МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

# ИССЛЕДОВАНИЕ ПРИМЕНИМОСТИ ФУНКЦИИ SOFTPLUS ДЛЯ РЕШЕНИЯ ЗАДАЧИ ПРИБЛИЖЕНИЯ МНОГОЧЛЕНОВ ИСКУССТВЕННЫМИ НЕЙРОННЫМИ СЕТЯМИ ПРЯМОГО РАСПРОСТРАНЕНИЯ

БАКАЛАВРСКАЯ РАБОТА

студента 4 курса 411 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Стоколесова Максима Сергеевича

Научный руководитель
доцент, к. ф.-м. н. _____ С. В. Миронов

Заведующий кафедрой
к. ф.-м. н. _____ С. В. Миронов

Саратов 2019

## СОДЕРЖАНИЕ

# ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ИНС — Искусственная нейронная сеть;

FANN — Feedforward Artificial Neural Network;

$k = \overline{1, N} \Leftrightarrow k = 1, 2, \ldots N$.

# ВВЕДЕНИЕ

В настоящее время искусственные нейронные сети приобрели большую популярность благодаря впечатляющим результатам их применения при решении самых разных задач — от приближения простейших зависимостей и классификации объектов до обнаружения и сегментации сложных образов. ИНС активно используются в поисковых системах, обработке видео и изображений, распознавании акустических и визуальных сигналов, анализе различных показаний, симуляторах, автопилотируемых системах и т.д. Архитектуры сетей и подходы к вычислению их параметров стремительно развиваются, а вместе с ними неуклонно растет и область применения ИНС, вытесняя собой стандартные подходы, которые часто оказываются менее эффективными.

Однако в настоящее время процесс определения оптимальной для поставленной задачи архитектуры ИНС нередко носит эмпирический характер, при котором возможности полученной сети могут быть только оценены на основе результатов некоторого набора испытаний, что не всегда удовлетворяет требованиям. Более того, существенное укрепление теоретической основы нейронных сетей в конечном итоге может стать необходимым условием для их дальнейшего развития.

Таким образом, вышеприведенные утверждения являются причиной появления настоящей работы, призванной внести свой вклад в развитие этого направления.

Объектом исследования в данной работе является активационная функция `softplus` [1], которая часто используется в глубоких нейронных сетях и является гладкой аппроксимацией еще более популярной в последнее время функции `ReLU` [2, 3]. Целью настоящей работы является исследование применимости указанной активационной функции для решения задачи приближения с заданной точностью многочленов произвольной степени с помощью классической полносвязной трехслойной искусственной нейронной сети прямого распространения. Данная цель определяет следующие задачи:

— изучение актуальных источников исследуемого направления;
— формирование необходимой теоретической основы;
— практическое подтверждение доказанных утверждений.

Каждый из вышеуказанных пунктов подробно рассмотрен в последующих разделах.

# 1 Формирование теоретической основы

## 1.1 Обзор используемых источников

В настоящее время существует немалое количество авторитетных источников, содержащих результаты исследований, связанных с приближением многочленов искусственными нейронными сетями прямого распространения, и к некоторым из них в данной работе осуществляются отсылки. Так, источник [4] является одним из основополагающих в данном направлении, а в [5] можно найти его интересное продолжение.

В качестве же основного источника в настоящей работе используется статья *B. Malakooti и Y. Zhou «Approximating polynomial functions by Feedforward Artificial Neural Networks: Capacity analysis and design»* [6], в котором приводятся доказательство теоремы о взаимосвязи между количеством нейронов скрытого слоя ИНС и степенью приближаемого ей многочлена, а также алгоритм построения такой нейронной сети.

В то время как в [6] рассматривалась ИНС с сигмоидальной функцией активации, в настоящей работе представлены результаты исследования применимости для решения той же задачи функции `softplus`, для которой в разделах 1.4 и 1.5 соответственно приводятся аналогичные доказательство справедливости вышеуказанной теоремы и алгоритм построения нейронной сети. Затем в разделе 2 подробно описывается способ реализации самой ИНС и двух алгоритмов вычисления ее весов — итерационного и однопроходного, результаты выполнения которых используются для подтверждения доказанных утверждений на практике.

## 1.2 Основные свойства функции softplus

Функция `softplus` имеет следующий вид [1]:

$$\varphi(x) = \ln(1 + e^x).$$

Данная функция является *гладкой*, в отличие от `ReLU` [7], которую она аппроксимирует, что означает существование непрерывной производной `softplus` на всем ее множестве определения, а это, в свою очередь, является очень важной особенностью, учитываемой при проектировании любой ИНС [8, 9]. Сравнение поведения двух функций вблизи начала координат представлено на рис. 1.

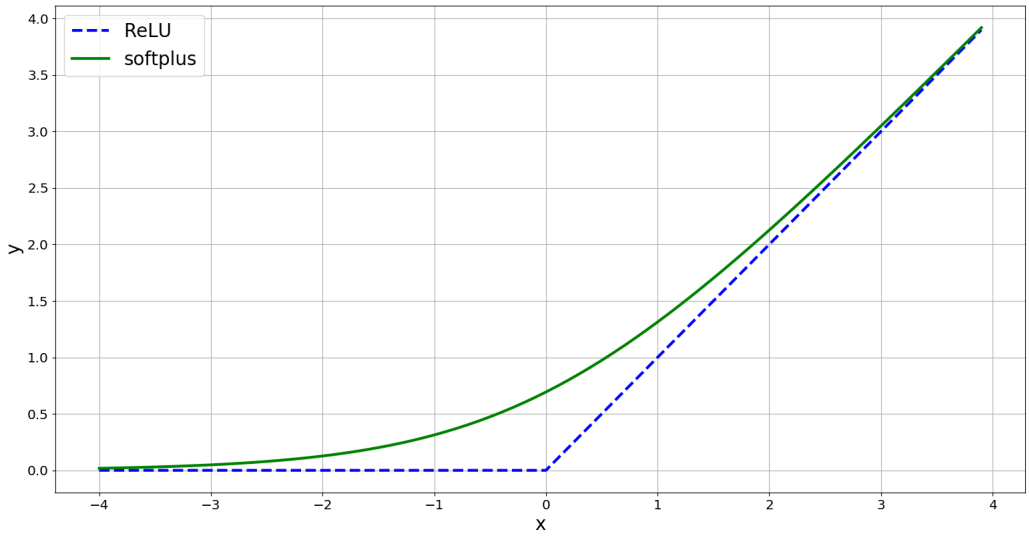


Рисунок 1 – Сравнение графиков функций `ReLU` и `softplus`

Как нетрудно убедиться, ассимптотами функции `softplus` действительно являются стороны угла, образованного графиком функции `ReLU`:

$$\begin{aligned}\lim_{x\to\infty}\frac{\varphi(x)}{x} &= \lim_{x\to\infty}\frac{\ln(1+e^x)}{x} = \lim_{x\to\infty}\frac{\ln'(1+e^x)}{x'} = \\ &= \lim_{x\to\infty}\frac{e^x}{1+e^x} = \lim_{x\to\infty}\left(1-\frac{1}{1+e^x}\right) = 1;\end{aligned}$$

$$\lim_{x\to-\infty}\varphi(x) = \lim_{x\to-\infty}\ln(1+e^x) = 0.$$

### 1.3 Архитектура рассматриваемой ИНС

Рассмотрим ИНС, представленную на рис. 2. Данная ИНС обладает свойством *полносвязности* [8] и состоит из трех слоев — входного, скрытого и выходного соответственно.

Рассмотрим каждый из слоев более подробно:

— входной слой представлен одним сенсорным [8] нейроном, предназначенным для передачи без изменения получаемого сигнала $x$ во все исходящие связи;

— скрытый слой содержит $N$ вычислительных нейронов, комбинированный вход $n_j$ каждого из которых выражается следующим равенством:

$$n_j = n_j(x) = xw_j + \theta_j,$$

где $j$ — номер нейрона, $\theta_j$ — его пороговое значение, а $w_j$ — вес связи со

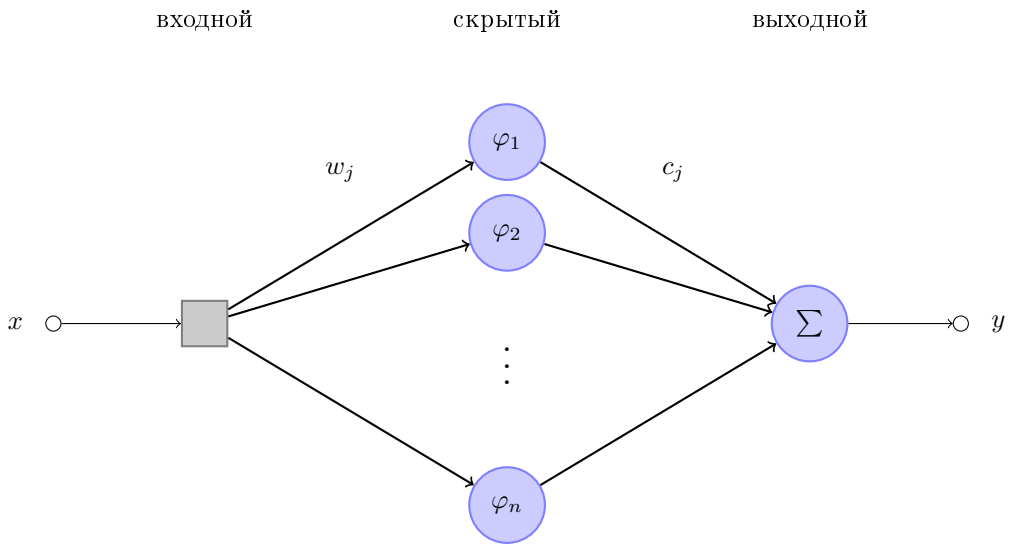


Рисунок 2 – Общий вид рассматриваемой ИНС

входным узлом. Также внутри каждого скрытого нейрона используется активационная функция `softplus` (далее $\varphi_j$), формирующая его выходное значение:

$$\varphi_j = \varphi_j(x) = \ln(1 + e^{n_j(x)});$$

— выходной слой состоит из единственного нейрона, не имеющего активационной функции, который производит на выход значение $y$, представляющее собой ответ ИНС на входной сигнал:

$$y = F(x) = \sum_{j=1}^{N} c_j \varphi_j(x).$$

В настоящей работе ИНС с вышеописанной архитектурой будет использоваться для приближения многочленов, определяемых следующим выражением:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_r x^r,$$

где $a_i$, $i = \overline{0, r}$ — коэффициенты многочлена, а $r$ — его степень.

### 1.4 Исследование применимости функции softplus

Следуя идеям из статьи [6], а также некоторым разъяснениям из работы [10], произведем доказательства справедливости следующего набора лемм

и теоремы для случая использования функции `softplus` в качестве активационной.

**Лемма 1.** *Производная порядка $s$ функции $F(x)$ может быть представлена следующим образом:*

$$\frac{d^s F(x)}{dx^s} = \sum_{j=1}^{N} c_j w_j^s Q_s(\psi_{kj}),$$

*где*

$$k = k(s) = \begin{cases} 1, & s > 0, \\ 0, & s = 0, \end{cases}$$

*функция $\psi_{kj}$ имеет вид*

$$\psi_{kj} = \psi_{kj}(x) = \begin{cases} \sigma_j(x), & k = 1, \\ \varphi_j(x), & k = 0, \end{cases}$$

*где $\sigma_j(x) = \frac{\varphi_j'(x)}{w_j}$, а $Q_s(\psi_{kj})$ выражается равенством*

$$Q_s(\psi_{kj}) = \begin{cases} \psi_{kj}(1 - \psi_{kj}) \frac{dQ_{s-1}(\psi_{kj})}{d\psi_{kj}}, & \text{для } s = 2, 3, \ldots, \\ \psi_{kj}, & \text{для } s = 0, 1. \end{cases}$$

*Доказательство.* Поскольку функция `softplus` является *первообразной* для *сигмоиды* [1], производная которой может быть выражена через исходную функцию [11], это означает, что данное свойство аналогичным образом можно применить и в случае `softplus`. Для этого вычислим ее производную:

$$\frac{d\varphi_j}{dx} = \frac{d \ln(1 + e^{w_j x + \theta_j})}{dx} = \frac{w_j e^{w_j x + \theta_j}}{1 + e^{w_j x + \theta_j}} = \varphi_j'.$$

Далее произведем следующую замену:

$$\frac{e^{w_j x + \theta_j}}{1 + e^{w_j x + \theta_j}} = \frac{1}{1 + e^{-(w_j x + \theta_j)}} = \sigma_j(x) = \sigma_j,$$

где $\sigma_j = \frac{\varphi_j'}{w_j}$ — *сигмоида*, для которой справедливо следующее свойство [6]:

$$\sigma_j' = w_j \sigma_j (1 - \sigma_j).$$

Теперь воспользуемся *методом математической индукции* для доказательства существования производной произвольного порядка функции $F(x)$.

