«Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И.Ульянова (Ленина)»
(СПбГЭТУ «ЛЭТИ»)

Направление	01.03.02 – Прикладная математика и информатика
Профиль	Без профиля
Факультет	КТИ
Кафедра	МО ЭВМ

К защите допустить

Зав. кафедрой Кринкин К.В.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

ТЕМА: ИСПОЛЬЗОВАНИЕ НЕЙРОННЫХ СЕТЕЙ ДЛЯ АППРОКСИМАЦИИ ПОЛИНОМИАЛЬНЫХ ФУНКЦИЙ

Студент		_____ *подпись*	Кирсанов А.Я.
Руководитель	к.т.н., доцент	_____ *подпись*	Сучков А.И.
Консультанты	к.ф.-м.н	_____ *подпись*	Григорьева Н.Ю.
		_____ *подпись*	Смолова О.В.
	к.т.н.	_____ *подпись*	Заславский М.М.

Санкт-Петербург

2022

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Утверждаю

Зав. кафедрой МО ЭВМ

_____ Кринкин К.В.

«___»_____2022 г.

Студент Кирсанов А.Я. Группа 7383

Тема работы: Использование нейронных сетей для аппроксимации полиномиальных функций

Место выполнения ВКР: СПбГЭТУ «ЛЭТИ», кафедра МО ЭВМ

Исходные данные (технические требования):

Случайным образом сгенерированная выборка значений полиномиальной функции.

Содержание ВКР:

Введение, постановка задачи, приближённые алгоритмы, анализ, заключение, список литературы

Перечень отчетных материалов: пояснительная записка, иллюстративный материал

Дополнительные разделы: обеспечение качества разработки, продукции, программного продукта

Дата выдачи задания «22» апреля 2021 г.

Дата представления ВКР к защите «___»_____20__ г.

Студент _____ Кирсанов А.Я.

Руководитель к.т.н., доцент _____ Лисс А.А.

Консультант к.ф.-м.н  Григорьева Н.Ю.

КАЛЕНДАРНЫЙ ПЛАН ВЫПОЛНЕНИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Утверждаю
Зав. кафедрой МО ЭВМ
_____ Кринкин К.В.
«___»_____2022 г.

Студент Кирсанов А.Я. Группа 7383

Тема работы: Использование нейронных сетей для аппроксимации полиномиальных функций.

№ п/п	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	11.02 – 05.03
2	Разработка модели нейронной сети	06.03 – 25.03
3	Реализация модели нейронной сети	26.03 – 20.04
3	Исследование влияния архитектуры модели и метода обучения нейронной сети на результат обучения	21.04 – 10.05
4	Исследование влияния вида исходных данных на результат обучения	11.05 – 17.05
5	Сравнение полученных результатов и выводы	18.05 – 22.05
5	Оформление пояснительной записки	22.05 – 24.05
6	Оформление иллюстративного материала	25.05 – 27.05

Студент _____ Кирсанов А.Я.

Руководитель к.т.н., доцент _____ Лисс А.А.

Консультант к.ф.-м.н  Григорьева Н.Ю.

РЕФЕРАТ

Пояснительная записка 88 стр., 31 рис., 16 табл., 38 ист., 2 прил.

ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ, АДДИТИВНЫЕ ЦЕПОЧКИ, АЛГОРИТМЫ, АЛГОРИТМ БРАУЭРА, АЛГОРИТМ ЯО

Объектом исследования являются полиномы.

Предметом исследования является аппроксимация полиномов.

Цель работы – исследование возможности предсказания значений полиномиальной функции с помощью нейронной сети, реализация такой нейронной сети, изучение и реализация алгоритмов нахождения аддитивных цепочек для заданного числа.

В данной работе была исследована аппроксимация полиномиальных функций с помощью глубинных нейронных сетей, изучено использование аддитивных цепочек при вычислении степеней полинома, реализованы приближенные алгоритмы построения аддитивных цепочек для возведения в степень за небольшое число операций. Были проведены измерения скорости всех реализованных алгоритмов. Результатом работы является реализация нейронной сети, выполняющей аппроксимацию, а также реализация алгоритмов Брауэра, Яо построения аддитивных цепочек, алгоритма возведения в степень вида 2^k с помощью замены переменных, а также стандартного алгоритма возведения в степень за одну операцию на языке Python 3.9. Было измерено и выполнено сравнение времени работы всех реализованных алгоритмов.

ABSTRACT

In this paper, we investigated the approximation of polynomial functions using deep neural networks, studied the use of additive chains in calculating the powers of a polynomial, implemented approximate algorithms for constructing additive chains for raising to a power in a small number of operations. The speed measurements of all implemented algorithms were carried out. The result of the work is the implementation of a neural network that performs approximation, as well as the implementation of the Brauer and Yao's algorithms for constructing additive chains, an exponentiation algorithm of the form 2^k using variable substitution, as well as a standard exponentiation algorithm in one operation in Python 3.9. The running times of all implemented algorithms were measured and compared.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяют следующие обозначения и сокращения:

Adam – метод адаптивной оценки моментов (англ. Adaptive Moment Estimation)

SGD – Стохастический градиентный спуск (англ. Stochastic gradient descent)

НС – нейронная сеть

ИНС – искусственная нейронная сеть

АЦ – аддитивная цепочка

ReLU – линейный выпрямитель (англ. Rectified linear unit)

ВВЕДЕНИЕ

Благодаря доступности огромных объёмов обучающих данных и значительному увеличению вычислительной мощности, глубокие НС стали современной технологией для решения широкого круга практических задач машинного обучения. НС эффективно реализует отображение, приближающее функцию, которая изучается на основе заданного набора пар значений ввода-вывода. Замечательным свойством глубоких НС является то, что они обеспечивают экспоненциальную точность аппроксимации, то есть ошибка аппроксимации экспоненциально спадает с увеличением количества ненулевых весов в сети – совершенно разных функций, таких как операция возведения в квадрат, умножение, полиномы и так далее. Таким образом, глубокие НС обеспечивают теоретически оптимальное приближение очень широкого диапазона функций и классов функций, используемых в математической обработке сигналов.

