Tekst do prezentacji

Wstęp

Sieć neuronowa jest właściwie funkcją wielu zmiennych: przyjmuje wejście, wykonuje obliczenia i generuje wynik. Lubimy ją wizualizować jako neurony w różnych warstwach, gdzie każdy neuron w warstwie jest połączony z wszystkimi neuronami w poprzedniej i następnej warstwie. Wszystkie obliczenia odbywają się wewnątrz tych neuronów i zależą od wag, które łączą neurony ze sobą. Więc wszystko, co musimy zrobić, to nauczyć się odpowiednich wag, aby uzyskać pożądany wynik.

W związku z tym, poniżej zaimplementujemy bardzo prostą sieć z 2 warstwami. Aby to zrobić, potrzebujemy również bardzo prostego zestawu danych, więc w naszym przykładzie użyjemy zestawu danych XOR.

XOR

A i B to 2 wejścia NN, a A XOR B to wyjście. Spróbujemy nauczyć naszą sieć neuronową wag tak, aby dla dowolnej pary wejść A i B zwracała odpowiedni wynik.

Najpierw musimy zdefiniować strukturę naszej sieci neuronowej. Ponieważ nasz zestaw danych jest stosunkowo prosty, sieć z jedną ukrytą warstwą będzie wystarczająca. Będziemy mieli warstwę wejściową, ukrytą warstwę i warstwę wyjściową. Następnie potrzebujemy funkcji aktywacji. Funkcja sigmoidalna jest dobrym wyborem dla ostatniej warstwy, ponieważ wyprowadza wartości między 0 a 1, podczas gdy tanh (tangens hiperboliczny) działa lepiej w warstwie ukrytej, ale każda inna powszechnie używana funkcja również będzie działać (np. ReLU). Struktura naszej sieci neuronowej będzie wyglądać następująco:

Tutaj parametrami, które należy nauczyć, są wagi W1, W2 i odchylenia b1, b2. Jak można zauważyć, W1 i b1 łączą warstwę wejściową z warstwą ukrytą, podczas gdy W2, b2 łączą warstwę ukrytą z warstwą wyjściową. Z podstawowej teorii wiemy, że aktywacje A1 i A2 są obliczane następująco:

gdzie g i h to dwie funkcje aktywacji, które wybraliśmy (dla nas sigmoid i tangens hiperboliczny), a W1, W2, b1, b2 to zazwyczaj macierze.

Teraz przejdźmy do właściwego kodu.

XOR w języku Python

Najpierw zaimplementujemy naszą funkcję aktywacji sigmoidalną zdefiniowaną następująco: g(z) = 1/(1+e^(-z)), gdzie z będzie zazwyczaj macierzą. Na szczęście numpy obsługuje obliczenia z macierzami, więc kod jest stosunkowo prosty:

import numpy as np

def sigmoid(z):  
 return 1/(1 + np.exp(-z))

Następnie musimy zainicjować nasze parametry. Macierze wag W1 i W2 będą inicjowane losowo z rozkładu normalnego, podczas gdy odchylenia b1 i b2 będą inicjowane na zero. Funkcja initialize\_parameters(n\_x, n\_h, n\_y) przyjmuje jako dane wejściowe liczbę jednostek w każdej z 3 warstw i odpowiednio inicjuje parametry:

def initialize\_parameters(n\_x, n\_h, n\_y):

inicjalizuje macierz wag W1 o wymiarach (n\_h, n\_x) z losowymi wartościami z rozkładu normalnego (średnia = 0, odchylenie standardowe = 1)  
 W1 = np.random.randn(n\_h, n\_x)  
inicjalizuje wektor biasów b1 o wymiarach (n\_h, 1) z wartościami zerowymi  
 b1 = np.zeros((n\_h, 1))  
inicjalizuje macierz wag W2 o wymiarach (n\_y, n\_h) z losowymi wartościami z rozkładu normalnego (średnia = 0, odchylenie standardowe = 1)  
 W2 = np.random.randn(n\_y, n\_h)  
inicjalizuje wektor biasów b2 o wymiarach (n\_y, 1) z wartościami zerowymi  
 b2 = np.zeros((n\_y, 1))

