# Analiza i wizualizacja danych

Klasyfikacja danych z zastosowaniem sztucznych sieci neuronowych i metody wstecznej propagacji błędu z momentem

Simlat Arkadiusz

## 1. Opis teoretyczny analizowanego problemu

Sztuczne sieci neuronowe powstały na bazie inspiracji neurobiologicznych. Zbudowane są z połączonych ze sobą sztucznych neuronów, które jednak z punktu widzenia neurobiologii są skrajnie uproszczonym modelem rzeczywistego neuronu. Mimo to występuje szereg analogii zarówno w budowie neuronu jak i struktur, które są za pomocą tego elementu tworzone. Skrajne uproszczenie modelu neuronu nie powoduje jednak zniszczenia najważniejszej cechy, czyli zdolności uczenia się. Obecnie sieci neuronowe są narzędziem pozwalającym dokonać odwzorowania bardzo złożonych funkcji. W porównaniu z dotychczas istniejącymi metodami modelowania liniowego, sieci neuronowe mają dużo większe możliwości z powodu swojego nieliniowego charakteru. W dziedzinie nauk technicznych sztuczne sieci neuronowe wykorzystuje się m.in. do klasyfikacji i rozpoznawania. Przez pojęcie klasyfikacji rozumie się dzielenie dowolnego zbioru elementów na grupy, do których zalicza się elementy różniące się, ale podobne tj. mające własności wyróżniające daną grupę. Zbiór elementów należących do jednej grupy nazywany jest klasą, a każdy element klasy - obiektem. Elementy klasy mogą różnić się między sobą, z wyjątkiem tych własności, na których opiera się klasyfikacja. W zależności od rodzaju dostępnej informacji w ramach klasyfikacji można wyodrębnić dwa zagadnienia:

- klasyfikację wzorcową; struktura kategorii jest znana, czyli dysponuję się charakterystyką klas, z których pochodzą obiekty; zagadnienie to nazywane jest uczeniem (rozpoznawaniem) z nauczycielem lub też pod nadzorem.
- klasyfikację bezwzorcową, znaną jako taksonomia albo analiza skupień (klasteryzacja), określaną również jako uczenie lub rozpoznawanie bez nauczyciela.

## 2. Klasyczny algorytm wstecznej propagacji błędu

Algorytm wstecznej propagacji błędu jest uogólnieniem reguły delta na potrzeby wielowarstwowych sieci neuronowych. Trenowanie sieci typu MLP jest w rzeczywistości numeryczną procedurą optymalizacji określonej funkcji celu. Należy on do tzw. metod gradientowych, które wykorzystują prawidłowość głoszącą, iż gradient funkcji wskazuje kierunek jej najszybszego wzrostu, a w przypadku zmiany znaków składowych na przeciwne, czyli pomnożeniu przez -1, kierunek jej najszybszego spadku. Właściwość ta pozwala na minimalizację funkcji celu przez modyfikację jej zmiennych, a w przypadku sieci

współczynników wagowych, w kierunku najszybszego spadku funkcji, czyli zgodnie z regułą delty, proporcjonalnie do gradientu funkcji celu. Funkcja celu sieci neuronowej może mieć różne postaci, w klasycznej jednak wersji używa się kwadratu błędu sieci, lub błędu średniokwadratowego, rozumianego jako różnica pomiędzy wartościami wyjść wygenerowanymi dla określonego wektora wejść, a wartościami wzorcowymi, czyli takimi do jakich sieć neuronowa powinna dążyć przetwarzając przyporządkowany do nich wektor wejść. W metodach uczenia sieci wielowarstwowych wyróżnia się dwie podstawowe techniki, modyfikujące nieznacznie sam algorytm i niektóre wzory. Jedną z nich jest uczenie przyrostowe (*ang. incremental learning*), w którym minimalizuje się funkcję błędu osobno dla każdej pary wektorów wartości zmiennych wejściowych i wzorcowych, a modyfikacja wag sieci następuje po każdorazowej prezentacji pojedynczej pary wektorów wejść i wzorców ze zbioru uczącego. Odpowiadająca jej forma funkcji celu to suma kwadratów błędów na wyjściach obliczana osobno dla każdej obserwacji:

$$Q(k) = \frac{1}{2}\sum_{j=1}^{N_L}(y_{i,j}^{(L)} - p_{i,j}^{(L)})^2$$

Algorytm wstecznej propagacji błędu sprowadza się do modyfikacji każdej z wag proporcjonalnie do wartości pochodnej cząstkowej funkcji celu. Modyfikacja wag w k-tej iteracji polega na odjęciu od wartości wag z iteracji poprzedniej (k - 1) wektora gradientu obliczonego dla bieżącej obserwacji. Algorytm jest sparametryzowany dzięki użyciu tzw. współczynnika uczenia, lub inaczej długości kroku, oznaczonego w poniższym równaniu symbolem α. Współczynnik uczenia przyjmuje na ogół wartości z zakresu od 0,5 do 0,9, ale w specjalnych przypadkach może nawet osiągać wartość α = 3:

$$w_i^{(n)}(k) = w_i^{(n)}(k-1) - \alpha\frac{\partial Q(k)}{w_i^{(n)}}$$

$$\alpha > 0$$

Wektor gradientu, stanowiący kluczowy składnik powyższego wzoru można przedstawić w postaci iloczynu pochodnej funkcji Q względem wyjścia j-tego neuronu z n-tej warstwy i pochodnej cząstkowej funkcji wyjścia tego samego neurony względem własnego wektora wag:

$$\frac{\partial Q(k)}{\partial w_i^{(n)}} = \frac{\partial Q(k)}{\partial y_{i,j}^{(n)}}\frac{\partial y_{i,j}^{(n)}}{\partial w_i^{(n)}}$$

Powyższe wyrażenie wygodnie jest z kolei przedstawić w formie sumy iloczynów pochodnych funkcji Q względem wyjść neuronów z warstwy następnej (n + 1) i pochodnych funkcji wyjść tych samych neuronów względem tych z ich wejść, do których trafia sygnał wyjściowy pochodzący z j-tego neurony warstwy poprzedniej (n-tej):

$$\frac{\partial Q(k)}{\partial y_{i,j}^{(n)}} = \sum_{j=1}^{N_{n+1}} \frac{\partial Q(k)}{\partial x_{i,j}^{(n+1)}} = \sum_{j=1}^{N_{n+1}} \frac{\partial Q(k)}{\partial y_{i,j}^{(n+1)}} \frac{\partial y_{i,j}^{(n+1)}}{\partial x_{i,j}^{(n+1)}}$$

$$\frac{\partial Q(k)}{\partial y_{i,j}^{(n)}} = \sum_{j=1}^{N_{n+1}} \frac{\partial Q(k)}{\partial y_{i,j}^{(n+1)}} \frac{\partial y_{i,j}^{(n+1)}}{\partial s_{i,j}^{(n+1)}} \frac{\partial s_{i,j}^{(n+1)}}{\partial x_{i,j}^{(n+1)}}$$

$$\frac{\partial Q(k)}{\partial y_{i,j}^{(n)}} = \sum_{j=1}^{N_{n+1}} \frac{\partial Q(k)}{\partial y_{i,j}^{(n+1)}} y_{i,j}^{(n+1)} (1 - y_{i,j}^{(n+1)}) w_{i,j}^{(n+1)}$$

Analiza wyprowadzenia powyższego wzoru ukazuje, że możliwe jest obliczenie pochodnej funkcji celu Q w stosunku do wyjść dowolnego neurony niezależnie od liczby warstw. Opisana procedura w praktyce przebiega w kierunku odwrotnym, od neuronów warstwy o numerze najwyższym do neuronu docelowego w warstwie o numerze niższym. W iteracyjnym wyznaczaniu kolejnych pochodnych wykorzystuje się wcześniej obliczone pochodne z warstw o wyższej numeracji i jedynym wyjątkiem od tej reguły jest pochodna funkcji celu Q po wyjściach ostatniej warstwy perceptronowej:

$$\frac{\partial Q(k)}{\partial y_{i,j}^{(L)}} = y_{i,j}^{(L)} - p_{i,j}^{(L)}$$

Finałowym elementem, niezbędnym do wyznaczenia potrzebnego w algorytmie gradientu, jest druga cześć iloczynu ze wzory, czyli pochodna wyjścia neuronu po jego wagach. Wykorzystując sigmoidalną funkcję aktywacji, która posiada względnie prostą pierwszą pochodną, potrzebny wzór ma postać:

$$\frac{\partial y_{i,j}^{(n)}}{\partial w_i^{(n)}} = \frac{\partial y_{i,j}^{(n)}}{\partial s_{i,j}^{(n)}} \frac{\partial s_{i,j}^{(n)}}{\partial w_i^{(n)}} = y_{i,j}^{(n)} (1 - y_{i,j}^{(n)}) x_i^{(n)}$$

