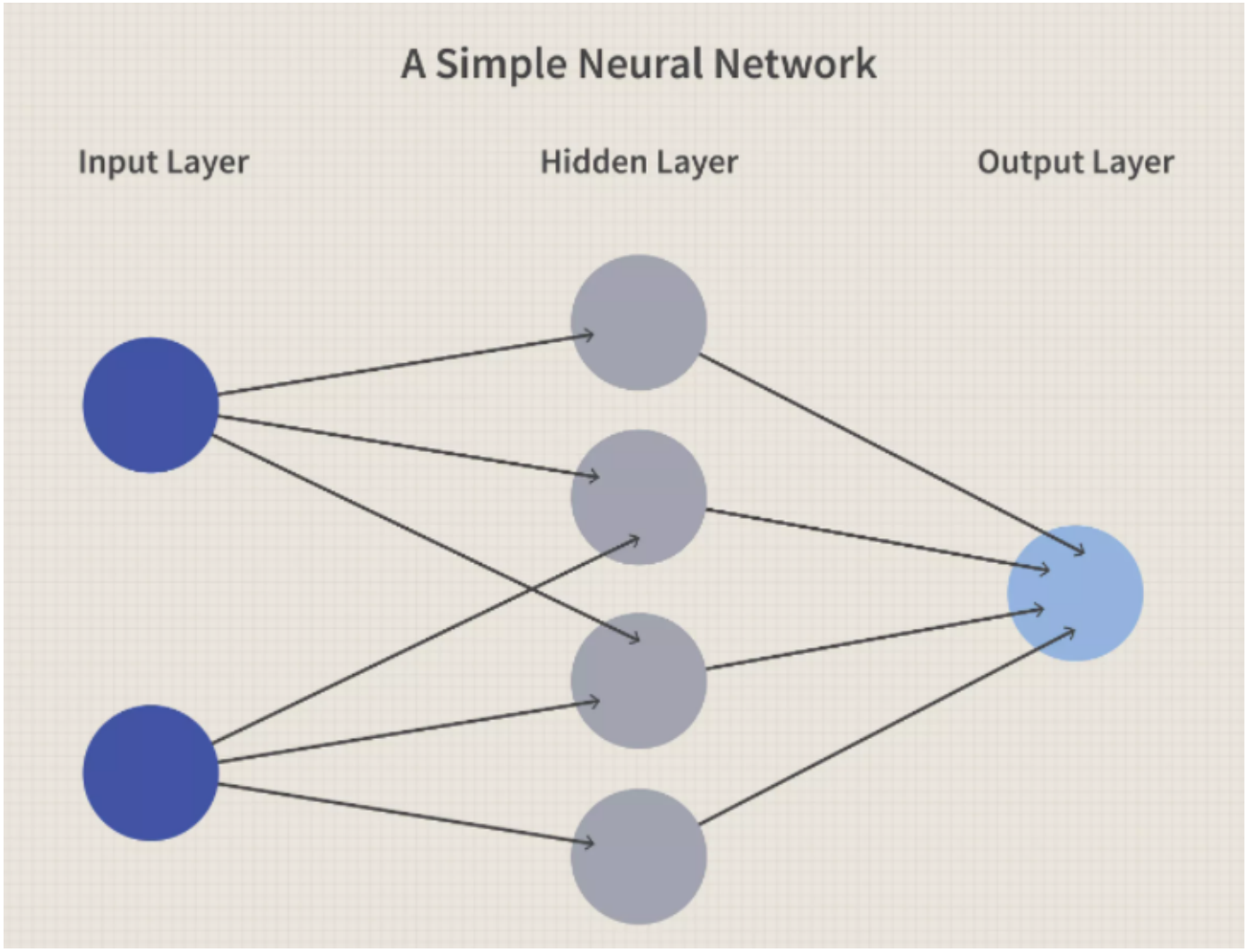
Sprawozdanie projekt SI

Nasz projekt związany jest z rozpoznawaniem cyfr ze zbioru MNIST. Wykorzystaliśmy do tego KNN, SVM i sieci neuronowe.

