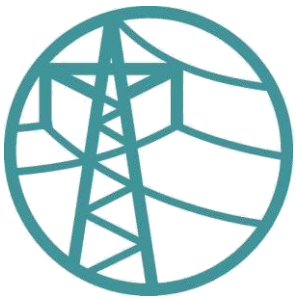


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Kacper Ziółkowski 165014
Paweł Prokop 165090

Realizacja radialnej sieci neuronowej służącej do rozpoznawania liter i cyfr 0-9.

Projekt z przedmiotu Programowanie w Python

Rzeszów, 2023

Spis treści

1. Opis projektu

Celem projektu jest stworzenie radialnej sieci neuronowej potrafiącej rozpoznawać litery oraz cyfry 0-9. Algorytm zostanie zrealizowany w języku Python i wykorzysta do uczenia bazę odręcznie pisanych znaków EMNIST.

2. Opis danych

Dane do trenowania oraz testowania sieci pochodzą z zbioru danych EMNIST https://www.kaggle.com/datasets/crawford/emnist.

Zestaw posiada odręcznie napisane cyfry i litery z przypisanymi do nich klasami. Zbiór zawiera 10 klas dla cyfr oraz 37 klas dla liter. W użytym zbiorze małe i wielkie litery, które wyglądają podobnie są traktowane jako jedna klasa (tj. C, I, J, K, L, M, O, P, S, U, V, W, X, Y, Z).

Dane te możemy zaczytać za pomocą biblioteki gzip. Odczytujemy rozmiar zdjęć (28x28 px), za pomocą którego następnie formatujemy odczytywany ciąg danych do tablicy numpy.

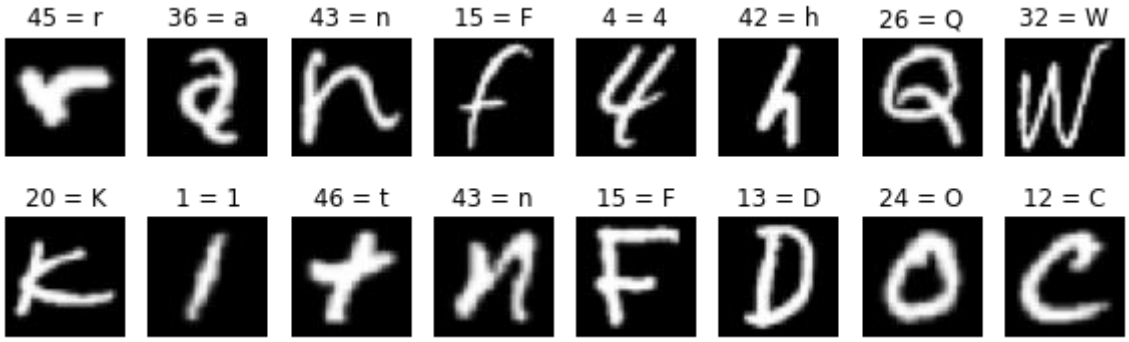
```
def read_emnist(file):
    with gzip.open(file, 'rb') as f:
        print("Plik: ", file)
        # Odczyt big-endian
        z, dtype, dim = struct.unpack('>HBB', f.read(4))
        print("Wymiar: ", dim)
        # Rozmiar każdego wymiaru
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dim))
        print("Kształt: ", shape)
        # Zwraca dane jako przekształcony ciąg numpy
        return np.frombuffer(f.read(), dtype=np.uint8).reshape(shape)

raw_train_X = read_emnist(str(pathlib.Path().resolve()) + '/Dane/emnist-balanced-train-images-idx3-ubyte.gz')
raw_train_Y = read_emnist(str(pathlib.Path().resolve()) + '/Dane/emnist-balanced-train-labels-idx1-ubyte.gz')
raw_test_X = read_emnist(str(pathlib.Path().resolve()) + '/Dane/emnist-balanced-test-images-idx3-ubyte.gz')
raw_test_Y = read_emnist(str(pathlib.Path().resolve()) + '/Dane/emnist-balanced-test-labels-idx1-ubyte.gz')
```

Każde zdjęcie zapisane jest w tablicy o rozmiarze 28x28, gdzie komórka reprezentuje pojedynczy piksel zdjęcia przedstawiającego znak. Wartości pikseli mieszczą się w przedziale od 0 do 255.

