Klasyfikacja danych ze zbioru liczb od 0 do 9, z wykorzystaniem sztucznej sieci neuronowej (2 warstwy ukryte) uczonej metodą wstecznej propagacji błędu.

Spis treści

1 Wstęp

Problematyką projektu było stworzenie sztucznej sieci neuronowej z dwiema warstwami ukrytymi, która uczona była metodą wstecznej propagacji błędu. Na podstawie wprowadzonych danych sieć miała nauczyć się przewidywać cyfry od 0 do 9 zapisane jako tablice binarne o wymiarach 7x5, gdzie 1 oznacza ślad atramentu w danym polu.

Celem pracy było zastosowanie stworzonej sieci neuronowej oraz sprawdzenie skuteczności jej nauczenia. Analizowana była dokładności predykcji cyfr w celu oceny jak dobrze sieć nauczyła się rozpoznawać liczby.

W realizacji projektu został użyty język programowania Python [1], który ze względu na swoją elastyczność oraz mnogość bibliotek był doskonałym wyborem do utworzenia sieci neuronowej. Pomocna była biblioteka Matplotlib [2], która w prosty sposób pozwoliła na stworzenie potrzebnych wykresów. Zastosowana była również biblioteka NumPy [3], która została wykorzystana do operacji na tablicach.

2 Dane

Bazą danych były cyfry od 0 do 9 zapisane jako tablice binarne o wymiarach 7x5, gdzie 1 oznaczało ślad atramentu w danym polu. Zostały wykorzystane dane idealne oraz zaszumione. W celu stworzenia bazy został wykorzystany język programowania Python. Na początku zostały utworzone dane idealne w postaci tablicy tablic z wartościami binarnymi reprezentujące cyfry od 0 do 9, przedstawione to zostało w listingu 2.1.

```
lista_liczb =[
#0
            [[1,1,1,1,1],
            [1,0,0,0,1],
            [1,0,0,0,1],
            [1,0,0,0,1],
            [1,0,0,0,1],
            [1,0,0,0,1],
            [1,1,1,1,1]],
#1
            [[0,0,0,0,1],
            [0,0,0,1,1],
            [0,0,1,0,1],
            [0,1,0,0,1],
            [0,0,0,0,1],
            [0,0,0,0,1],
            [0,0,0,0,1]],
#2
            [[0,1,1,1,0],
            [1,0,0,0,1],
            [1,0,0,1,0],
            [0,0,1,0,0],
            [0,1,0,0,0],
            [1,0,0,0,0],
            [1,1,1,1,1]],
#3
```

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

```
                [1,1,1,1,1],
                [0,0,0,0,1],
                [0,0,0,0,1],
                [1,1,1,1,1]],
]
```

Listing 2.1 Dane idealne w postaci tablicy tablic

W kolejnym kroku dane idealne zostały wielokrotnie zaszumione poprzez losowe zmienianie znaków oraz każda z zaszumionych cyfr została zapisana w pliku „baza.txt" jako jedna tablica. Dane testowe powstały poprzez ponowne uruchomienie programu co spowodowało zaszumienie ich w inny, losowy sposób oraz zapisanie ich do pliku „test.txt". Program przedstawiony został w listingu 2.2.

```
f = open("baza.txt","wt")
a=0
while(a<10):
    z=0
    while(z<1000):
        z=z+1
        f.write("["+str(a)+",")
        i=0
        for x in lista_liczb[a]:
            for y in x:
                if(random.randrange(0,12)==1 and z>1):
                    if(int(y)==0):
                        f.write("1")
                    else:
                        f.write("0")
                else:
                    f.write(str(y))
                i=i+1
                if(i<35):
                    f.write(",")
        f.write("]\n")
    a=a+1

f.close()
```

Listing 2.2 Program do generowania bazy danych

3 Algorytm

Algorytmem zastosowanym w sztucznej sieci neuronowej była wsteczna propagacja błędu, która często używana jest do problemu klasyfikacji danych. Działanie sieci polega na początkowy wykorzystaniu sygnałów wejściowych do obliczenia macierzy predykcji oraz zmierzenia błędy w stosunku do prawdziwych danych. Następnie stosuje się funkcję aktywacji, która wprowadza nieliniowość. W kolejnym kroku propagowany jest gradient błędu w poszczególnych warstwach. Pod koniec dostosowywane są wagi połączeń w sieci wykonując krok w przeciwnym kierunku do gradientu błędu aktywacji w wyniku czego minimalizowany jest błąd wyjściowy.[4]