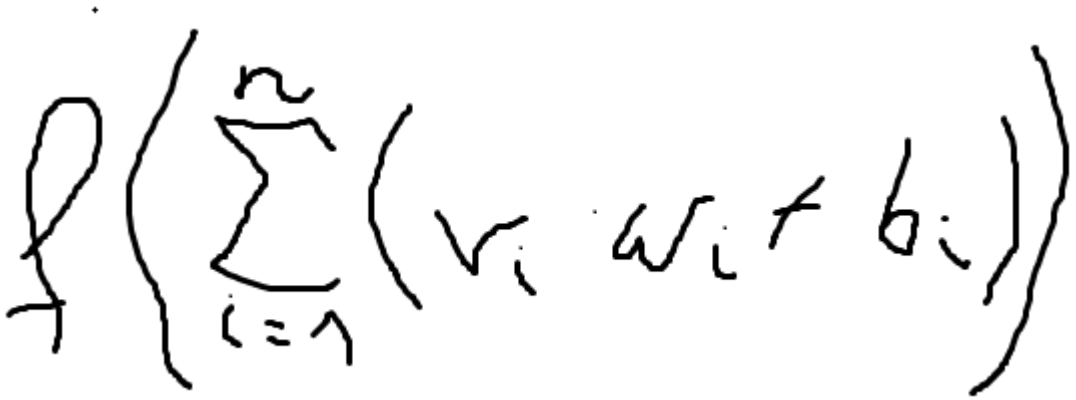
1) sieci neuronowe



Sieć neuronowa składa się z input layer, hidden layers i output layer. Sieć FCNN (fully connected neural networks) to taka sieć, w której każdy neuron jednej warstwy jest połączony dokładnie raz z neuronami z kolejnej warstwy tak jak na rysunku powyżej. Na krawędziach znajdują się wagi (w) i biasy (b). W neuronach są wartości (v). Output layer wylicza sumę:

$$\sum_{i=1}^{n} (v_i \cdot w_i + b_i)$$

No i to co robi sieć neuronowa podczas treningu to dopasowuje wagi i biasy każdej krawędzi, tak żeby otrzymywać coraz więcej i więcej poprawnych odpowiedzi na wyjściu tzn. poprawnych tych sum właśnie. Jednak łatwo zauważyć, że ta ważona suma jest po prostu liniowa co nas ogranicza w różnych aspektach, dlatego też korzysta się z innych funkcji aktywacji, które zapewniają nieliniowość. Działają w ten sposób:



Mamy kilka rodzajów funkcji aktywacji np.:

1. funkcje liniowe:

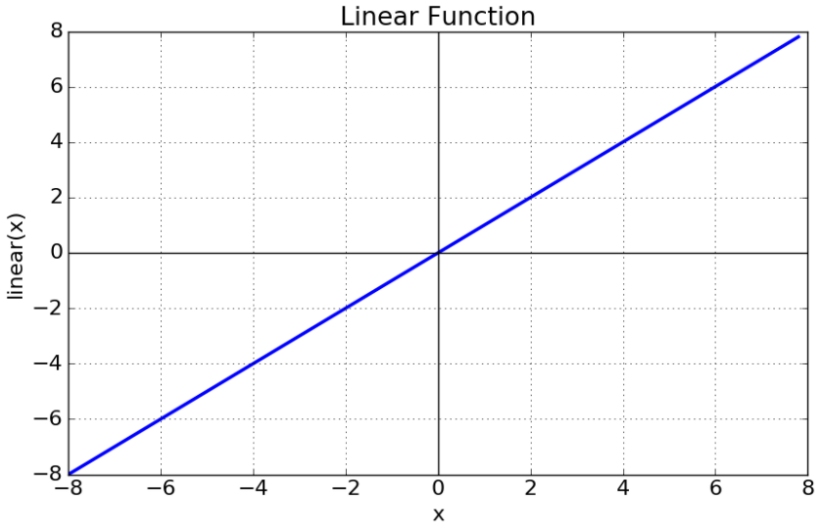


Fig: Linear Activation Function

Equation : $f(x) = x$

2. funkcje nieliniowe:
   - sigmoid

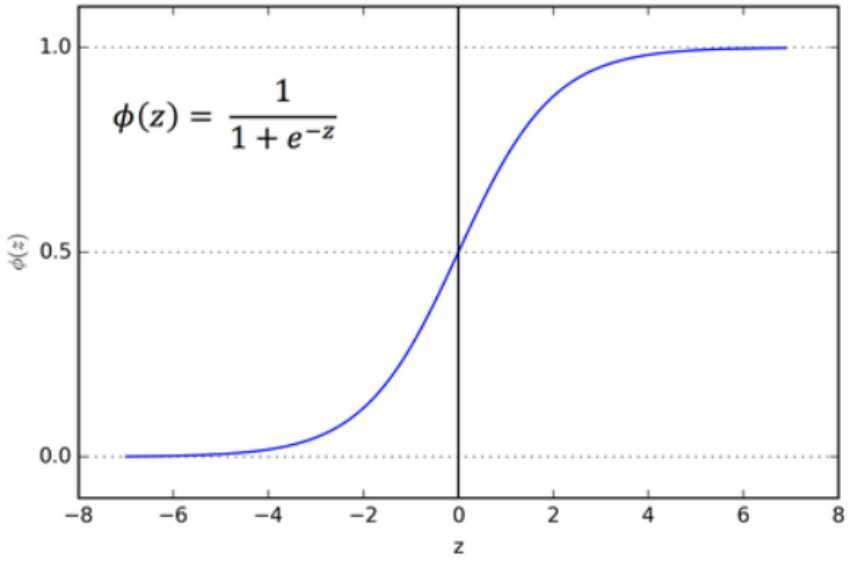
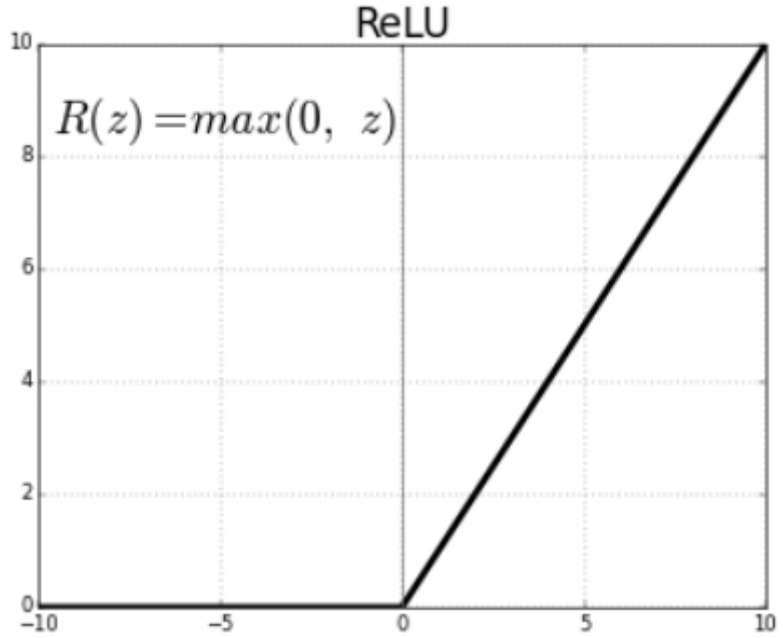


