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, zaczniemy od implementacji prostej sieci z 2 warstwami. Aby to zrobić, potrzebujemy również prostego zestawu danych, więc w naszym przykładzie użyjemy zestawu danych XOR.

XOR

Czyli takiej funkcji, która mówi nam kiedy dokładnie jedno z wejść jest prawdziwe. Jak możemy zauważyć, nie da się wyznaczyć linii prostej oddzielającej przestrzeń danych, z których jedna odpowiada sygnałowi 1 na wyjściu, a druga 0 na wyjściu. Nie da się zatem zrealizować funkcji logicznej XOR za pomocą jednego perceptronu.  
Rozwiązaniem tego problemu jest rozszerzenie sieci neuronowej o jeden neuron w warstwie aby stworzyć sieć neuronów o wagach realizujących następujący podział przestrzeni:

Czyli dla 1-go neuronu: (u – prosta realizująca separowalność liniową)  
u1 = W11x1 + W12x 2 + b1 > 0  
u1 = W21x1 + W22x 2 + b1 < 0

a dla 2-go neuronu:  
u2 = W21x1 + W22x 2 + b2 > 0  
u2 = W21x1 + W22x 2 + b2 < 0

I teraz przestrzeń, która na wykresie przedstawia się jako część wspólna zbiorów u1>0 i u2>0 odpowiada sygnałowi 1 na wyjściu, a przestrzeń poza tym obszarem 0 na wyjściu. Na rysunku 2 można zobaczyć pełną strukturę sieci neuronowej mogącej realizować funkcję XOR.

Najpierw omówię schemat uczenia sieci. Naszą sieć będziemy uczyć metodami nadzorowanymi. Wykorzystamy do tego Algorytm propagacji wstecznej, który składa się z dwóch naprzemiennych faz:  
1. Fazy propagacji sygnału od wejść do wyjścia  
2. Fazy wstecznej propagacji błędu od wyjścia y w kierunku wejść sieci.

Na początku 1. fazy pobudzamy sygnałami wejściowymi A i B neurony w 1. warstwie i obliczamy ich wartości wyjściowe, poprzez ich połączenia z neuronami w warstwie ukrytej podajemy te wartości jako sygnały wejściowe dla neuronu w 2. warstwie. Neuron w warstwie wyjściowej pobudzamy sygnałami wyjściowymi warstwy drugiej i na tej podstawie wyznaczamy wyjście, czyli odpowiedź sieci na sygnał wejściowy. Wartość tą porównujemy z wzorcową i wyznaczymy błąd. Chodzi nam o znalezienie minimum błędu średniokwadratowego. Korzystając z metod gradientowych propagujemy błąd wstecz przez sieć i dokonujemy korekty wartości wag połączeń synaptycznych. Przechodząc wstecz z warstwy wyjściowej do ukrytej wartość błędu jest ważona zgodnie z wagą połączenia między neuronami i sumowana w tych neuronach. Na koniec dokonywana jest korekta wartości wag sieci.

Obraz zawierający Czcionka, biały, tekst, pismo odręczne

Opis wygenerowany automatycznieObraz zawierający Czcionka, tekst, biały, pismo odręczne

Opis wygenerowany automatycznie

Za chwilkę przejdziemy do implementacji.

Jeśli chodzi o sposób obliczania wartości wyjściowych dla neuronów to potrzebujemy oczywiście funkcji aktywacji. W sieciach neuronowych, funkcje aktywacji są wykorzystywane w celu wprowadzenia nieliniowości do modelu.

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 ze względu na jego symetryczny kształt, który umożliwia jednoczesne przetwarzanie informacji zarówno w kierunku dodatnim, jak i ujemnym

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.   
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 zainicjalizować nasze parametry. Funkcja initialize\_parameters(n\_x, n\_h, n\_y) przyjmuje jako dane wejściowe liczbę neuronów 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)) (jedna kolumna)  
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))

tworzymy 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  
def forward\_prop(X, parameters):

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

Funkcja aktywacji użyta w pierwszej warstwie to tangens hiperboliczny, a w drugiej warstwie sigmoid.  
  
