POLITECHNIKA WARSZAWSKA

Wydział Elektroniki i Technik Informacyjnych

Sieci Neuronowe w Zastosowaniach Biomedycznych

31.Przewidywanie skuteczność immunoterapii w leczeniu brodawek za pomocą sieci uczonej z nauczycielem (MLP)

Etap 2: ”Implementacja sieci neuronowej”

Mateusz Małkowski 321358

Spis treści

[Wstęp 3](#_Toc167226971)

[Implementacja sieci neuronowej 3](#_Toc167226972)

[Warstwa wejściowa 3](#_Toc167226973)

[Warstwa ukryta 4](#_Toc167226974)

[Warstwa wyjściowa 4](#_Toc167226975)

[Funkcja straty 5](#_Toc167226976)

[Metoda propagacji wstecznej 5](#_Toc167226977)

[Wyniki 6](#_Toc167226978)

[Wnioski 8](#_Toc167226979)

# Wstęp

Celem projektu było zaprojektowanie i zaimplementowanie sztucznej sieci neuronowej z nauczycielem do przewidywanie skuteczności immunoterapii w leczeni brodawek.

Dane wykorzystane do stworzenia sieci zostały opisane i przygotowane w poprzednim etapie projektu.

# Implementacja sieci neuronowej

Zgodnie z założeniami z poprzedniego etapu stworzono sieć z jedną warstwą ukrytą i po jednej warstwie wyjścia i wejścia.

Obraz zawierający szkic, rysowanie, diagram, linia

Opis wygenerowany automatycznie

Rysunek 1:Schemat sieci neuronowej

## Warstwa wejściowa

Składa się z 7 neuronów reprezentujących nasze dane wejściowe, czyli: Sex, Age, Time elapsed before treatment, Number of warts, types of wart, Surface area of the warts i induration diameter of initial test. Wszystkie te dane zostały wcześniej przygotowane i znormalizowane do przedziału [0,1], więc nie trzeba ich modyfikować w tej warstwie.

## Warstwa ukryta

Podczas tworzenia warstwy ukrytej inicjalizujemy jej wartości początkowe sposobem Kaiminga He, czyli ustawiamy biasy na 0 i wagi jako liczbę losową z rozkładem prawdopodobieństwa Gaussa ze średnią 0 i odchyleniem standardowym

(1)

Metoda Kaiminga He jest jednym z podejść do zapewnienia odpowiednich wartości początkowych wag, szczególnie w przypadku funkcji aktywacji ReLU, która jest często stosowana ze względu na swoją skuteczność i prostotę obliczeniową. Powinna pomagać z problemem zanikających lub eksplodujących gradientów.

Do stworzenia sieci neuronowej jest potrzebna funkcja aktywacji, ponieważ bez niej wagi rosły by liniowo.

Używamy funkcji aktywacji ReLU (Rectified Linear Unit). Jest to funkcja nieliniowa opisana wzorem:

Gdy dane wejściowe nie są większe od zera funkcja ReLU nie wprowadza żadnych zmian (jedynie przepuszcza pozytywne wartości), co prowadzi do skutecznej eliminacji problemu zanikającego gradientu.

Jest ona prosta obliczeniowo, co przyspiesza proces uczenia się sieci neuronowej i dla większości problemów działa bardzo dobrze.

## Warstwa wyjściowa

W warstwie wyjściowej stosujemy inicjalizacje Xawiera. Metoda ta zakłada inicjalizowanie wag w warstwach sieci neuronowej zgodnie z rozkładem normalnym o średniej zero i wariancji:

gdzie ​ to liczba neuronów w warstwie poprzedniej, a ​ to liczba neuronów w warstwie obecnej. Następnie wagi są losowane z tego rozkładu.

Istotą tej metody jest odpowiednie skalowanie wag początkowych w taki sposób, aby sygnał wejściowy do warstwy były mniej więcej tej samej wielkości, co sygnał wyjściowy. Ma to na celu zapobieżenie problemowi zanikającego lub eksplodującego gradientu podczas propagacji wstecznej w procesie uczenia się.