Fig: Sigmoid Function

   - relu (Rectified Linear Units)



- softmax

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

I teraz jak już mamy ten wynik to możemy sprawdzić jak bardzo jest on dobry albo jak bardzo jest zły, czyli po prostu możemy obliczyć błąd. Do tego służą loss functions. Też mamy kilka rodzajów tych funkcji np.:

- **Mean Squared Error (MSE)**
- **Binary Crossentropy (BCE)**
- **Categorical Crossentropy (CC)**
- **Sparse Categorical Crossentropy (SCC)**

Wspomnieliśmy o tym, że sieć neuronowa podczas treningu dopasowuje wagi i biasy, tak żeby otrzymywać na wyjściu coraz więcej poprawnych odpowiedzi. Ale jak sieć neuronowa dopasowuje/aktualizuje wagi i biasy na krawędziach? Do tego służą optimizery. Optimizery to są algorytmy, które odpowiadają za zmienianie tych wartości Mamy również kilka takich algorytmów między innymi:
- adam (adaptive moment estimation)
- RMS (root mean squared propagation)
- SGD (stochastic gradient descent)

Warto jeszcze wspomnieć, że epoka to trening wszystkich danych treningowych dokładnie raz.

Z wiedzy opisanej powyżej postaramy się skorzystać i dostarczyć różnych statystyk, które zaprezentujemy poniżej:

Pierwsza statystyka dotyczy porównania optimaizerów tych trzech, które wymieniliśmy wyżej.

Dana sieć neuronowa:

```
model = tf.keras.models.Sequential()  # it creates an empty model object

model.add(tf.keras.layers.Flatten())  # it converts an N-dimentional layer to a 1D layer

# hidden layers
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='sigmoid'))

# output layer - the number of neurons in the softmax must be equal to the number of classes
model.add(tf.keras.layers.Dense(10, activation='softmax'))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
```

Optimizer adam dla danych treningowych w ostatniej 5 epoce:

```
Epoch 5/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0415 - accuracy: 0.9866
```

A tu dla naszych danych testowych:

```
Accuracy = 100.0%
```

To samo dla optimizera SGD:

```
Epoch 5/5
1875/1875 [==============================] - 2s 937us/step - loss: 0.2954 - accuracy: 0.9143
```

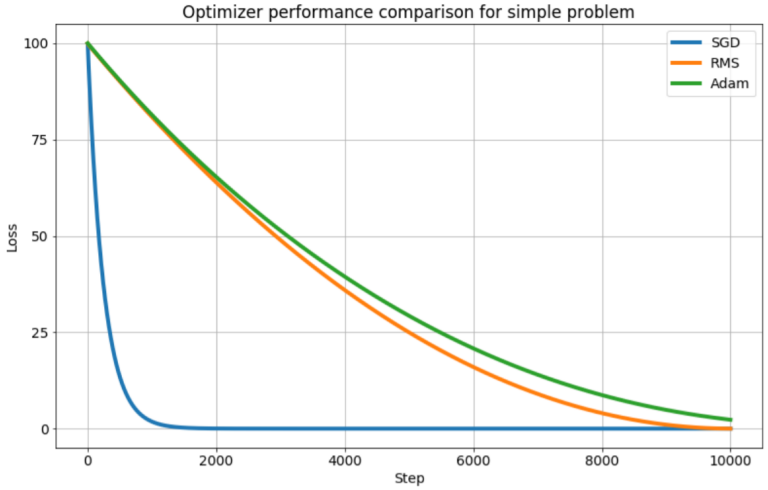
```
Accuracy = 90.0%
```

I dla RMS:

```
Epoch 5/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0561 - accuracy: 0.9827
```

```
Accuracy = 100.0%
```

Podsumujmy to wykresem:



W dalszych rozważaniach zostaniemy przy optimizerze adam.