Za pomocą matplotlib możemy wyświetlić wczytany zbiór danych wraz z odpowiadającymi im etykietami.

```
plt.figure(figsize=(10, 4))
for i in range(16):
    plt.subplot(2, 8, i + 1)
    plt.imshow(raw_train_X[i].T, cmap='gray') # Dane emnist są transponowane
    title = '%d = %s' % (raw_train_Y[i], labels[raw_train_Y[i]])
    plt.title(title)
```

```
    plt.axis('off')
plt.subplots_adjust(hspace = -.3)
plt.show()
```



Rysunek 2.1 Reprezentacja graficzna elementów zbioru danych

Przed użyciem tych danych w celu szkolenia sieci musimy je znormalizować z zakresu 0-255 do zakresu 0-1, tak aby funkcjonowanie sieci nie było zależne od zakresu danych, jaki będziemy podawać na wejściu.

```
# Konwersja z uint8 na float32
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
# Normalizacja
train_X /= 255
test_X /= 255
```

Dane przedstawiające obraz należy również spłaszczyć, tj. przekształcić z macierzy 28x28 do macierzy 784x1.

```
img_height = raw_train_X[0].shape[0]
img_width = raw_train_X[0].shape[1]
input_shape = img_height * img_width
train_X = train_X.reshape(len(train_X), input_shape)
test_X = test_X.reshape(len(test_X), input_shape)
```

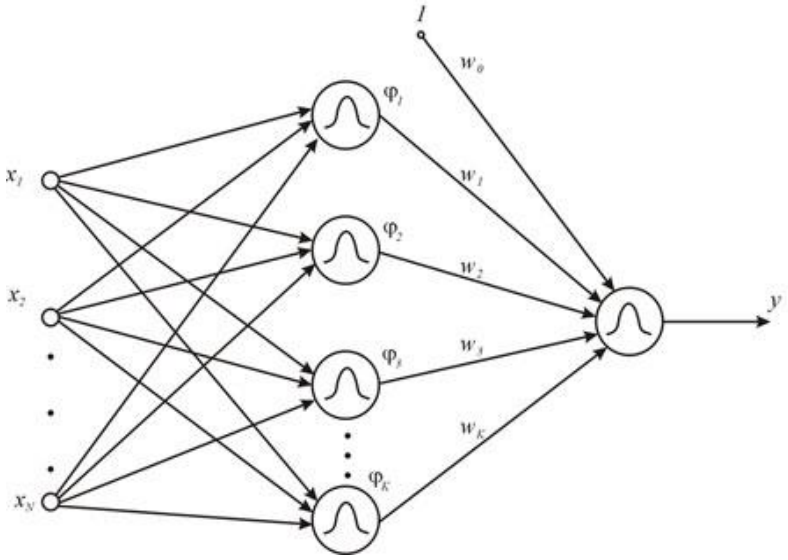
W projekcie zostało użyte 1410 zdjęć (po 30 zdjęć dla każdej klasy) do trenowania oraz 470 (po 10 zdjęć dla każdej klasy) oddzielnych od zbioru trenującego zdjęć do testowania.

3. Opis teoretyczny

1. Działanie sieci RBF

Typowa sieć radialna zbudowana jest z dwóch warstw: ukrytej z neuronami radialnymi oraz wyjściowej: liniowej, która sumuje wagi sygnałów z neuronów ukrytych.

Jako wejście podawana jest lista wartości pikseli zdjęć znaków. Każdy element x_n odpowiada pojedynczej wartości z listy i ma znormalizowany zakres 0-1 reprezentujący jasność piksela. Wartości te przekazywane są do warstwy ukrytej.