Funkcją aktywacji warstwy wyjściowej jest sigmoid, który opisany jest wzorem:

Funkcja sigmoidalna przekształca dowolną wartość rzeczywistą na zakres od 0 do 1, co pozwala nam na określenie prawdopodobieństwa

Większość danych wejściowych była dla pacjentów dla których leczenie było sukcesem, dlatego sieć ma więcej wartości bliższych 1 niż 0. By sobie z tym poradzić korzystamy nie dzielimy danych od prawdopodobieństwa 50%, tylko sami decydujemy jakie ustawiamy odcięcie przy inicjalizacji sieci.

## Funkcja straty

Jako funkcje straty używamy binarnej entropii krzyżowe (ang. binary cross entropy). Mierzy różnicę między rzeczywistymi etykietami klas a prognozowanymi przez model prawdopodobieństwami należenia do klasy pozytywnej i opisana jest wzorem:

Gdzie, to rzeczywista etykieta klasy (0 lub 1), a to prognozowane prawdopodobieństwo przynależności do klasy pozytywnej przez model.

Dobrze nadaje się do problemów klasyfikacji binarnej i poprzez uwzględnienie prawdopodobieństw przynależności do klasy, funkcja binarnej entropii krzyżowej pozwala na ocenę pewności predykcji modelu. Wartość funkcji straty będzie większa, gdy model będzie bardziej pewny swoich przewidywań i popełni błąd. Jest też łatwa obliczeniowo co przyśpiesza proces optymalizacji modelu.

Dla dobrze douczonego modelu jej wartości dla danych trenujących i testowych powinien być podobny i mały.

## Metoda propagacji wstecznej

Metoda propagacji wstecznej (ang. Backpropagation) jest fundamentalnym algorytmem używanym do uczenia się w sieciach neuronowych. Służy do obliczania gradientów funkcji straty względem wag sieci neuronowej, co umożliwia aktualizację wag w kierunku minimalizacji funkcji straty.

Jej kroki to:

1. Przekazanie danych wejściowych: Na początku, dane wejściowe są przekazywane przez sieć neuronową, a każda warstwa oblicza swoje wyjście na podstawie aktualnych wag i funkcji aktywacji.
2. Obliczenie funkcji straty: Następnie, obliczana jest funkcja straty, która mierzy różnicę między rzeczywistymi etykietami a prognozowanymi przez sieć neuronową wartościami. Funkcja ta jest używana jako miara błędu sieci.
3. Propagacja wsteczna: W tej fazie, algorytm propagacji wstecznej oblicza gradient funkcji straty względem każdej wagi w sieci neuronowej, zaczynając od warstwy wyjściowej i przechodząc wstecz przez sieć.
4. Aktualizacja wag: Na podstawie obliczonych gradientów, algorytm stosuje regułę aktualizacji wag (np. metodę gradientu prostego) w celu dostosowania wartości wag w kierunku minimalizacji funkcji straty.
5. Powtarzanie procesu: Proces propagacji wstecznej i aktualizacji wag jest powtarzany aż do momentu, gdy sieć neuronowa osiągnie akceptowalny poziom dokładności lub zbiegnie do minimum funkcji straty.

Główne zalety metody propagacji wstecznej to jej skuteczność w uczeniu się z danych, zdolność do uczenia się na dużej skali i możliwość stosowania w różnych architekturach sieci neuronowych.

# Wyniki

Tabela 1: Wyniki dla 1000 epok i współczynnika uczenia 0.1

|  |  |  |
| --- | --- | --- |
| Dla 1000 epok i współczynnika uczenia 0.1 | | |
|  | Dane trenujące | Dane testowe |
| Czułość | 0,92 | 0,57 |
| Specyficzność | 0,76 | 0,83 |
| funkcja straty | 0,313 | 0,528 |

Obraz zawierający tekst, linia, Wykres, diagram

Opis wygenerowany automatycznie

Rysunek 2:Wartość funkcji straty do ilości epokObraz zawierający tekst, diagram, linia, Plan

Opis wygenerowany automatycznie

Rysunek 3:Czułość i specyficzność do wartości odcięcia

Obraz zawierający tekst, diagram, linia, Czcionka

Opis wygenerowany automatycznie

Rysunek 4:Czułość i specyficzność do ilości epok przez które rozwijał się algorytm

# 

# Wnioski

W typ projekcie stworzono sieć z jedna ukrytą warstwą, ale kod został przygotowany w taki sposób, by można było przeanalizować wynik w innej architekturze sieci neuronowej w kolejnym etapie.

