**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5

по дисциплине: **«**Введение в нейронные сети**»**

**Анализ временных рядов и аппроксимация**

Выполнила: студент гр. ИТИ-21

Тесёлкин Б. А.

Принял: ассистент

Карабчикова Е.А.

Гомель 2020

**Цель работы**: разработать программное обеспечение для решения задачи аппроксимации и анализа временных рядов.

# **Задание**

Задача 1. Для аппроксимации использовать радиально базисную сеть, представленную на рисунке 1 и персептрон – рисунок 2.

Функция для аппроксимации . Количество нейронов на скрытом слое радиально базисной сети и персептрона соответственно: 7 и 7.

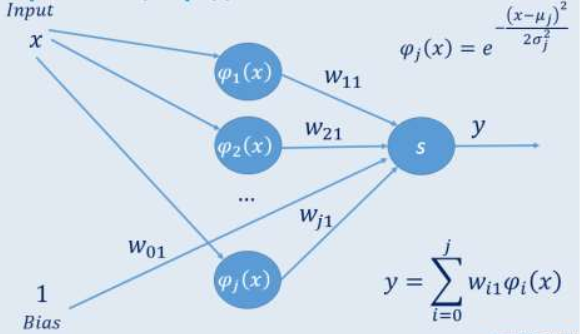


Рисунок 1 – Радиально-базисная сеть

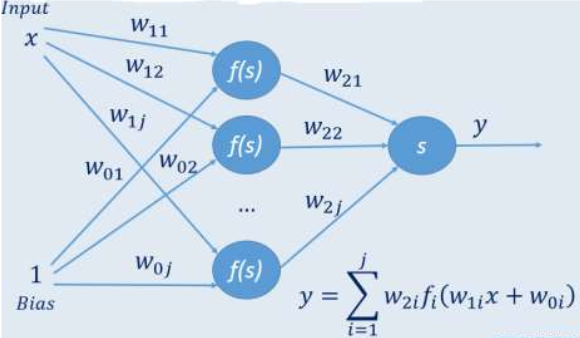


Рисунок 2 – Персептрон

Разработать соответствующее программное обеспечение.

Отобразить результаты аппроксимации графически на одном поле (3 графика).

Дополнить персептрон ещё одним скрытым слоем (количество нейронов выбрать самостоятельно).

Сравнить результаты аппроксимации с помощью радиально-базисной сети, однослойным персептроном и двухслойным персептроном.

Исследовать влияние количества нейронов на скрытом слое на погрешность аппроксимации. Сделать вывод.

Задача 2. Используя метод скользящего окна выполнить экстраполяцию данных на январь 2020 года персептроном (рисунок 3) и сетью Элмана (рисунок 4). Исследовать качество экстраполяции от представления данных, глубины погружения временного ряда и структуры нейронной сети.

Экстраполируемые данные: курс доллара к юаню.

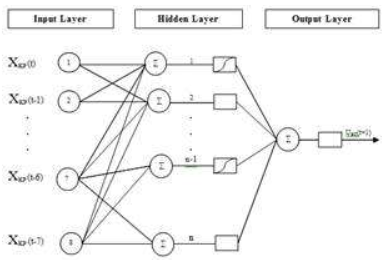


Рисунок 3 – Персептрон со скользящим окном

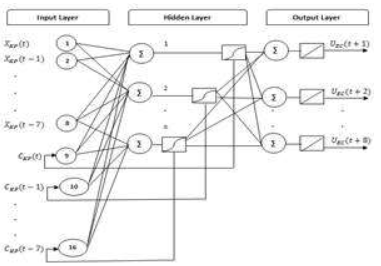


Рисунок 4 – Сеть Элмана

**Ход работы**

В ходе выполнения лабораторной работы была разработана радиально-базисная сеть, а также 2 персептрона с одним и двумя скрытыми слоями. Данные нейронные сети предназначены для аппроксимации функции. Для обучения нейронных сетей были взяты значения функции на интервале от -5 до 5. После обучения нейронные сети выдают значения близкие к значениям функции. График аппроксимации после обучения сетей из Задания 1 приведён на рисунке 5.