В данной работе представляет интерес построение НС для аппроксимации полиномов и её практические ограничения, а также способы уменьшения числа операций при возведении в степень. Для уменьшения числа операций при вычислении полинома в данной работе будут реализованы такие алгоритмы построения АЦ как алгоритм Брауэра и алгоритм Яо и произведён анализ скорости их работы по сравнению с алгоритмом, который не использует АЦ. На сегодняшний день использование АЦ – наиболее эффективный способ, при котором можно получить определённую степень за минимальное число операций.

Практической пользой данной работы является реализация на языке программирования и обучения НС, обеспечивающей экспоненциальную точность аппроксимации полиномов, а также реализация алгоритмов Брауэра и Яо. Причём, сам аппроксимируемый полином может быть неизвестен, достаточно иметь пары аргумент-значение для полинома.

1. АППРОКСИМАЦИЯ С ПОМОЩЬЮ НЕЙРОННЫХ СЕТЕЙ

В этом разделе будут рассмотрены такие понятия как нейронная сеть, аппроксимация и её применение по отношению к нейронным сетям.

1.1 Определение нейронной сети

В литературе описано множество архитектур нейронных сетей и функций активации. В этой работе используется только функция активации ReLU и рассматривается следующая общая архитектура сети.

Пусть $L,\ N_0,\ N_1,\ldots,N_L \in \mathbb{N},\ L \geq 2$. Отображение $\Phi : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$, заданное как

$$\Phi(x) = \begin{cases} W_2(\rho(W_1(x))), & L = 2 \\ W_L(\rho(W_{L-1}(\rho(\ldots\rho(W_1(x)))))), & L \geq 3 \end{cases}, \qquad (1)$$

с аффинными линейными отображениями $W_L : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$, $l \in \{1,2,\ldots,L\}$ и функцией активации ReLU $\rho(x) = \max(x,0)$, $x \in \mathbb{R}$, действующая покомпонентно, (т.е. $\rho(x_1,\ldots,x_N) := (\rho(x_1),\ldots,\rho(x_N))$) называется нейронной сетью ReLU [1]. Отображение W_l, соответствующее слою l, задается с помощью формулы $W_\ell(x) = A_\ell x + b_\ell$, где $A_\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ и $b_\ell \in \mathbb{R}^{N_\ell}$. На протяжении всей работы пишется $L \in \mathbb{N}$ для обозначения того, что соответствующее утверждение применимо к сетям с $L \in \mathbb{N}$, $L \geq 2$ слоями. Определим сетевую связность $\mathcal{M}(\Phi)$ как общее количество ненулевых элементов в матрицах A_ℓ, $\ell \in \{1,2,\ldots,L\}$, и в векторах $b_\ell, \ell \in \{1,2,\ldots L\}$. Глубина сети, или, что то же самое, количество слоёв это $\mathcal{L}(\Phi) := L$, а её ширина это $\mathcal{W}(\Phi) := \max_{\ell=0,\ldots,L} N_\ell$. Далее обозначим через $\mathcal{B}(\Phi) := \max_{\ell=1,\ldots,L} \max\left\{\|A_\ell\|_\infty, \|b_\ell\|_\infty\right\}$ максимальное абсолютное значение весов в сети.

1.2 Ограничение на связность в нейронной сети

Связность удовлетворяет неравенству $\mathcal{M}(\Phi) \leq \mathcal{L}(\Phi)\mathcal{W}(\Phi)\mathcal{W}((\Phi)+1)$.

1.3 Классификация нейронных сетей

N_0 в 1.1. – это размер входного слоя, $N_1,\ldots,N_{L-1}$ – это размеры $L-1$ скрытых слоёв, а N_L – размер выходного слоя. Стоит обратить внимание, что определение $\mathcal{L}(\Phi)$ не учитывает входной слой, который считается «0-м» слоем. Элемент матрицы $(A_\ell)_{i,j}$ представляет вес, связанный с ребром между j-м узлом в $(\ell-1)$-м слое и i-м узлом в ℓ-м слое, $(b_\ell)_i$ – вес, связанный с i-м узлом в ℓ-м слое. Эти обозначения схематически представлены на рис. 1.