Użycie innej formy funkcji przejścia zmienia jedynie pierwszą cześć wyrażenia, należy jednak pamiętać że musi to być funkcja ciągła i różniczkowalna, co nie jest spełnione np. w odniesieniu do funkcji progowej.

Algorytm wstecznej propagacji błędu posiada niestety szereg słabości. Najpoważniejszą z nich jest brak gwarancji na osiągnięcie minimum globalnego funkcji celu. Procedura najczęściej prowadzi do osiągnięcia minimum lokalnego, które może okazać się wystarczająco głębokie i całkiem użyteczne w zastosowaniach praktycznych, ale również całkowicie bezużyteczne. Dlatego jedną z metod poszukiwania dobrych właściwości sieci jest wielokrotne ponawianie procesu uczenia w oparciu o losowo generowane zestawy wag startowych, na ogół z przedziału (-0,5; 0,5). Sam dobór początkowych wartości wag jest także problematyczny, gdyż zbył duże wartości wejść lub wag mogą doprowadzić do negatywnego zjawiska nasycenia funkcji aktywacji, praktycznie do postaci funkcji stałych, i braku postępów w procesie ucznia. Punkt startowy w przestrzeni wag ma duży wpływ na przebieg trenowania, ponieważ to od kształtu powierzchni funkcji celu w jego sąsiedztwie uzależnione jest ryzyko bardzo wolnej zbieżności algorytmu lub porażka utkwienia w płytkim minimum lokalnym.

## 3. Momentowa metoda wstecznej propagacji błędu

Najpopularniejszą modyfikacją algorytmu propagacji wstecznej jest momentowa metoda wstecznej propagacji błędu, której główny pomysł sprowadza się do uzależnienia zmian wartości wag w bieżącej iteracji nie tylko od gradientu funkcji celu, ale także od zmiany tych samych wag w iteracji wcześniejszej. Omawiane rozszerzenie klasycznego algorytmu pozwala na względnie bezpieczne przyspieszenie procesu uczenia i częściowe zminimalizowanie ryzyka utknięcie w minimum lokalnym funkcji celu. Metoda dodaje do opisanego wcześniej sposobu modyfikacji wag drugi element, który jest zależny od parametru β, należącego do przedziału (0, 1].

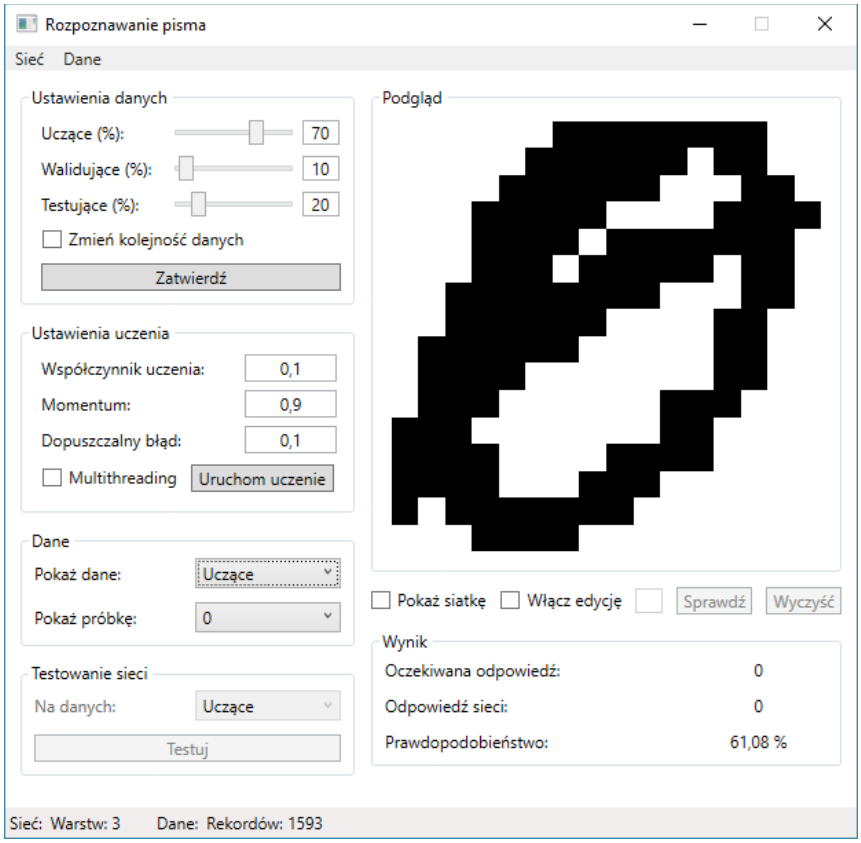
$$w_i^{(n)}(k) = w_i^{(n)}(k-1) - \alpha \frac{\partial Q(k)}{w_i^{(n)}} + \beta \Delta w_i^{(n)}(k-1)$$