— $s = 1$:

$$\begin{aligned} \frac{dF(x)}{dx} &= \sum_{j=1}^{N} c_j \frac{d\varphi_j}{dx} = \sum_{j=1}^{N} c_j w_j \sigma_j = \\ &= \sum_{j=1}^{N} c_j w_j Q_1(\sigma_j), \end{aligned}$$

где $Q_1(\sigma_j) = \sigma_j$.

— $s = 2$:

$$\begin{aligned} \frac{d^2F(x)}{dx^2} &= \sum_{j=1}^{N} c_j w_j \frac{d\sigma_j}{dx} = \sum_{j=1}^{N} c_j w_j^2 \sigma_j (1 - \sigma_j) = \\ &= \sum_{j=1}^{N} c_j w_j^2 Q_2(\sigma_j), \end{aligned}$$

где $Q_2(\sigma_j) = \sigma_j(1 - \sigma_j)\frac{dQ_1(\sigma_j)}{d\sigma_j}$, а $\frac{dQ_1(\sigma_j)}{d\sigma_j} = 1$.

— Предположим, что если $s = m$, то справедливо

$$\frac{d^mF(x)}{dx^m} = \sum_{j=1}^{N} c_j w_j^m Q_m(\sigma_j),$$

тогда при $s = m + 1$ имеем

$$\begin{aligned} \frac{d^{m+1}F(x)}{dx^{m+1}} &= \sum_{j=1}^{N} c_j w_j^m \frac{dQ_m(\sigma_j)}{dx} = \sum_{j=1}^{N} c_j w_j^m \frac{dQ_m(\sigma_j)}{d\sigma_j}\frac{d\sigma_j}{dx} = \\ &= \sum_{j=1}^{N} c_j w_j^{m+1} \sigma_j (1 - \sigma_j) \frac{dQ_m(\sigma_j)}{d\sigma_j} = \sum_{j=1}^{N} c_j w_j^{m+1} Q_{m+1}(\sigma_j), \end{aligned}$$

где

$$Q_{m+1}(\sigma_j) = \sigma_j(1-\sigma_j)\frac{dQ_m(\sigma_j)}{d\sigma_j}.$$

Таким образом, по индукции лемма будет справедлива для всех $s = 1, 2, \ldots$. □

*Примечание.* Пусть $\frac{d^s F(x)}{dx^s} = \frac{d^s f(x)}{dx^s}$ при $x = 0$.

Поскольку в точке $x = 0$ справедливо равенство $\frac{d^s f(x)}{dx^s} = s!a_s$, $s = \overline{0,r}$, то мы получаем следующее матричное выражение:

$$\underbrace{\begin{bmatrix} Q_0(\psi_{01}^0) & Q_0(\psi_{02}^0) & \ldots & Q_0(\psi_{0N}^0) \\ Q_1(\psi_{11}^0)w_1 & Q_1(\psi_{12}^0)w_2 & \ldots & Q_1(\psi_{1N}^0)w_N \\ \vdots & \vdots & \ddots & \vdots \\ Q_r(\psi_{11}^0)w_1^r & Q_r(\psi_{12}^0)w_2^r & \ldots & Q_r(\psi_{1N}^0)w_N^r \end{bmatrix}}_{\Omega} \cdot \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix}}_{c} = \underbrace{\begin{bmatrix} a_0 \\ 1!a_1 \\ \vdots \\ r!a_r \end{bmatrix}}_{a}, \qquad (1)$$

где $\psi_{kj}^0 = \psi_{kj}(0)$, $k = \overline{0,1}$, $j = \overline{1,N}$, а само выражение (1) можно представить в следующем виде:

$$\Omega c = a,$$

где $\Omega$ — крайняя левая матрица, $c = (c_1, c_2, \ldots, c_N)^T$, $a = (a_0, 1!a_1, \ldots, r!a_r)$.

**Лемма 2.** *Если $N = r+1$, то матрица $\Omega$ является невырожденной по крайней мере для одного набора $\psi_{kj}^0$, $k = \overline{0,1}$, $j = \overline{1,N}$, если $w_j \neq w_l$ при $j \neq l$ для $j, l = \overline{1,N}$.*

*Доказательство.* Обратим внимание на то, что при $N = r+1$ матрица $\Omega$ является квадратной. Докажем невырожденность матрицы $\Omega$, используя условие линейной независимости столбцов [12].

Пусть $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_N)$ — вектор произвольных скаляров. Обозначим $j$-ый столбец матрицы $\Omega$ через $\Omega_j$. И пусть $\sum_{j=1}^N \alpha_j \Omega_j = \overrightarrow{0}$, тогда справедливо

$$\sum_{j=1}^{N} \alpha_j Q_i(\psi_{kj}^0) w_j^i = 0, \quad i = \overline{0,r},\ k = k(i). \qquad (2)$$

Таким образом задача сводится к доказательству того, что равенство (2) выполняется только при $\alpha_j = 0$, $j = \overline{0, N}$.

Пусть $\psi^0_{kj} = \psi^0_k$, $k = \overline{0, 1}$, $j = \overline{1, N}$ такие, что $Q_i(\psi^0_k) \neq 0$, $i = \overline{0, r}$, тогда

$$Q_i(\psi^0_k) \sum_{j=1}^{N} \alpha_j w_j^i = 0, \quad i = \overline{0, r},$$

т.е.

$$\sum_{j=1}^{N} \alpha_j w_j^i = 0, \quad i = \overline{0, r}. \tag{3}$$

Перепишем полученное выражение (3) в матричном виде:

$$\begin{bmatrix} 1 & 1 & \dots & 1 \\ w_1 & w_2 & \dots & w_N \\ \vdots & \vdots & \ddots & \vdots \\ w_1^r & w_2^r & \dots & w_N^r \end{bmatrix} \cdot \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{4}$$

Заметим, что первая матрица выражения (4) является *матрицей Вандермонда*, определитель которой отличен от 0, если $w_j \neq w_l$ при $j \neq l$, $j, l = \overline{1, N}$. Т.е. при таком условии эта матрица является невырожденной, что эквивалентно условию линейной независимости ее строк, следовательно, $\alpha_1 = \alpha_2 = \dots = \alpha_N = 0$. □

**Лемма 3.** *Для каждого параметра $c_j$ из выражения* (1) *справедливо следующее неравенство:*

$$|c_j| \leqslant v_{\max} q_{\max} \sum_{i=0}^{r} |i! a_i| \delta^{-i}, \quad j = \overline{1, N},$$

*где $v_{\max}$ — положительное число, зависящее от $N$ и $r$, $q_{\max}$ — положительное число, зависящее от пороговых значений скрытых узлов, а $\delta$ выбрана таким образом, что $w_j = (j-1)\delta$ при $\delta > 0$.*

*Доказательство.* Пусть $\theta_j = \theta$, $w_j = (j-1)\delta$ для $j = \overline{1, N}$, где $\delta$ — положи-

тельная константа, тогда $\psi^0_{kj} = \psi^0_k$, $k = \overline{0,1}$, $j = \overline{1,N}$ и

$$\Omega = \begin{bmatrix} Q_0(\psi^0_{01}) & Q_0(\psi^0_{02}) & \dots & Q_0(\psi^0_{0N}) \\ Q_1(\psi^0_{11})w_1 & Q_1(\psi^0_{12})w_2 & \dots & Q_1(\psi^0_{1N})w_N \\ \vdots & \vdots & \ddots & \vdots \\ Q_r(\psi^0_{11})w_1^r & Q_r(\psi^0_{12})w_2^r & \dots & Q_r(\psi^0_{1N})w_N^r \end{bmatrix} =$$

$$= \begin{bmatrix} Q_0(\psi^0_0)\delta^0 & Q_0(\psi^0_0)\delta^0 & \dots & Q_0(\psi^0_0)\delta^0 \\ 0 & Q_1(\psi^0_1)\delta^1 & \dots & Q_1(\psi^0_1)(N-1)\delta^1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & Q_r(\psi^0_1)\delta^r & \dots & Q_r(\psi^0_1)(N-1)^r\delta^r \end{bmatrix} =$$

$$= \begin{bmatrix} Q_0(\psi^0_0)\delta^0 & 0 & \dots & 0 & 0 \\ 0 & Q_1(\psi^0_1)\delta^1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & Q_r(\psi^0_1)\delta^r \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & \dots & 1 \\ 0 & 1 & \dots & N-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0^r & 1^r & \dots & (N-1)^r \end{bmatrix}.$$

Таким образом,

$$\Omega^{-1} = \underbrace{\begin{bmatrix} 1 & 1 & \dots & 1 \\ 0 & 1 & \dots & N-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0^r & 1^r & \dots & (N-1)^r \end{bmatrix}^{-1}}_{V^{-1}} \times$$

$$\times \underbrace{\begin{bmatrix} \delta^{-0}/Q_0(\psi^0_0) & 0 & \dots & 0 & 0 \\ 0 & \delta^{-1}/Q_1(\psi^0_1) & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & \delta^{-r}/Q_r(\psi^0_1) \end{bmatrix}}_{Q},$$

т.е. $\Omega^{-1} = V^{-1} \cdot Q$, где $V$ — известная невырожденная матрица с параметрами $N$ и $r$, а $Q$ — диагональная матрица.

Введем следующие обозначения:

— $v_{\max} = \max\left\{|V^{-1}_{ij}| : i = \overline{0,r},\ j = \overline{0,N-1}\right\}$ — максимальный по модулю элемент матрицы $V^{-1}$;

— $q_{\max} = \max\left\{|1/Q_0(\psi^0_0)|, |1/Q_1(\psi^0_1)|, \dots, |1/Q_r(\psi^0_1)|\right\}$.

Тогда очевидно, что для заданных $\theta$, $N$ и $r$ значения $v_{\max}$ и $q_{\max}$ — известные положительные числа. Таким образом, из выражения (1) имеем

$$|c_j| \leqslant v_{\max} q_{\max} \sum_{i=0}^{r} |i! a_i| \delta^{-i}, \quad j = \overline{1, N}.$$

□

Используя утверждения, полученные в леммах 1–3, сформулируем и докажем теорему, позволяющую оценить *емкость ИНС*, под которой понимается наибольшая степень многочленов, которые данная нейронная сеть приближает с наперед заданной точностью.

**Теорема 1** (О емкости ИНС)**.** *Однослойная ИНС с $N > 0$ скрытыми нейронами может приближать любой многочлен $f(x)$ степени $N-1$ с любой точностью при $|x| < x_{\max}$. Т.е. ИНС может принять такую описывающую ее функцию $F(x)$, что $|F(x) - f(x)| < \varepsilon$, где $\varepsilon$ — требуемая верхняя граница ошибки приближения.*

*Доказательство.* Пусть $r = N - 1$. Поскольку $F(x)$ имеет любой порядок производной (лемма 1), то мы можем разложить $F(x)$ в *ряд Тейлора* [13]:

$$F(x) = F(0) + F'(0)x + \left(\frac{1}{2!}\right) F''(0)x^2 + \cdots + \left(\frac{1}{r!}\right) F^{(r)}(0)x^r + R_r(x),$$

где $R_r(x)$ — погрешность разложения:

$$R_r(x) = \frac{1}{(r+1)!} F^{(r+1)}(\zeta) x^{r+1} = \frac{x^{r+1}}{(r+1)!} \sum_{j=1}^{N} c_j w_j^{r+1} Q_{r+1}(\psi_{kj}(\zeta)),$$

где $-x_{\max} \leqslant \zeta \leqslant x_{\max}$, $k = k(r)$.

Пусть $w_j = (j-1)\delta$ для $j = \overline{1, N}$ и $0 < \delta < 1$. Из леммы 1: $Q_s(\psi_{kj}) = \psi_{kj}(1 - \psi_{kj}) \frac{dQ_{s-1}(\psi_{kj})}{d\psi_{kj}}$ для $s = 2, 3, \ldots$, и $Q_s(\psi_{kj}) = \psi_{kj}$ для $s = 0, 1$, $k = k(s)$. Поскольку $Q_s(\psi_{kj})$ — полиномиальная функция от $\psi_{kj}$, и $\psi_{kj}$ — всегда положительна и меньше единицы для любого $x$ при $s > 0$, то существует положительное число $M^0$ такое, что $|Q_s(\psi_{kj})| \leqslant M^0$ для конечного $s > 0$ и любого значения $\psi_{kj}$.