Wyniki z prostej sieci jednowarstwowej były mało zadowalające. Z tabeli 1 można zauważyć, że czułość dla danych testowych była mało zadowalająca. Wartość specyficzności była zdecydowanie lepsza. Jest to spowodowane mniejszą ilością wartości negatywnych w zbiorze danych. Dla danych testowych i trenujących wartość funkcji straty była całkiem mała, ale jednak większa dla trenujących, co pokazuje nam lekkie przetrenowanie sieci.

Mimo to, dla 1000 iteracji i współczynnika uczenia 0.1 osiągaliśmy najlepsze wyniki.

Z rysunku 1 można wywnioskować, że najważniejsze dla algorytmu jest pierwsze 500 epok podczas których wartość funkcji straty ostro malała. Dla kolejnych epok jest spadek nie był tak gwałtowny.

Na rysunku 3 widać, że czułości i specyficzności przy odcięciu 0,7 były bardziej miarodajne niż te dla odcięcia 0,5.

Z rysunku 4 widać, że po 2000 epok sieć była przetrenowana i specyficzność i czułość dla danych ostro spadała.

Duży wpływ na wyniki miał sposób inicjalizacji wag i biasów. W kolejnym etapie projektu trzeba będzie sprawdzić jak sieć będzie się zachowywała dla innych metod.

# Listing programów

import numpy as np  
import pandas as pd  
from matplotlib import pyplot as plt  
import contextlib  
import io  
  
def load\_data(filename\_X\_train:str, filename\_X\_test:str, filename\_Y\_train:str, filename\_Y\_test:str):  
 *"""  
 Load data from CSV files.  
  
 Args:  
 filename\_X\_train (str): File path for training data features.  
 filename\_X\_test (str): File path for test data features.  
 filename\_Y\_train (str): File path for training data labels.  
 filename\_Y\_test (str): File path for test data labels.  
  
 Returns:  
 tuple: Tuple containing X\_train, X\_test, Y\_train, Y\_test.  
 """* X\_train = pd.read\_csv(filename\_X\_train)  
 X\_test = pd.read\_csv(filename\_X\_test)  
 Y\_train = pd.read\_csv(filename\_Y\_train)  
 Y\_test = pd.read\_csv(filename\_Y\_test)  
 X\_train = np.array(X\_train).T  
 X\_test = np.array(X\_test).T  
 Y\_train = np.array(Y\_train).T  
 Y\_test = np.array(Y\_test).T  
 return X\_train, X\_test, Y\_train, Y\_test  
  
def relu(Z: np.array):  
 *"""ReLU activation function."""* return np.maximum(Z, 0)  
  
def sigmoid(Z: np.array):  
 *"""Sigmoid activation function."""* return 1/(1 + np.exp(-Z))  
  
def relu\_deriv(Z: np.array):  
 *"""Derivative of ReLU activation function."""* return Z > 0  
  
class Hidden\_Layer:  
 *"""Hidden layer of the neural network."""* def \_\_init\_\_(self, n\_inputs: int, n\_neurons: int, learning\_rate: float, bias\_offset = 0):  
 *"""  
 Initialize hidden layer parameters.  
  
 Args:  
 n\_inputs (int): Number of input features.  
 n\_neurons (int): Number of neurons in the layer.  
 learning\_rate (float): Learning rate for training.  
 bias\_offset (float, optional): Offset for bias initialization. Defaults to 0.5.  
 """* self.weights = np.random.randn(n\_neurons, n\_inputs) \* np.sqrt(2 / n\_inputs)  
 self.bias = np.zeros((n\_neurons, 1)) - bias\_offset  
 self.learning\_rate = learning\_rate  
 self.activation\_function = relu  
 self.activation\_function\_derivative = relu\_deriv  
  
 def forward\_propagation(self, inputs: np.array):  
 *"""  
 Perform forward propagation through the layer.  
  
 Args:  
 inputs (np.array): Input data.  
  
 """* self.Z = np.dot(self.weights, inputs) + self.bias  
 self.output = self.activation\_function(self.Z)  
  
 def backward\_propagation(self, X: np.ndarray, W\_next: np.ndarray, dZ\_next: np.ndarray):  
 *"""  
 Perform backward propagation through the layer.  
  