W projekcie zastosowana została sztuczna sieć neuronowa z dwiema warstwami ukrytymi uczona metodą wstecznej propagacji błędu. Pierwszym niezbędnym krokiem było zdefiniowanie wag, które zostały na początku wypełnione losowymi wartościami z rozkładu Gaussa. Rozmiar macierzy zależny jest od wprowadzonych wartości wielkosc_wejscia, k1, k2, wielkosc_wyjscia. Zostały zainicjowane również biasy, które zostały wypełnione zerami. [4] Kod odpowiedzialny za tą część przedstawiony został w listingu 3.1.

```
def __init__(self, wielkosc_wejscia, k1,k2,wielkosc_wyjscia):
    self.W1 = np.random.randn(wielkosc_wejscia, k1)
    self.b1 = np.zeros((1, k1))
    self.W2 = np.random.randn(k1, k2)
    self.b2 = np.zeros((1, k2))
    self.W3 = np.random.randn(k2, wielkosc_wyjscia)
    self.b3 = np.zeros((1, wielkosc_wyjscia))
```

Listing 3.1 Inicjowanie wartości sieci

Następnie została stworzona funkcja forward, która była odpowiedzialna z obliczanie wartości predykcji. Wynik mnożenia wejść przez odpowiednie wagi oraz dodanie biasu zostało zapisane w zmiennych z zgodnie ze wzorem(1):

$$z_i^l = \sum_{j=0}^{j} w_{ij}^l a_j^{l-1} + b_i^l \tag{1}$$

gdzie:

- z oznacza wynik mnożenia wejść oraz wag z dodaniem biasu,
- w oznacza wagi
- a oznacza wartość aktywacji
- b oznacza bias.

Natomiast wartość aktywacji obliczona była poprzez użycie funkcji aktywacji do poszczególnych wartości ‘z’. Następnie wartość ‘z3’ przekształcona została na wartości prawdopodobieństw. [5]

Kod odpowiedzialny za tą część przedstawiony został w listingu 3.2.

```
def forward(self, X):
    self.z1 = np.dot(X, self.W1) + self.b1
    self.a1 = np.maximum(0, self.z1)
    self.z2 = np.dot(self.a1, self.W2) + self.b2
```

```
    self.a2 = np.maximum(0, self.z2)
    self.z3 = np.dot(self.a2, self.W3) + self.b3
    c = np.max(self.z3, axis=1, keepdims=True)
    exp_scores = np.exp(self.z3 - c)
    self.probs = exp_scores / np.sum(exp_scores, axis=1,
keepdims=True)
    return self.probs
```

Listing 3.2 Funkcja forward

Na koniec została zapisana funkcja backward, która służyła do aktualizacji wag w sieci. Jako pierwsza obliczona została delta, która jest macierzą prawdopodobieństw. Następnie obliczone zostały gradienty, które są iloczynem transponowanych macierzy aktywacji. Zostały obliczone również biasy, które są sumą wartości w kolumnach macierzy delt. Na końcu nastąpiła aktualizacja wag oraz biasów poprzez odjęcie iloczynów gradientów i współczynników uczenia. [6]

Kod odpowiedzialny za tą część przedstawiony został w listingu 3.3.

```
def backward(self, X, y, lr):
    delta3 = self.probs
    delta3[range(X.shape[0]), y] -= 1
    dW3 = np.dot(self.a2.T, delta3)
    db3 = np.sum(delta3, axis=0, keepdims=True)
    delta2 = np.dot(delta3, self.W3.T) * (self.a2 > 0)
    dW2 = np.dot(self.a1.T, delta2)
    db2 = np.sum(delta2, axis=0)
    delta1 = np.dot(delta2, self.W2.T) * (self.a1 > 0)
    dW1 = np.dot(X.T, delta1)
    db1 = np.sum(delta1, axis=0)
    self.W1 -= lr * dW1
    self.b1 -= lr * db1
    self.W2 -= lr * dW2
    self.b2 -= lr * db2
    self.W3 -= lr * dW3
    self.b3 -= lr * db3
```