Тогда

$$\begin{aligned}
|R_r(x)| &\leqslant \frac{x_{\max}^{r+1}}{(r+1)!}\sum_{j=1}^{N}\Big(|c_j|\cdot|w_j^{r+1}|\cdot|Q_{r+1}(\psi_{kj}(\zeta))|\Big)\leqslant \\
&\leqslant \frac{x_{\max}^{r+1}}{(r+1)!}v_{\max}q_{\max}\sum_{j=1}^{N}\left(\sum_{i=0}^{r}\Big(|i!a_i|\delta^{-i}\Big)\cdot j^{r+1}\delta^{r+1}\cdot M^0\right)\leqslant \\
&\leqslant \delta\frac{x_{\max}^{r+1}}{(r+1)!}v_{\max}q_{\max}M^0\cdot\sum_{i=0}^{r}|i!a_i|\cdot\sum_{j=1}^{N}j^{r+1}.
\end{aligned}$$

Обозначим

$$M=\frac{x_{\max}^{r+1}}{(r+1)!}v_{\max}q_{\max}M^0\sum_{i=0}^{r}|i!a_i|\cdot\sum_{j=1}^{N}j^{r+1}.$$

Следовательно, $M$ — конечное число, зависящее от выбора параметров $N$, $r$ и $\theta$. Таким образом

$$|R_r(x)|\leqslant\delta M.$$

Если взять $\delta<\frac{\varepsilon}{M+1}$, то

$$|R_r(x)|<\varepsilon,$$

где $\varepsilon$ — верхняя граница ошибки приближения.

Поэтому для любого многочлена $f(x)$ степени $r$ мы можем использовать уравнение (1) для построения такой ИНС, что

$$|F(x)-f(x)|<\varepsilon,$$

поскольку для выбранных параметров $N$, $r$ и $\theta$ матрица $\Omega$ является невырожденной (лемма 2). □

### 1.5 Алгоритм вычисления весов ИНС

Результаты из приведенных выше доказательств могут быть использованы для построения *алгоритма вычисления весов ИНС*, приближающей *заданный* многочлен.

*Алгоритм вычисления весов ИНС* принимает на вход следующие параметры:

— список коэффициентов $a_i$, $i = \overline{0, r}$ приближаемого многочлена, где $r$ — его степень;

— значение требуемой верхней границы $\varepsilon$ ошибки приближения;

— максимальная по модулю граница $x_{\max}$ отрезка, на котором осуществляется приближение;

— шаг $d$ поиска подходящего порогового значения нейронов скрытого слоя.

*Алгоритм вычисления весов ИНС*

1. Согласно теореме 1 задать количество $N$ нейронов скрытого слоя равным $r + 1$.
2. Выбрать пороговое значение $\theta$ скрытых нейронов и вычислить значение функции $\psi_k^0$, $k = \overline{0, 1}$, определенной в лемме 3.
3. Вычислить значения $Q_s(\psi_k^0)$, $s = \overline{0, r}$ из леммы 3. Если хотя бы одно из получившихся значений равно нулю, то взять новое пороговое значение $\theta = \theta + d$ и повторить данный шаг.
4. Вычислить значения $v_{\max}$ и $q_{\max}$, определенные в лемме 3.
5. Вычислить значение $M$, определенное в теореме 1.
6. Задать значение $\delta = \frac{\varepsilon}{M+1}$ и вычислить веса $w_j = (j-1)\delta$, $j = \overline{1, N}$.
7. Вычислить веса $c_j$, $j = \overline{1, N}$, используя уравнение 1, таким образом, $c = \Omega^{-1}a$.

### 1.6 Итерационный алгоритм обучения ИНС

В настоящее время одним из самых популярных методов обучения ИНС является *алгоритм обратного распространения ошибки* [8], основная идея которого заключается в применении итерационного процесса, на каждом шаге которого производится вычисление *градиента*, используемого для коррекции весов нейронной сети с целью минимизации ошибки ее работы.

Каждая итерация алгоритма включает в себя два этапа:

— прямое распространение, при котором поданный *функциональный сигнал* распространяется от входов ИНС к ее выходам, посредством чего генерируется ответ сети;

— обратное распространение, при котором *сигнал ошибки*, вычисленный на выходе ИНС, распространяется по сети к ее входу, после чего на каждом слое осуществляется коррекция весов в соответствии с их влиянием

на значение *функции ошибки*.

В настоящей работе данный метод используется в режиме реализации *стохастического* градиентного спуска, при котором изменение весов производится для *каждого* экземпляра обучающей выборки. Выбор обусловлен тем, что такая реализация, по сравнению со своим *пакетным* вариантом, в общем случае позволяет добиться большей точности. Более подробно с алгоритмом обратного распространения ошибки можно ознакомиться в источнике [8].

## 2 Реализация ИНС

### 2.1 Определение представления ИНС в компьютере

Прежде чем приступить к реализации ИНС, необходимо определить способ ее представления в компьютере, т.е. установить, каким образом в памяти будет храниться ее состояние, и как будет производиться обучение сети и обработка запросов.

Для этого следует рассмотреть основные шаги алгоритма обратного распространения ошибки:

1) прямое распространение сигнала;
2) обратное распространение ошибки;
3) коррекция весов.

#### 2.1.1 Прямое распространение сигнала

Общий вид связи между соседними слоями $l$ и $p$, состоящими из $n$ и $m$ нейронов соответственно, представлен на рис. 3.

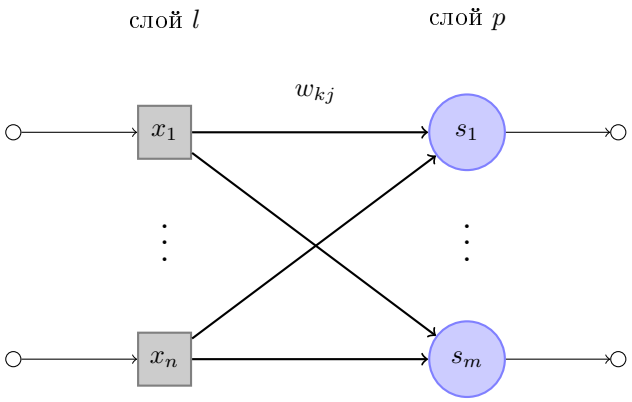


Рисунок 3 – Общий вид связи между соседними слоями $l$ и $p$, где $x_j$, $j = \overline{1,n}$ — выходы нейронов слоя $l$, $w_{kj}$, $k = \overline{1,m}$, $j = \overline{1,n}$ — вес связи $k$-го нейрона слоя $p$ с $j$-ым нейроном слоя $l$, $s_k$, $k = \overline{1,m}$ — комбинированные входы соответствующих нейронов слоя $p$

Комбинированный вход $k$-го нейрона слоя $p$ можно вычислить по следующей формуле:

$$s_k = \sum_{j=1}^{n} w_{kj} x_j + b_k, \quad k = \overline{1,m}. \tag{5}$$

Однако такой способ представления порогового значения $b_k$ не самый удачный по причине неудобного представления в памяти компьютера. Поэтому порог часто представляют как весовой коэффициент связи с дополнитель-

ным нейроном, на вход которого всегда подается единица [8]. Таким образом получается структура ИНС, изображенная на рис. 4, а выражение (5) принимает вид:

$$s_k = \sum_{j=1}^{n+1} w_{kj} x_j, \quad k = \overline{1, m}, \tag{6}$$

где $w_{k,n+1} = b_k$ — пороговое значение $k$-го нейрона.

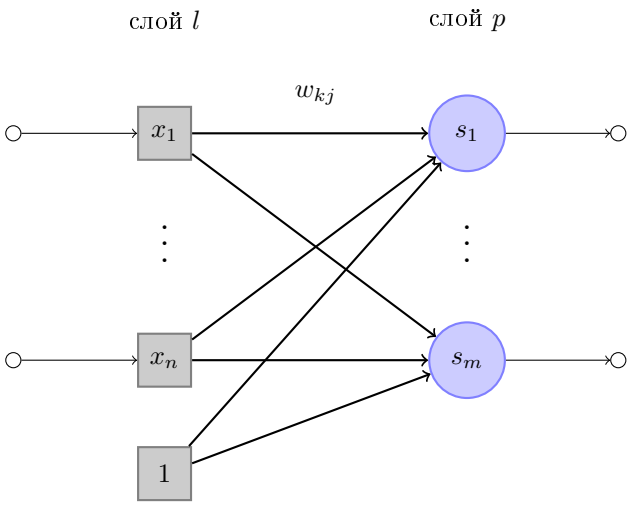


Рисунок 4 – Представление пороговых значений в виде весовых коэффициентов связей с дополнительным нейроном

Преобразуем полученное выражение (6) в матричное произведение:

$$\underbrace{\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1,n+1} \\ w_{21} & w_{22} & \dots & w_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{m,n+1} \end{bmatrix}}_{W} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n+1} \end{bmatrix}}_{X} = \underbrace{\begin{bmatrix} w_{11}x_1 + w_{12}x_2 + \dots + w_{1,n+1}x_{n+1} \\ w_{21}x_1 + w_{22}x_2 + \dots + w_{2,n+1}x_{n+1} \\ \vdots \\ w_{m1}x_1 + w_{m2}x_2 + \dots + w_{m,n+1}x_{n+1} \end{bmatrix}}_{S}.$$

Таким образом справедливо следующее равество:

$$S = W \cdot X, \tag{7}$$

где $S_{m\times 1}$ — вектор-столбец комбинированных входов нейронов слоя $p$, $W_{m\times(n+1)}$ — матрица весовых коэффициентов входных связей нейронов слоя $p$, $X_{(n+1)\times 1}$ — вектор-столбец выходов нейронов слоя $l$.

Помимо компактной записи и удобства представления в компьютере,

использование матриц предпочтительно по той причине, что операции над ними поддаются существенной оптимизации, позволяющей производить вычисления значительно быстрее, чем если бы они выполнялись в обычном цикле.

### 2.1.2 Обратное распространение ошибки

Общий вид обратной связи между соседними слоями $p$ и $l$, состоящими из $m$ и $n$ нейронов соответственно, представлен на рис. 5.

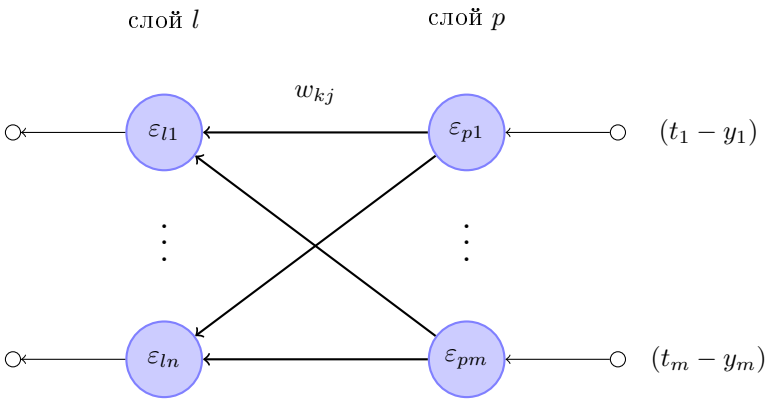


Рисунок 5 – Общий вид обратной связи между соседними слоями $p$ и $l$, где $w_{kj}$, $k = \overline{1,m}$, $j = \overline{1,n}$ — вес связи $k$-го нейрона слоя $p$ с $j$-ым нейроном слоя $l$, $y_k$, $k = \overline{1,m}$ — фактические выходы соответствующих нейронов слоя $p$, $t_k$, $k = \overline{1,m}$ — желаемые выходы соответствующих нейронов слоя $p$, $\varepsilon_{pk}$, $k = \overline{1,m}$ — величина ошибки $k$-го нейрона слоя $p$, $\varepsilon_{lj}$, $j = \overline{1,m}$ — величина ошибки $j$-го нейрона слоя $l$

Величина $\varepsilon_{pk}$ ошибки $k$-го нейрона слоя $p$ вычисляется по следующей формуле:

$$\varepsilon_{pk} = t_k - y_k, \quad k = \overline{1,m}.$$

Тогда, если произвести распространение ошибки от выходных нейронов к входным, то ошибку $j$-го нейрона слоя $l$ можно рассчитать по формуле:

$$\varepsilon_{lj} = \sum_{k=1}^{m} w_{kj}\varepsilon_{pk}, \quad j = \overline{1,n},$$

что эквивалентно следующей записи в матричной форме:

$$\underbrace{\begin{bmatrix} w_{11} & w_{21} & \dots & w_{m1} \\ w_{12} & w_{22} & \dots & w_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1n} & w_{2n} & \dots & w_{mn} \end{bmatrix}}_{W^T} \cdot \underbrace{\begin{bmatrix} \varepsilon_{p1} \\ \varepsilon_{p2} \\ \vdots \\ \varepsilon_{pm} \end{bmatrix}}_{E_p} = \underbrace{\begin{bmatrix} w_{11}\varepsilon_{p1} + w_{21}\varepsilon_{p2} + \dots + w_{m1}\varepsilon_{pm} \\ w_{12}\varepsilon_{p1} + w_{22}\varepsilon_{p2} + \dots + w_{m2}\varepsilon_{pm} \\ \vdots \\ w_{1n}\varepsilon_{p1} + w_{2n}\varepsilon_{p2} + \dots + w_{mn}\varepsilon_{pm} \end{bmatrix}}_{E_l}.$$

Заметим, что крайняя левая матрица полученного выражения равна транспонированной матрице весов из формулы (7), таким образом

$$E_l = W^T \cdot E_p,$$

где $E_l$ — вектор-столбец ошибок нейронов слоя $l$, $W^T$ — транспонированная матрица весовых коэффициентов входных связей нейронов слоя $p$, $E_p$ — вектор-столбец ошибок нейронов слоя $p$.