Rysunek 3.1 Budowa sieci radialnej

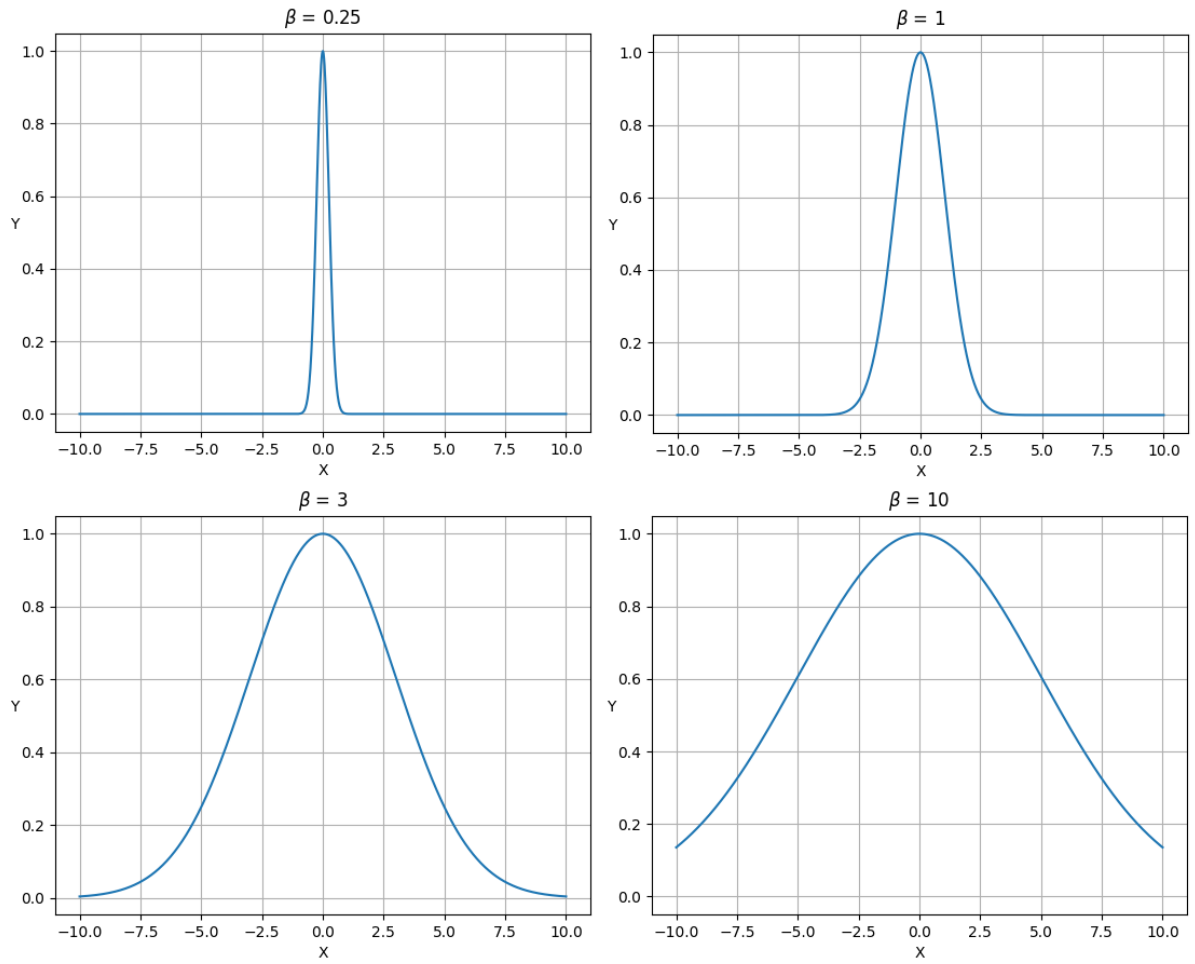
Neurony warstwy ukrytej φ_n realizują funkcję reprezentującą hipersferę wokół punktu centralnego, centroida. Obliczana ona jest za pomocą funkcji

$$e^{-\frac{(x-c)^2}{2\beta^2}}, \tag{1}$$

gdzie:
β jest parametrem kontrolującym szybkość opadania funkcji
x jest elementem podanym na wejściu
c jest centroidą neuronu
Wartość zwracana przez tą funkcję maleje wykładniczo wraz ze wzrostem odległości x od centrum. Zmieniając parametr skalarny β kontrolujemy jak szybko ta funkcja opada.

Za pomocą biblioteki matplotlib możemy narysować wykres przedstawiający wpływ parametru β na kształt funkcji radialnej. Na zakres osi Y został nałożony sztywnie zakres wartości od -0.05 do 1.05. Aby w tytule wstawić znak beta używamy składni TeX.

```
x1 = np.linspace(-10, 10, 1000)
y1 = np.exp(-(x1**2) / (2*5**2))
fig, ax = plt.subplots()
ax.plot(x1, y1)
plt.ylabel('Y', rotation=0)
plt.ylim([-0.05, 1.05])
ax.set(xlabel='X', title='$\\beta$ = 10')
ax.grid()
plt.show()
```