tworzy słownik parameters, który zawiera zainicjalizowane wartości wag i biasów.  
 parameters = {  
 "W1": W1,  
 "b1" : b1,  
 "W2": W2,  
 "b2" : b2 }

return parameters

Następnym krokiem jest zaimplementowanie propagacji sygnału (Forward Propagation). Funkcja forward\_prop(X, parameters) przyjmuje jako dane wejściowe macierz wejściową sieci neuronowej X i słownik parametrów, a następnie zwraca wynik działania sieci neuronowej A2 wraz z cache'em, który zostanie użyty później w propagacji wstecznej (backpropagation).

def forward\_prop(X, parameters):

W1 = parameters["W1"]  
 b1 = parameters["b1"]  
 W2 = parameters["W2"]  
 b2 = parameters["b2"]

W sieciach neuronowych, funkcje aktywacji są wykorzystywane w celu wprowadzenia nieliniowości do modelu. Nielinearność jest kluczowym elementem w tworzeniu modeli, które potrafią dobrze modelować złożone zależności między danymi wejściowymi i wyjściowymi.

W powyższej funkcji forward\_prop, funkcja aktywacji użyta w pierwszej warstwie to tangens hiperboliczny, a w drugiej warstwie sigmoid. Tangens hiperboliczny jest często używany jako funkcja aktywacji w pierwszej warstwie ze względu na jego symetryczny kształt, który umożliwia jednoczesne przetwarzanie informacji zarówno w kierunku dodatnim, jak i ujemnym. Sigmoida jest często używana w ostatniej warstwie sieci neuronowej, ponieważ pozwala na uzyskanie wyjścia w zakresie od 0 do 1, co może być interpretowane jako prawdopodobieństwo.   
 Z1 = np.dot(W1, X) + b1 iloczyn skalarny  
 A1 = np.tanh(Z1)  
 Z2 = np.dot(W2, A1) + b2  
 A2 = sigmoid(Z2)

słownik cache przechowuje wartości A1 i A2, czyli wyjścia z warstw 1 i 2 sieci neuronowej, odpowiednio. Są one potrzebne do obliczenia pochodnych funkcji kosztu względem wag podczas propagacji wstecznej  
 cache = {  
 "A1": A1,  
 "A2": A2 }

return A2, cache

Teraz musimy obliczyć funkcję kosztu. Będziemy korzystać z funkcji kosztu Cross-Entropy (Entropia krzyżowa). Funkcja calculate\_cost(A2, Y) przyjmuje jako argumenty wyjście sieci neuronowej A2 oraz oczekiwaną wartość (ground truth) Y i zwraca wartość kosztu Cross-Entropy:

def calculate\_cost(A2, Y):

cost = -np.sum(np.multiply(Y, np.log(A2)) + np.multiply(1-Y, np.log(1-A2)))/m

cost = np.squeeze(cost)

return cost

W tym przypadku, poszukiwanie minimalizacji funkcji kosztu sprowadza się do znalezienia takich wag i biasów sieci neuronowej, które pozwolą na jak najlepsze odwzorowanie rzeczywistych etykiet klas w postaci binarnej (0 lub 1).

ta część formuły: np.multiply(Y, np.log(A2)) + np.multiply(1-Y, np.log(1-A2))  
odpowiada za mierzenie podobieństwa pomiędzy przewidywaniami A2 a rzeczywistymi wartościami Y. Im bliżej siebie te wartości są, tym mniejszy będzie koszt.

Z kolei ten dodatek: -np.sum(...)  
odpowiada za przeliczenie kosztu dla całego zbioru treningowego. Ostatecznie, podzielone przez liczbę przykładów, koszt ma informować nas o tym, jak dobrze dany model poradził sobie z danym problemem klasyfikacji binarnej.

