Machine learning - zaliczenie

Michał Ślusarski, nr albumu 420090

Raport zawiera opis poszczególnych etapów tworzenia sieci, eksperymentów i definiowanych funkcji. Całość **kodu, wraz z danymi i zapisanymi wagami, umieściłem w folderze:**

https://www.dropbox.com/scl/fo/zxv1hiuylsr05wx36oaxz/h?rlkey=iue08eg6y78y20nafofm3nocz&dl=0

Przygotowanie danych

Pracę z danymi zaczynam od wizualnego przejrzenia zbioru. Określiłem, że znajdują się tam dwa rzędy wyjaśniające nazwę i typ prezentowanej zmiennej. Jest też sporo wartości pustych, najprawdopodobniej reprezentujących brak wartości.

Celem oczyszczenia zbioru, utworzyłem skrypt **clean_data**. W ramach operacji, ładuję nieoczyszczony zbiór basic.csv. Przeprowadzam podstawowe czyszczenie: usunięcie rzędów wyjaśniających oraz kolumny Z13; uzupełnienie braków danych zerami oraz profilaktyczne unormowanie typu danych w tabeli (*to_numeric*).

```
import pandas as pd
import numpy as np

df = pd.read_csv('basic.csv')

df = df.drop([0, 1])
df = df.reset_index(drop=True)
df = df.drop('Z13', axis=1)
df = df.fillna(0)
df = df.apply(pd.to_numeric)
```

Następnym krokiem jest standaryzacja danych, tak aby wartości mieściły się w podobnym zakresie. Bez standaryzacji wysokie wartości mogłyby zdominować proces uczenia się sieci.

Do standaryzacji wybieram kolumny liczbowe (za kategoryczne uznaję wszystkie: 0-1). Korzystam z metody max-min, która przeskalowuje wartości w określonych kolumnach do standardowego zakresu od 0 do 1. Odbywa się to poprzez odjęcie wartości minimalnej w kolumnach i jej podzielenie przez zakres wartości tam znajdowanych.

```
columns_to_standardize = ['Z11', 'Z14', 'Z15', 'Z16', 'Z17', 'Z18', 'Z19', 'Z20']
df[columns_to_standardize] = round((df[columns_to_standardize] -
df[columns_to_standardize].min()) / (df[columns_to_standardize].max() -
df[columns_to_standardize].min()), 6)
```

Zamieniam zmienną Z21 z numerycznej na kategoryczną, za pułap przyjmując medianę wartości liczby komentarzy. Dzięki temu można podzielić zbiór na dwie mniej więcej równe klasy.

```
median = df['Z21'].median()
df['Z21'] = np.where(df['Z21'] > median, 1, 0)
```

Następnie sprawdzam, czy uzyskane klasy rzeczywiście są w miarę równoliczne.

```
print(df['Z21'].value_counts())
```

Dla mediany wychodzą następujące liczebności klas:
0 341
1 301

Dla porównania, przy grupowaniu po średniej, klasy były znacząco nierówne:
0 488
1 154

Wartość mediany to **7**. To znaczy, że 341 postów ma równo 7 lub mniej komentarzy. 301 postów ma więcej. Klasy nie mogą być dokładnie równe, gdyż wartości zachodziłyby na siebie. **Zadaniem sieci będzie przewidzieć, czy przy danym poście liczba komentarzy jest większa niż mediana zbioru.**

Pod koniec procesu przetwarzania, 50 ostatnich pozycji ze zbioru przeznaczam na zbiór testowy. Usuwam je ze zbioru treningowego.

```
test_df = df[-50:]
df = df[:-50]
```

Zapisuję zbiory.

```
df.to_csv('training.csv')
test_df.to_csv('test.csv')
```

Budowa sieci neuronowej

Na początku przetestowałem architekturę opartą o kod z zajęć. Oczyściłem go z komentarzy, ponieważ był on omawiany na zajęciach. Jedyną modyfikacją była liczba inputów/wejść (11).

```
import pandas as pd
import numpy as np

class NeuralNetwork():

    def __init__(self):
        np.random.seed(1)
        self.synaptic_weights = 2 * np.random.random((11, 1)) - 1

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def train(self, training_inputs, training_outputs, training_iterations):

        for iteration in range(training_iterations):
            output = self.think(training_inputs)
            error = training_outputs - output
            adjustments = np.dot(training_inputs.T, error *
self.sigmoid_derivative(output))
            self.synaptic_weights += adjustments

    def think(self, inputs):
        inputs = inputs.astype(float)
        output = self.sigmoid(np.dot(inputs, self.synaptic_weights))
        return output
```

Trenowanie modelu

Tym co uległo zmianie, w relacji do skryptu z zajęć, jest sposób wprowadzania i walidacji danych.