### 2.1.3 Коррекция весов

Общий вид связи между соседними слоями $l$ и $p$ представлен на рис. 6.

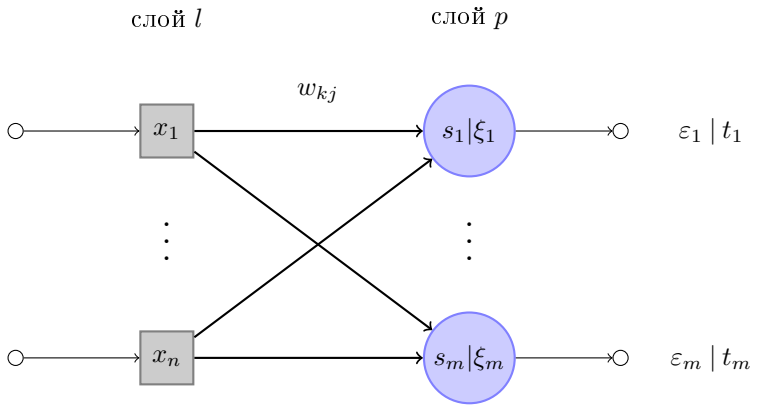


Рисунок 6 – Общий вид связи между соседними слоями $l$ и $p$, где $x_j$, $j = \overline{1,n}$ — выходы нейронов слоя $l$, $s_k$, $k = \overline{1,m}$ — комбинированные входы нейронов слоя $p$, $\xi_k$, $k = \overline{1,m}$ — выходы нейронов слоя $p$, $\varepsilon_k$, $k = \overline{1,m}$ — величина ошибки $k$-го нейрона слоя $p$

*Функция среднеквадратической ошибки* [14] сети на слое $p$ выражается следующим равенством [8]:

$$E = \frac{1}{2}\sum_{k=1}^{m} \varepsilon_k^2,$$

где $\varepsilon_k = t_k - \xi_k$, $k = \overline{1,m}$ — ошибка $k$-го нейрона слоя $p$.

Коррекцию весов будем осуществлять в соответствии со следующим правилом [8]:

$$w_{kj}(n+1) = w_{kj}(n) - \alpha \frac{dE}{dw_{kj}(n)},$$

где $\alpha$ — коэффициент скорости обучения, а $w_{kj}(n)$ и $w_{kj}(n+1)$ — значения весов после итерации $n$ и $n+1$ соответственно.

Для нахождения частной производной $\frac{dE}{dw_{kj}}$ воспользуемся *цепным правилом* дифференцирования сложной функции [8]:

$$\begin{aligned}
\frac{dE}{dw_{kj}} &= \frac{dE}{d\varepsilon_k} \cdot \frac{d\varepsilon_k}{d\xi_k} \cdot \frac{d\xi_k}{ds_k} \cdot \frac{ds_k}{dw_{kj}} = \\
&= \frac{1}{2} \frac{d\sum_{i=1}^{m} \varepsilon_i^2}{d\varepsilon_k} \cdot \frac{d(t_k - \xi_k)}{\xi_k} \cdot \frac{d\xi_k}{ds_k} \cdot \frac{d\sum_{i=1}^{n} w_{ki} x_i}{dw_{kj}} = \\
&= -\varepsilon_k \cdot \frac{d\xi_k}{ds_k} \cdot x_j,\ k = \overline{1,m},\ j = \overline{1,n},
\end{aligned}$$

где $\varepsilon_k$ — величина ошибки $k$-го нейрона слоя $p$, $x_j$ — выход $j$-го нейрона слоя $l$, а $\frac{d\xi_k}{ds_k}$ — частная производная выхода $k$-го нейрона по его комбинированному входу, значение которой зависит от используемой функции активации.

Теперь рассмотрим два случая, характерных для реализуемой ИНС:

— корректируемые нейроны не имеют функций активации, т.е. $\xi_k = s_k$, тогда

$$\frac{dE}{dw_{kj}} = -\varepsilon_k \cdot \frac{ds_k}{ds_k} \cdot x_j = -\varepsilon_k \cdot x_j,\ k = \overline{1,m},\ j = \overline{1,n}; \tag{8}$$

— корректируемые нейроны имеют функции активации, т.е. $\xi_k = \varphi(s_k)$, тогда

$$\frac{dE}{dw_{kj}} = -\varepsilon_k \cdot \frac{d\varphi(s_k)}{ds_k} \cdot x_j = -\varepsilon_k \cdot \varphi'(s_k) \cdot x_j,\ k = \overline{1,m},\ j = \overline{1,n}, \tag{9}$$

где $\varphi'(s_k) = \frac{1}{1+e^{-s_k}}$ — значение производной активационной функции `softplus` для комбинированного входа $s_k$.

## 2.2 Используемые инструменты

Для реализации описанной выше ИНС был выбран язык программирования `Python` версии 3 [15], поскольку он является высокоуровневым, простым и минималистичным, что позволяет сосредоточиться на решении задачи, а не на самом языке и особенностях используемой платформы.

К тому же, для языка `Python` существует множество готовых инструментов и библиотек, позволяющих быстро производить математические операции, а также осуществлять визуализацию имеющихся данных.

Таким образом, в настоящей работе вместе с собственными реализованными классами будут использоваться следующие модули:

— `sys` [16], обеспечивающий доступ к некоторым системным переменным и функциям;

— `random` [17], предоставляющий функции для генерации случайных чисел;

— `math` [18], предназначенный для работы с числами и содержащий множество констант;

— библиотека `NumPy` [19], позволяющая эффективно работать с многомерными матрицами;

— библиотека `Matplotlib` [20], с помощью которой можно визуализировать данные в виде графиков.

## 2.3 Класс ИНС

Предназначенная для приближения многочленов ИНС, описанная в разделе 1.3, была реализована в виде класса `NeuralNetwork`, общий вид которого представлен ниже:

```
1  # ИНС
2  class NeuralNetwork:
3
4      # Инициализация
5      def __init__(self)
6
7      # Указание весов
8      def set_ws(self)
9
10     # Обучение
11     def train(self)
12
13     # Прогноз
```

```
14      def predict(self)
```

Таким образом, класс `NeuralNetwork` включает в себя конструктор и 3 метода:

1) конструктор `__init__` по указанным параметрам формирует архитектуру нейронной сети;
2) метод `set_ws` позволяет явно задать весовые коэффициенты;
3) метод `train` осуществляет обучение сети на основе переданных в качестве параметров входного и желаемого выходного сигналов;
4) метод `predict` предназначен для осуществления прогноза.

Далее каждый метод рассматривается более подробно.

### 2.3.1 Инициализация

Конструктор, выполняющий инициализацию ИНС, имеет следующую сигнатуру:

```
def __init__(self, input_node_count, hidden_node_count,
                   output_node_count, learning_rate)
```

где `input_node_count`, `hidden_node_count`, `output_node_count` — количества входных, скрытых и выходных нейронов соответственно,
`learning_rate` — коэффициент скорости обучения.

Процесс инициализации ИНС включает в себя следующие шаги:

— сохранение указанного коэффициента скорости обучения:

```
self.l_rate = learning_rate
```

— установка `softplus` в качестве функции активации, используемой нейронами скрытого слоя:

```
self.act_fun = np.vectorize(lambda s: math.log(1 + math.exp(s)))
```

— определение частной производной функции ошибки для *выходного* слоя, представленной формулой (8) в разделе 2.1.3:

```
self.error_der_out = lambda e, x: np.dot(-1 * e, np.transpose(x))
```

— определение частной производной функции ошибки для *скрытого* слоя, представленной формулой (9) в разделе 2.1.3:

```
act_fun_der = np.vectorize(lambda s: 1 / (1 + math.exp(-s)))
self.error_der_hid = lambda e, s, x: np.dot(-1 * e * act_fun_der(s),
                                            np.transpose(x))
```

— установка случайных значений для весовых коэффициентов:

```
self.hl_ws = np.random.normal(0.0, pow(hidden_node_count, -0.5),
                              (hidden_node_count, input_node_count + 1))
self.ol_ws = np.random.normal(0.0, pow(output_node_count, -0.5),
                              (output_node_count, hidden_node_count))
```

Здесь `hl_ws` и `ol_ws` представляют собой матрицы весовых коэффициентов нейронов скрытого и выходного слоев соответственно.

Значения весов выбираются из нормального распределения с математическим ожиданием, равным нулю, и стандартным отклонением, величина которого обратно пропорциональна квадратному корню из количества входящих связей на узел. Данный прием инициализации является более предпочтительным, чем обычная генерация случайных чисел в определенном диапазоне, т.к. он способствует предотвращению чрезмерного разрастания весов [8, 21]. Важно обратить внимание на то, что на данном этапе мы также добавляем к исходной матрице весов вектор-столбец пороговых значений нейронов скрытого слоя.

### 2.3.2 Точное указание весовых коэффициентов

Метод `set_ws` выполняет функцию т.н. «сеттера» и предназначен для явного указания значений весовых коэффициентов сети. Данная возможность требуется при построении ИНС для приближения *заданного* многочлена и будет использована в разделе 2.7.2. Листинг данного метода представлен ниже:

```
1 def set_ws(self, hl_ws, ol_ws):
2     self.hl_ws = hl_ws
3     self.ol_ws = ol_ws
```

### 2.3.3 Прогнозирование

Метод `predict` осуществляет прогноз для переданного в качестве параметра вектора входных значений. Данный метод рассматривается раньше метода `train` по причине того, что первый содержит в себе часть функционала последнего.

Метод `predict` имеет следующую сигнатуру:

```
def predict(self, input_list)
```

где `input_list` представляет собой входной вектор значений.

Прогнозирование осуществляется путем описанного в разделе 2.1.1 прямого распространения поданного сигнала от входных нейронов ИНС к выход-

ным и состоит из следующих шагов:

— сначала ко входному вектору добавляется константа, равная 1, являющаяся значением дополнительного нейрона, предназначеного для реализации смещения. Затем производится транспонирование входных векторов, для того, чтобы преобразовать их в вектор-столбцы:

```
input_list.append(1)
input_list = np.array(input_list, ndmin=2).T
```

— затем путем умножения матрицы весовых коэффициентов на вектор-столбец входных значений вычисляются комбинированные входы нейронов скрытого слоя, к которым сразу же поэлементно применяется функция активации:

```
hidden_layer_inputs = np.dot(self.hl_ws, input_list)
hidden_layer_outputs = self.act_fun(hidden_layer_inputs)
```

— после чего аналогичным образом вычисляются комбинированные входы выходного слоя, но на этот раз функция активации к результату уже не применяется по причине ее отсутствия:

```
final_layer_inputs = np.dot(self.ol_ws, hidden_layer_outputs)
final_layer_outputs = final_layer_inputs
```

— результат возвращается в качестве прогноза:

```
return final_layer_outputs
```

### 2.3.4 Обучение

Метод `train` осуществляет одну из самых важных способностей ИНС — обучение. Он имеет следующую сигнатуру:

```
def train(self, input_list, target_list)
```

где `input_list` — входной вектор значений, `target_list` — желаемый выходной вектор.