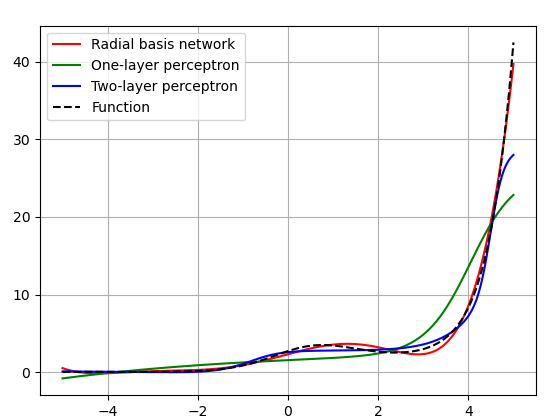


Рисунок 5 – Аппроксимация функции

По результату аппроксимации приведённом на рисунке 5 можно сделать вывод, что для качественного приближения функции необходим хотя бы персептрон с двумя скрытыми слоями (каждый скрытый слой содержит 5 нейронов), так как персептрон с одним скрытым слоем не способен нормально аппроксимировать заданную функцию. Радиально-базисная сеть из-за малого количества нейронов довольно неточно аппроксимировала функцию.

На рисунке 6 приведены графики аппроксимации функции с разным количеством нейронов на скрытом слое.

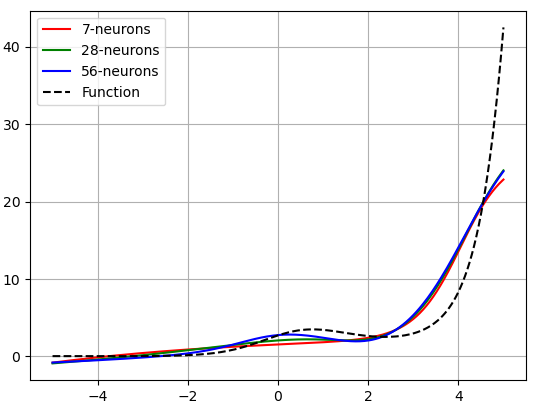


Рисунок 6 – Сети с разным количеством нейроном

Далее по ходу лабораторной работы были взяты данные температуры моря в Юрмале. Данные представлены на рисунке 7.

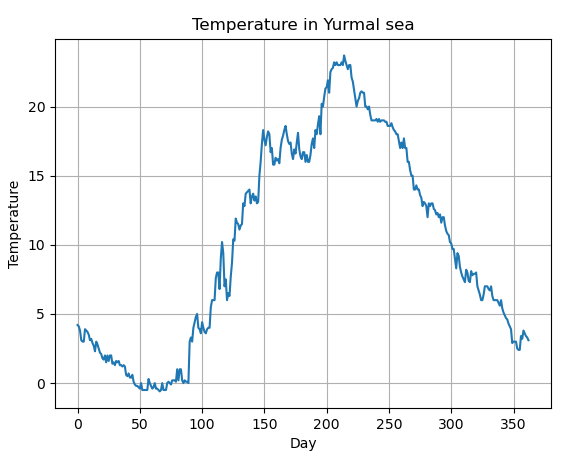


Рисунок 7 – График температуры моря в Юрмале

После чего были созданы сеть Элмана и персептрон, которые были обучены методом скользящего окна. После обучения сети были использованы для предсказания курса российского рубля к евро на следующий месяц. График экстраполяции данных Персептроном и сетью Элмана изображён на рисунке 8.

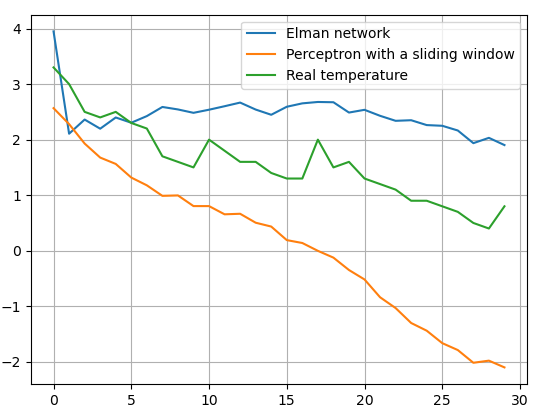


Рисунок 8 – Экстраполяция данных

По результату экстраполяции приведённом на рисунке 8 можно сделать вывод, что нейронные сети, обученные с использованием метода скользящего окна, достаточно хорошо могут выполнять прогноз, то есть выполнять задачу экстраполяции.