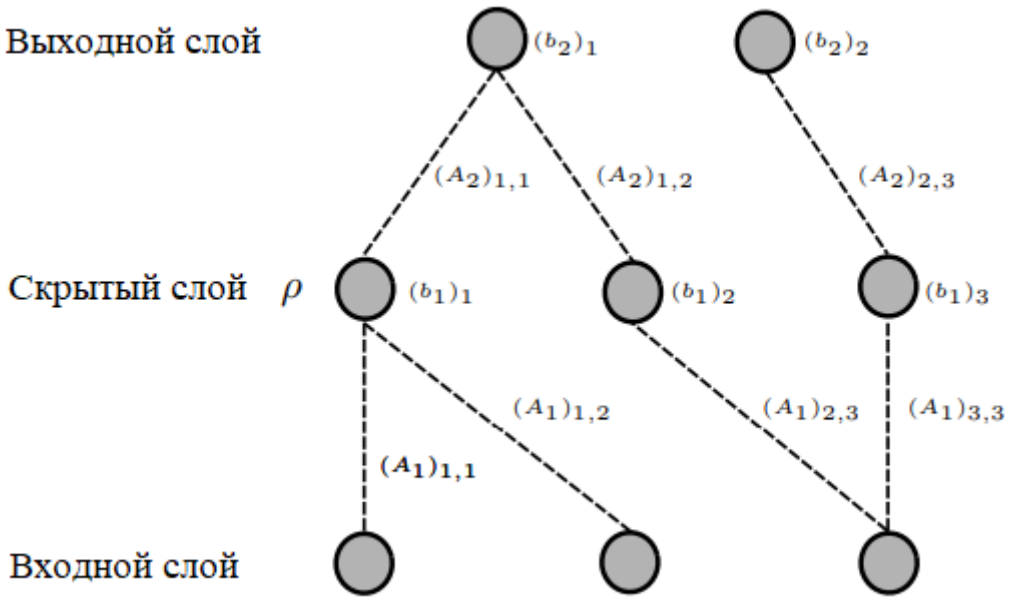

Рисунок 1 – Присвоение весов $(A_\ell)_{i,j}$ и $(b_\ell)_i$ двухуровневой сети рёбрам и узлам соответственно

Действительные числа $(A_\ell)_{i,j}$ и $(b_\ell)_i$ называют весами рёбер и весами узлов соответственно. Предполагается, что каждый узел во входном слое и в слоях $1,\ldots,L-1$ имеет хотя бы одно исходящее ребро и каждый узел в

выходном слое L имеет по крайней мере одно входящее ребро. Эти предположения о невырожденности являются базовыми, поскольку узлы, которые им не удовлетворяют, могут быть удалены без изменения функциональной взаимосвязи, реализуемой сетью.

Термин «сеть» происходит от интерпретации отображения Φ как взвешенного ациклического ориентированного графа с узлами, расположенными в иерархических слоях, и рёбрами только между соседними слоями.

Обозначим класс сетей ReLU $\Phi:\mathbb{R}^d \to \mathbb{R}^{N_L}$ с не более чем L слоями, связностью не более $\mathcal{M}$, входной размерностью d и выходной размерностью N_L через $\mathcal{NN}_{L,M,d,N_L}$. Кроме того, определим:

$$\mathcal{NN}_{\infty,M,d,N_L} := \bigcup_{L\in\mathbb{N}} \mathcal{NN}_{L,M,d,N_L},$$

$$\mathcal{NN}_{L,\infty,d,N_L} := \bigcup_{M\in\mathbb{N}} \mathcal{NN}_{L,M,d,N_L},$$

$$\mathcal{NN}_{\infty,\infty,d,N_L} := \bigcup_{L\in\mathbb{N}} \mathcal{NN}_{L,\infty,d,N_L}.$$

В работе рассматривается почти исключительно случай $\Phi:\mathbb{R}^d \to \mathbb{R}$, т.е. $N_L = 1$. Однако, результаты легко обобщаются на $N_L > 1$.

Теперь, имея функцию $f:\mathbb{R}^d \to \mathbb{R}$, интересует наилучшее возможное приближение f сетью Φ при различных ограничениях на топологию и веса Φ. В частности, понадобится понятие «полиномиально ограниченных весов» семейств сетей, которые уточняются следующим образом.

1.4 Полиномиальная ограниченность

Веса (ребра и вершины) семейства нейронных сетей Φ_z, $z=(z_1,z_2,\ldots,z_N)\in D\subseteq\mathbb{R}^N$ полиномиально ограничены по $z_1,z_2,\ldots,z_N$ если существует N - мерный многочлен π такой, что $\mathcal{B}(\Phi_z)\le\pi(z_1,z_2,\ldots,z_N)$ для

всех $z \in D$. Далее, веса семейства нейронных сетей $\Phi_{z,j}, z \in D \subseteq \mathbb{R}^N, j \in J$ равномерно (относительно $j \in J$) полиномиально ограничены по $z_1, z_2, \ldots, z_N$, если существует N- мерный многочлен π такой, что $\mathcal{B}\left(\Phi_{z.j}\right) \leq \pi\left(z_1, z_2, \ldots, z_N\right)$ для всех $z \in D,\ j \in J$.

Нейронная сеть Φ, как определено в (1), реализует функцию $\Phi : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$. Однако, для данной функции возможен различный выбор параметров $L, N_0, N_1, \ldots, N_L \in \mathbb{N}$ и аффинные отображения $W_1, W_2, \ldots, W_L$, реализующие эту функцию через нейронную сеть [3]. Несмотря на эту дихотомию, имеет смысл говорить о композициях и линейных комбинациях нейронных сетей. С этой целью записывается техническая лемма о композиции нейронных сетей, как это определено в [4].

1.5 Леммы о нейронных сетях

Для того, чтобы записать ограничения на нейронную сеть, аппроксимирующую многочлены, понадобится лемма о линейной комбинации нейронной сети. Она выводится из лемм о композиции и превосходящей глубине.