Процесс обучения включает в себя два этапа:

— прямое распространение сигнала, рассмотренное в методе `predict` (раздел 2.3.3):

```
1 input_list.append(1)
2 input_list = np.array(input_list, ndmin=2).T
3 target_list = np.array(target_list, ndmin=2).T
4
5 hidden_layer_inputs = np.dot(self.hl_ws, input_list)
6 hidden_layer_outputs = self.act_fun(hidden_layer_inputs)
```

```
7
8 final_layer_inputs = np.dot(self.ol_ws, hidden_layer_outputs)
9 final_layer_outputs = final_layer_inputs
```

— описанное в разделе 2.1.2 обратное распространение ошибки, при котором сначала вычисляется ошибка нейронов на выходном и скрытом слоях:

```
final_layer_errors = target_list - final_layer_outputs
hidden_layer_errors = np.dot(self.ol_ws.T, final_layer_errors)
```

После чего производится коррекция весов в соответствии с их влиянием на функцию ошибки (раздел 2.1.3):

```
self.ol_ws -= self.l_rate * self.error_der_out(final_layer_errors,
                                                hidden_layer_outputs)
self.hl_ws -= self.l_rate * self.error_der_hid(hidden_layer_errors,
                                                hidden_layer_inputs,
                                                input_list)
```

## 2.4 Класс менеджера данных

Для генерации тестовых и тренировочных наборов данных, их нормализации, а также визуализации полученных результатов в виде графиков, был реализован вспомогательный класс `PolyDataManager`, общий вид которого представлен ниже:

```
1  # Менеджер данных
2  class PolyDataManager:
3
4      # Нормализация
5      @staticmethod
6      def normalize(l1, l2)
7
8      # Денормализация
9      @staticmethod
10     def denormalize(list_pair, nc_pair)
11
12     # Вычисление значения многочлена в точке
13     @staticmethod
14     def comp_pln(cs, x0)
15
16     # Генерация тренировочного набора данных
17     @staticmethod
18     def generate_train_pairs(pln_cs, bounds, point_count)
19
20     # Генерация тестового набора данных
21     @staticmethod
```

```
22      def generate_test_pairs(pln_cs, bounds, point_count)
23
24      # Визуализация
25      @staticmethod
26      def plot(test_pairs, pred_pairs, x_nc, y_nc)
```

Класс `PolyDataManager` не имеет конструктора, а все его методы являются *статическими* и имеют следующие предназначения:

1) метод `normalize` производит нормализацию двух указанных списоков, масштабируя значения их элементов в диапазон $[-1, 1]$ с помощью общего нормирующего коэффициента;
2) метод `denormalize` принимает на вход пару списков и пару соответствующих нормирующих коэффициентов, по которым производит операцию, обратную нормализации;
3) метод `comp_pln` принимает в качестве параметров список коэффициентов многочлена и точку, в которой необходимо найти его значение, выполняет необходимые вычисления и возвращает результат;
4) метод `generate_train_pairs` предназначен для генерации *случайного* набора тренировочных экземпляров, представляющих собой список пар вида (вход, желаемый выход), по заданному своими коэффициентами многочлену;
5) метод `generate_test_pairs` предназначен для генерации *упорядоченной* последовательности тестовых экземпляров, представляющих собой список пар вида (вход, ожидаемый выход), по заданному своими коэффициентами многочлену;
6) метод `plot` предназначен для визуализации в виде графиков указанных наборов данных.

Как уже было отмечено, методы данного класса носят вспомогательный характер, поэтому подробно они рассматриваться не будут, более детально с ними можно ознакомиться в приложении Б.

### 2.5 Класс тестировщика ИНС

Для тестирования ИНС был реализован класс `NNTester`, общий вид которого представлен ниже:

```
1  # Тестировщик ИНС
2  class NNTester:
3
4      # Тестирование
```

```
5       @staticmethod
6       def test(nn, test_pairs, x_nc, y_nc)
```

Как видно, класс `NNTester` содержит в себе единственный *статический* метод `test`, предназначенный для оценивания качества способности ИНС совершать прогнозы. В список параметров метода `test` входят ИНС, которую необходимо проверить, а так же соответствующие тестовые данные и нормирующие коэффициенты для них.

Процесс тестирования ИНС осуществляется следующим образом:

— сначала задаются начальные значения для минимальной, максимальной и средней ошибок соответственно, создается пустой список, предназначенный для хранения ответов ИНС, представляющих собой пары вида (вход, прогноз):

```
1 min_error = sys.float_info.max
2 max_error = sys.float_info.min
3 avg_error = 0
4 pred_pairs = []
```

— в цикле для каждой пары из тестового набора сначала производится опрос сети для получения ее ответа на входной сигнал, затем этот ответ сохраняется. Следом за этим вычисляется ошибка, представляющая собой погрешность ответа ИНС по отношению к целевому значению, после чего осуществляется обновление статистики ошибок:

```
1 for test_pair in test_pairs:
2     pred = nn.predict([test_pair[0]])[0][0]
3     pred_pairs.append((test_pair[0], pred))
4     error = abs(test_pair[1] - pred)
5     if error < min_error:
6         min_error = error
7     if error > max_error:
8         max_error = error
9     avg_error += error
```

— после выхода из цикла производится довычисление среднего арифметического значения всех ошибок. Затем осуществляется денормализация значений ошибок и их последующий вывод в консоль:

```
1 avg_error /= len(test_pairs)
2 print("Результаты тестирования:")
3 print("min error:", min_error * y_nc)
4 print("max error:", max_error * y_nc)
5 print("avg error:", avg_error * y_nc)
```

— в завершение работы метода для визуального сравнения полученных ответов ИНС с целевыми строится график:

```
1 pdm.plot(test_pairs, pred_pairs, x_nc, y_nc)
```

## 2.6 Класс учителя ИНС

Для реализации регулируемого процесса обучения описанной в разделе 2.3 ИНС был создан отдельный класс `NNPolyTeacher`, предназначенный для формирования сети, приближающей *некоторую* полиномиальную функцию, заданную набором дискретных значений.

Общий вид класса `NNPolyTeacher` представлен ниже:

```
1  # Учитель ИНС
2  class NNPolyTeacher:
3
4      # Инициализация
5      def __init__(self)
6
7      # Обучение
8      def teach(self)
9
10     # Тестирование
11     def test(self)
```

Таким образом, класс `NNPolyTeacher` включает в себя конструктор и 2 метода:

1) конструктор `__init__` по указанным параметрам производит инициализацию тренировочных и тестовых данных;
2) метод `teach` осуществляет обучение ИНС;
3) метод `test` предназначен для оценки способности ИНС совершать верные прогнозы.

Далее каждый метод рассматривается более подробно.

### 2.6.1 Инициализация

Конструктор, выполняющий инициализацию тренировочных и тестовых данных, используемых для обучения ИНС, имеет следующую сигнатуру:

```
def __init__(self, pln_cs, bounds, point_counts)
```

где `pln_cs` — список коэффициентов приближаемого многочлена, `bounds` — границы аппроксимации, `point_count` — требуемые количества тренировочных и тестовых экземпляров.

Процесс инициализации объекта данного класса включает в себя следующие шаги:

— генерация тренировочного и тестового наборов данных:

```
train_x, train_y = pdm.generate_train_pairs(pln_cs, bounds,
                                            point_counts[0])
test_x, test_y = pdm.generate_test_pairs(pln_cs, bounds,
                                         point_counts[1])
```

— нормализация данных и сохранение нормирующих коэффициентов:

```
train_x, test_x, self.x_nc = pdm.normalize(train_x, test_x)
train_y, test_y, self.y_nc = pdm.normalize(train_y, test_y)
```

Стоит отметить, что нормализация данных является очень важной, сильно влияющей на процесс обучения операцией, определение которой зависит от используемой функции активации [8, 21];

— «сшивание» полученных данных в списки пар:

```
self.train_pairs = list(zip(train_x, train_y))
self.test_pairs = list(zip(test_x, test_y))
```

### 2.6.2 Обучение

Метод `teach` предназначен для управления обучением указанной ИНС и имеет следующую сигнатуру:

```
def teach(self, nn, epoch_count)
```

где `nn` — ИНС, которую требуется обучить, `epoch_count` — количество эпох, на протяжении которых будет обучаться ИНС.

Тело данного метода состоит из одного двойного цикла, в котором на каждой эпохе обучения сети последовательно передаются все экземпляры вида (вход, выход) тренировочных данных, после чего с целью повышения качества обучения они перемешиваются [8]. Ниже представлен листинг тела метода `teach`:

```
1 for i in range(epoch_count):
2     for train_pair in self.train_pairs:
3         nn.train([train_pair[0]], [train_pair[1]])
4     random.shuffle(self.train_pairs)
```

### 2.6.3 Тестирование

Для тестирования ИНС используется рассмотренный в разделе 2.5 *статический* метод класса `NNTester`, которому в качестве параметров переда-

ются сама сеть и тестовый набор данных с нормирующими коэффициентами. Листинг метода `test` представлен ниже:

```
1 def test(self, nn):
2     NNTester.test(nn, self.test_pairs, self.x_nc, self.y_nc)
```

### 2.6.4 Запуск обучения и тестирования

После ознакомления с особенностями реализации алгоритма обучения и опроса ИНС перейдем к обзору результатов работы описанных классов.

Для примера в качестве приближаемого многочлена возьмем функцию $f(x) = 1 - 15x + x^2 + x^3$, график которой представляет собой кубическую параболу. Будем рассматривать эту функцию на отрезке $[-5, 5]$, используя 30 случайно выбранных точек для обучения ИНС и 100 последовательных точек для проверки. Согласно доказанной в разделе 1.4 теореме 1, оптимальным количеством $n$ скрытых нейронов для такой сети будет $n = r + 1$, где $r$ — степень рассматриваемого многочлена.

Для того, чтобы запустить обучение сети, необходимо сначала создать ее экземпляр с помощью конструктора. В нашем случае создание ИНС будет выглядеть следующим образом:

```
nn = NeuralNetwork(1, 4, 1, 0.1)
```

где коэффициент скорости обучения, равный 0.1, был задан на основе результатов серии экспериментов.

Затем аналогичным образом необходимо создать экземпляр учителя нейронной сети:

```
teacher = NNPolyTeacher([1, -15, 1, 1], (-5, 5), (30, 100))
```

После чего, предварительно выбрав количество эпох, можно запускать обучение ИНС:

```
teacher.teach(nn, 5000)
```

Как только обучение завершено, получившуюся сеть следует протестировать:

```
teacher.test(nn)
```

В результате чего должен построиться график, схожий с тем, что представлен на рис. 7.

