**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**

Выполнил: студент гр. ИТИ-21

Шеметков К.Д.

Принял: преподаватель

Карабчикова Е.А.

Гомель 2020

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

# **Задание**

**Задание 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: 6 и 7.

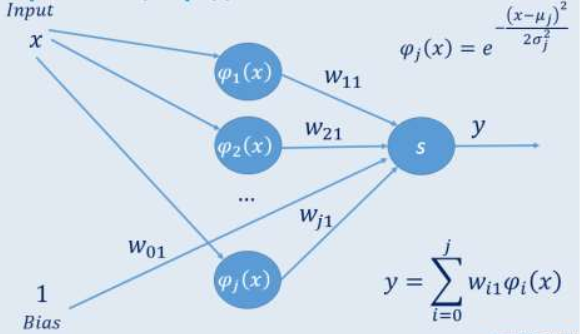


Рисунок 1 – Радиально-базисная сеть

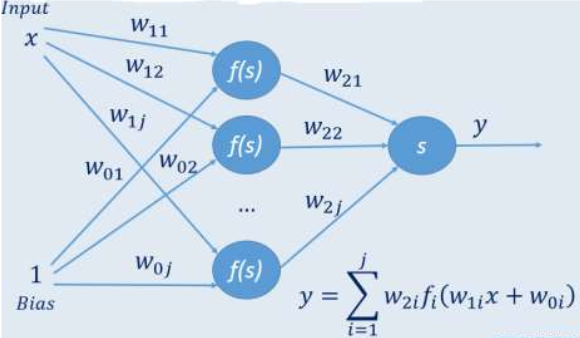


Рисунок 2 – Персептрон

Разработать соответствующее программное обеспечение.

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

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.

**Задание 2**. Используя метод скользящего окна выполнить экстраполяцию данных на январь месяц 2019 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: температура воздуха в Бресте.

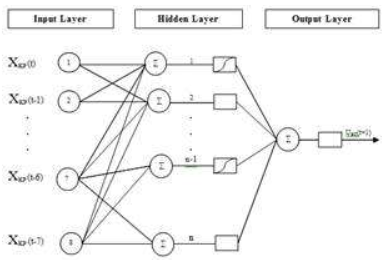


Рисунок 3 – Персептрон со скользящим окном

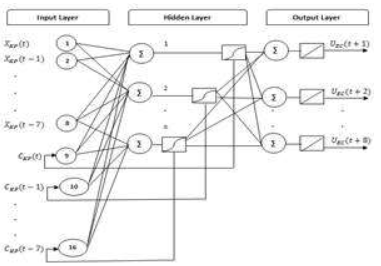


Рисунок 4 – Сеть Элмана

**Ход работы**

В ходе выполнения лабораторной работы была разработана радиально-базисная сеть и 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 -2.5 до 5.5. После обучения нейронные сети выдают значения примерно близкие к значениям функции. График аппроксимации обучения сетей показан на рисунке 5 ниже.

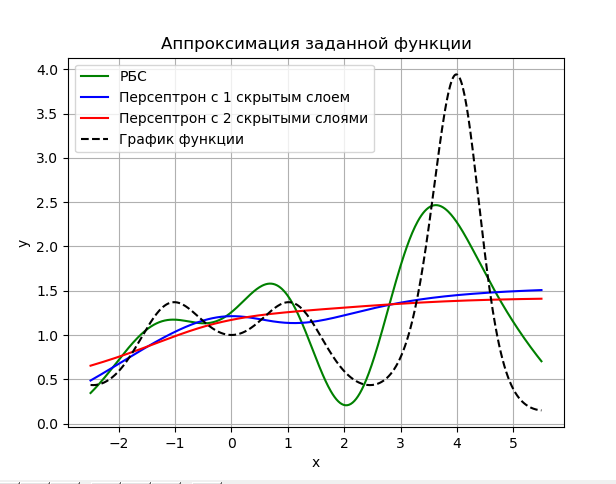


Рисунок 5 – Аппроксимация функции

По результату аппроксимации приведённом на рисунке 5 можно сделать вывод, что с качественным приближением функции персептрон не справляется. Радиально-базисная сеть сделала это более точно. Можно сравнить данные на примере персептрона с различным числом нейронов. График показан на рисунке 6.