Następna statystyka dotyczy liczby neuronów w warstwach ukrytych.

Dane warstwy ukryte (256 i 128 neuronów):

```
# hidden layers
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='sigmoid'))
```

Wyniki:

```
Epoch 5/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0415 - accuracy: 0.9866
```

```
Accuracy = 100.0%
```

Tutaj 16 i 8 neuronów:

```
# hidden layers
model.add(tf.keras.layers.Dense(16, activation='relu'))
model.add(tf.keras.layers.Dense(8, activation='sigmoid'))
```

Wyniki:

```
Epoch 5/5
1875/1875 [==============================] - 1s 649us/step - loss: 0.2560 - accuracy: 0.9306
```

```
Accuracy = 90.0%
```

A tu 1024 i 512 neuronów:

```
# hidden layers
model.add(tf.keras.layers.Dense(1024, activation='relu'))
model.add(tf.keras.layers.Dense(512, activation='sigmoid'))
```

Wyniki:

```
Epoch 5/5
1875/1875 [==============================] - 17s 9ms/step - loss: 0.0412 - accuracy: 0.9862
```

```
Accuracy = 90.0%
```

Widać że warstwy ukryte z 16 i 8 neuronami odbiegają znacząco od dwóch pozostałych par warstw ukrytych patrząc na skuteczność wytrenowania w 5 epoce. Natomiast jak widać to wcale nie jest tak, że im więcej neuronów tym lepiej, dlatego że warstwy ukryte z 256 i 128 neuronami poradziły sobie troszeczkę lepiej niż warstwy ukryte z 1024 i 512 neuronami.

Teraz porównamy jak liczba warstw ukrytych i funkcje aktywacji wpływają na wynik.

Jedna warstwa ukryta z funkcją aktywacji relu:

```
# hidden layers
model.add(tf.keras.layers.Dense(256, activation='relu'))
```

```
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0435 - accuracy: 0.9859
```

```
Accuracy = 100.0%
```

Jedna warstwa ukryta z funkcją aktywacji sigmoid:

```
# hidden layers
model.add(tf.keras.layers.Dense(256, activation='sigmoid'))
```

```
Epoch 5/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0752 - accuracy: 0.9780
```

```
Accuracy = 100.0%
```

Dla jednej warstwy widać, że relu radzi sobie znacznie lepiej niż sigmoid.

Dwie warstwy ukryte:

```
# hidden layers
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='sigmoid'))
```

```
Epoch 5/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0415 - accuracy: 0.9866
```

```
Accuracy = 100.0%
```

Dwie warstwy ukryte:

```
# hidden layers
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='relu'))
```

```
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0485 - accuracy: 0.9837
```

```
Accuracy = 100.0%
```

Natomiast w tym przypadku kombinacja dwóch różnych funkcji aktywacji daje lepszy wynik, mimo że dla jednej warstwy ukrytej relu poradził sobie znacznie lepiej. Wybór funkcji aktywacji zależy pewnie od sytuacji i od problemu. Jednak dla naszych danych tak właśnie się stało :).

Trzy warstwy ukryte:

```
# hidden layers
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='sigmoid'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
```

```
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0487 - accuracy: 0.9844
```

```
Accuracy = 100.0%
```

Cztery warstwy:

```
# hidden layers
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='sigmoid'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(32, activation='sigmoid'))
```

```
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0573 - accuracy: 0.9825
```

```
Accuracy = 90.0%
```

Wniosek ten sam co w przypadku liczby neuronów, czyli zwiększenie liczby warstw ukrytych tylko do pewnego momentu poprawia rezultat. W naszym przypadku taką granicą były dwie warstwy ukryte.

Podsumowując trzeba znaleźć złoty środek w doborze powyższych parametrów, żeby osiągnąć najlepsze rezultaty.

2) SVM:
W porównaniu do sieci neuronowych, SVM dokonuje liniowej separacji jedynie pomiędzy dwoma klasami. Rozwiązaniem w tym przypadku jest metoda 1 vs ALL. Natomiast gdy problem jest nieseparowalny liniowo wtedy można skorzystać z kernela, który przekształca daną przestrzeń do przestrzeni więcej wymiarowej, tak żeby móc poprowadzić hiperpłaszczyznę. W SVM mamy także dostępny parametr C, który pozwala na wprowadzenie naruszeń marginesu kosztem jego szerokości. Jest to przydatne gdy są jakieś elementy odstające i wtedy parametr C ułatwia poprowadzenie prostej. Ponadto wprowadzenie parametru C pozwala osiągnąć lepsze wyniki jak poniżej dla C=0.95.