Funkcja np.squeeze() usuwa jednostkowe wymiary z tablicy numpy. W tym konkretnym przypadku, gdy cost jest jednoelementową tablicą numpy, np.squeeze() usuwa ten jednostkowy wymiar i zwraca prostą liczbę, czyli wartość funkcji kosztu.

Najtrudniejsza część algorytmu sieci neuronowej, propagacja wsteczna.

Ta funkcja zwróci gradienty funkcji straty względem 4 parametrów naszej sieci (W1, W2, b1, b2):

Funkcja backward\_prop przyjmuje cztery argumenty: macierz wejściową X, macierz etykiet Y, słownik cache z wartościami zwróconymi przez funkcję forward\_prop, i słownik parameters z wartościami wag i biasów.

def backward\_prop(X, Y, cache, parameters):

Wartości aktywacji dla ukrytej warstwy A1 i końcowej warstwy A2 są pobierane ze słownika cache, który został zwrócony przez funkcję forward\_prop.

A1 = cache["A1"]

A2 = cache["A2"]

Macierz wag W2 jest pobierana ze słownika parameters, który zawiera wartości wag i biasów.

W2 = parameters["W2"]

Obliczamy różnicę między przewidywaniami modelu A2 a rzeczywistymi etykietami Y.

dZ2 = A2 – Y

Obliczamy gradienty dla wag W2 i biasów b2 korzystając ze wzorów matematycznych dla Backpropagation.

dW2 = np.dot(dZ2, A1.T)/m

db2 = np.sum(dZ2, axis=1, keepdims=True)/m

Obliczamy gradienty dla wag W1 i biasów b1 korzystając ze wzorów matematycznych dla Backpropagation.

dZ1 = np.multiply(np.dot(W2.T, dZ2), 1-np.power(A1, 2))

dW1 = np.dot(dZ1, X.T)/m

db1 = np.sum(dZ1, axis=1, keepdims=True)/m

Zwracamy gradienty jako słownik.

grads = {

"dW1": dW1,

"db1": db1,

"dW2": dW2,

"db2": db2

}

return grads

Świetnie, mamy już wszystkie gradienty funkcji straty, możemy przejść do faktycznego uczenia! Będziemy używać algorytmu Gradient Descent do aktualizacji naszych parametrów i nauczenia naszego modelu, z prędkością uczenia przekazaną jako parametr:

Funkcja update\_parameters służy do aktualizacji wag i biasów sieci neuronowej na podstawie obliczonych gradientów funkcji kosztu.

def update\_parameters(parameters, grads, learning\_rate):

Tutaj pobierane są aktualne wartości parametrów,

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

a tutaj pobierane są obliczone wcześniej gradienty

dW1 = grads["dW1"]

db1 = grads["db1"]

dW2 = grads["dW2"]

db2 = grads["db2"]

aktualizowane są wartości wag i biasów zgodnie z regułą Gradient Descent zadaną przez learning\_rate.

W1 = W1 - learning\_rate\*dW1

b1 = b1 - learning\_rate\*db1

W2 = W2 - learning\_rate\*dW2

b2 = b2 - learning\_rate\*db2

Ostatecznie, funkcja zwraca słownik z nowymi wartościami parametrów.

new\_parameters = {

"W1": W1,

"W2": W2,

"b1" : b1,

"b2" : b2

}

return new\_parameters

Do tej pory zaimplementowaliśmy wszystkie funkcje potrzebne do jednego cyklu treningowego. Teraz, co musimy zrobić, to po prostu połączyć je wszystkie wewnątrz funkcji o nazwie model() i wywołać model() z programu głównego.