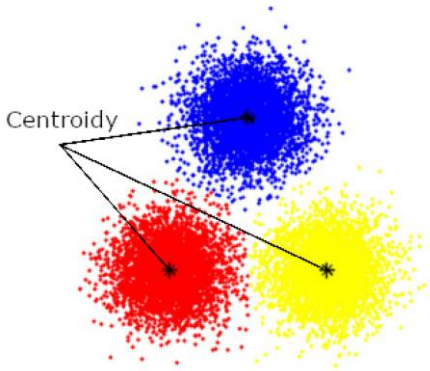
Rysunek 3.2 Wpływ parametru β na kształt funkcji radialnej

Warstwa wyjściowa sumuje wszystkie sygnały otrzymane od neuronów warstwy ukrytej pomnożone przez ich wagę. Waga wpływu neuronu ukrytego na wyjściowy może mieć wartości zarówno dodatnie jak i ujemne w zależności jak bardzo neuron ukryty odzwierciedla daną klasę. Neuron wyjściowy z największą sumą jest odczytywany jako wynik.

Proces uczenia sieci możemy kontrolować poprzez zmianę ilości centroid w warstwie ukrytej oraz współczynnika β używanego w funkcji radialnej.

2. Znajdowanie centroid za pomocą K-Means Clustering

K-Means clustering jest algorytmem uczenia nienadzorowanego, którego celem jest podzielenie n elementów na k grup, w których każdy element należy do klastra z najbliższym centroidem. Algorytm ma na celu zminimalizowanie kwadratu odległości euklidesowych między elementami a centroidami klastra, do którego należy.



Rysunek 3.3 Wizualizacja wyniku k-means clustering

Metoda wyznaczania środków centroid opiera się na czterech zasadach:

1. Losowym wyznaczeniu k punktów
2. Przypisaniu elementów do centroid, które są im najbliższe
3. Przeniesieniu centroida do środka przypisanych jemu elementów
4. Powtórzeniu kroku 2 i 3 aż do uzyskania zbieżności

3. Wyznaczanie wag neuronów

Mając wyznaczone centroidy φ możemy podać zbiór danych, aby otrzymać macierz X reprezentującą otrzymane wartości sygnałów RBF warstwy ukrytej. Z taką macierzą możemy wyznaczyć wagi sieci za pomocą regresji liniowej

$$w = (X^T X)^{-1} X^T y, \tag{2}$$

gdzie:
w oznacza macierz wag o rozmiarach [p, k], gdzie *p* oznacza ilość neuronów a *k* oznacza ilość klas
X jest macierzą sygnałów neuronów dla otrzymanych zdjęć p rozmiarach [x, p], gdzie *x* oznacza ilość elementów wejściowych
y jest macierzą zer z jedynką w miejscu reprezentującym klasę elementu o rozmiarach [x, k]

4. Algorytm

1. Kod na obliczenie euklidesowej odległości

Do obliczania odległości miedzy dwoma elementami używamy funkcji na odległość euklidesową wyrażaną wzorem

$$d = \sqrt{\sum_{i=1}^{p} (v_{1i} - v_{2i})^2}$$

W tym celu przechodzimy w pętli przez każdą oś dwóch wektorów i sumujemy do siebie kwadrat ich różnicy. Na końcu używamy numpy do obliczenia pierwiastka z zsumowanych odległości.

```
def euc_dist(x, y):
    d = 0 #Ustawiamy początkową odległość na 0
    for i in range(len(x)): #Przechodzimy przez wszystkie wartości w macierzy (1xn)
        d += (x[i] - y[i])**2 #Sumujemy kwadrat różnic między elementami
    return np.sqrt(d) #Wyciągamy pierwiastek z obliczonej sumy
```

2. Kod na wyznaczenie centroid za pomocą K-Means clustering

Kod jako środki centroid ustawia k losowych elementów z przekazanej tablicy X. Tworzymy listę zawierającą listy, w których zapisywać będziemy elementy przynależna do danej centroidy. Następnie dla każdego zdjęcie wyznaczamy odległość od wszystkich centroid a następnie dodajemy je do klastra najbliższej centroidy.