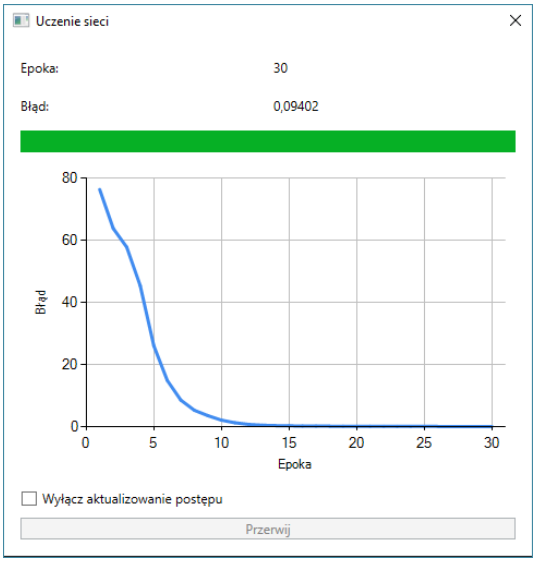
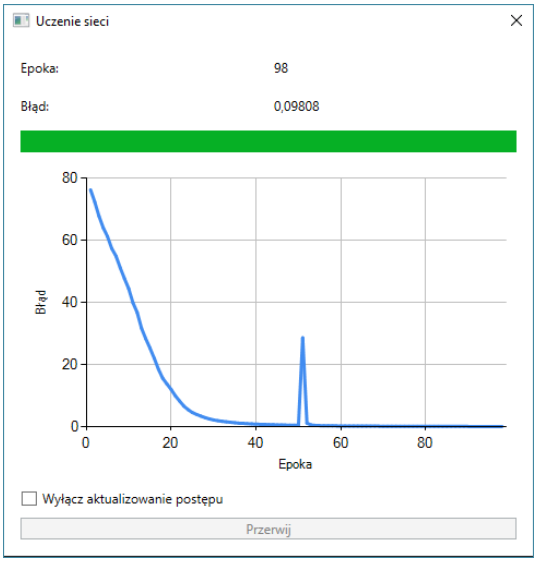
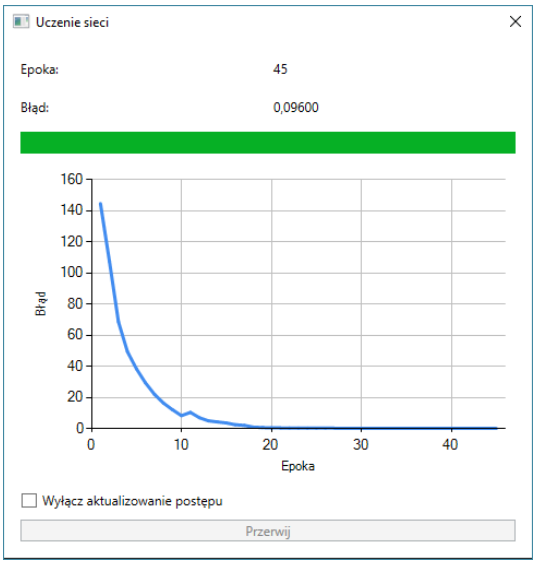
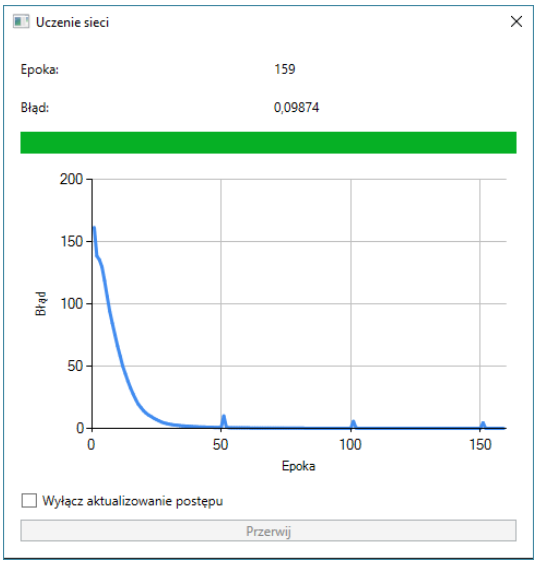