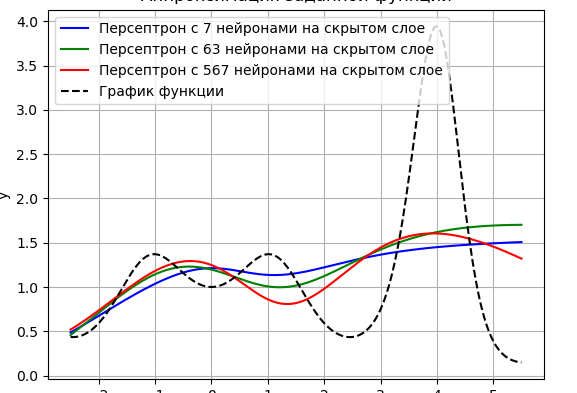


Рисунок 6 – Аппроксимация персептроном

Видно, что для более точного приближения необходимо большое количество нейронов на скрытом слое.

Далее по ходу лабораторной работы были взяты данные температуры воздуха в городе за месяц. Данные представлены на рисунке 7.

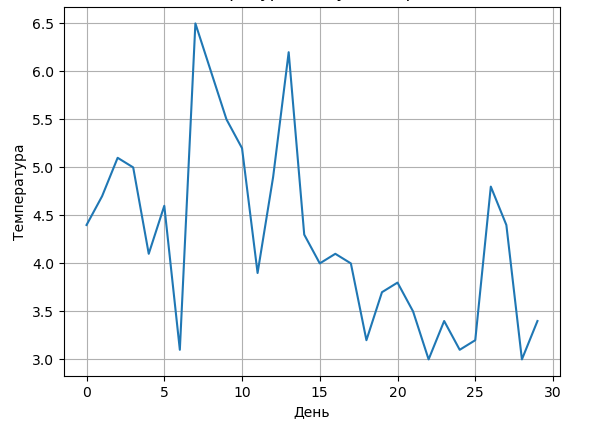


Рисунок 7 – График температуры города

После чего были созданы сеть Элмана и персептрон. Сети были использованы для предсказания погоды на следующий. График экстраполяции данных Персептроном и сетью Элмана изображён на рисунке 8.

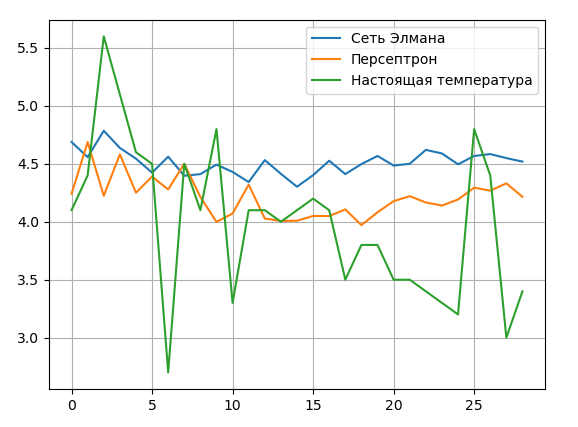


Рисунок 8 – Экстраполяция данных

По результату экстраполяции можно сделать вывод, что нейронные сети, обученные с использованием метода скользящего окна, могут выполнять прогноз с точностью 0.5, то есть выполнять экстраполяцию.

**Вывод:** в результате выполнения данной лабораторной работы были решены задачи экстраполяции и анализа временных рядов.

**Листинг файла lab5\_1.py:**

import numpy as np

import matplotlib.pyplot as plt

import NeuralNetwork as nn

def draw\_menu():

print("Меню:")

print("1. Обучить")

print("2. Аппроксимация + графики")

print("3. Графическая разница нейронов")

print("4. Выход")

def menu\_item\_1(fx, rbf, perceptron\_1, perceptron\_2, perceptron\_1x2, perceptron\_1x3):

x = np.random.uniform(-2.5, 5.5, size=[150000, 1])

y = fx(x)

rbf.sigma = np.std(y)

rbf.fit(x, y)

for i in range(150000):

xi = x[i].reshape(1, 1)

yi = y[i].reshape(1, 1)

perceptron\_1.forward(xi)

perceptron\_1.backward(yi)

perceptron\_1.update()

perceptron\_2.forward(xi)

perceptron\_2.backward(yi)

perceptron\_2.update()

perceptron\_1x2.forward(xi)

perceptron\_1x2.backward(yi)

perceptron\_1x2.update()

perceptron\_1x3.forward(xi)

perceptron\_1x3.backward(yi)

perceptron\_1x3.update()