Po przejściu przez wszystkie zdjęcia wyznaczane są pozycje centroid jako środek elementów klastra. Następnie sprawdzamy różnicę pomiędzy poprzednią iteracją. Jeśli nie ma żadnych zmian to oznacza że centroidy są już w optymalnej pozycji i możemy zakończyć obliczenia. Jeżeli jest różnica to ponownie wykonujemy wyznaczanie klastrów i centroid.

```
def kmeans(X, k):
    #Wybieramy k losowych elementów jako początki centroid
    centroids = X[np.random.choice(range(len(X)), k, replace=False)]

    #Funkcja ma 100 powtórzeń na znalezienie optymalnego rozkładu
    for i in range(100):
        #Lista [[][]...[]] przynależnych elementów do centroid
        clusters = [[] for _ in range(len(centroids))]

        for x_i, x in enumerate(X):  #Przejdź przez każdy element
            distances_list = []
            for c_i, c in enumerate(centroids): #Przejdz przez centroidy
                distances_list.append(euc_dist(c, x)) #Odległość centroidy od zdjęcia

            #Element przypisywany jest do najbliższego klastra / centroidy
            clusters[int(np.argmin(distances_list))].append(x)

        centroids_old = centroids.copy() # Kopiowanie do porównania

        #Kalkuluje średnią pozycję centroid na podstawie zdjęć przypisanych do klastra
        centroids = [np.mean(c, axis=0) for c in clusters]

        #Porównanie z poprzednią iteracją
        diff = np.abs(np.sum(centroids_old) - np.sum(centroids))

        if diff == 0: #Jeśli jest optymalne rozmieszczenie to zakończ
            break

    return centroids # Zwraca centroidy
```

Wynikiem są centroidy, które możemy przedstawić również jako zdjęcia. W niektórych przypadkach można w miarę łatwo przypisać do grupy liter lub cyfr, które wpłynęły na daną centroidę.

Rysunek 4.1 Wyznaczone centroidy

3. Kod klasy RBF

Sieć RBF zrealizowana jest jako klasa. Podczas inicjalizacji podajemy zbiory i etykiety do trenowania oraz testowania, jak i ilość neuronów (centroid) sieci.

Aby wyznaczyć centroidy mamy zdefiniowaną funkcję, która wywoła omówioną wcześniej funkcję do znajdowania centroid.

Do obliczenia wartości sieci danego elementu od centroidy wywołujemy funkcję get_rbf. Realizuje ona wzór warstwy ukrytej, realizującej hipersferę neuronu. W tym celu obliczamy odległość euklidesową pomiędzy zdjęciem i centroidą. Zwracamy wartość funkcji wykładniczej na podstawie tej odległości oraz parametru sieci β.

Funkcja get_rbf_matrix tworzy tablicę numpy zawierającą wartość siły neuronów do wszystkich zdjęć.

Do wyznaczania wag sieci używamy takową tablicę wartości neuronów. Za pomocą np.linalg.pinv obliczamy pseudo inwersję macierzy, którą mnożymy z transponowaną tablicą wartości neuronów oraz tablicą 0 / 1 oznaczającą za pomocą 1 wiersz etykiety.

```
class RBF:

    def __init__(self, X, Y, TX, TY, k):
        self.TrX = X #Zdjęcia do trenowania
        self.TrY = Y #Klasy zdjęć do trenowania
        self.TeX = TX #Zdjęcia do testowania
        self.TeY = TY #Klasy zdjęć do testowania
        self.k = k #Ilość centroid / neuronów

    #Zwraca wartość funkcji radialnej
    def get_rbf(self, x, c, B):
        distance = euc_dist(x, c) #Odległość elementu od centroida
        return np.exp(-(distance**2) / (2*B**2)) #Funckja radialna

    #Zwraca listę rbf dla podanych elementów
    def get_rbf_matrix(self, X, centroids, B):
        #Dla każdego elementu zwraca wartość rbf od wszystkich centroid
        return np.array([[self.get_rbf(x, c, B) for c in centroids] for x in X])

    #Wyznacza centroidy dla elementów
    def find_kmeans(self):
        self.centroids = kmeans(self.TrX, self.k)

    #Wyznaczanie wag
    def fit(self, B):
        #Zwraca macierz rbf dla elementów trenujących
        RBF_X = self.get_rbf_matrix(self.TrX, self.centroids, B)
        #Oblicza wagi na podstawie listy rbf i odpowiadających jej elementom klas
        self.W = np.linalg.pinv(RBF_X.T @ RBF_X) @ RBF_X.T @
keras.utils.np_utils.to_categorical(self.TrY)

    #Testowanie sieci
    def test(self, B):
        #Zwraca macierz rbf dla elementów testowych
```

```
        RBF_TX = self.get_rbf_matrix(self.TeX, self.centroids, B)
        self.PredTeY = RBF_TX @ self.W #Mnożenie rbf z wagami

        #Wybiera neurony z największą sumą jako wynik
        self.PredTeY = np.array([np.argmax(x) for x in self.PredTeY])

        #Oblicza różnicę między klasami przewidzianymi a prawdziwymi
        diff = self.PredTeY - self.TeY
        #Oblicza stosunek prawidłowych trafień
        self.Accuracy = len(np.where(diff == 0)[0]) / len(diff)
        print('Accuracy: ', self.Accuracy)

        #Tworzenie tabeli klasyfikacji
        self.accuracy_table = np.zeros((47,47))
        #Przechodzi przez wszystkie elementy testujące
        for i, y in enumerate(self.TeY):
            #Zwiększa numerację w odpowiednim miejscu tabeli
            self.accuracy_table[y, self.PredTeY[i]] += 1
            #Dodaje element po drugiej stronie dla symetryczności
            if y != self.PredTeY[i]:
                self.accuracy_table[self.PredTeY[i], y] += 1
```