1.5.1 Лемма о композиции нейронных сетей

Пусть $L_1, L_2, M_1, M_2, d_1, d_2, N_{L_1}, N_{L_2} \in \mathbb{N}$, $\Phi_1 \in \mathcal{NN}_{L_1, M_1, d_1, N_{L_1}}$ и $\Phi_2 \in \mathcal{NN}_{L_2, M_2, d_2, N_{L_2}}$, где $N_{L_1} = d_2$. Тогда существует сеть $\Psi \in \mathcal{NN}_{L_1+L_2, 2M_1+2M_2, d_1, N_{L_2}}$ такая, что $\mathcal{W}(\Psi) \leq \max\left\{2N_{L_1}, \mathcal{W}(\Phi_1), \mathcal{W}(\Phi_2)\right\}$ и $\mathcal{B}(\Psi) = \max\left\{\mathcal{B}(\Phi_1), \mathcal{B}(\Phi_2)\right\}$, удовлетворяющая условию $\Psi(x) = \Phi_2(\Phi_1(x))$ для всех $x \in \mathbb{R}^{d_1}$.

Перед тем как формализовать концепцию линейной комбинации нейронных сетей, нужен результат, который показывает, как увеличить

глубину сети, сохраняя при этом отношения ввода-вывода сети. Об этом следующая лемма.

1.5.2 Лемма о превосходящей глубине нейронной сети

Пусть $L,M,K,d \in \mathbb{N}$, $\Phi_1 \in \mathcal{NN}_{L,M,d,1}$ и $K > L$. Тогда существует соответствующая сеть $\Phi_2 \in \mathcal{NN}_{K,M+\mathcal{W}(\Phi_1)+2(K-L)+1,d,1}$ такая, что $\Phi_2(x) = \Phi_1(x)$ для всех $x \in \mathbb{R}^d$. Кроме того, $\mathcal{W}(\Phi_2) = \max\{2, \mathcal{W}(\Phi_1)\}$ и веса Φ_2 состоят из весов Φ_1 и ± 1-го.

Следующая лемма формализует понятие линейной комбинации нейронных сетей.

1.5.3 Лемма о линейной комбинации нейронных сетей

Пусть $N, L_i, M_i, d_i \in \mathbb{N}$, $a_i \in \mathbb{R}$, $\Phi_i \in \mathcal{NN}_{L_i,M_i,d_i,1}$, $i = 1,2,\ldots,N$, $d = \sum_{i=1}^{N} d_i$. Тогда существуют сети $\Phi^1 \in \mathcal{NN}_{L,M,d,N}$ и $\Phi^2 \in \mathcal{NN}_{L,M+N,d,1}$ с $L = \max_i L_i$, $\mathcal{W}(\Phi^1) = \mathcal{W}(\Phi^2) \le \sum_{i=1}^{N} \max\{2, \mathcal{W}(\Phi_i)\}$, $M = \sum_{i=1}^{N} \left(M_i + \mathcal{W}(\Phi_i) + 2(L - L_i) + 1\right)$ которые удовлетворяют

$$\Phi^1(x) = \left(a_1\Phi_1(x_1) \quad a_2\Phi_2(x_2) \quad \ldots \quad a_N\Phi_N(x_N)\right)^T \text{ и}$$

$$\Phi^2(x) = \sum_{i=1}^{N} a_i\Phi_i(x_i),$$

для всех $x = \left(x_1^T, x_2^T, \ldots, x_N^T\right)^T \in \mathbb{R}^d$ с $x_i \in \mathbb{R}^{d_i}$, $i = 1,2,\ldots,N$. Кроме того, веса Φ^1, Φ^2 состоят из весов Φ_i, $i = 1,2,\ldots,N$, $\{a_1, a_2, \ldots, a_N\}$ и ещё ± 1-го.

1.6 Аппроксимация умножения, полиномов и гладких функций

Сначала записывается аппроксимация операции умножения глубокими сетями ReLU, затем аппроксимация многочленов. В частности, имеются в виду сети, которые аппроксимируются с точностью до ошибки ε, имеют

конечную ширину и имеют масштабирование по глубине полилогарифмически по $\varepsilon-1$ (т.е. как полином от $\log(\varepsilon-1)$) и (краевые, узловые) веса, которые растут не быстрее, чем полиномиально по размеру области, в которой происходит аппроксимация. Такая комбинация требований приводит к экспоненциальной точности аппроксимации для отдельных вхождений, то есть ошибка аппроксимации экспоненциально спадает в количестве узлов в сети, и к оптимальной аппроксимации классов вхождений по отношению скоростей и искажений.

1.6.1 Аппроксимация возведения в квадрат с помощью сетей ReLU

Существует такая $C>0$ что для всех $\varepsilon\in(0,1/2)$ существует сеть $\Phi_\varepsilon\in\mathcal{NN}_{\infty,\infty,1,1}$, удовлетворяющая $\mathcal{L}(\Phi_\varepsilon)\le C\log\left(\varepsilon^{-1}\right)$, $\mathcal{W}(\Phi_\varepsilon)=4$, $\mathcal{B}(\Phi_\varepsilon)\le 4$, $\Phi_\varepsilon(0)=0$ и

$$\left\|\Phi_\varepsilon(x)-x^2\right\|_{L^\infty([0,1])}\le\varepsilon.$$

С помощью данного пункта можно показать, как операция умножения может быть реализована через глубокие сети ReLU. Затем это приведет к приближению произвольных многочленов.