Dla C = 1:

```
Accuracy = 0.9777777777777777
```

Dla C = 0.95:

```
Accuracy = 0.9785555555555555
```

Dla C = 0.75:

```
Accuracy = 0.9753333333333334
```

Dla C = 0.5:

```
Accuracy = 0.974
```

Oprócz skuteczności algorytmu SVM przydatną statystyką jest również confusion matrix. Przykład poniżej:

```
Confusion matrix:
[[ 892    1    0    0    0    1    1    0    0    1]
 [   0 1023    4    1    3    0    0    2    1    0]
 [   2    0  912    1    2    1    0    2    3    1]
 [   0    2   10  920    1    6    0    7    6    3]
 [   1    3    1    0  846    0    7    0    1   13]
 [   3    0    0    8    2  685    4    1    1    1]
 [   2    0    1    0    1    6  924    0    0    0]
 [   1    7    3    2    5    0    0  890    2    2]
 [   0    2    2    2    0    4    2    1  813    3]
 [   3    1    0    3    9    3    0   14    4  902]]
```

Interpretacja:
- każdy wiersz przedstawia kolejne klasy, czyli w naszym przypadku od 0 - 9
- każda kolumna przedstawia przewidywania algorytmu dotyczące danej klasy (danego wiersza)

Czyli na powyższym rysunku widzimy, że dla klasy 0, SVM przewidział 892 razy faktycznie 0, ale raz przewidział 1, 5, 6 czy 9.

3) KNN

KNN (ang. K nearest neighbours) - algorytm najbliższych sąsiadów. W Statystyce jest on używany do prognozowania wartości pewnej zmiennej losowej. W przypadku naszego projektu będziemy używać KNN do przypisywania obiektów do odpowiednich klas. Skorzystamy w tym celu ponownie ze zbioru MNIST i na jego podstawie będziemy klasyfikować obrazki jako odpowiednie liczby. Zbiór składa się z 60000 obrazów o wielkości 28x28 jednostek.

Implementacja polega na
1. sformatowaniu danych z ".csv", tak aby można było je przekazać do dalszej części programu.

```
36  ##main
37  # mnist_test - 10000 rows x 785 columns
38  mnist_train = pd.read_csv('mnist_train.csv')
39  mnist_test = pd.read_csv('mnist_test.csv')
40  #x_data_test = mnist_test.values
41  #y_data_test = mnist_test.values
42  x_data_test = mnist_train.values
43  y_data_test = mnist_train.values
44  y = x_data_test[:, 0]
45  x_data_test = x_data_test[:, 1:]
```

2. Podziale danych na odpowiednie zbiory - trenujący i testujący

```
X_train, X_test, y_train, y_test = train_test_split(x_data_test, y, tes
```

3. Utworzeniu klasyfikatora poprzez podanie liczby sąsiadów, których bierzemy pod uwagę i wytrenowaniu modelu

```
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
```

4. Wykonaniu predykcji na zbiorze testującym

```
y_prediction = model.predict(X_test)
```

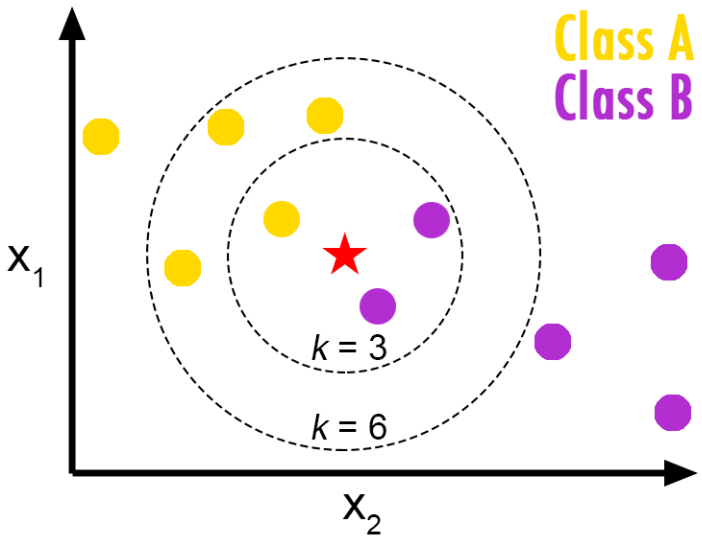
5. Ocenie uzyskanej dokładności modelu, a także dodatkowo wygenerowniu macierzy konfuzji i heatmapy dla niej.

```python
# score
acc = model.score(X_test, y_test)
print("Model accuracy:", acc)
model_balanced_acc = metrics.balanced_accuracy_score(y_test, y_prediction)
print("Model balanced accuracy:", model_balanced_acc)

# confusion matrix
fig, ax = plt.subplots(figsize=(10, 10))
confusion_matrix = confusion_matrix(y_true=y_test, y_pred=y_prediction)

sns = sns.heatmap(confusion_matrix, annot=True, cmap='nipy_spectral_r', fmt='g')
sns.set_title("Confusion matrix")
plt.show()
```

KNN dokonuje dopasowania obiektu do klasy na podstawie jego k-najbliższych sąsiadów. (źródło obrazka https://helloacm.com/a-short-introduction-to-k-nearest-neighbors-algorithm/). Klasyfikacja może ulec zmianie w zależności od “k”, czyli liczby sąsiednich obiektów branych pod uwagę. Zwiększenie niekoniecznie musi powodować, że algorytm działa lepiej i jego dokładność jest większa. Dla k=1 algorytm znany jest jako “Nearest Neighbor Alghoritm”. Dodatkowo zwiększenie liczby cech branych pod uwagę może powodować problemy wydajnościowe i problemy z dopasowaniem obiektu do odpowiedniej klasy.