4. Kod inicjalizujący, trenujący oraz testujący sieć

Poniższy kod tworzy sieć rbf, dla której znajduje rozmieszczenie 200 centroid za pomocą k-means i wyznacza dla nich wagi przy parametrze β funkcji radialnej równym 6. Na końcu wyznaczona sieć jest testowana oddzielnym zbiorem testowym.

```
RBFNN = RBF(train_X, train_Y, test_X, test_Y, 200)
RBFNN.find_kmeans()
RBFNN.fit(6)
RBFNN.test(6)
```

5. Obliczanie wielordzeniowe

W powyższej konfiguracji możemy trenować wyłącznie jedną sieć w danym momencie. Jest to dosyć długotrwały proces. Obliczając w wielu wątkach kilka sieci jednocześnie możemy skrócić całkowity czas uczenia wielu sieci.

W tym celu definiujemy funkcje, która wytrenuje sieć i zapisze jej wyniki do pliku.

```
def Fit(B,cl):
    RBF_CLASSIFIER = RBF(train_X, train_Y, test_X, test_Y, cl)
    RBF_CLASSIFIER.centroids = np.load('Output/30/centroids ' + str(cl) + ' - 30.npy')
    print("Fiting: cl" + str(cl) + " B" + str(B))
    RBF_CLASSIFIER.fit(B)
    RBF_CLASSIFIER.test(B)
    np.save('Output/30/weights ' + str(cl) + ' - 30 v3 B' + str(B) + " acc" +
str(RBF_CLASSIFIER.Accuracy) + ".npy", RBF_CLASSIFIER.W)
    np.save('Output/30/acc_table ' + str(cl) + ' - 30 v3 B' + str(B) + ".npy",
RBF_CLASSIFIER.accuracy_table)
```

Mając funkcję, która potrafi wytrenować sieć możemy ją uruchomić w wielu wątkach jednocześnie, podając do niej różne parametry sieci. W tym celu używamy biblioteki multiprocessing.

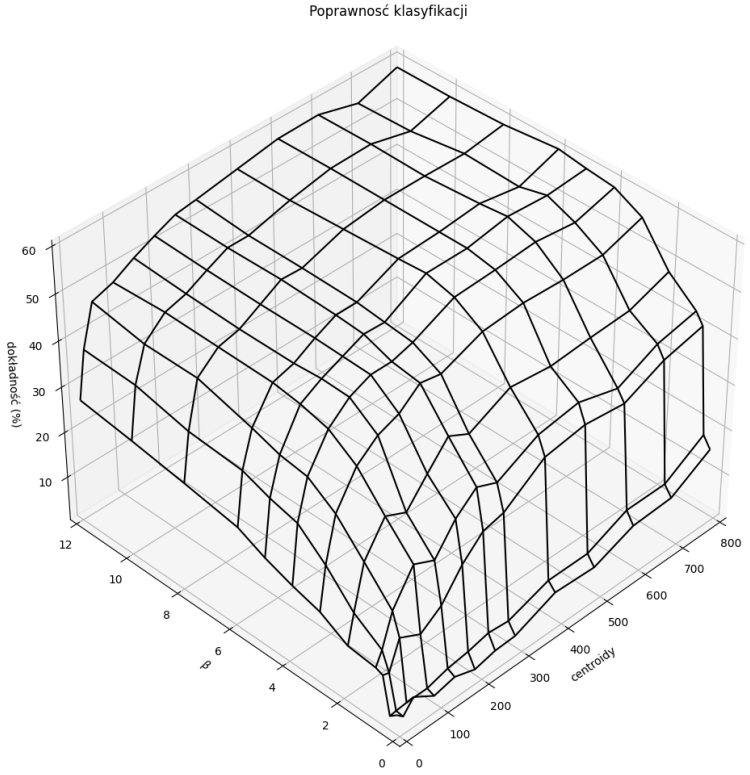
```
import multiprocessing as mp
if __name__ == '__main__':
    with mp.Pool(5) as pool:
        a = [0.25, 0.5, 0.75, 1, 2, 3, 4, 5, 6, 8, 10, 12]
        b = [10] * 20
        pool.starmap(Fit, zip(a,b))
```