Funkcja model() przyjmuje jako dane wejściowe macierz cech X, macierz etykiet Y, liczbę jednostek n\_x, n\_h, n\_y, liczbę iteracji, przez które chcemy, aby nasz algorytm Gradient Descent działał, oraz współczynnik uczenia Gradient Descent i łączy wszystkie powyższe funkcje, aby zwrócić wytrenowane parametry naszego modelu:  
def model(X, Y, n\_x, n\_h, n\_y, num\_of\_iters, learning\_rate):  
Na początku funkcja inicjalizuje wagi sieci neuronowej używając funkcji initialize\_parameters().  
 parameters = initialize\_parameters(n\_x, n\_h, n\_y)  
Następnie, pętla for wykonuje num\_of\_iters iteracji algorytmu Gradient Descent.  
 for i in range(0, num\_of\_iters+1):  
W każdej iteracji, wywołana zostaje funkcja forward\_prop() i wyznaczane są wyniki warstw sieci neuronowej.  
 a2, cache = forward\_prop(X, parameters)  
Następnie obliczana jest wartość funkcji kosztu wykorzystując funkcję calculate\_cost().  
 cost = calculate\_cost(a2, Y)  
W kolejnym kroku wyznaczane są pochodne funkcji kosztu z uwagi na parametry sieci neuronowej wykorzystując funkcję backward\_prop().  
 grads = backward\_prop(X, Y, cache, parameters)  
Na koniec, wagi sieci neuronowej są aktualizowane używając funkcji update\_parameters().  
 parameters = update\_parameters(parameters, grads, learning\_rate)  
Co sto iteracji, wyświetlany jest koszt sieci neuronowej.  
 if(i%100 == 0):  
 print('Cost after iteration# {:d}: {:f}'.format(i, cost))  
funkcja zwraca wytrenowane wagi sieci neuronowej.  
 return parameters

Trening został już zakończony. Funkcja powyżej zwróci wytrenowane parametry naszej sieci neuronowej. Teraz musimy tylko dokonać predykcji. Funkcja predict(X, parameters) przyjmuje jako dane wejściowe macierz X z dwoma liczbami, dla których chcemy obliczyć funkcję XOR, oraz wytrenowane parametry modelu i zwraca oczekiwany wynik y\_predict, wykorzystując próg 0,5:  
def predict(X, parameters):  
Wykonuje propagację w przód, aby uzyskać wartość wyjściową a2 i zapamiętane wartości aktywacji cache.  
 a2, cache = forward\_prop(X, parameters)  
Przypisuje yhat wartość a2.  
 yhat = a2  
Wykorzystuje funkcję np.squeeze(), aby zwrócić tablicę jednowymiarową o wyższym stopniu, co oznacza, że usuwa ona wymiary o jednym elemencie z tablicy yhat.  
 yhat = np.squeeze(yhat)  
Porównuje wartość yhat z wartością graniczną 0.5, aby określić, czy wartość wyjściowa powinna wynosić 0 czy 1, a następnie zwraca wartość y\_predict.  
 if(yhat >= 0.5):  
 y\_predict = 1  
 else:  
 y\_predict = 0

return y\_predict

Skończyliśmy z implementacją wszystkich potrzebnych funkcji. Teraz przejdźmy do głównego programu i zadeklarujmy nasze macierze X, Y oraz hiperparametry n\_x, n\_h, n\_y, num\_of\_iters, learning\_rate:

Funkcja np.random.seed(2) ustawia seed generatora liczb losowych biblioteki numpy na 2, co zapewni powtarzalność wyników w przypadku wielokrotnego uruchomienia programu z tą samą wartością seed'a.  
np.random.seed(2)  
  
Tworzymy macierze X i Y, które reprezentują 4 przypadki treningowe, gdzie X to macierz o wymiarach 2x4, gdzie każda kolumna zawiera 2 liczby (odpowiadające 2 wejściom w funkcji XOR) a Y to macierz o wymiarach 1x4 zawierająca oczekiwane wyjście funkcji XOR dla każdego przypadku treningowego.  
# The 4 training examples by columns  
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])

