Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №4

По дисциплине: «МиАПР»

Тема: «Нелинейные ИНС в задачах прогнозирования.»

Выполнил:

Студент 2 курса

Группы ПО-7(2)

Панкратов Р.С.

Проверил:

Крощенко А.А.

2021

Вариант 3

**Цель работы:**

Изучить обучение и функционирование нелинейной ИНС при решении задач распознавания образов.

**Задание:**

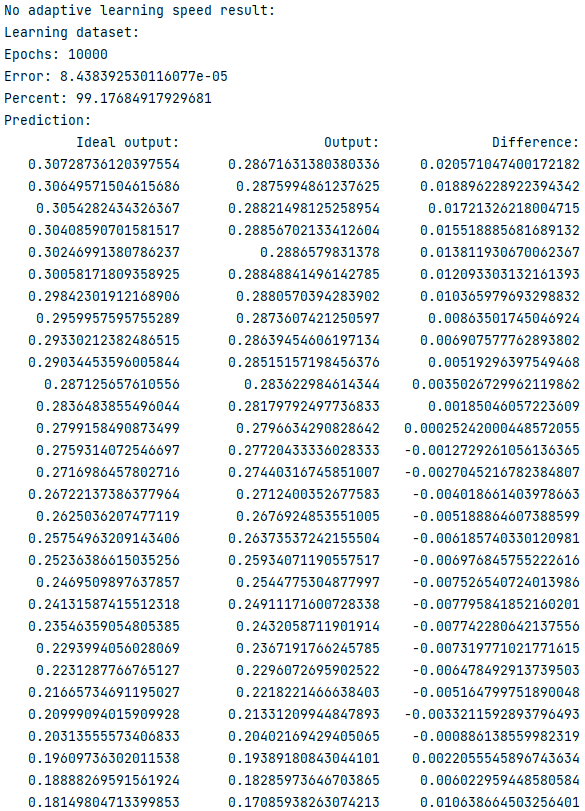
Спрогнозировать нелинейный временной ряд, применяя параметры лабораторной работы №3. При этом необходимо использовать алгоритм обучения многослойной ИНС с адаптивным шагом.

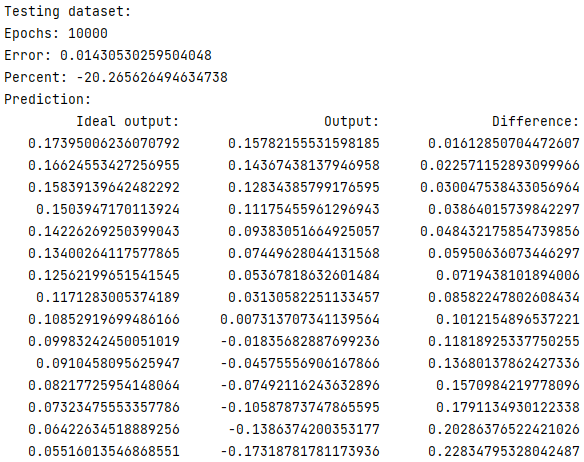
**Код программы:**

from typing import Tuple, Callable, Any  
from functools import partial  
  
import numpy as np  
from numpy.typing import NDArray  
  
LEARNING\_SPEED = 0.1  
  
WeightsType = NDArray[NDArray[np.float64]]  
TType = NDArray[np.float64]  
  
  
def func(a: float, b: float, c: float, d: float, x) -> float:  
 return a \* np.cos(b \* x) + c \* np.sin(d \* x)  
  
  
def sigmoid(s):  
 return 1 / (1 + np.exp(-s))  
  
  
def d\_sigmoid(y):  
 return y \* (1 - y)  
  
  
def lin(s):  
 return s  
  
  
def d\_lin(\_):  
 return 1.0  
  
  
def generate\_dataset(  
 start: int, stop: int, count: int, func: Callable[[Any], float], step: float  
) -> Tuple[NDArray[NDArray[np.float64]], NDArray[NDArray[np.float64]]]:  
 inputs, outputs = [], []  
 for i in range(start, stop):  
 inputs\_sample = func(np.arange(i, count + i) \* step)  
 outputs\_sample = np.array([func((count + i) \* step)])  
  
 inputs.append(inputs\_sample)  
 outputs.append(outputs\_sample)  
  
 return np.array(inputs), np.array(outputs)  
  
  
def calc\_error(  
 outputs: NDArray[NDArray[np.float64]], ideal\_outputs: NDArray[NDArray[np.float64]]  
) -> NDArray[np.float64]:  
 errors = (outputs - ideal\_outputs) \*\* 2  
 return errors.sum(axis=0) / len(ideal\_outputs)  
  
  
def calc\_percent(  
 outputs: NDArray[NDArray[np.float64]], ideal\_outputs: NDArray[NDArray[np.float64]]  
) -> NDArray[np.float64]:  
 percents = (outputs \* 100) / ideal\_outputs  
 return percents.sum(axis=0) / len(ideal\_outputs)  
  