return True

def menu\_item\_2(fx, rbf, perceptron\_1, perceptron\_2):

points = np.linspace(-2.5, 5.5, 500)

p1 = []

p2 = []

for i in points:

p1.append(perceptron\_1(i.reshape(1, 1)).reshape(-1))

p2.append(perceptron\_2(i.reshape(1, 1)).reshape(-1))

plt.plot(points, rbf.predict(points), "g", label="РБС")

plt.plot(points, p1, "b", label="Персептрон с 1 скрытым слоем")

plt.plot(points, p2, "r", label="Персептрон с 2 скрытыми слоями")

plt.plot(points, fx(points), "k--", label="График функции")

plt.xlabel("x")

plt.ylabel("y")

plt.title(r"Аппроксимация заданной функции")

plt.legend()

plt.grid()

plt.show()

def menu\_item\_3(fx, perceptron\_1, perceptron\_1x2, perceptron\_1x3):

points = np.linspace(-2.5, 5.5, 500)

p1 = []

p1\_x2 = []

p1\_x3 = []

for i in points:

p1.append(perceptron\_1(i.reshape(1, 1)).reshape(-1))

p1\_x2.append(perceptron\_1x2(i.reshape(1, 1)).reshape(-1))

p1\_x3.append(perceptron\_1x3(i.reshape(1, 1)).reshape(-1))

plt.plot(points, p1, "b", label="Персептрон с 7 нейронами на скрытом слое")

plt.plot(points, p1\_x2, "g", label="Персептрон с 63 нейронами на скрытом слое")

plt.plot(points, p1\_x3, "r", label="Персептрон с 567 нейронами на скрытом слое")

plt.plot(points, fx(points), "k--", label="График функции")

plt.xlabel("x")

plt.ylabel("y")

plt.title(r"Аппроксимация заданной функции")

plt.legend()

plt.grid()

plt.show()

def create\_nets():

rbf = nn.RBFNet(6)

hidden\_neurons = 7

lr = 0.00001

perceptron\_1 = nn.NN(lr)

perceptron\_1.add\_layer(1, hidden\_neurons, "tanh", need\_bias=True)

perceptron\_1.add\_layer(hidden\_neurons, 1)

perceptron\_2 = nn.NN(lr / 2)

perceptron\_2.add\_layer(1, hidden\_neurons, "tanh", need\_bias=True)

perceptron\_2.add\_layer(hidden\_neurons, hidden\_neurons, "tanh", need\_bias=True)

perceptron\_2.add\_layer(hidden\_neurons, 1)

perceptron\_1x2 = nn.NN(lr)

perceptron\_1x2.add\_layer(1, hidden\_neurons \* 3 \*\* 2, "tanh", need\_bias=True)

perceptron\_1x2.add\_layer(hidden\_neurons \* 3 \*\* 2, 1)

perceptron\_1x3 = nn.NN(lr)

perceptron\_1x3.add\_layer(1, hidden\_neurons \* 3 \*\* 4, "tanh", need\_bias=True)

perceptron\_1x3.add\_layer(hidden\_neurons \* 3 \*\* 4, 1)

return rbf, perceptron\_1, perceptron\_2, perceptron\_1x2, perceptron\_1x3

if \_\_name\_\_ == "\_\_main\_\_":

training\_complete = False

fx = lambda x: pow(2,x\*np.cos(x)\*np.sin(x))/(pow(np.sin(x),2)+pow(np.cos(x),2))

rbf, perceptron\_1, perceptron\_2, perceptron\_1x2, perceptron\_1x3 = create\_nets()

flag = False

k = 0

while not flag:

draw\_menu()

try:

k = int(input("Выберите пункт меню: "))

except ValueError:

pass

print()

if k == 1:

training\_complete = menu\_item\_1(fx, rbf, perceptron\_1, perceptron\_2, perceptron\_1x2, perceptron\_1x3)

print("\nОбучение завершено успешно.\a\n")

elif k == 2:

if training\_complete:

menu\_item\_2(fx, rbf, perceptron\_1, perceptron\_2)

else:

print("Ошибка. Необходимо обучить сеть.\n")

elif k == 3:

if training\_complete:

menu\_item\_3(fx, perceptron\_1, perceptron\_1x2, perceptron\_1x3)

else:

print("Ошибка. Необходимо обучить сеть.\n")

elif k == 4:

flag = True

**Листинг файла lab5\_2.py:**

import numpy as np

import matplotlib.pyplot as plt

import NeuralNetwork as nn

def deserialize(filename):