**Вывод:** в результате выполнения данной лабораторной работы было разработано программное обеспечение для решения задачи экстраполяции и анализа временных рядов.

**Приложение:**

**Код программы Task\_Lab5.py:**

import numpy as np  
import matplotlib.pyplot as plt  
  
import NeuralNetworkModule as nnm  
  
  
def create\_networks(fx, data\_length):  
 hidden\_num = 7  
 learning\_rate = 0.00001  
  
 neural\_network\_1 = nnm.NeuralNetwork(learning\_rate)  
 neural\_network\_1.add\_layer(1, hidden\_num, "tanh", isBias=True)  
 neural\_network\_1.add\_layer(hidden\_num, 1)  
  
 neural\_network\_2 = nnm.NeuralNetwork(learning\_rate)  
 neural\_network\_2.add\_layer(1, hidden\_num, "tanh", isBias=True)  
 neural\_network\_2.add\_layer(hidden\_num, hidden\_num, "tanh", isBias=True)  
 neural\_network\_2.add\_layer(hidden\_num, 1)  
  
 neural\_network\_1\_4 = nnm.NeuralNetwork(learning\_rate)  
 neural\_network\_1\_4.add\_layer(1, hidden\_num \* 4, "tanh", isBias=True)  
 neural\_network\_1\_4.add\_layer(hidden\_num \* 4, 1)  
  
 neural\_network\_1\_8 = nnm.NeuralNetwork(learning\_rate)  
 neural\_network\_1\_8.add\_layer(1, hidden\_num \* 8, "tanh", isBias=True)  
 neural\_network\_1\_8.add\_layer(hidden\_num \* 8, 1)  
  
 rbn = nnm.RadialBasisNetwork(7)  
  
 en = nnm.ElmanNetwork(data\_length, 40, 1, 0.002)  
  
 neural\_network\_window = nnm.NeuralNetwork(0.002)  
 neural\_network\_window.add\_layer(data\_length, 50, "tanh")  
 neural\_network\_window.add\_layer(50, 1)  
  
 x = np.random.uniform(-5, 5, size=[150000, 1])  
 y = fx(x)  
 rbn.sigma = np.std(y)  
 rbn.fit(x, y)  
 for i in range(150000):  
 xi = x[i].reshape(1, 1)  
 yi = y[i].reshape(1, 1)  
 neural\_network\_1.forward(xi)  
 neural\_network\_1.backward(yi)  
 neural\_network\_1.update()  
 neural\_network\_2.forward(xi)  
 neural\_network\_2.backward(yi)  
 neural\_network\_2.update()  
 neural\_network\_1\_4.forward(xi)  
 neural\_network\_1\_4.backward(yi)  
 neural\_network\_1\_4.update()  
 neural\_network\_1\_8.forward(xi)  
 neural\_network\_1\_8.backward(yi)  
 neural\_network\_1\_8.update()  
 return rbn, neural\_network\_1, neural\_network\_2, neural\_network\_1\_4, neural\_network\_1\_8, en, neural\_network\_window  
  
  
def display\_charts(fx, rbn, nn\_1, nn\_2):  
 points = np.linspace(-5, 5, 400)  
 p1 = []  
 p2 = []  
 for i in points:  
 p1.append(nn\_1.predict(i.reshape(1, 1)).reshape(-1))  
 p2.append(nn\_2.predict(i.reshape(1, 1)).reshape(-1))  
 plt.plot(points, rbn.predict(points), "r", label="Radial basis network")  
 plt.plot(points, p1, "g", label="One-layer perceptron")  
 plt.plot(points, p2, "b", label="Two-layer perceptron")  
 plt.plot(points, fx(points), "k--", label="Function")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 plt.legend()  
 plt.grid()  
 plt.show()  
  