Listing 3.3 Funkcja backward

4 Badania

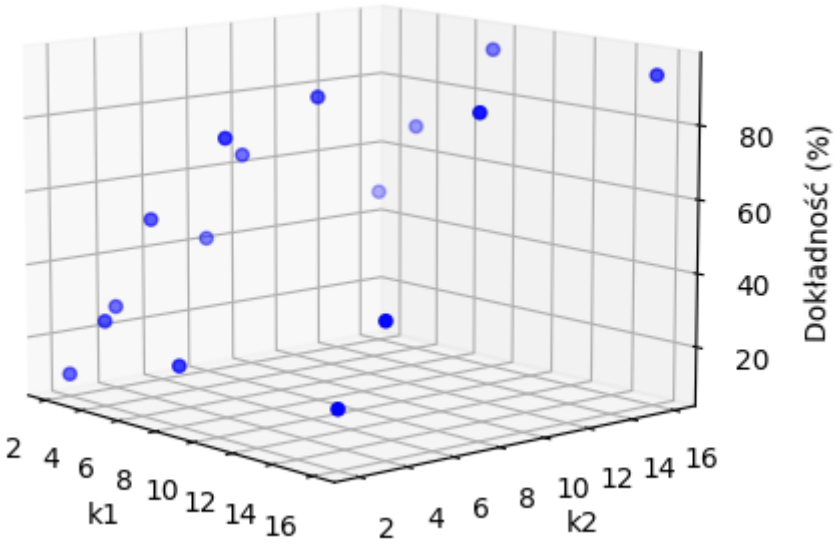
Badania przeprowadzone były na wybranych parametrach dotyczących sieci neuronowej. Pierwszymi parametrami były ilości neuronów w poszczególnych warstwach ukrytych, ich badanie przeprowadzone było przez stworzenie skryptu, który dla wybranych ilości neuronów testował je w całym programie. Następnym parametrem był współczynnik uczenia, który również był badany poprzez napisanie algorytmu, który dla wybranych wartości oraz wcześniej już wybranej ilości neuronów z największą dokładnością, testował wybrane wartości współczynnika uczenia. Kolejnym eksperymentem było badanie dokładności nauczenia sieci w zależności od ilości danych uczących. Ostatnim badanym parametrem była liczba epok i sprawdzenie w jakim stopniu sieć się nauczyła w zależności od jej ilości.

4.1 Eksperyment I

Eksperyment I polegał na sprawdzeniu podstawowych zależności w danej sieci, w tym przypadku były to liczby neuronów w warstwach ukrytych oraz współczynnik uczenia.

4.1.1 Faza I

W fazie pierwszej przetestowane były zależności występujące między różnymi ilościami neuronów w poszczególnych warstwach, w tym przypadku były to kombinacje liczb 2, 4, 8, 16 w każdej z warstw ukrytych. Wynikowy wykres wraz z dokładnością testowania danych przedstawiony został na rysunku 4.1.1.



Rysunek 4.4.1 Wykres zależności k1, k2 oraz ich dokładności w różnych konfiguracjach

Z przeprowadzone eksperymentu oraz po analizie wykresu największa procentowa dokładność 93.4% uzyskała kombinacja k1 = 8, k2 = 16, jednak była tylko nieznacznie większa od k1 = 16, k2 = 16, gdzie dokładność wyniosła 93.3%.