W deklaracji `if __name__ == "__main__":` zaczynam od zaimportowania danych z setu treningowego.

```
    df = pd.read_csv("training.csv")
```

Wyciągam zmienne Z9-Z20 jako zmienne objaśniające/predykcyjne i Z21 jako objaśnianą (labels).

```
    input_features = df[['Z9', 'Z10','Z11', 'Z12', 'Z14', 'Z15', 'Z16', 'Z17',
'Z18', 'Z19', 'Z20']]
    output_labels = df[['Z21']]
```

Tak jak w oryginalnym skrypcie, wartości z poszczególnych kolumn są przypisywane do wejść i wyjścia sieci.

```
    training_inputs = input_features.values
    training_outputs = output_labels.values
```

Inicjalizowany jest obiekt sieci i wywoływana jest metoda trenująca.

```
    neural_network = NeuralNetwork()
    neural_network.train(training_inputs, training_outputs, 11000)
```

Uzyskane wagi zapisuję w pliku weights.npy. Rozszerzenie .npy jest częścią biblioteki *numpy*.

```
np.save("weights.npy", neural_network.synaptic_weights)
```

Testowanie modelu

Przechodzę do walidacji modelu. Na początku inicjuję sieć, wczytując zapisane wagi.

```
    neural_network = NeuralNetwork()
    neural_network.synaptic_weights = np.load("weights.npy")
```

Analogicznie do zbioru treningowego przygotowuję zbiór testowy.

```
    test_df = pd.read_csv("test.csv")
    test_input_features = test_df[['Z9', 'Z10','Z11', 'Z12', 'Z14', 'Z15', 'Z16',
'Z17', 'Z18', 'Z19', 'Z20']]
    test_output_labels = test_df[['Z21']]
    test_inputs = test_input_features.values
    test_outputs = test_output_labels.values
```

Wartości testowe zapuszczane są do modelu i przypisane do zmiennej *predictions*. Jako że są to wartości zmiennoprzecinkowe, zamieniam je na binarne, wywołując metodę *astype()*. Wartości, które przekroczą próg 0.5 zliczane są jako 1 - pozostałe jako 0.

```
    predictions = neural_network.think(test_inputs)
    binary_predictions = (predictions > 0.5).astype(int)
```

Metryką, którą posłużę się do sprawdzenia modelu jest *accuracy*, czyli dokładność. Obliczam dokładność modelu wyciągając średnią z przypadków gdzie predykcja jest równa rzeczywistości. Następnie, wynik mnożony jest przez 100, aby otrzymać wartość w procentach.

Na przykład, jeśli accuracy wynosi 85, to oznacza, że model poprawnie sklasyfikował 85% próbek ze zbioru testowego.

```
    accuracy = np.mean(binary_predictions == test_outputs) * 100
    print("Mean accuracy:", accuracy)
```

Średnia dokładność modelu wyniosła **70%**. Nie jest to wynik wybitny, ale całkiem przyzwoity jak na jedną warstwę.

Modyfikacja modelu

Dodawanie warstwy pośredniej

Pierwszą modyfikacją architektury, która mogłaby potencjalnie usprawnić jego dokładność, jest warstwa pośrednia, zwana też ukrytą. Dodanie takiej warstwy rozpoczynam od zadeklarowania nowego zestawu wag dla każdej z warstw.

Warstwa wejściowa ma 11 neuronów przyjmujących input treningowy. Tych 11 neuronów połączonych jest z 5 neuronami w warstwie ukrytej, które z kolei łączą się z jednym neuronem wyjściowym. Liczba 5 neuronów jest wynikiem dalszych eksperymentów.

```
    def __init__(self):
        np.random.seed(1)
        self.synaptic_weights1 = 2 * np.random.random((11, 5)) - 1
        self.synaptic_weights2 = 2 * np.random.random((5, 1)) - 1
```

Modyfikacji ulega funkcja treningowa. Dodaję output warstwy ukrytej. Jego deklaracja jest taka sama jak outputu całej funkcji w wersji jednowarstwowej. Tam występuje jednak tylko w funkcji *think*. Tutaj musi być uzgadniany co iterację z warstwą ukrytą. Należy pamiętać, że output całej funkcji przyjmuje teraz output ukryty, a nie treningowy.

```
 def train(self, training_inputs, training_outputs, training_iterations):

        for iteration in range(training_iterations):
            hidden_output = self.sigmoid(np.dot(training_inputs,
self.synaptic_weights1))
            output = self.sigmoid(np.dot(hidden_output, self.synaptic_weights2))
```

Tak jak wcześniej, błąd w warstwie wyjściowej jest obliczany na podstawie różnicy między oczekiwanymi wynikami a przewidywanymi wynikami. Następnie dostosowania/korekty (*output_adjustments*) są obliczane przy użyciu pochodnej funkcji aktywacji warstwy wyjściowej, tak samo jak w przypadku jednowarstwowej sieci.