1.6.2 Аппроксимация умножения с помощью сетей ReLU

Существует такая $C>0$ что для всех $D\in\mathbb{R}_+$ и $\varepsilon\in(0,1/2)$ существует сеть $\Phi_{D,\varepsilon}\in\mathcal{NN}_{\infty,\infty,2,1}$, удовлетворяющая $\mathcal{L}(\Phi_\varepsilon)\le C\log\left(\lceil D\rceil^2\varepsilon^{-1}\right)$, $\mathcal{W}(\Phi_{D,\varepsilon})\le 12$, $\mathcal{B}(\Phi_{D,\varepsilon})\le\max\left\{4,2\lceil D\rceil^2\right\}$, $\Phi_{D,\varepsilon}(0,x)=\Phi_{D,\varepsilon}(x,0)$ для всех $x\in\mathbb{R}$ и

$$\left\|\Phi_{D,\varepsilon}(x,y)-xy\right\|_{L^\infty\left([-D,D]^2\right)}\le\varepsilon$$

Теперь, когда известно как аппроксимировать операцию возведения в квадрат и умножение с помощью глубоких сетей ReLU, можно реализовать произвольные степени посредством композиции сетей возведения в квадрат и умножения, взяв взвешенные линейные комбинации степеней x согласно лемме 1.5.3. о линейной комбинации нейронных сетей. В частности, многочлены могут быть аппроксимированы сетями ReLU конечной ширины и глубины, логарифмически растущей по отношению к ошибке аппроксимации.

1.6.3 Аппроксимация многочленов с помощью сетей ReLU

Существует такая $C>0$, что для всех $m\in\mathbb{N}$, $A\in\mathbb{R}_+$, $p_m(x)=\sum_{i=0}^{m}a_i x^i$ где $\max_{i=0,\ldots,m}|a_i|=A$, $D\in\mathbb{R}_+$ и $\varepsilon\in(0,1/2)$ существует сеть $\Phi_{p_m,D,\varepsilon}\in\mathcal{NN}_{\infty,\infty,1,1}$, которая удовлетворяет

$$\mathcal{L}\left(\Phi_{p_m,D,\varepsilon}\right)\le Cm\left(\log\left(\lceil A\rceil\right)+\log\left(\varepsilon^{-1}\right)+m\log\left(\lceil D\rceil\right)+\log(m)\right), \qquad (2)$$

$\mathcal{W}\left(\Phi_{p_m,D,\varepsilon}\right)\le 16$, $\mathcal{B}\left(\Phi_{p_m,D,\varepsilon}\right)\le\max\left\{A,8\lceil D\rceil^{2m-2}\right\}$ и

$$\left\|\Phi_{p_m,D,\varepsilon}-p_m\right\|_{L^\infty([-D,D])}\le\varepsilon.$$

Далее, аппроксимационная теорема Вейерштрасса утверждает, что любую непрерывную функцию на отрезке можно аппроксимировать с произвольной точностью полиномом. Из этого вытекает следующий пункт.

1.6.4 ([7]) Аппроксимация полинома функцией

Пусть $[a,b]\subseteq\mathbb{R}$ и $f\in C([a,b])$. Тогда для любого $\varepsilon>0$ существует полином π такой, что

$$\|f-\pi\|_{L^\infty([a,b])}\le\varepsilon.$$

Таким образом, пункт 1.6.3 позволяет заключить, что любая непрерывная функция на отрезке может быть аппроксимирована с

произвольной точностью глубокой сетью ReLU шириной не более 16. Это представляет собой вариант универсальной аппроксимационной теоремы [5][6] для глубоких сетей ReLU конечной ширины. Однако, аппроксимационная теорема Вейерштрасса неколичественна. Количественное утверждение можно получить для гладких функций, определяемых следующим образом.

Для $D \in \mathbb{R}_+$ пусть множество $S_D \subseteq C^\infty\left([-D,D],\mathbb{R}\right)$ задаётся формулой

$$S_D = \{ f \in C^\infty\left([-D,D],\mathbb{R}\right) : \left\| f^{(n)}(x) \right\|_{L^\infty([-D,D])} \leq n!, \text{ для всех } n \in \mathbb{N}_0 \}.$$

1.6.5 Лемма об аппроксимации функции сетью ReLU

Пусть существует такая $C > 0$ и полином π такой, что для всех $D \in \mathbb{R}_+$, $f \in S_D$ и $\varepsilon \in (0,1/2)$, существует сеть $\Psi_{f,\varepsilon} \in \mathcal{NN}_{\infty,\infty,1,1}$, удовлетворяющая $\mathcal{L}\left(\Psi_{f,\varepsilon}\right) \leq C\lceil D \rceil \left(\log\left(\varepsilon^{-1}\right)\right)^2$, $\mathcal{W}\left(\Psi_{f,\varepsilon}\right) \leq 23$, $\mathcal{B}\left(\Psi_{f,\varepsilon}\right) \leq \max\left\{1/D, \lceil D \rceil\right\} \pi\left(\varepsilon^{-1}\right)$ и

$$\left\| \Psi_{f,\varepsilon} - f \right\|_{L^\infty([-D,D])} \leq \varepsilon.$$

Лемма сформулирована для симметричных интервалов $[-D,D]$ для простоты. Расширение на функции $f \in C^\infty\left([a,b],\mathbb{R}\right)$ с $\left\| f^{(n)} \right\|_{L^\infty([a,b])} \leq n!$, для всех $n \in \mathbb{N}_0$ на произвольных интервалах $[a,b]$ получается симметризацией носителя f согласно $g(x) = f\left(x + \frac{b+a}{2}\right)$ и последующим применением леммы выше к $g(x)$ с $D = \frac{b-a}{2}$.