Скорость и качество процесса обучения, в зависимости от начальных значений весовых коэффициентов, распределения тренировочных данных,

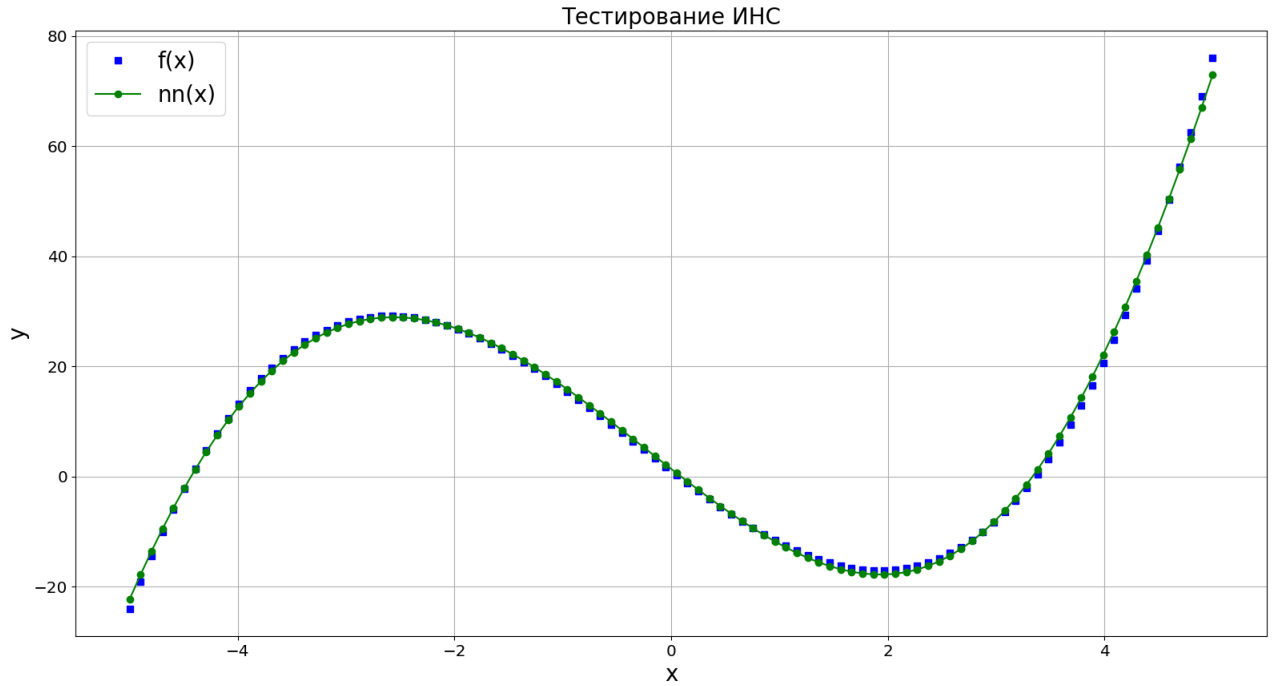


Рисунок 7 – Результат работы ИНС на тестовой выборке после процесса обучения, где $f(x) = 1 - 15x + x^2 + x^3$ — приближаемый многочлен, $nn(x)$ — функция, построенная ИНС. Результаты ошибок:
$\min \approx 0.008$, $\max \approx 3.06$, $\text{avg} \approx 0.55$

результатов осуществления градиентного спуска, погрешности округлений и прочих влияющих на него факторов, при разных запусках всегда отличаются, однако в общем случае, прежде чем выдать результат, представленный на рис. 7, ИНС от эпохи к эпохе последовательно проходит состояния, отображенные на рис. 8 и 9 соответственно. Но в отдельных случаях по причине критического сочетания вышеперечисленных факторов может произойти т.н. «паралич» сети (рис. 10), который проявляется в том, что с увеличением количества пройденных эпох качество работы ИНС остается почти неизменным, т.е. сеть перестает обучаться.

И все же, несмотря на вышеперечисленные особенности, присущие любой нейронной сети, результаты, представленные на рис. 7–9 демонстрируют жизнеспособность такого подхода к построению ИНС для приближения многочленов.

### 2.7 Класс вычислителя ИНС

Для реализации полученного в разделе 1.5 однопроходного алгоритма вычисления весовых коэффициентов ИНС был создан отдельный класс `NNPolyBuilder`, предназначенный для формирования сети, приближающей с

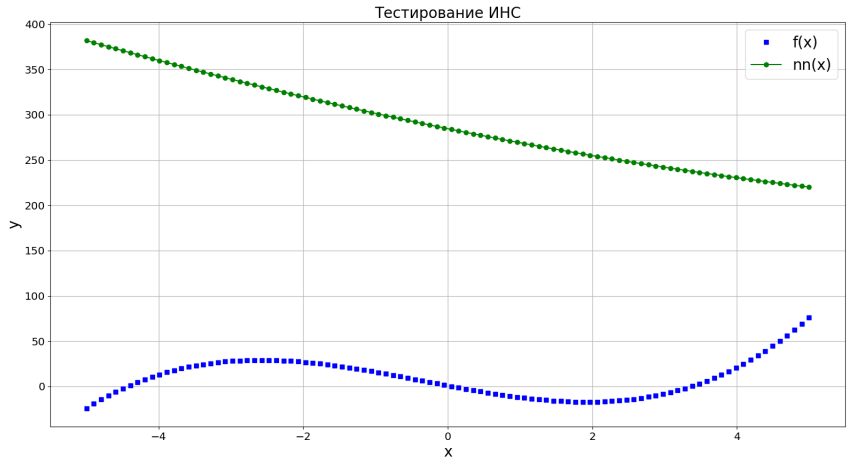


Рисунок 8 – Производительность ИНС на начальных эпохах

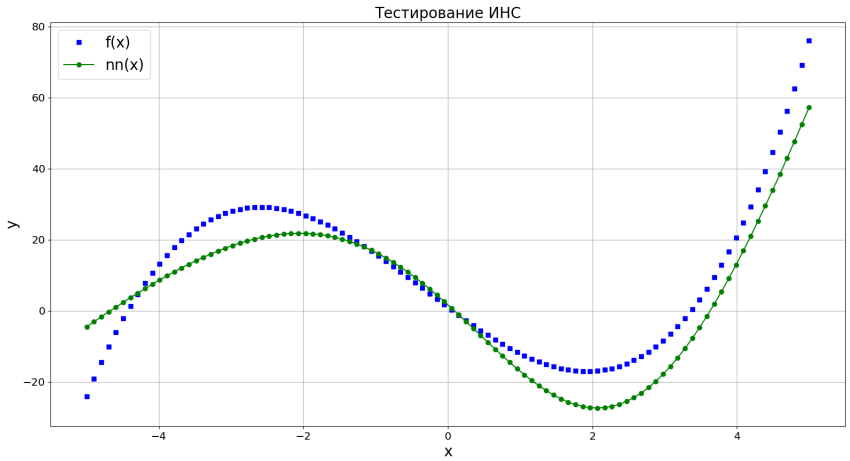


Рисунок 9 – Производительность ИНС на средних эпохах

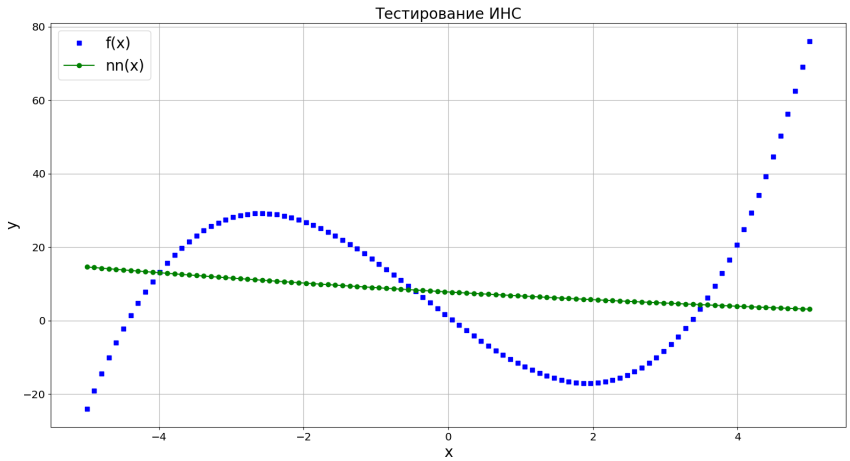


Рисунок 10 – Пример наступления «паралича» ИНС

указанной точностью *заданную* полиномиальную функцию.

Общий вид класса `NNPolyBuilder` представлен ниже:

```
1 # Вычислитель ИНС
2 class NNPolyBuilder:
```

```
3
4       # Инициализация
5       def __init__(self)
6
7       # Вспомогательные методы
8       def build_inv_v(self)
9       def der(self)
10      def wrapper(self)
11      def build_q_base(self)
12      def build_q_diag(self)
13      def comp_sum1(self)
14      def comp_sum2(self)
15      def comp_ws(self)
16      def build_omega(self)
17      def comp_cs(self)
18      def build_hl_ws(self)
19      def build_ol_ws(self)
20
21      # Вычисление весов ИНС
22      def build_for(self)
23
24      # Тестирование ИНС
25      def test(self)
```

Таким образом, класс `NNPolyBuilder` включает в себя конструктор и 2 основных метода:

1) конструктор `__init__` производит определение необходимых для алгоритма функций и значений;
2) метод `build_for` осуществляет вычисление весов ИНС;
3) метод `test` предназначен для тестирования ИНС и является аналогом одноименного метода класса `NNPolyTeacher`, рассмотренного в разделе 2.6;
4) остальные методы являются вспомогательными и производят вычисления промежуточных значений, используемых в алгоритме. Более подробно с ними можно ознакомиться в приложении Д.

Конструктор `__init__` и метод `build_for` далее рассматриваются более подробно.

### 2.7.1 Инициализация

Конструктор класса `NNPolyBuilder` имеет следующую сигнатуру:

```
def __init__(self, pln_cs)
```

где `pln_cs` — список коэффициентов приближаемого многочлена.

Процесс инициализации объекта данного класса включает в себя следующие шаги:

— определение необходимых для работы алгоритма функций, зависящих от порогового значения, в нашем случае это пара $\psi_k^0, k = \overline{0,1}$:

```
self.act_fun_0 = lambda t: math.log(1 + math.exp(t))
self.fun_0 = lambda t: 1 / (1 + math.exp(-t))
```

— установка начального порогового значения для нейронов скрытого слоя:

```
self.t_init = -1.4
```

— сохранение указанных коэффициентов приближаемого многочлена:

```
self.pln_cs = pln_cs
```

### 2.7.2 Вычисление весов

Метод `build_for` является центральным методом данного класса и реализует его основное предназначение — вычисление весовых коэффициентов ИНС для приближения с указанной точностью *заданного* многочлена.

Рассматриваемый метод имеет следующую сигнатуру:

```
def build_for(self, nn, eps, x_max, d)
```

где `nn` — ИНС, для которой требуется вычислить веса, `eps` — верхняя граница ошибки приближения, `x_max` — граница диапазона приближения многочлена, `d` — размер шага поиска подходящего порогового значения.

Процесс вычисления весовых коэффициентов ИНС состоит из следующих шагов:

— определение количества нейронов скрытого слоя:

```
n = len(self.pln_cs)
```

— вычисление первого коэффициента формулы:

```
k = (x_max ** n) / math.factorial(n)
```

— вычисление порогов и последовательности значений $Q_s$:

```
ts, q_base = self.build_q_base(n, d)
```

— вычисление значений $v_{\max}$ и $q_{\max}$:

```
v_max = np.amax(self.build_inv_v(n))
q_max = np.amax(self.build_q_diag(q_base))
```

— определение коэффициента $m_0$:

```
m0 = abs(q_base[-1])
```

— вычисление двух сумм:

```
sum1 = self.comp_sum1(self.pln_cs)
sum2 = self.comp_sum2(n)
```

— вычисление величины $m$:

```
m = k * v_max * q_max * m0 * sum1 * sum2
```

— вычисление значения величины $\delta$ и его последующая корректировка:

```
delta = eps / (m + 1)
cc = 10 ** (2 * (n - 3))
delta *= cc
```

— вычисление весов нейронов скрытого и выходного слоев:

```
ws = self.comp_ws(delta, n)
omega = self.build_omega(ws, q_base)
cs = self.comp_cs(omega, self.pln_cs)
```

— установка весов ИНС:

```
nn.set_ws(self.build_hl_ws(ws, ts), self.build_ol_ws(cs))
```

### 2.7.3 Запуск вычислений весов и тестирование

Аналогично действиям из раздела 2.6.4, осуществим запуск реализованного алгоритма для вычисления весовых коэффициентов ИНС, а затем протестируем полученный результат.

Как и в разделе 2.6.4, в качестве приближаемого многочлена будем использовать функцию $f(x) = 1 - 15x + x^2 + x^3$ на отрезке $[-5, 5]$, рассматривая 100 последовательных точек для проверки. Однако в этот раз не имеет значения, каким количеством скрытых нейронов инициализировать сеть, т.к. в процессе выполнения алгоритма матрица весов будет задана явным образом.

Ниже представлен листинг кода, осуществляющего вызов метода для вычисления весов ИНС, приближающей вышеуказанный многочлен с ошибкой не более $\varepsilon = 0.05$, и последующий запуск проверки результата:

```
1 nn = NeuralNetwork(1, 4, 1, 0.1)
2 builder = NNPolyBuilder([1, -15, 1, 1])
3 builder.build_for(nn, 0.05, 5, 0.1)
4 builder.test(nn, (-5, 5), 100)
```

В результате выполнения приведенного кода должен построиться график, представленный на рис. 11. В данном случае максимальная ошибка построенной ИНС составляет $\approx 0.01$, что удовлетворяет заданному требованию.