4.1.2 Faza II

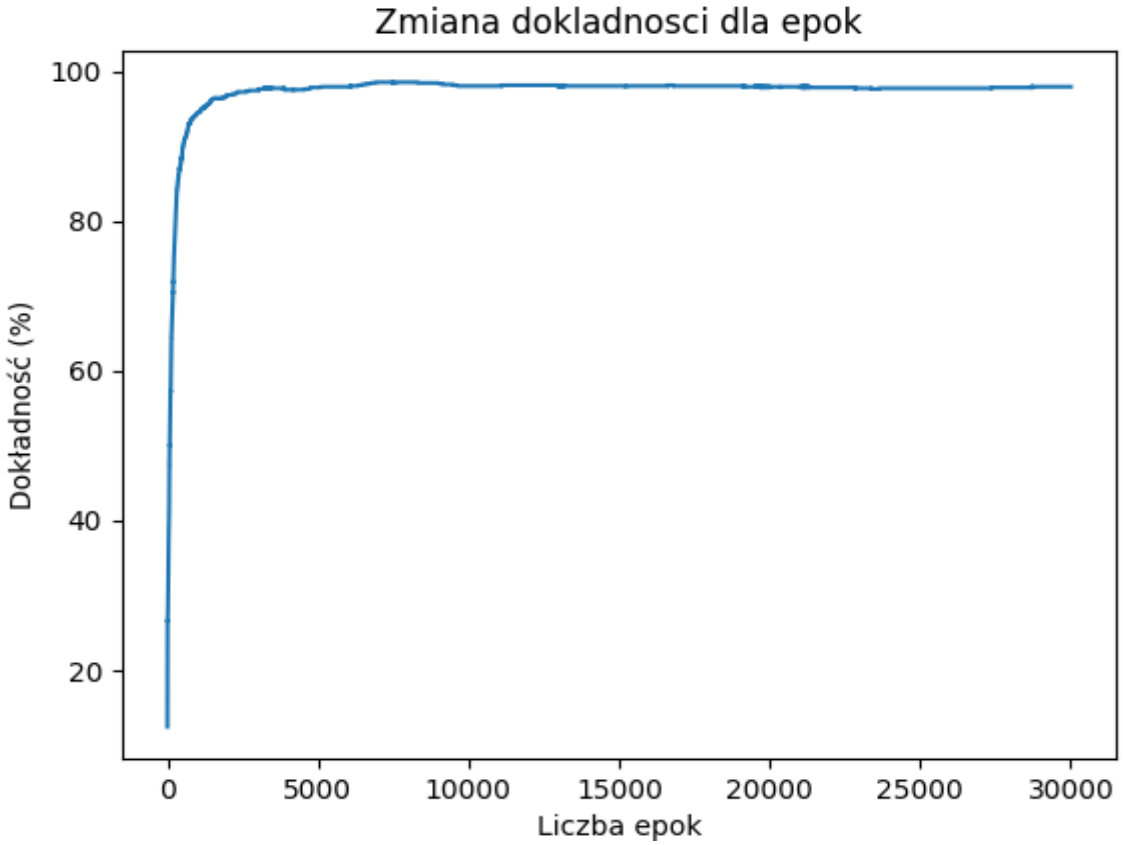
W fazie drugiej testowaniu podległ współczynnik uczenia, który przyjmował wartości: 0.001, 0.0001, 0.00001, 0.000001. Wyniki tego eksperymentu zostały przedstawione na rysunku 4.1.2.

Rysunek 4.4.2 Wykres zależności między współczynnikiem uczenia a dokładnością

Ta faza eksperymentu pokazała, że z mniejszym współczynnikiem uczenia uzyskuje się lepszą dokładność. Wykres przedstawił, że najlepszy wynik 97.8% uzyskała wartość lr wynosząca 0.00001.

4.2 Eksperyment II

W kolejnym eksperymencie testowane były epoki od 0 do 30000. W tym eksperymencie została użyta optymalna liczba neuronów w poszczególnych warstwach czyli 8 i 16, która została wybrana przez pierwszy eksperyment. Współczynnik uczenia także został wybrany przez wcześniej przeprowadzony eksperyment i wyniósł on 0.00001. Wyniki przedstawione zostały na rysunku 4.2.