5. Eksperymenty

1. Wpływ parametrów na poprawność klasyfikacji

Otrzymaną dokładność z testów wytrenowanych sieci dla różnych parametrów możemy przedstawić za pomocą matplotlib. Używamy krawędziowego wykresu 3D, gdzie na osiach poziomych mamy ilość neuronów oraz parametr β, a na osi pionowej mamy dokładność sieci.

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.set_xlabel('centroidy')
ax.set_ylabel('$\\beta$')
ax.set_title("Poprawnosć klasyfikacji")
ax.set_zlabel('dokładność (%)', rotation=-90)
ax.plot_wireframe(X, Y, Z, edgecolor='black')
plt.show()
```



Rysunek 5.1 Poprawność klasyfikacji sieci dla różnej ilości centroid oraz współczynnika β

Sieć jest bardzo czuła na zbyt niski parametr β. Dla małych wartości nie ma ona możliwości uogólniania. Neurony nie są w stanie reprezentować grupę, a tylko wartości bardzo bliskie centroidom. Zdjęcia, które powinny być zaliczone do grupy reprezentowanej przez dany neuron są zbyt oddalone, aby zostać „uwzględnione" w funkcji radialnej.

Zwiększanie szerokości funkcji radialnej daje znaczącą poprawę. Sieć bardzo szybko otrzymuje charakterystykę uogólniającą i już przy parametrze β równym 6 sieć ma najlepsze wyniki. Sieć z 800 neuronami oraz parametrem β równym 6 Dalsze zwiększanie tego parametru nie poprawia dokładności sieci.

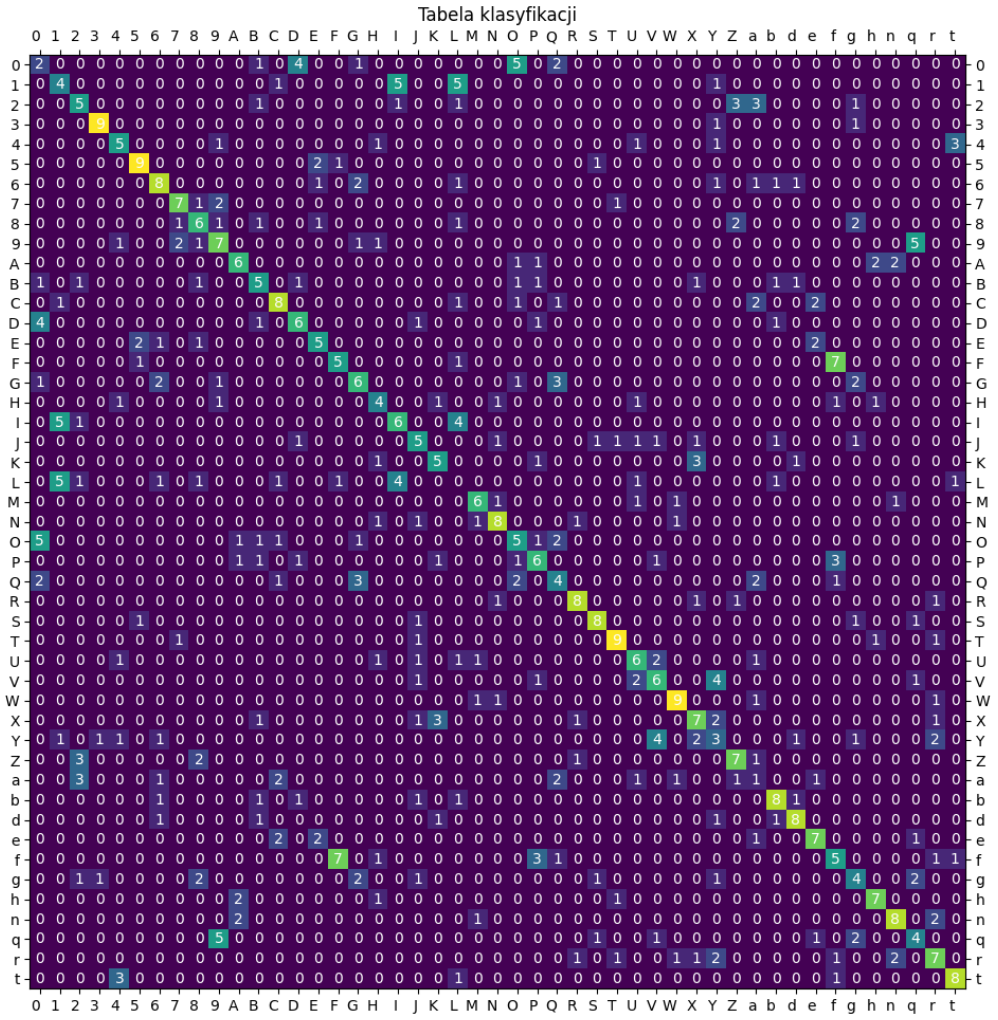
Sieć z małą ilością neuronów nie jest w stanie reprezentować wszystkich klas. Liczba neuronów jest mniejsze od liczby klas, więc centroidy zamiast reprezentować jedną klasę są ustawione między kilkoma podobnie wyglądającymi, lecz odmiennymi klasami.

Dla większej ilości centroid każda klasa ma więcej grup ją reprezentujących. Dalsze zwiększanie ilości centroid ma pozytywny wpływ na dokładność sieci. Te same klasy zdjęć z różnymi stylami zapisów mogą mieć swoje oddzielne neurony co wpływa pozytywnie na zdolności klasyfikacji znaków z wieloma stylami zapisów.

2. Błędy klasyfikacji sieci

Aby sprawdzić problemy sieci z błędną klasyfikacją utworzono tabelę reprezentującą klasy elementów testujących oraz przewidzianych przez sieć. Użyto sieci, dla której otrzymano najlepsze rezultaty, tj. 800 neuronów z parametrem β równym 6. Do przedstawienia danych użyto