 Args:  
 X (np.ndarray): Input data.  
 W\_next (np.ndarray): Weights of the next layer.  
 dZ\_next (np.ndarray): Gradient from the next layer.  
  
 Returns:  
 tuple: Updated weights and gradient for the previous layer.  
 """* m = X.shape[1]  
 W = self.weights  
  
 dZ = np.dot(W\_next.T, dZ\_next) \* self.activation\_function\_derivative(self.Z)  
 dW = np.dot(dZ, X.T)/m  
 db = np.sum(dZ, axis=1, keepdims=True)/m  
  
 self.weights -= self.learning\_rate \* dW  
 self.bias -= self.learning\_rate \* db  
 return W, dZ  
  
  
class Output\_layer(Hidden\_Layer):  
 *"""Output layer of the neural network."""* def \_\_init\_\_(self, n\_inputs: int, n\_neurons: int, learning\_rate: float, bias\_offset = 0):  
 *"""Initialize output layer parameters.  
  
 Args:  
 n\_inputs (int): Number of input features.  
 n\_neurons (int): Number of neurons in the layer.  
 learning\_rate (float): Learning rate for training.  
 bias\_offset (float, optional): Offset for bias initialization. Defaults to 0.  
 """* super().\_\_init\_\_(n\_inputs, n\_neurons, learning\_rate, bias\_offset)  
 self.activation\_function = sigmoid  
 limit = limit = np.sqrt(2 / (n\_inputs + n\_neurons))  
 self.weights = np.random.uniform(-limit, limit, (n\_neurons, n\_inputs))  
  
 def backward\_propagation(self, Y: np.ndarray, X: np.ndarray):  
 *"""Perform backward propagation through the output layer.  
  
 Args:  
 Y (np.ndarray): True labels.  
 X (np.ndarray): Input data.  
  
 Returns:  
 tuple: Tuple containing dW and dZ.  
 """* m = Y.shape[1]  
 W = self.weights  
  
 dZ = self.output - Y  
 dW = np.dot(dZ, X.T) / m  
 db = np.sum(dZ, axis=1, keepdims=True) / m  
  
 self.weights -= self.learning\_rate \* dW  
 self.bias -= self.learning\_rate \* db  
  
 return W, dZ  
  
  
class Neural\_Network:  
 *"""Neural Network class."""* def \_\_init\_\_(self, hidden\_layers: list, output\_layer: Output\_layer, threshold = 0.7):  
 *"""  
 Initialize the neural network.  
  
 Args:  
 hidden\_layers (list): List of hidden layers.  
 output\_layer (Output\_layer): Output layer of the network.  
 threshold (float, optional): Threshold for binary classification. Defaults to 0.7.  
 """* self.hidden\_layers = hidden\_layers  
 self.output\_layer = output\_layer  
 self.threshold = threshold  
  
 def cross\_entropy(self, Y\_true: np.array, Y\_prediction: np.array):  
 *"""  
 Calculate the binary cross-entropy loss.  
  
 Args:  
 Y\_true (np.array): True labels.  
 Y\_prediction (np.array): Predicted probabilities.  
  
 Returns:  
 float: Cross-entropy loss.  
 """* Y\_prediction = np.clip(Y\_prediction, 1e-15, 1 - 1e-15)  
 return -np.mean(Y\_true \* np.log(Y\_prediction) + (1 - Y\_true) \* np.log(1 - Y\_prediction))  
  
 def calculate\_sensitivity\_specificity(self, Y\_true: np.array, y\_prediction: np.array):  
 *"""  
 Calculate sensitivity and specificity.  
  
 Args:  
 Y\_true (np.array): True labels.  
 y\_prediction (np.array): Predicted labels.  
  
 Returns:  
 tuple: Tuple containing sensitivity and specificity.  
 """* TP = np.sum((Y\_true == 1) & (y\_prediction == 1))  
 TN = np.sum((Y\_true == 0) & (y\_prediction == 0))  
 FP = np.sum((Y\_true == 0) & (y\_prediction == 1))  
 FN = np.sum((Y\_true == 1) & (y\_prediction == 0))  
  
 sensitivity = TP / (TP + FN) if (TP + FN) > 0 else 0  
 specificity = TN / (TN + FP) if (TN + FP) > 0 else 0  
  
 return sensitivity, specificity  
 def train(self,X: np.array, Y: np.array, epochs: int):  
 *"""  
 Train the neural network.  
  