W zależności od danych oraz dobranej metryki należy tak dopasować k, aby dopasowanie było zgodne z oczekiwaniami autora programu. Ponadto, zwiększenie k powoduje dodatkowy narzut związany z wykonaniem algorytmu i jest to nieefektywne rozwiązanie. Drugim, wyżej wspomnianym czynnikiem wpływającym na klasyfikację jest metryka czyli sposób obliczania odległości. Klasyfikacja następuje na podstawie porównania odległości między cechami w przestrzeni, a sposób wyliczania tej odległości może się różnić. Implementacja KNN z biblioteki sklearn pozwala na zastosowanie następujących odległości, które porównam.

Metrics intended for real-valued vector spaces:

identifier	class name	args	distance function
“euclidean”	EuclideanDistance	•	`sqrt(sum((x - y)^2))`
“manhattan”	ManhattanDistance	•	`sum(|x - y|)`
“chebyshev”	ChebyshevDistance	•	`max(|x - y|)`
“minkowski”	MinkowskiDistance	p	`sum(|x - y|^p)^(1/p)`
“wminkowski”	WMinkowskiDistance	p, w	`sum(|w * (x - y)|^p)^(1/p)`
“seuclidean”	SEuclideanDistance	V	`sqrt(sum((x - y)^2 / V))`
“mahalanobis”	MahalanobisDistance	V or VI	`sqrt((x - y)' V^-1 (x - y))`

Test dokładności w zależności od k (wybrana metryka - minkowski) - zbiór 60 000

dla k=2

```
Model accuracy: 0.9638888888888889
Model balanced accuracy: 0.9632042924347972
```

dla k=5

```
Model accuracy: 0.9697474747474748
Model balanced accuracy: 0.9692865415034694
```

dla k=7

```
Model accuracy: 0.9679797979797979
Model balanced accuracy: 0.9674570581856872
```

dla k = 20

```
Model accuracy: 0.959040404040404
Model balanced accuracy: 0.9583939836444809
```

Widać, że zmiana parametru k nie powoduje istotnego wzrostu dokładności naszego modelu, a co więcej dla k>5 zaczyna spadać, a czas wykonania programu staje się istotnie dłuższy.

Test dokładności w zależności od metryki (wybrane k = 5) - zbiór 60 000

- Minkowski

```
Model accuracy: 0.9697474747474748
Model balanced accuracy: 0.9692865415034694
```

- Euklides

```
Model accuracy: 0.9697474747474748
Model balanced accuracy: 0.9692865415034694
```

- Manhattan

```
Model accuracy: 0.9318181818181818
Model balanced accuracy: 0.9308022302363236
```

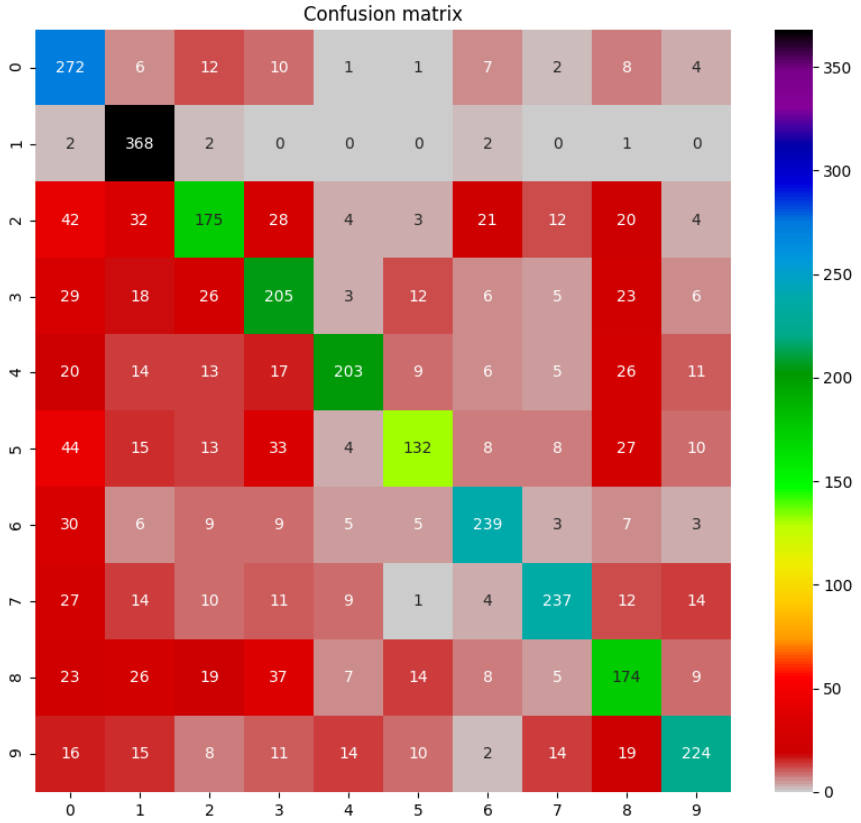
- Chebyshev

```
Model accuracy: 0.6754545454545454
Model balanced accuracy: 0.6696262996410502
```