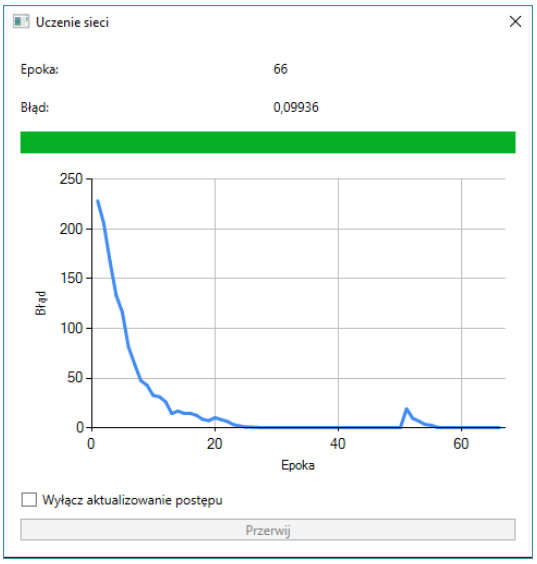
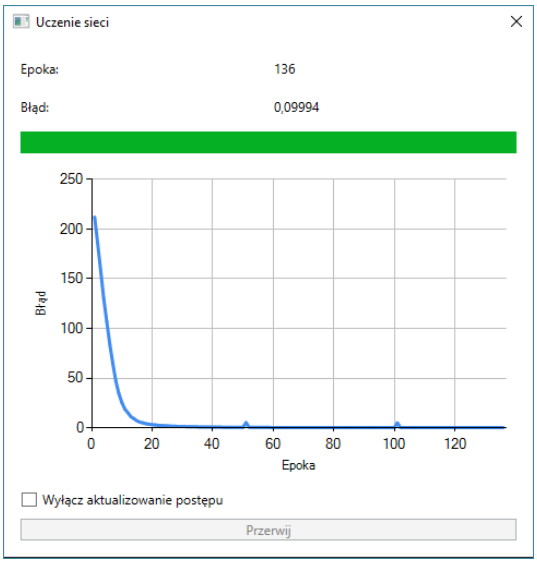
$$\Delta w_i^{(n)}(k-1) = w_i^{(n)}(k-1) - w_i^{(n)}(k-2)$$