 Args:  
 X (np.array): Input data.  
 Y (np.array): True labels.  
 epochs (int): Number of training epochs.  
 """* input = list()  
 input.append(X)  
 self.loss\_history = list()  
  
 for epoch in range(epochs):  
 for i in range(len(self.hidden\_layers)):  
 self.hidden\_layers[i].forward\_propagation(input[i])  
 input.append(self.hidden\_layers[i].output)  
 self.output\_layer.forward\_propagation(input[-1])  
  
 loss = Neural\_Network.cross\_entropy(self,Y, self.output\_layer.output)  
 self.loss\_history.append(loss)  
  
 W, dZ = self.output\_layer.backward\_propagation(Y, input[-1])  
 for i in reversed(range(len(self.hidden\_layers))):  
 W, dZ = self.hidden\_layers[i].backward\_propagation(input[i], W, dZ)  
  
 if epoch % 100 == 0:  
 print('Epoch ', epoch, ', Loss: ', loss)  
  
 def predict(self, X: np.array):  
 *"""  
 Make predictions using the trained model.  
  
 Args:  
 X (np.array): Input data.  
  
 Returns:  
 np.array: Predicted labels.  
 """* input = list()  
 input.append(X)  
  
 for i in range(len(self.hidden\_layers)):  
 self.hidden\_layers[i].forward\_propagation(input[i])  
 input.append(self.hidden\_layers[i].output)  
 self.output\_layer.forward\_propagation(input[-1])  
  
 return self.output\_layer.output  
  
 def plot\_loss(self):  
 *"""Plot the loss function over epochs."""* plt.plot(self.loss\_history, linestyle='-')  
 plt.title('Loss function graph')  
 plt.xlabel('Epoch')  
 plt.ylabel('Loss function value')  
 plt.grid(True)  
  
  
def test\_network(X\_train: np.array, X\_test: np.array, Y\_train: np.array, Y\_test: np.array):  
 *"""  
 Test the neural network on training and test data.  
  
 Args:  
 X\_train (np.array): Training data features.  
 X\_test (np.array): Test data features.  
 Y\_train (np.array): Training data labels.  
 Y\_test (np.array): Test data labels.  
 """* learning\_rate = 0.01  
 epochs = 5000  
  
 layer1 = Hidden\_Layer(7, 8, learning\_rate, bias\_offset=0)  
 output\_layer = Output\_layer(8, 1, learning\_rate)  
  
 network = Neural\_Network([layer1], output\_layer, threshold=0.7)  
  
 network.train(X\_train, Y\_train, epochs)  
  
 predictions\_train = network.predict(X\_train)  
 predictions\_test = network.predict(X\_test)  
  
 loss\_train = network.cross\_entropy(Y\_train, predictions\_train)  
 loss\_test = network.cross\_entropy(Y\_test, predictions\_test)  
  
 predictions\_train = (predictions\_train[0,:]> network.threshold).astype(int)  
 predictions\_test = (predictions\_test[0,:]> network.threshold).astype(int)  
  
 sensitivity\_train, specificity\_train = network.calculate\_sensitivity\_specificity(Y\_train, predictions\_train)  
 sensitivity\_test, specificity\_test = network.calculate\_sensitivity\_specificity(Y\_test, predictions\_test)  
  
 print('\nTrain data:')  
 print('Predictions: ', predictions\_train)  
 print('True values: ', Y\_train.astype(int)[0, :])  
 print("Sensitivity: ", sensitivity\_train)  
 print("Specificity: ", specificity\_train)  
 print("loss function value: ", loss\_train)  
 print('\nTest data:')  
 print('Predictions: ', predictions\_test)  
 print('True values: ', Y\_test.astype(int)[0, :])  
 print("Sensitivity: ", sensitivity\_test)  
 print("Specificity: ", specificity\_test)  
 print("loss function value: ", loss\_test)  
 network.plot\_loss()  
 plot\_specificity\_sensitivity(X\_train, X\_test, Y\_train, Y\_test, network)  
def plot\_specificity\_sensitivity(X\_train: np.array, X\_test: np.array, Y\_train: np.array, Y\_test: np.array, network: Neural\_Network):  
 *"""  
 Plot sensitivity and specificity by epoch and threshold.  
  