Все результаты в этом разделе имеют аппроксимирующие сети конечной ширины и полилогарифмического масштабирования по глубине в $1/\varepsilon$. Благодаря тому, что

$$\mathcal{M}(\Phi) \leq \mathcal{L}(\Phi)\mathcal{W}(\Phi)\mathcal{W}\left((\Phi)+1\right)$$

это означает, что связность масштабируется не быстрее, чем полилогарифмическая по $1/\varepsilon$. Отсюда следует, что ошибка аппроксимации ε убывает (по крайней мере) экспоненциально быстро в связности, или, что эквивалентно, в количестве параметров аппроксимант (то есть нейронной сетью). То есть сеть обеспечивает точность экспоненциальной аппроксимации.

1.7 Выводы

Выкладки в данном разделе не учитывают налагаемые на нейронную сеть языком разработки и средой выполнения ограничения. Так, в разделе 3 будет показано, что из-за ограничений языка Python, глубина нейронной сети не может быть больше 997, а обучение сети глубиной более 30 слоёв занимает неоправданно большое количество времени. Также не учтена ограниченный объём обучающих данных. Тем не менее, даже учитывая ограничения выше, возможно создать сеть, удовлетворяющую условиям в 1.6.3, тем самым обеспечив точность экспоненциальной аппроксимации.

2. АДДИТИВНЫЕ ЦЕПОЧКИ

Этот раздел посвящён методам, призванным уменьшить число операций для нахождения полинома за счёт уменьшения числа операций при вычислении степени. В этом разделе будет рассмотрено понятие аддитивной цепочки (АЦ), а также приближённые алгоритмы нахождения АЦ.

Так, можно вычислить степень x^{27182} за 18 операций умножения, или степень x^{31415} за 19 умножений.

В этом разделе будут описаны алгоритмы, которые используют операции умножения для вычисления степеней, а если обобщить, то для вычисления произведения степеней, последовательностей степеней, или произведения последовательностей степеней.

2.1 Аддитивная цепочка

АЦ длины ℓ – это последовательность $\ell+1$ чисел, в которой первое число это 1, а каждое последующее число это сумма двух ранее представленных чисел. Другими словами, существует последовательность $c_0, c_1, \ldots, c_\ell$ такая, что $c_0 = 1$, а для каждого $k \in \{1, 2, \ldots, \ell\}$, существуют $i, j \in \{0, 1, \ldots, k-1\}$ такие, что $c_k = c_i + c_j$.

Например, $1, 2, 3, 5, 7, 14, 28, 56, 63$ это АЦ длины 8, потому что $2 = 1+1$, $3 = 2+1$, $5 = 3+2$, $7 = 5+2$, $14 = 7+7$, $28 = 14+14$, $56 = 28+28$ и $63 = 56+7$.

2.1.1 Использование АЦ для вычисления степеней

Короткие АЦ – это модель быстрых алгоритмов модульного возведения в степень. Для каждой АЦ $c_0, c_1, \ldots, c_\ell$ существует алгоритм, который вычисляет c_ℓ-ю степень за ℓ произведений, вычисляя последовательно c_1-ю степень, c_2-ю и так далее.

Например, дана m и взято число x между 0 и $m-1$, можно последовательно вычислить:

$$x^2 \bmod m = (x \cdot x) \bmod m,$$

$$x^3 \bmod m = \left(\left(x^2 \bmod m \right) \cdot x \right) \bmod m,$$

$$x^5 \bmod m = \left(\left(x^3 \bmod m \right) \cdot \left(x^2 \bmod m \right) \right) \bmod m,$$

$$x^7 \bmod m = \left(\left(x^5 \bmod m \right) \cdot \left(x^2 \bmod m \right) \right) \bmod m,$$

$$x^{14} \bmod m = \left(\left(x^7 \bmod m \right) \cdot \left(x^7 \bmod m \right) \right) \bmod m,$$

$$x^{28} \bmod m = \left(\left(x^{14} \bmod m \right) \cdot \left(x^{14} \bmod m \right) \right) \bmod m,$$

$$x^{56} \bmod m = \left(\left(x^{28} \bmod m \right) \cdot \left(x^{28} \bmod m \right) \right) \bmod m,$$

$$x^{63} \bmod m = \left(\left(x^{56} \bmod m \right) \cdot \left(x^7 \bmod m \right) \right) \bmod m,$$

получив $x^{63} \bmod m$ через 8 умножений по модулю m.

Минимизация времени возведения в степень – это не то же самое, что минимизация длины АЦ по нескольким причинам:

1) Время умножения обычно не константа. Умножение $x^e \bmod m$ на $x^f \bmod m$ может занять меньшее время, если $e = f$, например, или если $x^e \bmod m$ небольшой, или если $x^e \bmod m$ был прочитан или записан часто, или если $x^e \bmod m$ участвовал в предыдущем произведении.

2) Существует несколько способов выбрать (i, j) для $c_k = c_i + c_j$. Например, АЦ $1,2,3,4,7$ может иметь $4 = 3 + 1$ и $4 = 2 + 2$. Выбор иногда влияет на скорость возведения в степень, а АЦ не устанавливает выбора слагаемых.