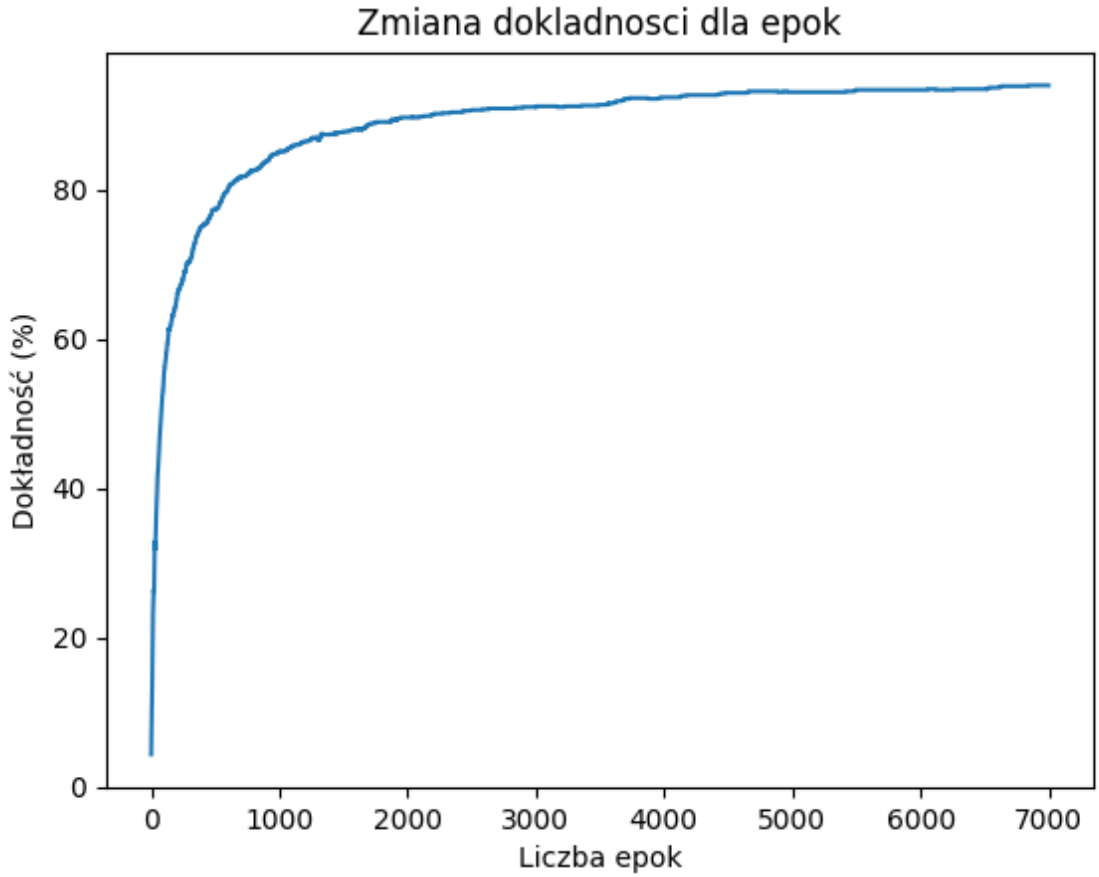
Rysunek 4.2 Wykres zależności między liczbą epok a dokładnością

Po skończeniu działania programu, który porównywał wartości wszystkich epok okazało się, że największą dokładność wynoszącą 98.2% uzyskała epoka 6912. Analizując wykres można było dostrzec, że po 2000 epoce różnica w dokładności była minimalna.