  
def training(  
 inputs: NDArray[np.float64],  
 outputs: NDArray[np.float64],  
 w\_output: WeightsType,  
 w\_hidden: WeightsType,  
 T\_output: TType,  
 T\_hidden: TType,  
 is\_adaptive\_learning\_speed: bool = False,  
) -> None:  
 hidden\_outputs = sigmoid(np.dot(w\_hidden, inputs) - T\_hidden)  
 output\_outputs = lin(np.dot(w\_output, hidden\_outputs) - T\_output)  
  
 error\_output = output\_outputs - outputs  
 error\_hidden = error\_output \* d\_sigmoid(hidden\_outputs) \* w\_output  
  
 learning\_speed\_output = LEARNING\_SPEED  
 learning\_speed\_hidden = LEARNING\_SPEED  
 if is\_adaptive\_learning\_speed:  
 ls\_output\_numerator = (np.square(error\_output) \* d\_lin(hidden\_outputs)).sum()  
 ls\_output\_denominator = 1 + np.square(error\_output \* d\_lin(hidden\_outputs)).sum()  
 learning\_speed\_output = ls\_output\_numerator / ls\_output\_denominator  
  
 ls\_hidden\_numerator = 4 \* (np.square(error\_hidden) \* d\_sigmoid(hidden\_outputs)).sum()  
 ls\_hidden\_denominator = (1 + np.square(output\_outputs).sum()) \* np.square(error\_hidden \* d\_sigmoid(hidden\_outputs)).sum()  
 learning\_speed\_hidden = ls\_hidden\_numerator / ls\_hidden\_denominator  
  
 w\_output -= learning\_speed\_output \* np.dot(error\_output.reshape(-1, 1), hidden\_outputs.reshape(1, -1))  
 T\_output += learning\_speed\_output \* error\_output.reshape(-1)  
  
 w\_hidden -= learning\_speed\_hidden \* np.dot(error\_hidden.reshape(-1, 1), inputs.reshape(1, -1))  
 T\_hidden += learning\_speed\_hidden \* error\_hidden.reshape(-1)  
  
  
def learn(  
 inputs: NDArray[NDArray[np.float64]],  
 ideal\_outputs: NDArray[NDArray[np.float64]],  
 w\_output: WeightsType,  
 w\_hidden: WeightsType,  
 T\_output: TType,  
 T\_hidden: TType,  
 epochs: int = 1000,  
 is\_print\_intermediate\_result: bool = True,  
 is\_adaptive\_learning\_speed: bool = False  
):  
 for epoch in range(epochs + 1):  
 for input, output in zip(inputs, ideal\_outputs):  
 training(input, output, w\_output, w\_hidden, T\_output, T\_hidden, is\_adaptive\_learning\_speed)  
  
 if is\_print\_intermediate\_result and not epoch % (epochs // 10):  
 outputs = v\_predict(inputs, w\_hidden, w\_output, T\_hidden, T\_output)  
 errors = calc\_error(outputs, ideal\_outputs)  
 percents = calc\_percent(outputs, ideal\_outputs)  
 print(f"{epoch} error: {errors}, percent: {percents}")  
  
 return w\_output, w\_hidden, T\_output, T\_hidden  
  
  
def predict(  
 inputs: NDArray[np.float64], w\_hidden: WeightsType, w\_output: WeightsType, T\_hidden: TType, T\_output: TType  
) -> NDArray[np.float64]:  
 hidden\_outputs = sigmoid(np.dot(w\_hidden, inputs) - T\_hidden)  
 return lin(np.dot(w\_output, hidden\_outputs) - T\_output)  
  
  
def print\_results(epochs, inputs, ideal\_outputs, w\_output, w\_hidden, T\_output, T\_hidden):  
 outputs = v\_predict(inputs, w\_hidden, w\_output, T\_hidden, T\_output)  
 difference = ideal\_outputs - outputs  
  
 error = calc\_error(outputs, ideal\_outputs)  
 percent = calc\_percent(outputs, ideal\_outputs)  
 print(f"Epochs: {epochs}")  
 print(f"Error: {error[0]}")  
 print(f"Percent: {percent[0]}")  
 print("Prediction:")  
 print(" Ideal output: Output: Difference:")  
 for i in range(len(difference)):  
 print(f"{ideal\_outputs[i][0]: 22}{outputs[i][0]: 25}{difference[i][0]: 25}")  
  