 Args:  
 X\_train (np.array): Training data features.  
 X\_test (np.array): Test data features.  
 Y\_train (np.array): Training data labels.  
 Y\_test (np.array): Test data labels.  
 network (Neural\_Network): Trained neural network.  
 """* specificity\_history\_train = list()  
 sensitivity\_history\_train = list()  
 specificity\_history\_test = list()  
 sensitivity\_history\_test = list()  
 specificity\_history\_train\_th = list()  
 sensitivity\_history\_train\_th = list()  
 specificity\_history\_test\_th = list()  
 sensitivity\_history\_test\_th = list()  
  
 for epoch in range(100, 5000, 100):  
 f = io.StringIO()  
 # Używamy contextlib.redirect\_stdout do przekierowania stdout  
 with contextlib.redirect\_stdout(f):  
 network.train(X\_train, Y\_train, epoch)  
  
 predictions\_train = network.predict(X\_train)  
 predictions\_test = network.predict(X\_test)  
 predictions\_train = (predictions\_train[0, :] > network.threshold).astype(int)  
 predictions\_test = (predictions\_test[0, :] > network.threshold).astype(int)  
  
 sensitivity\_train, specificity\_train = network.calculate\_sensitivity\_specificity(Y\_train, predictions\_train)  
 sensitivity\_history\_train.append(sensitivity\_train)  
 specificity\_history\_train.append(specificity\_train)  
  
 sensitivity\_test, specificity\_test = network.calculate\_sensitivity\_specificity(Y\_test, predictions\_test)  
 specificity\_history\_test.append(specificity\_test)  
 sensitivity\_history\_test.append(sensitivity\_test)  
  
 epochs = range(100, 5000, 100)  
 plt.figure()  
 plt.plot(epochs, specificity\_history\_train, label='Specificity\_train')  
 plt.plot(epochs, sensitivity\_history\_train, label='Sensitivity\_train')  
 plt.plot(epochs, specificity\_history\_test, label='Specificity\_test')  
 plt.plot(epochs, sensitivity\_history\_test, label='Sensitivity\_test')  
 plt.xlabel('Epochs')  
 plt.ylabel('Values')  
 plt.grid(True)  
 plt.title('Sensitivity and specificity by epoch')  
 plt.legend()  
  
 thresholds = np.linspace(0, 1, 100)  
 for threshold in thresholds:  
 network.threshold = threshold  
 f = io.StringIO()  
 # Używamy contextlib.redirect\_stdout do przekierowania stdout  
 with contextlib.redirect\_stdout(f):  
  
 network.train(X\_train, Y\_train, 1000)  
 predictions\_train = network.predict(X\_train)  
 predictions\_test = network.predict(X\_test)  
 predictions\_train = (predictions\_train[0, :] > network.threshold).astype(int)  
 predictions\_test = (predictions\_test[0, :] > network.threshold).astype(int)  
  
 sensitivity\_train, specificity\_train = network.calculate\_sensitivity\_specificity(Y\_train, predictions\_train)  
 sensitivity\_history\_train\_th.append(sensitivity\_train)  
 specificity\_history\_train\_th.append(specificity\_train)  
  
 sensitivity\_test, specificity\_test = network.calculate\_sensitivity\_specificity(Y\_test, predictions\_test)  
 specificity\_history\_test\_th.append(specificity\_test)  
 sensitivity\_history\_test\_th.append(sensitivity\_test)  
  
 plt.figure()  
 plt.plot(thresholds, specificity\_history\_train\_th, label='Specificity\_train')  
 plt.plot(thresholds, sensitivity\_history\_train\_th, label='Sensitivity\_train')  
 plt.plot(thresholds, specificity\_history\_test\_th, label='Specificity\_test')  
 plt.plot(thresholds, sensitivity\_history\_test\_th, label='Sensitivity\_test')  
 plt.xlabel('Thresholds')  
 plt.ylabel('Values')  
 plt.grid(True)  
 plt.title('Sensitivity and specificity by threshold')  
 plt.legend()  
 plt.show()  
  
  
  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 X\_train, X\_test, Y\_train, Y\_test = load\_data('X\_train.csv', 'X\_test.csv', 'Y\_train.csv', 'Y\_test.csv')  
  
 test\_network(X\_train, X\_test, Y\_train, Y\_test)