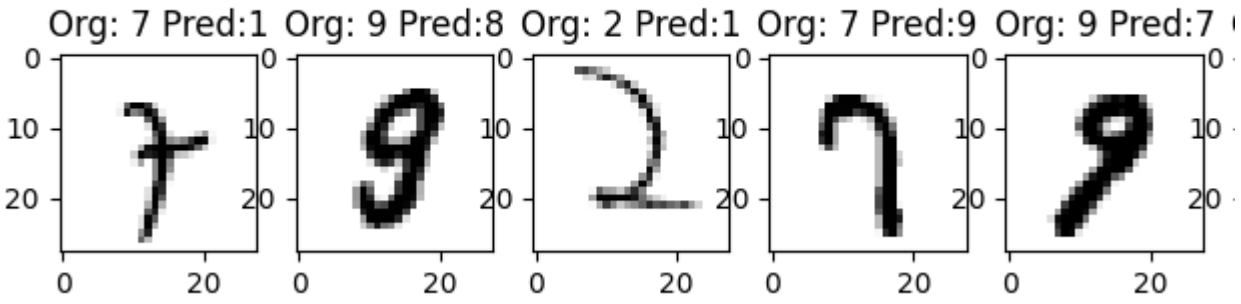
Zmiana metryki w naszym przypadku nie miała drastycznego wpływu na jego dokładność, oprócz metryki Chebyshev'a. Nastąpił znaczny spadek wydajności i dokładności w przypadku Chebysheva, który rośnie w miarę wzrostu parametru k. Wydajnościowo metryka Manhattan również radzi sobie słabo.

Dodatkowo w programie następuje generowanie przykładowych źle sklasyfikowanych obrazów, a także macierzy konfuzji, która obrazuje w jakich przypadkach algorytm się myli. Do tego generowana jest heat-mapa pozwalająca na dobre zobrazowanie macierzy konfuzji.

Heat-mapa dla macierzy konfuzji dla k=5 i metryki Chebyshev’a



Przykładowe źle sklasyfikowane obrazy

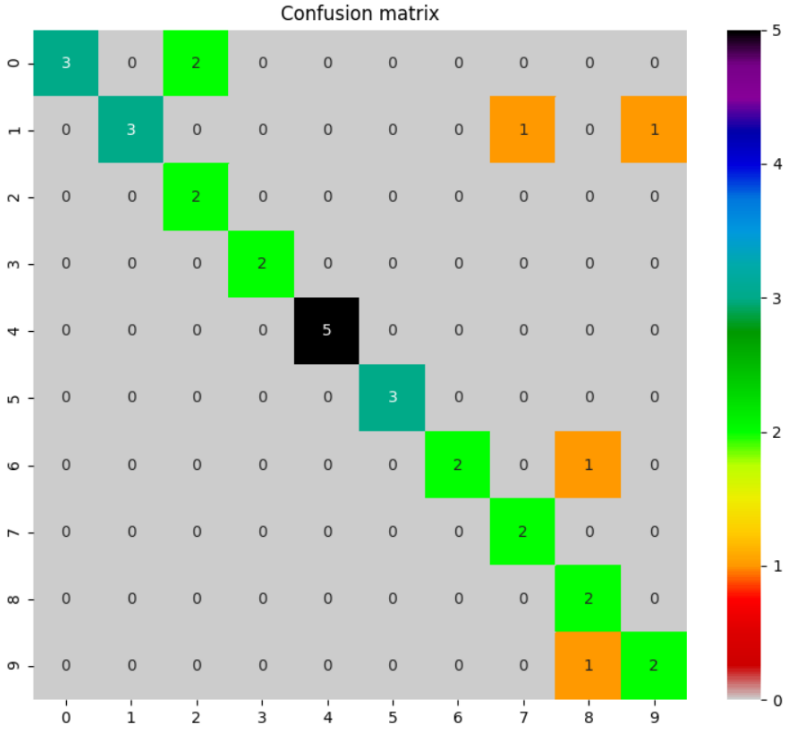


Na koniec stworzyliśmy własny zbiór cyfr narysowanych w paincie o wymiarach 28x28 pikseli i na podstawie tych danych porównamy jeszcze te trzy modele (SVM vs KNN vs sieci neuronowe) dla najlepiej dobranych eksperymentalnie parametrów danego modelu.