3) Тот факт, что существует некая АЦ длины ℓ, содержащая n, не означает, что программист знает АЦ. Часто нужен алгоритм, который может вычислить n-ю степень или несколько n-х степеней, при заданном n в

качестве входных данных. Время, требующееся, чтобы найти АЦ добавляется ко времени, потраченном на произведения и должно быть минимизировано соответственно.

4) Иногда существуют алгоритмы быстрее для вычисления $x^n \bmod m$. Если m простое и n заметно больше m, например, тогда нужно сначала заменить n на $1+\big((n-1)\bmod(m-1)\big)$. Возможно есть более быстрые алгоритмы, даже когда n довольно мало.

Тем не менее, модель оказалась полезной. Например, Брауэр доказал в [8], что существует АЦ длины $\big(1+o(1)\big)\lg n$, содержащая n.

2.1.2 Поиск кратчайших АЦ

Рассматривается множество $\left(n_1,\ldots,n_p,\ell\right)$ такой, что существует АЦ длины ℓ, содержащая $n_1,\ldots,n_p$. Дауни, Леонг и Сети в [9] доказали, что такое множество NP-полное.

Многие ошибочно заявили, что Дауни, Леонг и Сети доказали, что $p=1$ подмножество NP-полное. Но этого пока никто не доказал. Возможно, есть быстрый алгоритм для нахождения АЦ, содержащей n минимальной длины.

Многие также неверно представляют NP-полноту как препятствие для поиска цепочек, которые обычно являются кратчайшими из возможных [2].

2.2 Алгоритм Брауэра

В 1939 Альфред Брауэр в [8] опубликовал алгоритм, который вычисляет n-ю степень через

$$\lg n+\frac{\big(1+o(1)\big)(\lg n)}{\lg\lg n}$$

произведений.

Кратчайшая АЦ, содержащая n имеет длину $\lg n$. Пал Эрдёш в [10] доказал, что почти для всех n, кратчайшая АЦ, содержащая n имеет длину $\lg n + (1+o(1))(\lg n)/\lg\lg n$. Следовательно, цепь Брауэра всегда находится в пределах $\lg n + (1+o(1))(\lg n)/\lg\lg n$ из самых коротких и почти всегда в пределах $1+o(1)/\lg\lg n$.

Цепь Брауэра фактически представляет собой семейство цепец, параметризованное положительным целым числом k и задающаяся рекурсивно как

$$B_k(n) = \begin{cases} 1,2,3,\ldots,2^k-1 & n < 2^k \\ B_k(q), 2q, 4q, 8q, \ldots, 2^k q, n & n \geq 2^k, q = \lfloor n/2^k \rfloor \end{cases}.$$

Другими словами: запись n в системе счисления 2^k как $2^{jk}c_j + \ldots + 2^{2k}c_2 + 2^k c_1 + c_0$ с $c_j \neq 0$. Получается цепь $1,2,3,\ldots,2^k-1$, затем $2c_j, 4c_j, 8c_j, \ldots, 2^k c_j$, затем $2^k c_j + c_{j-1}$, затем $2^{k+1}c_j + 2c_{j-1}, \ldots, 2^{2k}c_j + 2^k c_{j-1}$, затем $2^{2k}c_j + 2^k c_{j-1} + c_{j-2}$ и так далее.

Цепь Брауэра имеет длину $j(k+1) + 2^k - 2$ если $jk \leq \lg n < (j+1)k$. Длина минимизирована для k вблизи $\lg\lg n - 2\lg\lg\lg n$.

На рис. 2 показана цепь Брауэра для $k=3$ и $n=31415$, рис. 3 показывает цепь Брауэра для $k=3$ и $n=27182$. Обе эти цепи имеют длину 22.

Если $2^{511} \leq n < 2^{512}$, тогда цепь Брауэра имеет длину 649 для $k=4$, 642 для $k=5$ и 657 для $k=6$. Для $k=5$ можно записать $n = 2^{510}c_{102} + \ldots + 2^5 c_1 + c_0$, где каждый из 103 коэффициентов находится между 0 и 31 включительно. Цепь Брауэра включает 30 предварительных шагов, 510 удвоений, начиная с c_{102} и 102 сложения чисел $c_{101}, c_{100}, \ldots, c_0$.