$$\beta \in (0, 1]$$

Uzależnienie procedury od zmiany wag w iteracji poprzedniej wprowadza do algorytmu element bezwładności, który zmniejsza chwilowe i gwałtowne zmiany kierunku wskazywanego przez gradient funkcji celu względem zestawu wag sieci. Powyższa właściwość wydaje się szczególnie korzystna w przypadku zastosowania metody uczenia przyrostowego, w której modyfikowanie wag po każdorazowej prezentacji pojedynczego wektora ze zbioru treningowego naraża cały proces na częste zbaczanie z głównej ścieżki minimalizacji. Utrzymywanie względnie stałego kierunku zmian w przestrzeni wag, zmniejsza tendencję do wchodzenia kolejnych zestawów ich wartości w płytkie minima lokalne. Dodatkową zaletą wprowadzenia opisywanej zmiany jest zdolność do znacznego przyspieszania procesu uczenie na obszarach tzw. płaskowyżu funkcji celu, czyli zakresu wartości w przestrzeni wag na którym występują minimalne wahania wielkości funkcji celu, a jej kształt można porównać do płaszczyzny. Można też wykazać analitycznie, iż dla typowej wartości $\beta = 0{,}9$, na idealnie płaskich obszarach funkcji celu, szybkość procesu ucznia może wzrosnąć nawet 10 razy.

## 4. Aplikacja



| | Momentum = 0,9 | Momentum = 0 |
|---|---|---|
| Dane uczące 10%, walidujące 10% |  |  |
| Dane uczące 20%, walidujące 10% |  |  |
| Dane uczące 30%, walidujące 10% |  |  |

Test dla różnych wartości momentum był przeprowadzony z tymi samymi wartościami wag oraz na tych samych danych.

## 5. Kod źródłowy algorytmu

```
private double backpropagate(List<int> indexOrder)
{
   Dictionary<Neuron, double[]> weightsDictionary =
      new Dictionary<Neuron, double[]>();

   double error = 0;

   foreach (int index in indexOrder)
   {
      DataItem dataItem = dataItems[index];

      neuralNetwork.SetInputs(dataItem.Inputs);
      neuralNetwork.Propagate();
      double[] netOutputs = neuralNetwork.GetOutputs();
      double[] expectedOutputs = dataItem.GetDoubleOutputs();

      List<Layer> layers = neuralNetwork.Layers;
      List<Neuron> outputNeurons = layers[layers.Count - 1].Neurons;

      // STEP 1:
      // Calculate error
      for (int i = 0; i < outputNeurons.Count; i++)
         outputNeurons[i].OutputError = netOutputs[i] - expectedOutputs[i];

      for (int i = layers.Count - 1; i > 1; i--)
      {
         List<Neuron> prevNeurons = layers[i - 1].Neurons;
         List<Neuron> currNeurons = layers[i].Neurons;

         for (int j = 0; j < prevNeurons.Count; j++)
         {
            double neuronError = 0;

            foreach (Neuron currNeuron in currNeurons)
               if (!currNeuron.IsBias)
                  neuronError += currNeuron.OutputError *
                     currNeuron.InputWeights[j];

            prevNeurons[j].OutputError = neuronError;
         }
      }

      // STEP 2:
      // Correrct weights
      for (int i = 1; i < layers.Count; i++)
      {
         List<Neuron> prevNeurons = layers[i - 1].Neurons;

         foreach (Neuron currNeuron in layers[i].Neurons)
         {
            if (!currNeuron.IsBias)
            {
               double[] prevWeights = null;
               double[] newWeights = new double[prevNeurons.Count];

               weightsDictionary.TryGetValue(currNeuron, out prevWeights);
```

```
            for (int j = 0; j < prevNeurons.Count; j++)
            {
               double delta = learningRate * currNeuron.OutputError *
                  currNeuron.Derivative * prevNeurons[j].Output;

               double prevDelta = prevWeights == null ? 0 : prevWeights[j] -
                  currNeuron.InputWeights[j];

               double momentumDelta = prevDelta * momentum;

               newWeights[j] = currNeuron.InputWeights[j] -
                  (delta + momentumDelta);
            }

            if (prevWeights == null)
               weightsDictionary.Add(currNeuron, currNeuron.InputWeights);
            else
               weightsDictionary[currNeuron] = currNeuron.InputWeights;

            currNeuron.InputWeights = newWeights;
         }
      }
   }

   neuralNetwork.Propagate();
   netOutputs = neuralNetwork.GetOutputs();
   error += calculateError(netOutputs, expectedOutputs);
  }

return error;
}
```

## 6. Bibliografia


- http://www.fizyka.umk.pl/publications/kmk/01phd-ra.pdf
- http://www.rosczak.com/mlp/mlp.html