  
def train\_nets(data\_length, train\_data, en, nn\_w, mean, std):  
 for epoch in range(400):  
 q = np.random.randint(0, data\_length)  
 for i in range(q, len(train\_data) - data\_length, data\_length):  
 x = (np.array(train\_data[i: i + data\_length]).reshape(1, data\_length) - mean) / std  
 y = (np.array(train\_data[i + data\_length: i + data\_length + 1]).reshape(1, 1) - mean) / std  
 en.forward(x)  
 en.backward(y)  
 en.update()  
 nn\_w.forward(x)  
 nn\_w.backward(y)  
 nn\_w.update()  
 return True  
  
def display\_nets1(fx, nn\_1, nn\_1\_4, nn\_1\_8):  
 points = np.linspace(-5, 5, 400)  
 p1 = []  
 p2 = []  
 p3 = []  
 for i in points:  
 p1.append(nn\_1.predict(i.reshape(1, 1)).reshape(-1))  
 p2.append(nn\_1\_4.predict(i.reshape(1, 1)).reshape(-1))  
 p3.append(nn\_1\_8.predict(i.reshape(1, 1)).reshape(-1))  
 plt.plot(points, p1, "r", label="7-neurons")  
 plt.plot(points, p2, "g", label="28-neurons")  
 plt.plot(points, p3, "b", label="56-neurons")  
 plt.plot(points, fx(points), "k--", label="Function")  
 plt.xlabel("x")  
 plt.xlabel("y")  
 plt.legend()  
 plt.grid()  
 plt.show()  
  
def display\_temp(train\_data):  
 plt.plot(train\_data)  
 plt.xlabel("Day")  
 plt.ylabel("Temperature")  
 plt.title("Temperature in Yurmal sea")  
 plt.grid()  
 plt.show()  
  
def display\_ext\_data(data\_length, train\_data, test\_data, en, nn\_w, mean, std):  
 en\_y = []  
 nn\_w\_y = []  
 for i in train\_data[-data\_length:]:  
 en\_y.append((i - mean) / std)  
 nn\_w\_y.append((i - mean) / std)  
 for i in range(len(test\_data)):  
 out\_en = en.forward(np.array(en\_y[i:i+data\_length]).reshape(1, data\_length)).reshape(-1)  
 out\_perceptron = nn\_w.predict(np.array(nn\_w\_y[i:i+data\_length]).reshape(1, data\_length)).reshape(-1)  
 en\_y.append(out\_en[0])  
 nn\_w\_y.append(out\_perceptron[0])  
 for i in range(len(en\_y)):  
 en\_y[i] = (en\_y[0] - (en\_y[i] - en\_y[0])) - 0.25  
 plt.plot(np.array(en\_y[data\_length:]) \* std + mean, label="Elman network")  
 plt.plot(np.array(nn\_w\_y[data\_length:]) \* std + mean, label="Perceptron with a sliding window")  
 plt.plot(test\_data, label="Real temperature")  
 plt.legend()  
 plt.grid()  
 plt.show()  
  
  
def read(file):  
 with open(file, "r") as file:  
 rows = file.read().split()  
 data = []  
 for i, k in enumerate(rows[:-1]):  
 data.append(float(k))  
 return data  
  
  
def display\_menu():  
 print("Menu")  
 print("1. Display charts")  
 print("2. Display different networks")  
 print("3. Temperature in Yurmal")  
 print("4. Extrapolated data")  
 print("5. Exit")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 train\_data = read("train.txt")  
 test\_data = read("test.txt")  
 mean = np.mean(train\_data)  
 std = np.std(train\_data)  
 data\_length = 40  
  
 fx = lambda x: 2 \*\* x \* np.exp(np.cos(x))  
 print("Training...", end='')  
 rbn, nn\_1, nn\_2, nn\_1\_4, nn\_1\_8, en, nn\_w = create\_networks(fx, data\_length)  
 train\_nets(data\_length, train\_data, en, nn\_w, mean, std)  
 print("Done!")  
  
