pic_list)) {
  pic_list[[i]] <- resize(pic_list[[i]], 28, 28)
}

# narysuj przekształcone cyfry
par(mfrow = c(2, 5))
for (i in 1:length(pic_list)) {
  plot(pic_list[[i]])
}
```



```
par(mfrow = c(1, 1))
# zmiana kodowania koloru
```

```
for (i in 1:length(pic_list)) {
 pic_list[[i]] <- 1 - pic_list[[i]][,,1]
}

# zmiana kształtu danych wejściowych do (10, 28, 28, 1)
for (i in 1:length(pic_list)) {
  pic_list[[i]] <- array_reshape(pic_list[[i]], c(28, 28, 1))
}

# połączenie w jeden zbiór danych
my_handwritten_digits <- NULL

for (i in 1:length(pic_list)) {
  my_handwritten_digits <- rbind(my_handwritten_digits, pic_list[[i]])
}
```

Widać, że operacje te spowodowały znaczne pogorszenie jakości cyfr, jednak dla człowieka w dalszym ciągu są od siebie odróżnialne. Kolejne przekształcenia służą doprowadzeniu zdjęć do takiej samej postaci, jak dane ze zbioru treningowego (normalizacja do zakresu [0, 1] oraz zmiana struktury danych).

```
x_handwritten <- ceiling(my_handwritten_digits)

x_handwritten <- array_reshape(x_handwritten, c(nrow(x_handwritten), 28, 28, 1))
```

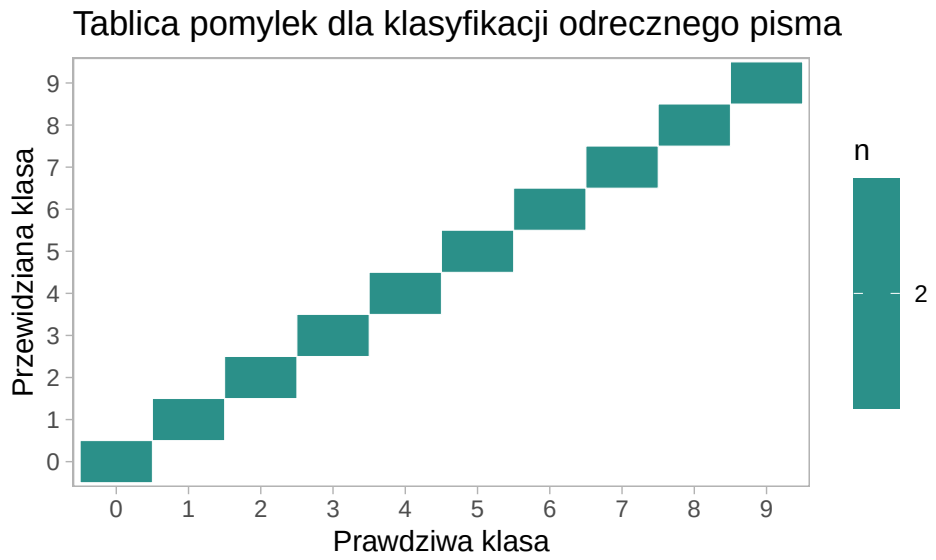
```
## Warning in py_module_import(module, convert = convert): '.Random.seed[1]' nie
## jest poprawną liczbą całkowitą, więc został zignorowany
```

```
y_handwritten <- rep(0:9, 2)
```

Predykcję klas przeprowadzono za pomocą modelu, który osiągnął najlepszy wynik w rankingu Kaggle i wykreślono tablicę pomyłek:

```
# predykcja klas
pred_handwritten <- load_model_hdf5("model_aug_lr_reduction.h5") %>%
          predict_classes(x_handwritten)