# The outputs of the XOR for every example in X  
Y = np.array([[0, 1, 1, 0]])

# No. of training examples  
liczba przypadków treningowych  
m = X.shape[1]

# Set the hyperparameters  
definiujemy hiperparametry n\_x, n\_h, n\_y, num\_of\_iters i learning\_rate, które będą używane do trenowania sieci neuronowej  
n\_x = 2 #No. of neurons in first layer  
n\_h = 2 #No. of neurons in hidden layer  
n\_y = 1 #No. of neurons in output layer

num\_of\_iters = 1000

learning\_rate = 0.3

Mając ustawione powyżej, trenowanie modelu na tych danych sprowadza się do wykonania jednej linii kodu:  
trained\_parameters = model(X, Y, n\_x, n\_h, n\_y, num\_of\_iters, learning\_rate)

Testujemy dla (1,1)

# Test 2X1 vector to calculate the XOR of its elements.   
# Try (0, 0), (0, 1), (1, 0), (1, 1)  
W tym kodzie tworzony jest wektor testowy X\_test z elementami (1,1), czyli wektor testowy będzie zawierał wartości [1,1].  
X\_test = np.array([[1], [1]])  
Następnie wywoływana jest funkcja predict() z wektorem testowym X\_test i nauczonymi parametrami "trained\_parameters". Funkcja ta zwróci wynik predykcji y\_predict dla podanego wektora testowego X\_test.  
y\_predict = predict(X\_test, trained\_parameters)  
Na końcu wypisywany jest wynik predykcji w formacie tekstowym z użyciem print(). Wynik zostanie wyświetlony w postaci 0 lub 1, oznaczającej predykcję dla danego wektora testowego.  
print('Neural Network prediction for example ({:d}, {:d}) is {:d}'.format(X\_test[0][0], X\_test[1][0], y\_predict))

**SKLEARN**

import numpy as np.  
Importuje klasę MLPClassifier z modułu neural\_network biblioteki scikit-learn.  
from sklearn.neural\_network import MLPClassifier  
Ustawia wartość ziarna losowego generatora liczb losowych NumPy na 2, dzięki czemu wyniki można replikować.  
np.random.seed(2)  
Definiuje macierz wejściową X, która zawiera cztery możliwe kombinacje 0 i 1 dla dwóch zmiennych binarnych (0, 0), (0, 1), (1, 0) i (1, 1).  
# The 4 training examples by columns  
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  
Definiuje wektor wyjściowy Y, który zawiera odpowiadające wyniki operacji XOR dla każdego wiersza X.  
# The outputs of the XOR for every example in X  
Y = np.array([0, 1, 1, 0])  
  
# Set the hyperparameter  
n\_x = 2 #No. of neurons in first layer liczbę neuronów w warstwie wejściowej  
n\_h = 2 #No. of neurons in hidden layer liczbę neuronów w warstwie ukrytej  
n\_y = 1 #No. of neurons in output layer liczbę neuronów w warstwie wyjściowej  
  
hiperparametry, w tym liczbę neuronów w warstwie ukrytej, funkcję aktywacji dla warstwy ukrytej, maksymalną liczbę iteracji dla solvera, algorytm solvera (algorytm quasi-Newtonowski, który jest często używany do optymalizacji nieliniowych funkcji celu) i parametr kary L2.  
model = MLPClassifier(hidden\_layer\_sizes=(n\_h,), activation='logistic', max\_iter=1000, solver='lbfgs', alpha=0.001)  
Następuje uczenie modelu MLPClassifier, korzystając z macierzy wejściowej X i wektora wyjściowego Y.  
model.fit(X, Y)  
Dokonujemy predykcji za pomocą nauczonych modeli na macierzy wejściowej X.  
predictions = model.predict(X)  
print(predictions)

W przypadku programu z użyciem biblioteki sklearn, jej główne zalety to:

* Szybkość implementacji modelu sieci neuronowej, która zwykle wymagałaby więcej kodu
* Możliwość łatwej zmiany hiperparametrów sieci, takich jak liczba neuronów w ukrytej warstwie, funkcja aktywacji i wiele innych
* Automatyczna obsługa różnych solverów i metod optymalizacji (w tym tutaj "lbfgs"), co pozwala na łatwe testowanie różnych opcji.