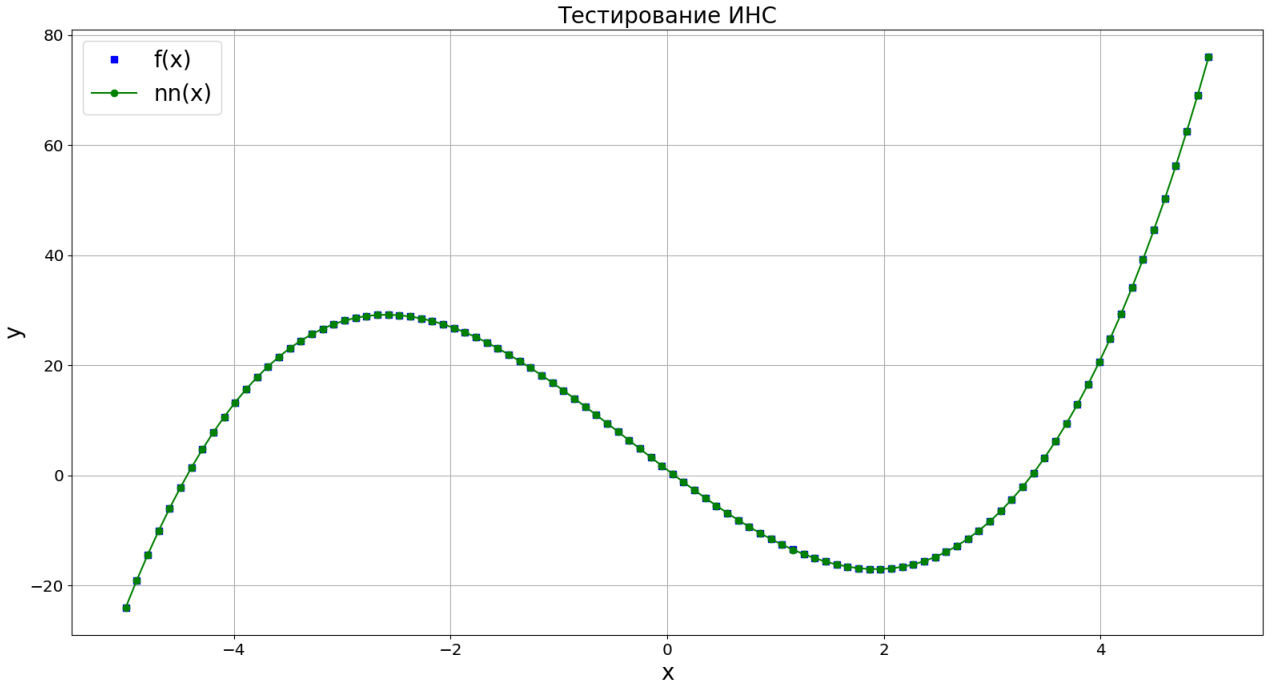


Рисунок 11 – Результат работы ИНС на тестовой выборке после процесса вычисления весов, где $f(x) = 1 - 15x + x^2 + x^3$ — приближаемый многочлен, $nn(x)$ — функция, построенная ИНС. Результаты ошибок:
$\min \approx 8.17e^{-6}$, $\max \approx 0.01$, $\text{avg} \approx 0.003$

Как следует из результатов, представленных на рис. 7 и 11, в случае известных степени и коэффициентов приближаемого многочлена, способ прямого вычисления весов ИНС в общем случае справляется с задачей значительно лучше, чем его итерационный аналог, если в качестве критерия оценки рассматривать затраченное на построение готовой сети время и полученную точность.

Однако на практике далеко не всегда известны параметры приближаемой функции, поэтому способ последовательного обучения ИНС своей актуальности ничуть не теряет. К тому же, в случаях, когда имеется априорная информация о форме графика приближаемой функции, можно применить комбинацию рассмотренных методов, реализовав таким образом своеобразный «перенос обучения», при котором веса ИНС для функции, близкой к целевой, сначала вычисляются *напрямую* а затем производится *дообучение* полученной модели, в результате чего схождение алгоритма происходит быстрее.

Таким образом, вышеприведенные результаты подтверждают справедливость алгоритма, полученного в разделе 1.5, а также демонстрируют его практическую применимость.

# ЗАКЛЮЧЕНИЕ

В ходе настоящей работы была доказана справедливость теоремы о взаимосвязи между количеством нейронов скрытого слоя трехслойной ИНС прямого распространения и степенью приближаемого ей многочлена в случае применения активационной функции `softplus`, а также были продемонстрированы результаты работы реализованных ИНС и двух алгоритмов вычисления ее весов, подтверждающие полученные теоретические выводы на практике.

Таким образом, представленные в данной работе результаты могут послужить основой для дальнейшего исследования активационной функции `softplus`, например, в контексте популярного в последнее время *«глубокого обучения»* [22] многослойных нейронных сетей, при котором существует проблема т.н. *«затухающих градиентов»* (vanishing gradients), наиболее свойственная *сигмоиде*, в связи с чем часто отдается предпочтение `softplus` [23].

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1 *Dugas, C.* Incorporating Second-order Functional Knowledge for Better Option Pricing / C. Dugas, Y. Bengio, F. Belisle, C. Nadeau, R. Garcia. — MIT Press, 2000.

2 *Hahnloser, R. H. R.* Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit / R. H. R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, H. S. Seung. — Nature, 2000.

3 *Hahnloser, R. H. R.* Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks / R. H. R. Hahnloser, H. S. Seung. — Neural Computation, 2003.

4 *Cybenko, G.* Approximation by superpositions of a sigmoidal function / G. Cybenko. — Springer London, 1989.

5 *Namig, G.* On the approximation by single hidden layer feedforward neural networks with fixed weights / G. Namig, V. Ismailov. — Elsevier, 2018.

6 *Malakooti, B.* Approximating polynomial functions by Feedforward Artificial Neural Networks: Capacity analysis and design / B. Malakooti, Y. Zhou. — Applied Mathematics and Computation, 1998.

7 *Glorot, X.* Deep Sparse Rectifier Neural Networks / X. Glorot, A. Bordes, Y. Bengio. — PMLR, 2011.

8 *Haykin, S.* Neural Networks: A Comprehensive Foundation / S. Haykin. — Prentice Hall, 1999.

9 *Галушкин, А. И.* Нейронные сети: основы теории / А. И. Галушкин. — Москва: Горячая Линия - Телеком, 2012.

10 *Узенцова, Н. С.* Об одновременном приближении алгебраических многочленов и их производных нейронными сетями прямого распространения сигнала с одним скрытым слоем / Н. С. Узенцова, С. П. Сидоров. — Известия Саратовского университета. Новая серия. Серия Математика. Механика. Информатика, 2013.

11 *Han, J.* The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning / J. Han, C. Moraga. — From Natural to Artificial Neural Computation, 1995.

12 *Курош, А. Г.* Курс высшей алгебры / А. Г. Курош. — Москва: Наука, 1968.

13 *Фихтенгольц, Г. М.* Основы математического анализа / Г. М. Фихтенгольц. — Москва: Наука, 1968.

14 *Хайкин, С.* Нейронные сети: Полный курс / С. Хайкин. — Вильямс, 2006.

15 Python 3.7.3 documentation [Электронный ресурс]. — URL: `https://docs.python.org/3/` (Дата обращения 15.05.2019). Загл. с экр. Яз. англ.

16 System-specific parameters and functions [Электронный ресурс]. — URL: `https://docs.python.org/3/library/sys.html` (Дата обращения 15.05.2019). Загл. с экр. Яз. англ.

17 Generate pseudo-random numbers [Электронный ресурс]. — URL: `https://docs.python.org/3/library/random.html` (Дата обращения 15.05.2019). Загл. с экр. Яз. англ.

18 Mathematical functions [Электронный ресурс]. — URL: `https://docs.python.org/3/library/math.html` (Дата обращения 15.05.2019). Загл. с экр. Яз. англ.

19 Numpy and Scipy Documentation [Электронный ресурс]. — URL: `https://docs.scipy.org/doc/` (Дата обращения 15.05.2019). Загл. с экр. Яз. англ.

20 Matplotlib documentation [Электронный ресурс]. — URL: `https://matplotlib.org/` (Дата обращения 15.05.2019). Загл. с экр. Яз. англ.

21 *Rashid, T.* Make Your Own Neural Network / T. Rashid. — 2016.

22 *Goodfellow, I.* Deep Learning / I. Goodfellow, Y. Bengio, A. Courville. — The MIT Press, 2016.

23 *Zheng, H.* Improving deep neural networks using softplus units / H. Zheng, Z. Yang, W. Liu, J. Liang, Y. Li. — 2015 International Joint Conference on Neural Networks (IJCNN), 2015.

# ПРИЛОЖЕНИЕ А

## Листинг модуля neural_network.py

Следующий код представляет листинг модуля neural_network.py.

```
# Импортирование модулей
import math
import numpy as np


# ИНС
class NeuralNetwork:

    # Инициализация
    def __init__(self, input_node_count, hidden_node_count,
        output_node_count, learning_rate):
        # сохранение коэффициента скорости обучения
        self.l_rate = learning_rate

        # определение функции активации
        self.act_fun = np.vectorize(lambda s: math.log(1 + math.exp(s)))

        # определение частной производной функции ошибки выходного слоя
        self.error_der_out = lambda e, x: np.dot(-1 * e, np.transpose(x))

        # определение частной производной функции ошибки скрытого слоя
        act_fun_der = np.vectorize(lambda s: 1 / (1 + math.exp(-s)))
        self.error_der_hid = lambda e, s, x: np.dot(-1 * e * act_fun_der(s),
                                                    np.transpose(x))

        # инициализация весов
        self.hl_ws = np.random.normal(0.0, pow(hidden_node_count, -0.5),
                                      (hidden_node_count, input_node_count + 1))
        self.ol_ws = np.random.normal(0.0, pow(output_node_count, -0.5),
                                      (output_node_count, hidden_node_count))

        pass

    # Указание весов
    def set_ws(self, hl_ws, ol_ws):
        self.hl_ws = hl_ws
        self.ol_ws = ol_ws
        pass

    # Обучение
    def train(self, input_list, target_list):
        # добавление константного входа
        input_list.append(1)
        # транспонирование для получения вектор-столбцов
```

```
44          input_list = np.array(input_list, ndmin=2).T
45          target_list = np.array(target_list, ndmin=2).T
46
47          # вычисление выходов нейронов скрытого слоя
48          hidden_layer_inputs = np.dot(self.hl_ws, input_list)
49          hidden_layer_outputs = self.act_fun(hidden_layer_inputs)
50
51          # вычисление выходов нейронов выходного слоя
52          final_layer_inputs = np.dot(self.ol_ws, hidden_layer_outputs)
53          final_layer_outputs = final_layer_inputs
54
55          # вычисление ошибок
56          final_layer_errors = target_list - final_layer_outputs
57          hidden_layer_errors = np.dot(self.ol_ws.T, final_layer_errors)
58
59          # коррекция весов
60          self.ol_ws -= self.l_rate * self.error_der_out(final_layer_errors,
61                                                         hidden_layer_outputs)
62          self.hl_ws -= self.l_rate * self.error_der_hid(hidden_layer_errors,
63                                                         hidden_layer_inputs,
64                                                         input_list)
65
66          pass
67
68      # Прогнозирование
69      def predict(self, input_list):
70          # добавление константного входа
71          input_list.append(1)
72          input_list = np.array(input_list, ndmin=2).T
73
74          # вычисление выходов нейронов скрытого слоя
75          hidden_layer_inputs = np.dot(self.hl_ws, input_list)
76          hidden_layer_outputs = self.act_fun(hidden_layer_inputs)
77
78          # вычисление выходов нейронов выходного слоя
79          final_layer_inputs = np.dot(self.ol_ws, hidden_layer_outputs)
80          final_layer_outputs = final_layer_inputs
81
82          # возврат результата
83          return final_layer_outputs
```