Na podstawie tych dostosowań oblicza się błąd w warstwie ukrytej przy użyciu macierzy wag między warstwą ukrytą a wyjściową. Mnożąc te korekty przez wagi *synaptic_weights2*, błąd z warstwy wyjściowej przenoszony jest z powrotem do warstwy ukrytej. Ten krok pozwala przypisać część błędu w warstwie wyjściowej do każdego neuronu w warstwie ukrytej.

```
            output_error = training_outputs - output
            output_adjustments = output_error * self.sigmoid_derivative(output)

            hidden_error = np.dot(output_adjustments, self.synaptic_weights2.T)
            hidden_adjustments = hidden_error *
self.sigmoid_derivative(hidden_output)
```

Output ukryty deklarowany jest tak samo, jak wyjściowy:

```
    def think(self, inputs):
        inputs = inputs.astype(float)
        hidden_output = self.sigmoid(np.dot(inputs, self.synaptic_weights1))
        output = self.sigmoid(np.dot(hidden_output, self.synaptic_weights2))
        return output
```

Testowanie modelu odbywa się na tej samej zasadzie co poprzednio. Okazuje się jednak, że model traci na dokładności. Po kilku eksperymentach z liczbą neuronów pośrednich, najlepsze wyniki uzyskałem z 5 neuronami. Wyniki te są i tak gorsze niż w jednowarstwowej sieci, wynosząc **jedynie 54%**. Możliwe, że wynika to z jakiegoś błędu logicznego z mojej strony. Przyznaję, że dodanie tej warstwy stanowiło spore wyzwanie, szczególnie w kwestii zrozumienia mechanizmu propagacji wstecznej.

Jeśli jednak zakładać, że sieć działa poprawnie, najprawdopodobniej, dodatkowa warstwa wprowadza niepotrzebny poziom skomplikowania do prostej zależności, którą obserwuje się w danych. Ponadto, modele wielowarstwowe mogą wymagać większej ilości danych treningowych. Ewentualnie, można spróbować zdefiniować inną, prostszą funkcję aktywacji w warstwie ukrytej.

Zmiana funkcji aktywacji

Prostą funkcją aktywacji, którą można zastosować w warstwie ukrytej jest ReLU. Przyporządkowuje ona wartościom dodatnim ich wartość w sposób liniowy, a wartościom ujemnym wartość 0. Do definicji funkcji wykorzystuję metodę *maximum* z biblioteki *numpy*.

```
    def relu(self, x):
        return np.maximum(0, x)
```

Pochodna z ReLU to pochodna z *X* dla liczb dodatnich i pochodna z 0 dla liczb ujemnych. Jej wartości to więc odpowiednio 1 i 0.

```
    def relu_derivative(self, x):
        return np.where(x > 0, 1, 0)
```

Jedyną zmianą w stosunku do sieci dwuwarstwowej jest konieczność podmienienia funkcji sigmoid na ReLU we wszystkich miejscach, w których figurowała jako warstwa ukryta. Przykład poniżej:

```
hidden_output = self.sigmoid(np.dot(inputs, self.synaptic_weights1))
```

Zamieniam na:

```
hidden_output = self.relu(np.dot(training_inputs, self.synaptic_weights1))
```

Eksperymentując z liczbą neuronów pośrednich, przy 7 neuronach model osiągnął maksymalny wynik **64%** dokładności. Lepiej, niż w przypadku dwóch warstw z

aktywacją sigmoidalną, ale wciąż gorzej niż w przypadku jednowarstwowej sieci. Jak widać prostsze, nie znaczy gorsze.

Finalnie, najlepszym rozwiązaniem jest pozostanie przy oryginalnej architekturze jednowarstwowej sieci.

Źródła

[1] Trusk, Andrew, *A Neural Network in 11 lines of Python*, https://iamtrask.github.io/2015/07/12/basic-python-network/, dostęp: 26.06.2023

[2] Spencer-Harper, Milo, *How to build a multi-layered neural network in Python*, https://medium.com/technology-invention-and-more/how-to-build-a-multi-layered-neural-network-in-python-53ec3d1d326a, dostęp: 26.06.2023

[3] Shi, Angela & Kezhan, *Neural network, why deeper isn't always better*, *https://towardsdatascience.com/neural-network-why-deeper-isnt-always-better-2f862f40e2c4,* dostęp: 26.06.2023

[4] *Why would adding more neurons per hidden layer in a neural network hinder convergence within the actual training set?*, https://www.quora.com/Why-would-adding-more-neurons-per-hidden-layer-in-a-neural-network-hinder-convergence-within-the-actual-training-set, dostęp: 26.06.2023