W przypadku programu z implementacją sieci neuronowej bez użycia biblioteki, jej główne zalety to:

* Dostępność pełnej kontroli nad każdym elementem sieci, włącznie z inicjalizacją, warstwami i propagacją
* Pełna elastyczność i możliwość modyfikacji każdego elementu sieci, co pozwala na łatwe dostosowywanie jej do różnych zadań i danych.
* Możliwość zrozumienia i nauczenia się algorytmu krok po kroku, co może pomóc w zrozumieniu uczenia maszynowego w ogóle.

**NEURALNET**  
Wczytanie biblioteki neuralnet do R.  
library(neuralnet)

# przygotowanie danych treningowych  
input <- data.frame(x1 = c(0, 0, 1, 1), x2 = c(0, 1, 0, 1))  
output <- data.frame(y = c(0, 1, 1, 0))

# utworzenie i trenowanie sieci neuronowej  
Określono wzór y ~ x1 + x2, co oznacza, że kolumny wejściowe x1 i x2 są wykorzystywane do prognozowania kolumny wyjściowej y. Funkcja cbind() łączy kolumny input i output, a następnie te dane zostają przekazane jako argument data. Określono również, że sieć neuronowa powinna zawierać 3 ukryte warstwy neuronów, a funkcja aktywacji powinna być funkcją logistyczną.  
nn <- neuralnet(y ~ x1 + x2, data = cbind(input, output), hidden = 3, act.fct = "logistic")  
Wygenerowanie wykresu przedstawiającego strukturę sieci neuronowej.  
plot(nn)

# testowanie sieci neuronowej  
Przygotowanie danych testowych za pomocą danych wejściowych test\_input zawierających te same kombinacje binarne, co w przypadku danych treningowych. Funkcja compute() używa wcześniej wytrenowanej sieci neuronowej nn i danych testowych test\_input do wygenerowania prognozowanych wartości dla kolumny wyjściowej y. Wynik tego procesu zostaje przypisany do predicted\_output.  
test\_input <- data.frame(x1 = c(0, 0, 1, 1), x2 = c(0, 1, 0, 1))  
predicted\_output <- compute(nn, test\_input)  
Zaokrąglenie prognozowanych wartości kolumny wyjściowej y do najbliższych liczb całkowitych. Wynik działania sieci neuronowej jest zwykle wartością ciągłą, a zaokrąglenie umożliwia przypisanie prognozowanej wartości do jednej z dwóch kategorii (0 lub 1) zgodnie z ustalonym progiem.  
round(predicted\_output$net.result)

Biblioteka neuralnet:

* Zaleta: Biblioteka neuralnet jest prostsza w użyciu i pozwala na łatwe wizualizowanie sieci neuronowej. Ponadto, neuralnet oferuje dokładniejsze informacje na temat uczenia się sieci neuronowej, takie jak wartości wag, wartości błędów i biasów.
* Wada: Biblioteka neuralnet jest mniej elastyczna niż sklearn i oferuje mniejszą liczbę hiperparametrów do dostosowania.

**ZGADYWANIE CYFR**

**SKLEARN**

from sklearn.neural\_network import MLPClassifier  
import numpy as np.  
  
Funkcja toArray przyjmuje ciąg 49 znaków (7x7) reprezentujący ręcznie napisaną cyfrę (z '#' reprezentującym czarny piksel i ' ' reprezentującym biały piksel) i zwraca tablicę numpy o kształcie (49,), gdzie każdy element to 0 lub 1 w zależności od tego, czy odpowiadający piksel w ciągu wejściowym jest biały czy czarny.  
def toArray(string):

if len(string) != 7 \* 7:  
 raise ValueError('string in wrong size')  
 return np.array([toNumber(char) for char in string])