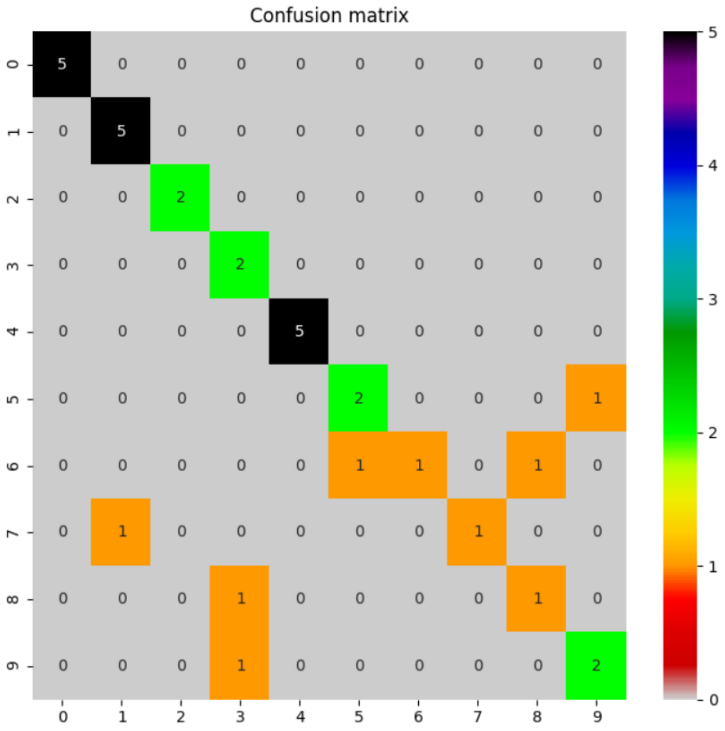
	Accuracy	Training Time
KNN	81,25%	0.01210331916809082 s
SVM	81,25%	168.52578735351562 s
NN	84,375%	57.199819564819336 s

Heatmapa macierzy konfuzji poszczególnych modeli dla narysowanych przez nas cyfr

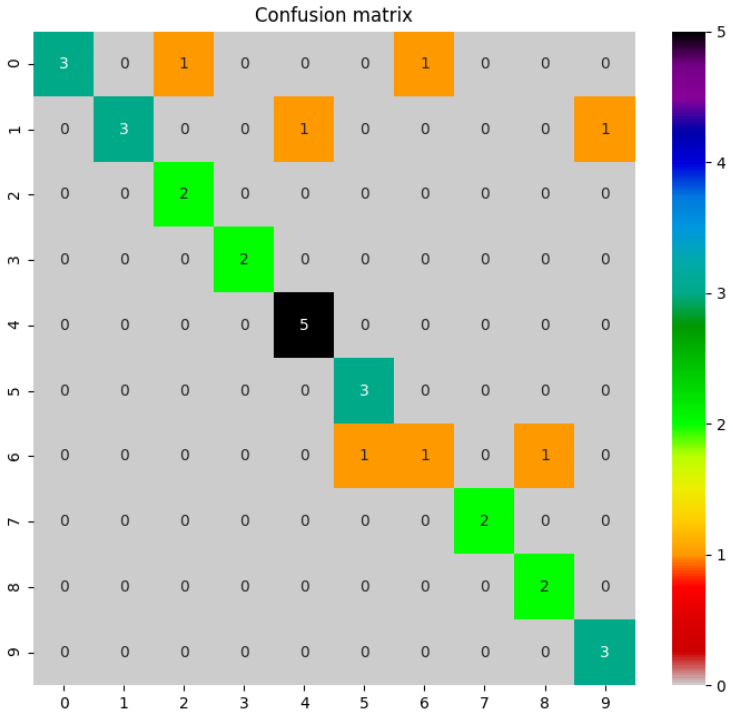
1) NN



2) SVM


Confusion matrix
0 5 0 0 0 0 0 0 0 0 0
1 0 5 0 0 0 0 0 0 0 0
2 0 0 2 0 0 0 0 0 0 0
3 0 0 0 2 0 0 0 0 0 0
4 0 0 0 0 5 0 0 0 0 0
5 0 0 0 0 0 2 0 0 0 1
6 0 0 0 0 0 1 1 0 1 0
7 0 1 0 0 0 0 0 1 0 0
8 0 0 0 1 0 0 0 0 1 0
9 0 0 0 1 0 0 0 0 0 2
0 1 2 3 4 5 6 7 8 9
5 4 3 2 1 0

3) KNN


Confusion matrix
0 3 0 1 0 0 0 1 0 0 0
1 0 3 0 0 1 0 0 0 0 1
2 0 0 2 0 0 0 0 0 0 0
3 0 0 0 2 0 0 0 0 0 0
4 0 0 0 0 5 0 0 0 0 0
5 0 0 0 0 0 3 0 0 0 0
6 0 0 0 0 0 1 1 0 1 0
7 0 0 0 0 0 0 0 2 0 0
8 0 0 0 0 0 0 0 0 2 0
9 0 0 0 0 0 0 0 0 0 3
0 1 2 3 4 5 6 7 8 9
5 4 3 2 1 0