with open(filename, "r") as file:

rows = file.read().split()

data = []

for i, k in enumerate(rows[:-1]):

data.append(float(k))

return data

def draw\_menu():

print("Меню")

print("1. Температура в Бресте")

print("2. Экстраполированные данные")

print("3. Выход")

def train\_nets(seq\_length, train, rnn, perceptron, mean, std):

for epoch in range(300):

q = np.random.randint(0, seq\_length)

for i in range(q, len(train) - seq\_length, seq\_length):

x = (np.array(train[i : i+seq\_length]).reshape(1, seq\_length) - mean) / std

y = (np.array(train[i+seq\_length : i+seq\_length+1]).reshape(1, 1) - mean) / std

rnn(x)

rnn.backward(y)

rnn.update()

perceptron.forward(x)

perceptron.backward(y)

perceptron.update()

return True

def menu\_item\_1(train):

plt.plot(train)

plt.xlabel("День")

plt.ylabel("Температура")

plt.title("Температура воздуха в городе")

plt.grid()

plt.show()

def menu\_item\_2(seq\_length, train, test, rnn, perceptron, mean, std):

rnn\_y = []

perceptron\_y = []

for i in train[-seq\_length:]:

rnn\_y.append((i - mean) / std)

perceptron\_y.append((i - mean) / std)

for i in range(len(test)):

out\_rnn = rnn(np.array(rnn\_y[i:i+seq\_length]).reshape(1, seq\_length)).reshape(-1)

out\_perceptron = perceptron(np.array(perceptron\_y[i:i+seq\_length]).reshape(1, seq\_length)).reshape(-1)

rnn\_y.append(out\_rnn[0])

perceptron\_y.append(out\_perceptron[0])

plt.plot(np.array(rnn\_y[seq\_length:]) \* std + mean, label="Сеть Элмана")

plt.plot(np.array(perceptron\_y[seq\_length:]) \* std + mean, label="Персептрон")

plt.plot(test, label="Настоящая температура")

plt.legend()

plt.grid()

plt.show()

def create\_nets(seq\_length):

rnn = nn.RNN(seq\_length, 30, 1, 0.003)

perceptron = nn.NN(0.003)

perceptron.add\_layer(seq\_length, 30, "tanh")

perceptron.add\_layer(30, 1)

return rnn, perceptron

if \_\_name\_\_ == "\_\_main\_\_":

train = deserialize("train.txt")

test = deserialize("test.txt")

mean = np.mean(train)

std = np.std(train)

seq\_length = 30

rnn, perceptron = create\_nets(seq\_length)

flag = False

k = 0

while not flag:

draw\_menu()

try:

k = int(input("Введите номер пункта меню: "))

except ValueError:

pass

print()

if k == 1:

menu\_item\_1(train)

elif k == 2:

train\_nets(seq\_length, train, rnn, perceptron, mean, std)

print("\nОбучение завершено успешно.\a\n")

menu\_item\_2(seq\_length, train, test, rnn, perceptron, mean, std)

elif k == 3:

flag = True

**Листинг нейросети NN.py:**

import numpy as np

class LinearLayer:

def \_\_init\_\_(self, input\_number, output\_number, activation="none", need\_bias=False):

self.weight = np.random.uniform(low=-np.sqrt(1 / input\_number),

high=np.sqrt(1 / input\_number), size=[input\_number, output\_number])

self.need\_bias = need\_bias

self.bias = 0

if need\_bias:

self.bias = np.random.uniform(low=-np.sqrt(1 / input\_number),

high=np.sqrt(1 / input\_number), size=[1, output\_number])

self.activation = lambda x: x

self.der\_activation = lambda x: 1

if activation == "tanh":

self.activation = lambda x: np.tanh(x)

self.der\_activation = lambda x: 1 - self.activation(x) \*\* 2

def forward(self, x, acc, mu):

self.input = x

self.mid = x @ (self.weight - mu \* acc["weight"])

if self.need\_bias:

self.mid += (self.bias - mu \* acc["bias"])

self.out = self.activation(self.mid)

return self.out

def backward(self, grad, acc, mu):

dout = grad \* self.der\_activation(self.mid)

self.dw = self.input.T @ dout

if self.need\_bias:

self.db = (dout - mu \* acc["bias"])

dinp = dout @ (self.weight - mu \* acc["weight"]).T

return dinp

class NN:

def \_\_init\_\_(self, lr=0.01, mu=0.85):

self.lr = lr

self.layers = []

self.params = []

self.optim = self.NAG

self.criterion = self.MSELoss

self.out = 0

self.mu = mu

def add\_layer(self, input\_number, output\_number, activation="none", need\_bias=False):

self.layers.append(LinearLayer(input\_number, output\_number, activation, need\_bias))

self.params.append({"accumulated": {"weight": 0, "bias": 0}})

def forward(self, x):

z = x

for layer, param in zip(self.layers, self.params):

z = layer.forward(z, param["accumulated"], self.mu)

self.out = z

return z

def predict(self, x):

z = x

for layer in self.layers:

z = layer.forward(z, {"weight": 0, "bias": 0}, 0)

self.out = z

return z

def \_\_call\_\_(self, \*args):

return self.predict(\*args)

def backward(self, y):

grad = self.criterion(y, True)

for layer, param in zip(list(reversed(self.layers)), list(reversed(self.params))):

grad = layer.backward(grad, param["accumulated"], self.mu)

def MSELoss(self, y, der=False):

if not der:

return np.mean((self.out - y) \*\* 2)

return self.out - y

def update(self):

for layer, param in zip(self.layers, self.params):

self.optim(layer.weight, layer.dw, param)

if layer.need\_bias:

self.optim(layer.bias, layer.db, param, "bias")

def NAG(self, weight, dw, param, key="weight"):

param["accumulated"][key] = self.mu \* param["accumulated"][key] + self.lr \* dw

weight -= param["accumulated"][key]

class RBFNet:

def \_\_init\_\_(self, hidden\_number, sigma=1.0):

self.hidden\_number = hidden\_number

self.sigma = sigma

self.centers = 0

self.weights = 0

def rbf(self, point, center):

return np.exp(-np.linalg.norm((point - center) \*\* 2 / (2 \* self.sigma \*\* 2)))

def calculate\_interpolation\_matrix(self, x):

g = np.zeros((len(x), self.hidden\_number))

for i, point in enumerate(x):

for j, center in enumerate(self.centers):

g[i, j] = self.rbf(point, center)

return g

def fit(self, x, y):

self.centers = x[np.random.choice(len(x), self.hidden\_number)]

g = self.calculate\_interpolation\_matrix(x)

inv\_g = np.linalg.pinv(g)

self.weights = inv\_g @ y

def predict(self, x):

g = self.calculate\_interpolation\_matrix(x)

return g @ self.weights

class RNN:

def \_\_init\_\_(self, input\_number, hidden\_number, output\_number, lr=0.01):

self.lr = lr

self.w\_ih = np.random.uniform(-np.sqrt(1 / input\_number), np.sqrt(1 / input\_number),

size=[input\_number, hidden\_number])

self.b\_ih = np.random.uniform(size=[1, hidden\_number])

self.w\_hh = np.random.uniform(-np.sqrt(1 / hidden\_number), np.sqrt(1 / hidden\_number),

size=[hidden\_number, hidden\_number])

self.b\_hh = np.random.uniform(size=[1, hidden\_number])

self.h = np.zeros(shape=[1, hidden\_number])

self.h\_t\_1 = np.zeros(shape=[1, hidden\_number])

self.w = np.random.uniform(-np.sqrt(1 / hidden\_number), np.sqrt(1 / hidden\_number),

size=[hidden\_number, output\_number])

def forward(self, x):

self.x = x

self.h\_t\_1 = self.h

self.h = self.x @ self.w\_ih + self.b\_ih + self.h\_t\_1 @ self.w\_hh + self.b\_hh

self.h = np.tanh(self.h)

self.out = self.h @ self.w

return self.out

def \_\_call\_\_(self, \*args):

return self.forward(\*args)

def backward(self, y):

dloss = self.out - y

self.dw = self.h.T @ dloss

dh = dloss @ self.w.T

grad = (1 - np.tanh(self.h) \*\* 2) \* dh

self.dw\_ih = self.x.T @ grad

self.db\_ih = 1 \* grad

self.dw\_hh = self.h\_t\_1.T @ grad

self.db\_hh = 1 \* grad

def update(self):

self.w -= self.lr \* self.dw

self.w\_ih -= self.lr \* self.dw\_ih

self.b\_ih -= self.lr \* self.db\_ih

self.w\_hh -= self.lr \* self.dw\_hh

self.b\_hh -= self.lr \* self.db\_hh