Projekt z NAI: Rozpoznawanie cyfr
Konstantin Puchko s19575

Cel badania: Zaprojektować i przeprowadzić uczenie dwuwarstwowej sieci neuronowej do rozpoznawania 10 cyfr od 0 do 9. Cyfry są reprezentowane przez mapy bitowe o wymiarze 6 × 5.

Zadanie 1. Tworzenie zbioru treningowego i testowego.

Dla celów uczenia sieci neuronowej stworzyłem dwa zbiory: testowy oraz treningowy (zawartość plików test_set.txt oraz dataset.txt).
Wektory ze zbiory testowego są unikatowe i nie występują w zbiorze treningowym.
Także wektory w zbiorze treningowym nie powtarzają się.

Zbior treningowy zawiera 50 wektorów (po 5 na każdą cyfrę)
Zbior testowy po 2 przykłady na każdą cyfrę (czyli razem 20)

Program wczytywuje dane z plików i zapisuje je jako ArrayList zawierający każdą wartość wektorów oraz jako ostatni element poprawną odpowiedz.

Zadanie 2. Definiowanie sieci neuronowej

Program wspiera zmianę architektury i może być wykorzystany dla rozwiązania innych problemów klasyfikacyjnych.

Dla rozwiązania podanego tematu najlepiej się sprawdziła architektura na 30 wejść, 1-ą ukrytą warstwę na 12 neuronów oraz 10 wyjść (cyfry od 0 – 9).

Funkcję aktywacji zaimplementowałem jako:

```
x -> 1 / (1 + Math.exp(-x))
```

Zadanie 3. Trenowanie sieci

Implementację algorytmu wstecznej propagacji błedu zamieściłem w funkcji backpropagation(double targets[]):

```
public void backpropagation(double[] targets){
  for (int i = 0; i < layers[layers.length - 1].size; i++) {
    errors[i] = targets[i] - layers[layers.length - 1].neurons[i];
  }
  for (int k = layers.length - 2; k >= 0; k--) {
    Layer l = layers[k];
    Layer l1 = layers[k + 1];
    double[] errorsNext = new double[l.size];
    double[] gradients = new double[l1.size];
    for (int i = 0; i < l1.size; i++) {
      gradients[i] = errors[i] * derivative.apply(layers[k + 1].neurons[i]);
      gradients[i] *= learningRate;
    }
    double[][] deltas = new double[l1.size][l.size];
```

```
            for (int i = 0; i < l1.size; i++) {
                for (int j = 0; j < l.size; j++) {
                    deltas[i][j] = gradients[i] * l.neurons[j];
                }
            }
            for (int i = 0; i < l.size; i++) {
                errorsNext[i] = 0;
                for (int j = 0; j < l1.size; j++) {
                    errorsNext[i] += l.weights[i][j] * errors[j];
                }
            }
            errors = new double[l.size];
            System.arraycopy(errorsNext, 0, errors, 0, l.size);
            double[][] weightsNew = new double[l.weights.length][l.weights[0].length];
            for (int i = 0; i < l1.size; i++) {
                for (int j = 0; j < l.size; j++) {
                    weightsNew[j][i] = l.weights[j][i] + deltas[i][j];
                }
            }
            l.weights = weightsNew;
            for (int i = 0; i < l1.size; i++) {
                l1.biases[i] += gradients[i];
            }
        }
    }
```

Obserwując zmianę kooficjętu błędu wnioskuję że bląd dąży do zera, ale nie schodzi do niego (czyli można powiedzieć $\lim\{x\to\infty\} f(x) = 0$)

Wykres kooficjęta błedu można zobaczyć w pliku wykres_bledu_rozpoznawania.xlsx
Wykres pokazuje zmianę na 1000 epok ze względu na długi czas kopiowania większej liczby linii.

Zadanie 4 oraz Zadanie 5: Testowanie sieci I Raportowanie wyników

Zostawiłem liczbę epok na 1 000 000, chociaż przy 100 000 sieć już rozpoznaję 100% przykładów ze zbioru testowego.
Przy liczbie epok 1000 sieć pokazuję rezultat w 80% błedu.
Zmiana procentu błedu odnośnie liczby epok można określić funkcją logarytmiczną.