4.3 Eksperyment III

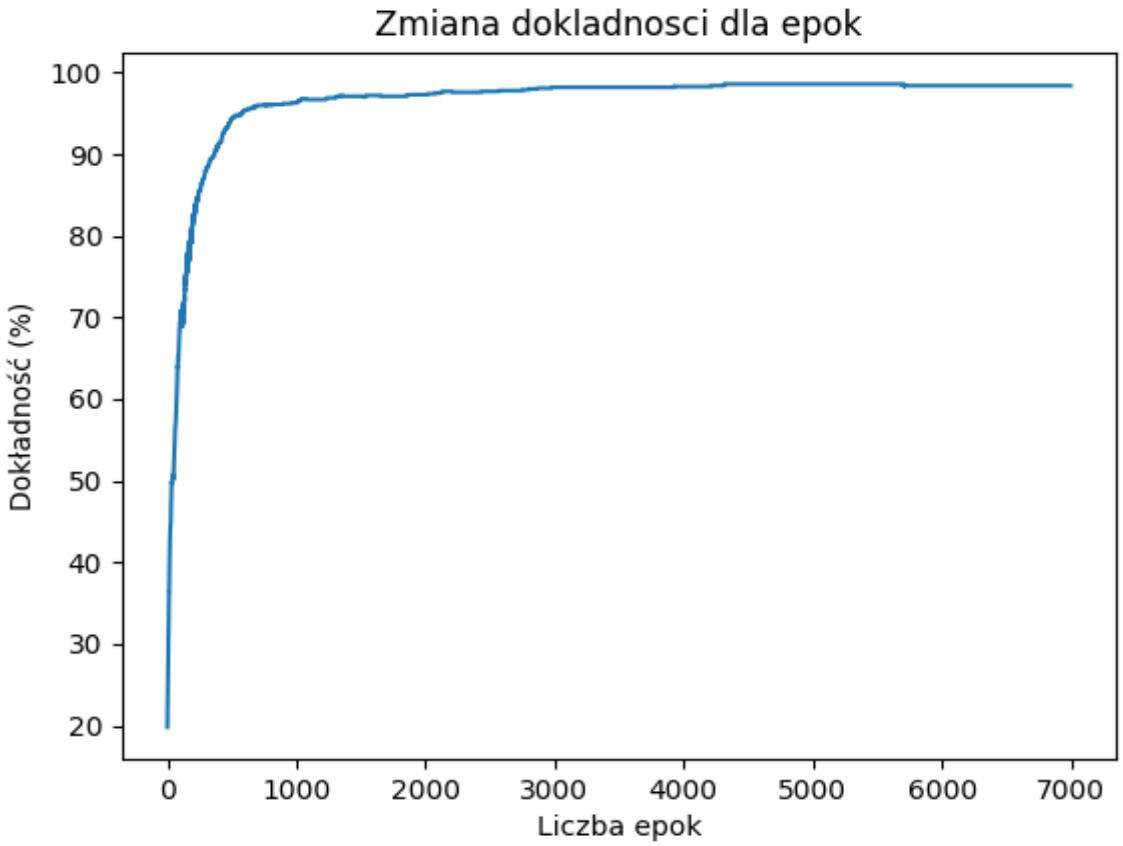
Ostatnim eksperymentem było sprawdzenie dokładności uczenia sieci w zależności od różnej ilości danych wejściowych. Testujące był następujące ilości: 1000, 10000, 100000. Ilość neuronów w poszczególnych warstwach była ta sama co w poprzednim eksperymencie, czyli 8 oraz 16. Współczynnik uczenia także został ten sam, czyli 0.00001. Ilość epok wyniosła 7000, gdyż po wcześniejszym eksperymencie można było zauważyć, że po tej epoce dokładność zmienia się nieznacznie.

Wynik uczenia sieci dla 1000 danych uczących przedstawiony został na rysunku 4.3.1. Zauważalne było na nim to, że maksymalną dokładnością osiągniętą przez algorytm było 94%.