 Z1 = np.dot(W1, X) + b1 iloczyn skalarny  
 A1 = np.tanh(Z1)  
 Z2 = np.dot(W2, A1) + b2  
 A2 = sigmoid(Z2)

słownik cache [kesz] 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 jednostkowy wymiar, gdy cost jest jednoelementową tablicą 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):

Na początku wartości aktywacji dla ukrytej warstwy A1 i końcowej warstwy A2 są pobierane ze słownika cache

A1 = cache["A1"]

A2 = cache["A2"]

Pobieramy macierz wag ze słownika parameters

W2 = parameters["W2"]

Obliczamy różnicę między przewidywaniami modelu A2 a rzeczywistymi etykietami Y.

dZ2 = A2 – Y

Obliczamy gradienty dla wag W2 i biasów b2

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

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

Obliczamy gradienty dla wag W1 i biasów b1

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:

Na podstawie obliczonych gradientów funkcji kosztu, Funkcja update\_parameters aktualizuje wagi i biasy

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()

Funkcja model() przyjmuje jako dane wejściowe macierz cech X, macierz etykiet Y, liczbę neuronów w każdej warstwie 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 wskazana liczbę 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  
 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 model() 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.  
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 liczbę, a nie tablicę jednowymiarową  
 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.  
  
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 = 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 #miara uczenia

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)

Z każdą kolejna iteracją koszt się zmiejsza.  
Testujemy dla (1,1)

# Test 2X1 vector to calculate the XOR of its elements.   
# Try (0, 0), (0, 1), (1, 0), (1, 1)  
Tworzymy wektor testowy X\_test z elementami (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)  
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))

Czy są jakieś pytania? Teraz przejdziemy do implementacji tego modelu z użyciem biblioteki sklearn.

**SKLEARN**Biblioteka scikit-learn oferuje wiele gotowych algorytmów uczenia maszynowego.  
Multi-layer Perceptron classifier implementuje sieć neuronową o wielu warstwach perceptronów wielowarstwowych (MLP) i jest łatwy w użyciu jak się zaraz przekonamy.  
  
import numpy as np.  
Importuje klasę MLPClassifier z modułu neural\_network biblioteki scikit-learn.  
from sklearn.neural\_network import MLPClassifier  
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])  
  
n\_h = 2 #No. of neurons in hidden layer liczbę neuronów w warstwie ukrytej   
  
MLPClassifier posiada wiele parametrów, które można dostosować do potrzeb, jak np.liczbę neuronów w warstwie ukrytej, funkcję aktywacji dla warstwy ukrytej (logistic to sigmoid), maksymalną liczbę iteracji dla solvera, algorytm solvera (Korzystamy z algorytmu quasi-Newtonowski, który jest często używany do optymalizacji gradientowej 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)

Jakieś pytania? Zajmiemy się teraz implementacją tego problemu w języku R z użyciem biblioteki neuralnet  
 **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  
Określony wzór y ~ x1 + x2 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. Argument hidden określa, że sieć powinna zawierać 2 ukryte warstwy neuronów, a act.fct, że funkcja aktywacji to sigmoid.  
nn <- neuralnet(y ~ x1 + x2, data = cbind(input, output), hidden = 3, act.fct = "logistic")  
Generujemy wykres przedstawiający strukturę sieci neuronowej.  
plot(nn)

# testowanie sieci neuronowej  
Przygotowanie danych testowych test\_input zawierających te same kombinacje binarne, co w przypadku danych treningowych.   
test\_input <- data.frame(x1 = c(0, 0, 1, 1), x2 = c(0, 1, 0, 1))  
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.  
predicted\_output <- compute(nn, test\_input)  
Zaokrąglamy prognozowane wartości kolumny wyjściowej y do najbliższych liczb całkowitych. Wynik działania sieci neuronowej jest zwykle wartością ciągłą, a zaokrąglenie umożliwia przypisanie prognozowanej wartości do 0 lub 1  
round(predicted\_output$net.result)

Zalety każdej z przedstawionych metod implementacji.

**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.