 isExit = False  
 k = 0  
 while not isExit:  
 display\_menu()  
 try:  
 k = int(input("Enter menu number:"))  
 except ValueError:  
 pass  
 print()  
  
 if k == 1:  
 display\_charts(fx, rbn, nn\_1, nn\_2)  
 elif k == 2:  
 display\_nets1(fx, nn\_1, nn\_1\_4, nn\_1\_8)  
 elif k == 3:  
 display\_temp(train\_data)  
 elif k == 4:  
 display\_ext\_data(data\_length, train\_data, test\_data, en, nn\_w, mean, std)  
 elif k == 5:  
 isExit = True

**Код программы NeuralNetworkModule.py:**

import numpy as np  
  
  
class RadialBasisNetwork:  
 def \_\_init\_\_(self, hiddenNum, sigma=1.0):  
  
 self.hiddenNum = hiddenNum  
 self.sigma = sigma  
 self.centers = 0  
 self.weights = 0  
  
 def fit(self, x, y):  
 self.centers = x[np.random.choice(len(x), self.hiddenNum)]  
 inter\_matrix = self.calc\_interpolation\_matrix(x)  
 inverse\_inter\_matrix = np.linalg.pinv(inter\_matrix)  
 self.weights = inverse\_inter\_matrix @ y  
  
 def predict(self, x):  
 inter\_matrix = self.calc\_interpolation\_matrix(x)  
 return inter\_matrix @ self.weights  
  
 def calc\_interpolation\_matrix(self, x):  
 inter\_matrix = np.zeros((len(x), self.hiddenNum))  
 for i, point in enumerate(x):  
 for j, center in enumerate(self.centers):  
 inter\_matrix[i, j] = self.radial\_basis\_func(point, center)  
 return inter\_matrix  
  
 def radial\_basis\_func(self, point, center):  
 return np.exp(-np.linalg.norm((point - center) \*\* 2 / (2 \* self.sigma \*\* 2)))  
  
  
class NeuralNetwork:  
 def \_\_init\_\_(self, learningRate=0.01, mu=0.85, ):  
  
 self.learningRate = learningRate  
 self.mu = mu  
  
 self.output = 0  
 self.layers = []  
 self.parameters = []  
  
 def add\_layer(self, inputNum, outputNum, activationFunc="nope", isBias=False):  
 self.layers.append(Layer(inputNum, outputNum, activationFunc, isBias))  
 self.parameters.append({"main": {"weight": 0, "bias": 0}})  
  
 def forward(self, x):  
 tmp\_input = x  
 for layer, parameter in zip(self.layers, self.parameters):  
 tmp\_input = layer.forward(tmp\_input, parameter["main"], self.mu)  
 self.output = tmp\_input  
 return tmp\_input  
  
 def backward(self, expected):  
 grad = self.mean\_square\_error(expected, True)  
 for layer, parameter in zip(list(reversed(self.layers)), list(reversed(self.parameters))):  
 grad = layer.backward(grad, parameter["main"], self.mu)  
  
 def predict(self, x):  
 tmp\_input = x  
 for layer in self.layers:  
 tmp\_input = layer.forward(tmp\_input, {"weight": 0, "bias": 0}, 0)  
 self.output = tmp\_input  
 return tmp\_input  
  
 def update(self):  
 for layer, parameter in zip(self.layers, self.parameters):  
 self.nesterov\_gradient(layer.weights, layer.deltaWeights, parameter)  
 if layer.isBias:  
 self.nesterov\_gradient(layer.bias, layer.deltaBias, parameter, "bias")  
  
 def mean\_square\_error(self, expected, der=False):  
 if not der:  
 return np.mean((self.output - expected) \*\* 2)  
 return self.output - expected  
  
 def nesterov\_gradient(self, weights, deltaWeights, parameter, key="weight"):  
 parameter["main"][key] = self.mu \* parameter["main"][key] + self.learningRate \* deltaWeights  
 weights -= parameter["main"][key]  
  