Funkcja toNumber to prosta funkcja pomocnicza, która zwraca 1, jeśli podany znak to '#' i 0 w przeciwnym razie.  
def toNumber(character):

return 1 if character == '#' else 0

Definiowanie tablic dla cyfr od 0 do 9, używając funkcji toArray do konwersji ciągów ręcznie pisanych cyfr na tablice numpy.  
zero = toArray( '#######' + '# #' + '# #' + '# #' + '# #' + '# #' + '#######')

…

nine = toArray( '#######' + '# #' + '# #' + '###### ' + ' # ' + ' # ' + ' # ')

Tworzenie tablicy cech wejściowych X poprzez pionowe ustawienie tablic cyfr, a także tworzenie tablicy celów wyjściowych y z odpowiadającymi etykietami cyfr.  
X = np.array([zero, one, two, three, four, five, six, seven, eight, nine])

y = np.array(['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine'])  
  
Tworzenie klasyfikatora MLP z jedną warstwą ukrytą o 100 neuronach, funkcją aktywacji logistyczną i solwerem 'lbfgs'. Maksymalna liczba iteracji jest ustawiona na 500.  
net = MLPClassifier(hidden\_layer\_sizes=(100,), activation='logistic', solver='lbfgs', max\_iter=500)  
  
Dopasowanie klasyfikatora do danych wejściowych i wyjściowych  
net.fit(X, y)  
  
Testowanie klasyfikatora na trzech dodatkowych ręcznie napisanych cyfrach za pomocą net.predict([toArray(ciag\_wejsciowy)]), gdzie ciag\_wejsciowy to ciąg 49 znaków reprezentujący ręcznie napisaną cyfrę.  
result = net.predict([toArray( '#######' + '# #' + '# #' + '## ###' + '# #' + '# #' + '#######' )])

print(result)

Zalety korzystania z biblioteki sklearn to:

1. Skuteczność i łatwość użycia - biblioteka zawiera wiele predefiniowanych algorytmów i metod, co ułatwia pracę.
2. Efektywność obliczeniowa - biblioteka jest zoptymalizowana pod kątem szybkości działania i wydajności.
3. Wsparcie dla wielu typów modeli i zastosowań - biblioteka zawiera szeroką gamę modeli i narzędzi, które można stosować w różnych zadaniach uczenia maszynowego.
4. Dostępność dokumentacji i narzędzi - biblioteka posiada dobrze udokumentowane funkcje oraz narzędzia, takie jak metryki do oceny jakości modeli.
5. Rozszerzalność - biblioteka jest często używana przez społeczność Pythona i posiada wiele dostępnych rozszerzeń.

Oto kilka zalet korzystania z biblioteki neuralnet w języku R:

1. Prosta w użyciu: Neuralnet jest łatwy w użyciu nawet dla początkujących użytkowników R, co czyni go popularnym narzędziem do eksploracji sieci neuronowych.
2. Szeroka funkcjonalność: Neuralnet oferuje wiele opcji i algorytmów do wyboru, takich jak algorytm propagacji wstecznej, algorytm Levenberga-Marquardta, regularyzacja L1 i L2, modele wielowarstwowe i wiele innych.
3. Dobra wydajność: Neuralnet działa szybko, nawet dla dużych zbiorów danych, dzięki wykorzystaniu wielu procesorów i wątków.
4. Obsługa zarówno danych kategorycznych, jak i ciągłych: Neuralnet umożliwia obsługę zarówno danych kategorycznych, jak i ciągłych, co czyni go bardziej uniwersalnym i elastycznym.
5. Możliwość tworzenia modeli złożonych: Neuralnet umożliwia tworzenie skomplikowanych modeli z wieloma warstwami i nieliniowymi funkcjami aktywacji, co pozwala na modelowanie złożonych zależności między zmiennymi.
6. Dobrze udokumentowana: Neuralnet jest dobrze udokumentowany, co ułatwia użytkownikom naukę i rozwiązywanie problemów.
7. Dobra integracja z innymi pakietami R: Neuralnet dobrze integruje się z innymi popularnymi pakietami R, takimi jak ggplot2, caret i dplyr, co ułatwia analizę i wizualizację wyników.