  
VPredictType = Callable[  
 [NDArray[NDArray[np.float64]], WeightsType, WeightsType, TType, TType], NDArray[NDArray[np.float64]]  
]  
v\_predict: VPredictType = np.vectorize(predict, signature="(n)->(m)", excluded=[1, 2, 3, 4])  
  
  
def main():  
 EPOCHS = 10\_000  
  
 A, B, C, D = 0.3, 0.3, 0.07, 0.3  
 STEP = 0.1  
  
 nn\_inputs = 10  
 nn\_hidden = 4  
 nn\_output = 1  
  
 training\_values, testing\_values = 30, 15  
  
 wrapped\_func = partial(func, A, B, C, D)  
 inputs, ideal\_outputs = generate\_dataset(0, training\_values, nn\_inputs, wrapped\_func, STEP)  
 test\_inputs, test\_ideal\_outputs = generate\_dataset(  
 training\_values, training\_values + testing\_values, nn\_inputs, wrapped\_func, STEP  
 )  
  
 w\_hidden: WeightsType = np.random.normal(-1, 1, (nn\_hidden, nn\_inputs))  
 w\_output: WeightsType = np.random.normal(-1, 1, (nn\_output, nn\_hidden))  
 T\_hidden: TType = np.random.normal(-1, 1, nn\_hidden)  
 T\_output: TType = np.random.normal(-1, 1, nn\_output)  
  
 not\_adaptive\_result = learn(  
 inputs, ideal\_outputs, w\_output.copy(), w\_hidden.copy(), T\_output.copy(), T\_hidden.copy(), EPOCHS, False, False  
 )  
 print("No adaptive learning speed result:")  
 print("Learning dataset:")  
 print\_results(EPOCHS, inputs, ideal\_outputs, \*not\_adaptive\_result)  
 print()  
 print("Testing dataset:")  
 print\_results(EPOCHS, test\_inputs, test\_ideal\_outputs, \*not\_adaptive\_result)  
  
 adaptive\_result = learn(  
 inputs, ideal\_outputs, w\_output.copy(), w\_hidden.copy(), T\_output.copy(), T\_hidden.copy(), EPOCHS, False, True  
 )  
 print()  
 print("Adaptive learning speed result:")  
 print("Learning dataset:")  
 print\_results(EPOCHS, inputs, ideal\_outputs, \*adaptive\_result)  
 print()  
 print("Testing dataset:")  
 print\_results(EPOCHS, test\_inputs, test\_ideal\_outputs, \*adaptive\_result)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

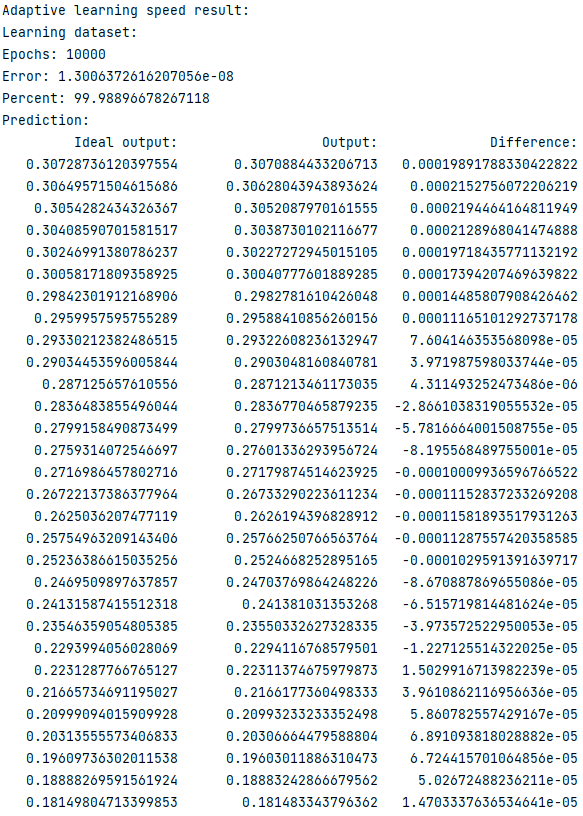
**Результат:**

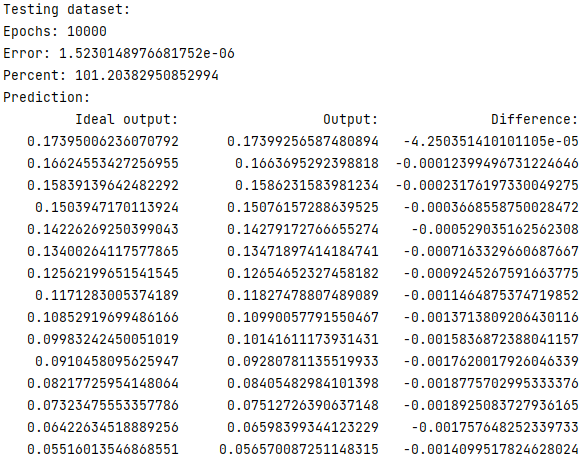
Результаты без адаптивного шага обучения:



****

Результаты c адаптивным шагом обучения:

****

****

**Вывод:**

Нн с адаптивным шагом обучения обучается горазда эффективнее нн без адаптивного шага обучения.