Politechnika Warszawska

Wydział Elektroniki i Technik Informacyjnych

Wstęp do Sztucznej Inteligencji

ćwiczenie 5 - Sztuczne sieci neuronowe

Autor:

Jarosław Jaworski, Arkadiusz Niedzielski

Data oddania ćwiczenia:

27.05.2025

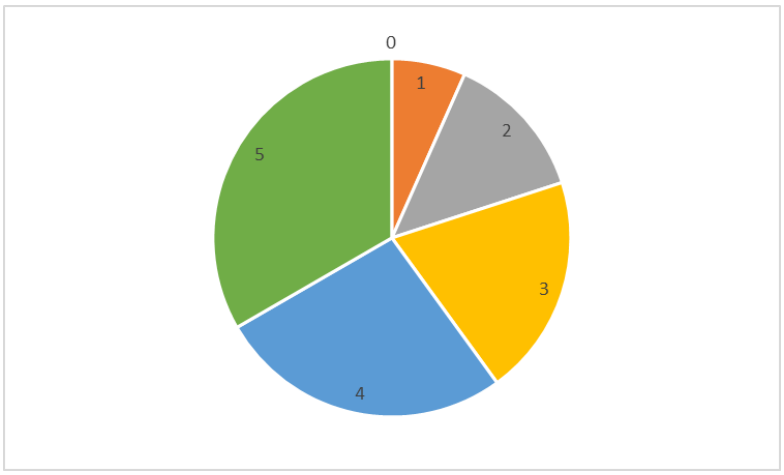
SPIS TREŚCI

1 Cel ćwiczenia

Celem ćwiczenia było utworzenie programu realizującego funkcjonalność sztucznej sieci neuronowej opartej o model perceptonu wielowarstwowego. W ramach ćwiczenia należało dobrać odpowiedni algorytm z rodziny *gradient descent*, następnie korzystając z niego do realizacji propagacji wstecznej (optymalizacji modelu).

2 Analiza źródła danych, realizacja perceptonu wielowarstwowego

Ćwiczenie rozpoczęto od analizy źródła danych w celu identyfikacji nadmiarowości reprezentacji poszczególnych klas. W modelu założono, że klasy reprezentują wartości kolumny quality, oraz przyjęto resztę danych (kolumn) jako cechy (features). Dostępne klasy oraz ich udział w zbiorze przedstawiawiono na Rys. 1.



Rys. 1. Udział poszczególnych klas 0,1,2,3,4,5 w zbiorze danych

Następnie przystąpiono do realizacji konstruktora klasy sieci, realizującego inicjalizację wszystkich wag oraz *biasów* z wykorzystaniem rozkładu normalnego

funkcji *random.randn()* z biblioteki *numpy*. Zdecydowano się na stworzenie perceptonu o trzech warstwach (wejściowej, jednej ukrytej i wyjściowej) o ilości neuronów, przypadających na warstwę odpowiednio: wymiar wektora cech, 8, wymiar wektora klas. Do podziału danych użyto wbudowanej funkcji biblioteki *scikit_learn - train_test_split()*. Dane podzielono w stosunku 4:1 - zestaw treningowy:testowy. Całość przedstawiono w Wycinku kodu 1.

```
class Network(object):
    def __init__(self, sizes: list,
        cost=CrossEntropyCost):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in
            self.sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(self.sizes[:-1],
                            self.sizes[1:])]
        self.cost=cost
    /.../
//(main.py:)
X_train, X_test, y_train, y_test = train_test_split(X,
    y_onehot, test_size=0.2, random_state=42)

num_features = X_train.shape[1]
num_classes = y_train.shape[1]
net = Network([num_features, 8, num_classes])
```

Wycinek kodu 1: Konstruktor klasy i utworzenie obiektu sieci

Kolejnym etapem realizacji perceptonu wielowarstwowego była implementacja metody *feedforward* generującej predykcje i wystawiającej aktywacje potrzebne

do późniejszej optymalizacji. Ze względu na niską złożoność modelu zastosowano w tym celu funkcję sigmoidalną do obliczania aktywacji. Skorzystano z zależności (1)

$$\frac{1}{1+e^{-z}} \tag{1}$$

,gdzie:

z - suma iloczynu wagi i aktywacji oraz *biasu*.

Ostatnim krokiem realizacji modelu sieci neuronowej była implementacja wybranej formy algorytmu *gradient descent* oraz wykorzystanie jej do wykonania propagacji wstecznej. Ze względu na niewielką ilość neuronów w każdej z warstw postanowiono zastosować się **klasyczny algorytm spadku gradientu** zamiast jego modyfikacji, takich jak obliczanie w batchach (ograniczone spojrzenie na całość kontekstu) lub SGD (dodatkowa niedokładność). W każdym kroku aktualizacji wag i biasów obliczano całkowite sumy gradientów dla wszystkich próbek. Następnie uśredniano te gradienty aby zauktalizowaćwagi i biasy zgodnie ze wzorem spadku gradientu (patrz - **Ćwiczenie 1.**). Podejście to dodatkowo umożliwiło obliczanie funkcji straty przez zwykłe zastosowanie aproksymatora (różnica wartości rzeczywistej i wartości wystawionej przez model). Proces powtarzano przez zadaną jako hiperparametr liczbę epok.

3 Wyniki pomiarów

Dla modelu obliczono metrykę dokładności (patrz - **Ćwiczenie 4.**) przy różnej ilości epok. Obliczenia przeprowadzono zarówno dla zbioru testowego jak i trenującego. Wyniki przedstawiono w Tabeli 1.

Tabela 1. Wyniki pomiarów dokładności dla różnej ilości epok działania algorytmu spadku gradientu

Ilość epok [-]	Dokładność z. trenujący [%]	Dokładność z. testujący[%]
100	50,00	43,63
500	57.25	51.96
1000	61.18	50.98
2500	61.18	50.98

4 Wnioski

Model wykazywał się średnio zadowalającą dokładnością dla zadania predykcji dla danych wejściowych treningowych i testowych. Było to podyktowane najprawdopodobniej nierównym rozkładem danych, co oznacza, że część poszczególnych rekordów była nadreprezentowana przy uczeniu. Kolejnym elementem wpływającym na wynik działania programu jest ilość danych. Zbiór zawiera ich stosunkowo niewielką liczbę, co przy fakcie nierównego ich rozkładu wzmacnia efekt nieidealnego wyuczenia modelu.

Sieć neuronową w podstawowej formie utworzonej w ramach laboratorium można potraktować jako funkcję złożoną, którą optymalizujemy. W związku z powyższym kluczowym elementem programu jest dobór i zapewnienie poprawności działania algorytmu *gradient descent* odpowiedzialnego za *tuning* modelu. Zmiany hiperparametru ilości epok ma znaczny wpływ na efektywność algorytmu. W ramach ćwiczeniach optymalną ilością co do czasu działania programu i uzyskanej dokładności było wymuszenie działania przez 1000 epok.

Pominięcie zbioru walidacyjnego mogło mieć wpływ na niedokładność modelu, przez brak dodatkowego punktu odniesienia przy trenowaniu modelu.