```
acc_table = np.load(str(pathlib.Path().resolve()) + "/Output/30/acc_table 800 - 30 v3 B6.npy")
labels = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabdefghnqrt'

fig, ax = plt.subplots()
im = ax.imshow(acc_table)

# Oznaczenia wierszy i kolumn
ax.set_xticks(np.arange(len(labels)), labels=labels)
ax.set_yticks(np.arange(len(labels)), labels=labels)
plt.setp(ax.get_xticklabels(), rotation=0, ha="right", rotation_mode="anchor")

# Wstawia dane
for i in range(len(labels)):
    for j in range(len(labels)):
        text = ax.text(j, i, round(acc_table[i, j]), ha="center", va="center", color="w")

ax.set_title("Tabela klasyfikacji")
fig.tight_layout()
plt.show()
```



Rysunek 5.2 Tabela klasyfikacji sieci z 800 neuronami i współczynnikiem β=6

Sieć popełnia dużo błędów przy klasyfikacji znaków o podobnym kształcie, np. cyfrę 0 myli z literami O, D i Q, cyfrę 1 myli z literami I, L, itd. Błędy te ze względu na różne sposoby ręcznego zapisu znaków są trudne bądź też niemożliwe do wyeliminowania.

6. Bibliografia

[1] Euclidean Distance raw, normalized, and double-scaled coeficients https://www.pbarrett.net/techpapers/euclid.pdf

[2] https://towardsdatascience.com/most-effective-way-to-implement-radial-basis-function-neural-network-for-classification-problem-33c467803319

[3] Radial Basis Function Networks, K.-L. Du, M.N.s. Swamy

[4] Unsupervised learning Clustering, Shimon Ullman, Tomaso Poggio http://www.mit.edu/~9.54/fall14/slides/Class13.pdf

[5] https://notebook.community/WNoxchi/Kaukasos/research/idx-to-numpy