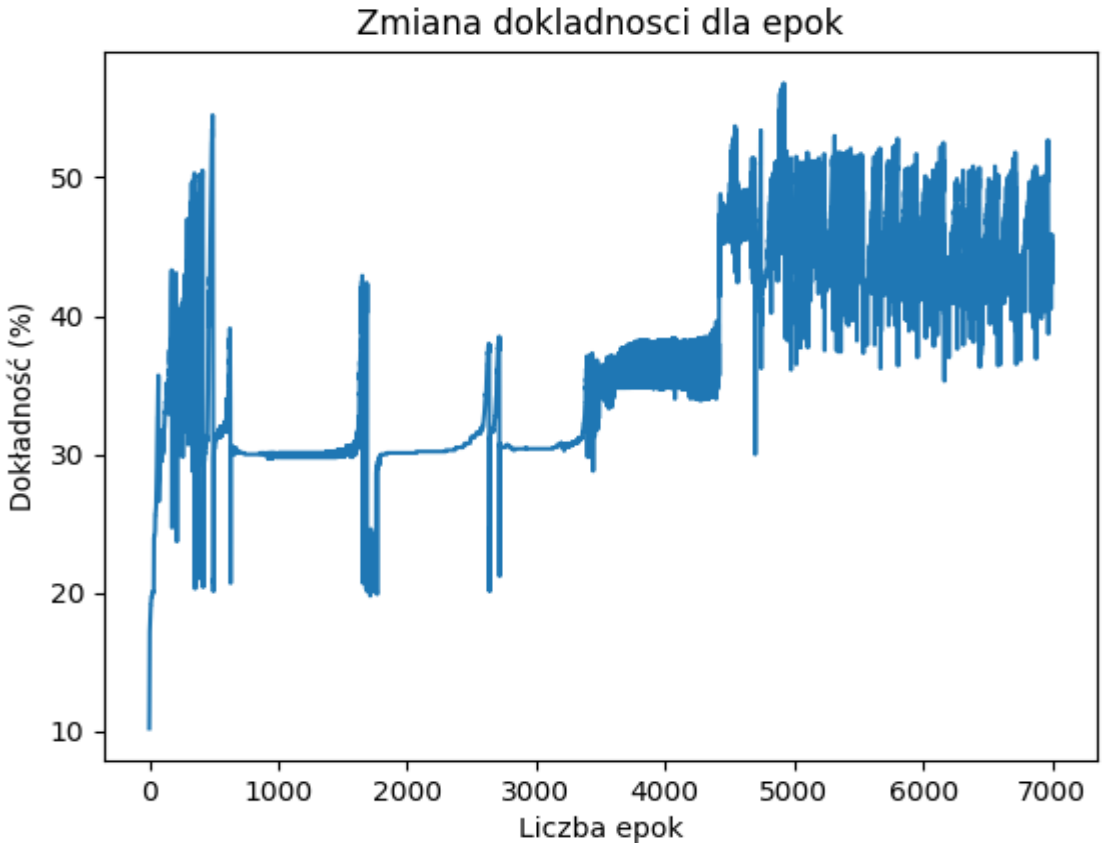
Rysunek 4.3.1 Wykres zależności między liczbą epok a dokładnością

Wynik uczenia sieci dla 10000 danych uczących przedstawiony został na rysunku 4.3.2. W tym przypadku widać było niewielką poprawę względem 1000 danych, ponieważ dokładność wyniosła 98.2%.



Rysunek 4.3.2 Wykres zależności między liczbą epok a dokładnością

Wynik uczenia sieci dla 100000 danych uczących przedstawiony został na rysunku 4.3.3. Dla tylu danych sieć została przeuczono co spowodowało bardzo słabe wyniki dopasowywania danych a największa dokładność wyniosła 56.2%.



Rysunek 4.3.3 Wykres zależności między liczbą epok a dokładnością

5 Podsumowanie i wnioski

Stworzony program osiągnął swoje założenie i z dokładnością 98.2% poprawnie rozpoznawał dane testowe, stało się tak dzięki odpowiednim przetestowaniu i dobraniu ilości neuronów w każdej warstwie ukrytej, współczynnika uczenia. Problemy z uczeniem sieci zostały rozwiązane poprzez eksperymentowanie z ilością danych uczących, w wyniku czego została dobrana ich odpowiednia ilość czyli 10000. Podczas realizacji projektu udało się dogłębnie przeanalizować działanie sieci neuronowej z algorytmem wstecznej propagacji co poskutkowało zdobyciem większej wiedzy na temat działania sztucznej inteligencji.

6 Bibliografia

[1] https://www.python.org Data dostępu: 16.06.2023.

[2] https://matplotlib.org/ Data dostępu: 16.06.2023.

[3] https://numpy.org/ Data dostępu: 16.06.2023.

[4] http://neuralnetworksanddeeplearning.com/ Data dostępu 16.06.2023

[5] https://towardsdatascience.com/backpropagation-step-by-step-derivation-99ac8fbdcc28 Data dostępu 16.06.2023

[6] https://www.javatpoint.com/pytorch-backpropagation-process-in-deep-neural-network Data dostępu 16.06.2023