  
class Layer:  
 def \_\_init\_\_(self, inputNum, outputNum, activationFunc="none", isBias=False):  
  
 self.bias = 0  
 self.isBias = isBias  
 if isBias:  
 self.bias = np.random.uniform(low=-np.sqrt(1 / inputNum),  
 high=np.sqrt(1 / inputNum), size=[1, outputNum])  
  
 self.weights = np.random.uniform(low=-np.sqrt(1 / inputNum),  
 high=np.sqrt(1 / inputNum), size=[inputNum, outputNum])  
  
 self.activationFunc = lambda x: x  
 self.derActivationFunc = lambda x: 1  
  
 if activationFunc == "tanh":  
 self.activationFunc = lambda x: np.tanh(x)  
 self.derActivationFunc = lambda x: 1 - self.activationFunc(x) \*\* 2  
  
 def forward(self, x, parameter, mu):  
 self.input = x  
 self.middleInput = x @ (self.weights - mu \* parameter["weight"])  
 if self.isBias:  
 self.middleInput += (self.bias - mu \* parameter["bias"])  
 self.output = self.activationFunc(self.middleInput)  
 return self.output  
  
 def backward(self, grad, parameters, mu):  
 delta\_output = grad \* self.derActivationFunc(self.middleInput)  
 self.deltaWeights = self.input.T @ delta\_output  
 if self.isBias:  
 self.deltaBias = (delta\_output - mu \* parameters["bias"])  
 delta\_input = delta\_output @ (self.weights - mu \* parameters["weight"]).T  
 return delta\_input  
  
  
class ElmanNetwork:  
 def \_\_init\_\_(self, inputNum, hiddenNum, outputNum, learningRate):  
 self.learningRate = learningRate  
  
 self.input\_hidden\_weights = np.random.uniform(-np.sqrt(1 / inputNum), np.sqrt(1 / inputNum),  
 size=[inputNum, hiddenNum])  
 self.input\_hidden\_bias = np.random.uniform(size=[1, hiddenNum])  
  
 self.hidden\_hidden\_weights = np.random.uniform(-np.sqrt(1 / hiddenNum), np.sqrt(1 / hiddenNum),  
 size=[hiddenNum, hiddenNum])  
 self.hidden\_hidden\_bias = np.random.uniform(size=[1, hiddenNum])  
  
 self.hidden = np.zeros(shape=[1, hiddenNum])  
 self.hidden\_transpose = np.zeros(shape=[1, hiddenNum])  
  
 self.weights = np.random.uniform(-np.sqrt(1 / hiddenNum), np.sqrt(1 / hiddenNum),  
 size=[hiddenNum, outputNum])  
  
 def forward(self, x):  
 self.input = x  
 self.hidden\_transpose = self.hidden  
 self.hidden = self.input @ self.input\_hidden\_weights + self.input\_hidden\_bias + self.hidden\_transpose @ self.hidden\_hidden\_weights + self.hidden\_hidden\_bias  
 self.hidden = np.tanh(self.hidden)  
 self.output = self.hidden @ self.weights  
 return self.output  
  
 def backward(self, y):  
 delta\_loss = self.output - y  
 self.delta\_weights = self.hidden.T @ delta\_loss  
 delta\_hidden = delta\_loss @ self.weights.T  
 grad = (1 - np.tanh(self.hidden) \*\* 2) \* delta\_hidden  
 self.delta\_weights\_ih = self.input.T @ grad  
 self.delta\_bias\_ih = 1 \* grad  
 self.delta\_weights\_hh = self.hidden\_transpose.T @ grad  
 self.delta\_bias\_hh = 1 \* grad  
  
 def update(self):  
 self.weights -= self.learningRate \* self.delta\_weights  
 self.input\_hidden\_weights -= self.learningRate \* self.delta\_weights\_ih  
 self.input\_hidden\_bias -= self.learningRate \* self.delta\_bias\_ih  
 self.hidden\_hidden\_weights -= self.learningRate \* self.delta\_weights\_hh  
 self.hidden\_hidden\_bias -= self.learningRate \* self.delta\_bias\_hh