# ПРИЛОЖЕНИЕ Б

## Листинг модуля poly_data_manager.py

Следующий код представляет листинг модуля poly_data_manager.py.

```
# Импортирование модулей
import random
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
# Настройка визуализации
params = {'legend.fontsize': 'x-large',
          'figure.figsize': (15, 5),
          'axes.labelsize': 'x-large',
          'axes.titlesize': 'x-large',
          'xtick.labelsize': 'x-large',
          'ytick.labelsize': 'x-large'}
pylab.rcParams.update(params)


# Менеджер данных
class PolyDataManager:

    # Нормализация
    @staticmethod
    def normalize(l1, l2):
        mx1 = abs(max(l1, key=abs))
        mx2 = abs(max(l2, key=abs))
        mx = max(mx1, mx2)
        l1_n = (np.array(l1) / mx).tolist()
        l2_n = (np.array(l2) / mx).tolist()
        return l1_n, l2_n, mx

    # Денормализация
    @staticmethod
    def denormalize(list_pair, nc_pair):
        list_pair[0] = (np.array(list_pair[0]) * nc_pair[0]).tolist()
        list_pair[1] = (np.array(list_pair[1]) * nc_pair[1]).tolist()
        return list_pair

    # Вычисление значения многочлена в точке
    @staticmethod
    def comp_pln(cs, x0):
        value = 0
        for r in range(len(cs)):
            value += cs[r] * (x0 ** r)
        return value

    # Генерация тренировочного набора данных
```

```
45      @staticmethod
46      def generate_train_pairs(pln_cs, bounds, point_count):
47          x = []
48          y = []
49          for i in range(point_count):
50              x0 = random.uniform(bounds[0], bounds[1])
51              x.append(x0)
52              y.append(PolyDataManager.comp_pln(pln_cs, x0))
53          return x, y
54
55      # Генерация тестового набора данных
56      @staticmethod
57      def generate_test_pairs(pln_cs, bounds, point_count):
58          x = []
59          y = []
60          length = bounds[1] - bounds[0]
61          step = length / (point_count - 1)
62          for x0 in np.arange(bounds[0], bounds[1], step):
63              x.append(x0)
64              y.append(PolyDataManager.comp_pln(pln_cs, x0))
65          x.append(bounds[1])
66          y.append(PolyDataManager.comp_pln(pln_cs, bounds[1]))
67          return x, y
68
69      # Визуализация
70      @staticmethod
71      def plot(test_pairs, pred_pairs, x_nc, y_nc):
72          test_pairs = PolyDataManager.denormalize(list(zip(*test_pairs)), (
                x_nc, y_nc))
73
74          pred_pairs = sorted(pred_pairs, key=lambda t: t[0])
75          pred_pairs = PolyDataManager.denormalize(list(zip(*pred_pairs)), (
                x_nc, y_nc))
76
77          plt.figure()
78
79          plt.plot(test_pairs[0], test_pairs[1], 'bs', label="f(x)")
80          plt.plot(pred_pairs[0], pred_pairs[1], 'go-', label="nn(x)")
81
82          plt.title("Тестирование ИНС", fontsize=20)
83          plt.xlabel("x", fontsize=20)
84          plt.ylabel("y", fontsize=20)
85          plt.grid(True)
86          plt.legend(loc=0, fontsize=20)
87
88          plt.show()
89          pass
```

# ПРИЛОЖЕНИЕ В

## Листинг модуля nn_tester.py

Следующий код представляет листинг модуля nn_tester.py.

```
# Импортирование модулей
import sys
from poly_data_manager import PolyDataManager as pdm


# Тестировщик ИНС
class NNTester:

    # Тестирование
    @staticmethod
    def test(nn, test_pairs, x_nc, y_nc):
        # инициализация ошибок
        min_error = sys.float_info.max
        max_error = sys.float_info.min
        avg_error = 0
        # инициализация списка прогнозов
        pred_pairs = []
        # для каждого тестового экземпляра
        for test_pair in test_pairs:
            # получение прогноза
            pred = nn.predict([test_pair[0]])[0][0]
            # сохранение прогноза
            pred_pairs.append((test_pair[0], pred))
            # вычисление ошибки
            error = abs(test_pair[1] - pred)
            # обновление статистики ошибок
            if error < min_error:
                min_error = error
            if error > max_error:
                max_error = error
            avg_error += error

        # довычисление средней ошибки
        avg_error /= len(test_pairs)

        # вывод значений ошибок в консоль
        print("Результаты тестирования:")
        print("min error:", min_error * y_nc)
        print("max error:", max_error * y_nc)
        print("avg error:", avg_error * y_nc)
        # визуализация результатов
        pdm.plot(test_pairs, pred_pairs, x_nc, y_nc)

        pass
```

# ПРИЛОЖЕНИЕ Г

## Листинг модуля nn_poly_teacher.py

Следующий код представляет листинг модуля nn_poly_teacher.py.

```
# Импортирование модулей
import random
from poly_data_manager import PolyDataManager as pdm
from nn_tester import NNTester


# Учитель ИНС
class NNPolyTeacher:

    # Инициализация
    def __init__(self, pln_cs, bounds, point_counts):
        # генерация тренировочного и тестового наборов данных
        train_x, train_y = pdm.generate_train_pairs(pln_cs, bounds,
            point_counts[0])
        test_x, test_y = pdm.generate_test_pairs(pln_cs, bounds,
            point_counts[1])

        # нормализация
        train_x, test_x, self.x_nc = pdm.normalize(train_x, test_x)
        train_y, test_y, self.y_nc = pdm.normalize(train_y, test_y)

        # сшивание в списки пар
        self.train_pairs = list(zip(train_x, train_y))
        self.test_pairs = list(zip(test_x, test_y))

        pass

    # Обучение
    def teach(self, nn, epoch_count):
        # для каждой эпохи
        for i in range(epoch_count):
            # вывод номера эпохи в консоль
            print("Эпоха:", i + 1)
            # обучить ИНС на каждом экземпляре тренировочных данных
            for train_pair in self.train_pairs:
                nn.train([train_pair[0]], [train_pair[1]])
            # перемешать тренировочные данные
            random.shuffle(self.train_pairs)
        pass

    # Тестирование
    def test(self, nn):
        NNTester.test(nn, self.test_pairs, self.x_nc, self.y_nc)
        pass
```

# ПРИЛОЖЕНИЕ Д

## Листинг модуля nn_poly_builder.py

Следующий код представляет листинг модуля nn_poly_builder.py.

```
# Импортирование модулей
import math
import numpy as np
from poly_data_manager import PolyDataManager as pdm
from nn_tester import NNTester


# Вычислитель ИНС
class NNPolyBuilder:

    # Инициализация
    def __init__(self, pln_cs):
        # определение функции активации
        self.act_fun_0 = lambda t: math.log(1 + math.exp(t))
        # определение производной функции активации
        self.fun_0 = lambda t: 1 / (1 + math.exp(-t))
        # определение начального порогового значения
        self.t_init = -1.4

        # сохранение коэффициентов многочлена
        self.pln_cs = pln_cs

    # Построение обратной матрицы V
    def build_inv_v(self, n):
        v = np.eye(n)
        v[0] = [1] * n
        for r in range(1, len(v)):
            v[r] = [i ** r for i in range(n)]
        return np.linalg.inv(v)

    # Численное дифференцирование
    def der(self, f, x0):
        dx = 1e-6
        return (f(x0 + dx) - f(x0)) / dx

    # Оболочка Q_s
    def wrapper(self, base):
        if base < 2:
            return lambda s: s
        return lambda s: s * (1 - s) * self.der(self.wrapper(base - 1), s)

    # Вычисление порогов и последовательности значений Q_s
    def build_q_base(self, n, d):
        t = self.t_init
```

```
45
46          while True:
47              af0 = self.act_fun_0(t)
48              q1 = [self.wrapper(0)(af0)]
49
50              f0 = self.fun_0(t)
51              q2 = [self.wrapper(i)(f0) for i in range(1, n + 1)]
52
53              if 0 in q1 + q2:
54                  t -= d
55              else:
56                  break
57
58          ts = [t] * n
59          q = np.array(q1 + q2)
60
61          return ts, q
62
63      # Построение редуцированной диагональной матрицы Q
64      def build_q_diag(self, q_base):
65          q_diag = abs(1 / q_base[:-1])
66          return q_diag
67
68      # Вычисление первой суммы
69      def comp_sum1(self, pln_cs):
70          sum1 = 0
71          for i in range(len(pln_cs)):
72              sum1 += abs(math.factorial(i) * pln_cs[i])
73          return sum1
74
75      # Вычисление второй суммы
76      def comp_sum2(self, n):
77          sum2 = 0
78          for j in range(1, n + 1):
79              sum2 += j ** n
80          return sum2
81
82      # Вычисление весов нейронов скрытого слоя
83      def comp_ws(self, delta, n):
84          ws = np.array([(j - 1) * delta for j in range(1, n + 1)])
85          return ws
86
87      # Построение матрицы <<Omega>>
88      def build_omega(self, ws, q_base):
89          omega = np.eye(len(ws))
90          omega[0] = [q_base[0]] * len(ws)
91          for r in range(1, len(omega)):
92              omega[r] = [q_base[r] * (w ** r) for w in ws]
```

```
93          return omega
94
95      # Вычисление весов нейронов выходного слоя
96      def comp_cs(self, omega, pln_cs):
97          a = np.array([math.factorial(i) * pln_cs[i] for i in range(len(
                pln_cs))], ndmin=2).T
98          cs = np.dot(np.linalg.inv(omega), a)
99          return cs
100
101     # Построение матрицы весов скрытого слоя
102     def build_hl_ws(self, ws, ts):
103         hl_ws = np.zeros((2, len(ws)))
104         hl_ws[0] = ws
105         hl_ws[1] = ts
106         return hl_ws.T
107
108     # Построение матрицы весов выходного слоя
109     def build_ol_ws(self, cs):
110         ol_ws = np.zeros((1, len(cs)))
111         ol_ws[0] = cs.T
112         return ol_ws
113
114     # Вычисление весов ИНС
115     def build_for(self, nn, eps, x_max, d):
116         # определение количества нейронов скрытого слоя
117         n = len(self.pln_cs)
118
119         # вычисление первого коэффициента
120         k = (x_max ** n) / math.factorial(n)
121
122         # вычисление порогов и последовательности значений Q_s
123         ts, q_base = self.build_q_base(n, d)
124
125         # вычисление значения v_max
126         v_max = np.amax(self.build_inv_v(n))
127         # вычисление значения q_max
128         q_max = np.amax(self.build_q_diag(q_base))
129
130         # определение значения m0
131         m0 = abs(q_base[-1])
132
133         # вычисление первой и второй сумм
134         sum1 = self.comp_sum1(self.pln_cs)
135         sum2 = self.comp_sum2(n)
136
137         # вычисление значения m
138         m = k * v_max * q_max * m0 * sum1 * sum2
139
```

```
140         # вычисление значения <<delta>>
141         delta = eps / (m + 1)
142         # корректировка
143         cc = 10 ** (2 * (n - 3))
144         delta *= cc
145 
146         # вычисление весов нейронов скрытого слоя
147         ws = self.comp_ws(delta, n)
148         # вычисление весов нейронов выходного слоя
149         omega = self.build_omega(ws, q_base)
150         cs = self.comp_cs(omega, self.pln_cs)
151 
152         # установка вычисленных весов
153         nn.set_ws(self.build_hl_ws(ws, ts), self.build_ol_ws(cs))
154 
155         pass
156 
157     # Тестирование ИНС
158     def test(self, nn, bounds, point_count):
159         # генерация тестового набора данных
160         test_x, test_y = pdm.generate_test_pairs(self.pln_cs, bounds,
                point_count)
161         # сшивание в список пар
162         test_pairs = list(zip(test_x, test_y))
163         # тестирование
164         NNTester.test(nn, test_pairs, 1, 1)
165         pass
```

# ПРИЛОЖЕНИЕ Е

## Листинг модуля main.py

Следующий код представляет листинг модуля `main.py`.

```
1  # Импортирование модулей
2  from neural_network import NeuralNetwork
3  from nn_poly_teacher import NNPolyTeacher
4  from nn_poly_builder import NNPolyBuilder
5
6
7  # Обучение ИНС
8  def teach_for_poly():
9      # создание объекта ИНС
10     nn = NeuralNetwork(1, 4, 1, 0.1)
11     # создание объекта учителя ИНС
12     teacher = NNPolyTeacher([1, -15, 1, 1], (-5, 5), (30, 100))
13     # обучение ИНС
14     teacher.teach(nn, 5000)
15     # тестирование ИНС
16     teacher.test(nn)
17
18
19 # Вычисление весов ИНС
20 def build_for_poly():
21     # создание объекта ИНС
22     nn = NeuralNetwork(1, 4, 1, 0.1)
23     # создание объекта вычислителя ИНС
24     builder = NNPolyBuilder([1, -15, 1, 1])
25     # вычисление весов ИНС
26     builder.build_for(nn, 0.05, 5, 0.1)
27     # тестирвование ИНС
28     builder.test(nn, (-5, 5), 100)
29     pass
30
31
32 # Запуск обучения ИНС
33 teach_for_poly()
34
35 # Запуск вычисления весов ИНС
36 build_for_poly()
```