handwritten_tibble <- tibble(y_handwritten, pred_handwritten)
```

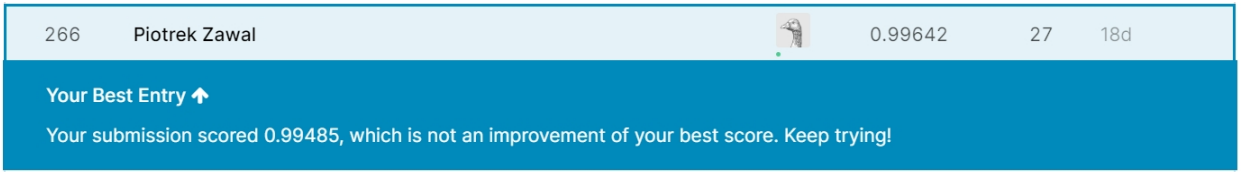
```
handwritten_tibble %>%
  count(y_handwritten, pred_handwritten) %>%
  ungroup() %>%
  ggplot(aes(x = as.factor(y_handwritten), y = as.factor(pred_handwritten))) +
  geom_tile(aes(fill = n), colour = "white") +
  scale_fill_viridis_c(na.value="#FFFFFF00") +
  theme(panel.grid.major = element_blank()) +
  ylab("Przewidziana klasa") +
  xlab("Prawdziwa klasa") +
  ggtitle("Tablica pomyłek dla klasyfikacji odręcznego pisma")
```



Na podstawie tablicy pomyłek widać wyraźnie, że model nie miał żadnych problemów z klasyfikacją cyfr - wszystkie zostały przyporządkowane do prawidłowych klas. Należy zwrócić jednak uwagę, że nie były to wymagające przypadki, wszystkie cyfry posiadały bowiem standardowy kształt, rozmiar i nie były pochylone bądź przesunięte względem środka obrazu.

Podsumowanie

W niniejszej pracy przedstawiono analizę eksploracyjną zbioru MNIST zawierającego zestaw odręcznie pisanych cyfr. Opracowane modele opierają się na głębokich sieciach neuronowych. Nawet mało skomplikowane struktury są w stanie z zadowalającą dokładnością (tj. powyżej 96%) rozpoznawać cyfry. Aby jednak zwiększyć dokładność predykcji, należało wprowadzić wiele poprawek do modelu: zmienić typ sieci z typu wielowarstwowego perceptronu na konwolucyjną sieć neuronową, dodać funkcje filtrujące, rozszerzyć zbiór danych o obrazy obrócone o pewien niewielki kąt oraz regulować tempo uczenia, aby funkcja lepiej przybliżała minimum. Podczas pracy przesłano wyniki opracowanego modelu do Kaggle - najlepszy model oparty o konwolucyjną sieć neuronową osiągnął dokładność 99.642%, co pozwoliło na uplasowanie na 266 miejscu w konkursie (na 2881 uczestników), co pozwoliło znaleźć się w gronie 10% najlepszych wyników.



W tym miejscu należy jednak zwrócić uwage, że duża część uczestników uzyskała dokładność 100%. Aby osiągnąć taki wynik, należy posunąć się do kilku "sztuczek". Najprawdopodobniej osoby te trening swoich modelu przeprowadzały na pełnym zbiorze MNIST (70 000 obserwacji), który podzielony jest na zbiór treningowy (60 000 obs.) i testowy (10 000 obs.). Zbiory na platformie Kaggle odzielone są odpowiednio na 42 000 i 28 000 obserwacji. Nieznane cyfry ze zbioru testowego Kaggle zawarte są w pełnym zbiorze MNIST, co z kolei powoduje, że model trenowany jest na tych samych danych, na których jest walidowany. Doprowadzając do overfittingu, w łatwy sposób uzyskać można dokładność rzędu 100%. Na dzień pisania tej pracy 61 pierwszych miejsc w rankingu ma taką wartość dokładności. Uznaje się, że granica ludzkich możliwościw rozpoznawaniu cyfr ze zbioru MNIST to 99.8%, wynikająca także z faktu, że część cyfr w oryginalnym zbiorze jest przypisana do niepoprawnej klasy. Zaproponowany model, niewymagający dużych nakładów obliczeniowych, wypada więc nieco gorzej, ale pokazano, że jest w stanie poprawnie sklasyfikować cyfry o "standardowym" kształcie.

Bibliografia

Podczas pisania niniejszej pracy korzystano z poniższych źródeł:

1. https://www.r-bloggers.com/exploring-handwritten-digit-classification-a-tidy-analysis-of-the-mnist-dataset/
2. http://repository.supsi.ch/5145/1/IDSIA-04-12.pdf
3. https://www.kaggle.com/kobakhit/digital-recognizer-in-r
4. https://rrighart.github.io/Digits/
5. https://www.kaggle.com/srlmayor/easy-neural-network-in-r-for-0-994
6. http://varianceexplained.org/r/digit-eda/
7. https://blog.prokulski.science/
8. https://www.kaggle.com/timokerremans/cnn-for-minst-dataset