$$
\begin{array}{rcll}
1 & = & 1_2 & \\
2 & = & 10_2 & \\
3 & = & 11_2 & \\
4 & = & 100_2 & \\
5 & = & 101_2 & \\
6 & = & 110_2 & \\
7 & = & 111_2 & \\
14 & = & 1110_2 & = 7 \cdot 2 \\
28 & = & 11100_2 & = 7 \cdot 2^2 \\
56 & = & 111000_2 & = 7 \cdot 2^3 \\
61 & = & 111101_2 & = 7 \cdot 2^3 + 5 \\
122 & = & 1111010_2 & = 7 \cdot 2^4 + 5 \cdot 2 \\
244 & = & 11110100_2 & = 7 \cdot 2^5 + 5 \cdot 2^2 \\
488 & = & 111101000_2 & = 7 \cdot 2^6 + 5 \cdot 2^3 \\
490 & = & 111101010_2 & = 7 \cdot 2^6 + 5 \cdot 2^3 + 2 \\
980 & = & 1111010100_2 & = 7 \cdot 2^7 + 5 \cdot 2^4 + 2 \cdot 2 \\
1960 & = & 11110101000_2 & = 7 \cdot 2^8 + 5 \cdot 2^5 + 2 \cdot 2^2 \\
3920 & = & 111101010000_2 & = 7 \cdot 2^9 + 5 \cdot 2^6 + 2 \cdot 2^3 \\
3926 & = & 111101010110_2 & = 7 \cdot 2^9 + 5 \cdot 2^6 + 2 \cdot 2^3 + 6 \\
7852 & = & 1111010101100_2 & = 7 \cdot 2^{10} + 5 \cdot 2^7 + 2 \cdot 2^4 + 6 \cdot 2 \\
15704 & = & 11110101011000_2 & = 7 \cdot 2^{11} + 5 \cdot 2^8 + 2 \cdot 2^5 + 6 \cdot 2^2 \\
31408 & = & 111101010110000_2 & = 7 \cdot 2^{12} + 5 \cdot 2^9 + 2 \cdot 2^6 + 6 \cdot 2^3 \\
31415 & = & 111101010110111_2 & = 7 \cdot 2^{12} + 5 \cdot 2^9 + 2 \cdot 2^6 + 6 \cdot 2^3 + 7
\end{array}
$$

Рисунок 2 – Вычисление числа 31415 через 22 произведения

$$
\begin{array}{rcll}
1 & = & 1_2 & \\
2 & = & 10_2 & \\
3 & = & 11_2 & \\
4 & = & 100_2 & \\
5 & = & 101_2 & \\
6 & = & 110_2 & \\
7 & = & 111_2 & \\
12 & = & 1100_2 & = 6 \cdot 2 \\
24 & = & 11000_2 & = 6 \cdot 2^2 \\
48 & = & 110000_2 & = 6 \cdot 2^3 \\
53 & = & 110101_2 & = 6 \cdot 2^3 + 5 \\
106 & = & 1101010_2 & = 6 \cdot 2^4 + 5 \cdot 2 \\
212 & = & 11010100_2 & = 6 \cdot 2^5 + 5 \cdot 2^2 \\
424 & = & 110101000_2 & = 6 \cdot 2^6 + 5 \cdot 2^3 \\
424 & = & 110101000_2 & = 6 \cdot 2^6 + 5 \cdot 2^3 + 0 \\
848 & = & 1101010000_2 & = 6 \cdot 2^7 + 5 \cdot 2^4 + 0 \cdot 2 \\
1696 & = & 11010100000_2 & = 6 \cdot 2^8 + 5 \cdot 2^5 + 0 \cdot 2^2 \\
3392 & = & 110101000000_2 & = 6 \cdot 2^9 + 5 \cdot 2^6 + 0 \cdot 2^3 \\
3397 & = & 110101000101_2 & = 6 \cdot 2^9 + 5 \cdot 2^6 + 0 \cdot 2^3 + 5 \\
6794 & = & 1101010001010_2 & = 6 \cdot 2^{10} + 5 \cdot 2^7 + 0 \cdot 2^4 + 5 \cdot 2 \\
13588 & = & 11010100010100_2 & = 6 \cdot 2^{11} + 5 \cdot 2^8 + 0 \cdot 2^5 + 5 \cdot 2^2 \\
27176 & = & 110101000101000_2 & = 6 \cdot 2^{12} + 5 \cdot 2^9 + 0 \cdot 2^6 + 5 \cdot 2^3 \\
27182 & = & 110101000101110_2 & = 6 \cdot 2^{12} + 5 \cdot 2^9 + 0 \cdot 2^6 + 5 \cdot 2^3 + 6
\end{array}
$$

Рисунок 3 – Вычисление числа 27182 через 22 произведения

Алгоритм Брауэра часто называют «2^k-ый метод слева направо», или, проще, «2^k-ый метод». Он довольно популярен и легко реализуем на практике. Построение цепочки для n заключается по сути в работе с битами числа n, поэтому не требует много памяти.

2.3 Алгоритм Яо

Эндрю Яо Цичжи в [11] опубликовал алгоритм, который, как и алгоритм Брауэра, вычисляет n-ю степень через

$$\lg n + \frac{\left(1+o(1)\right)(\lg n)}{\lg\lg n}$$

произведений.

Выбирается положительное целое число k. Число n записывается в 2^k-й системе счисления как $2^{jk}c_j + \ldots + 2^{2k}c_2 + 2^k c_1 + c_0$, где $c_j \neq 0$. Вводится $d(z)$ как сумма 2^{ik} для всех i, где $c_i = z$.

Цепь Яо начинается с $1,2,4,8,\ldots,2^{\lfloor \lg n \rfloor}$, складываются различные 2^{ik} для того, чтобы получить $d(z)$ для каждого $z \in \{1,2,3,\ldots,2^k-1\}$ такие, что $d(z)$ был ненулевым. Затем берутся $zd(z)$ для каждого z, и, наконец получается $n = d(1) + 2d(2) + 3d(3) + \ldots + (2^k-1)d(2^k-1)$.

Рис. 4 показывает цепь Яо для числа 27182. Длина цепи составляет 23.

1	=	1_2					
2	=	10_2					
4	=	100_2					
8	=	1000_2					
16	=	10000_2					
32	=	100000_2					
64	=	1000000_2	520	=	1000001000_2	=	$d(5)$
128	=	10000000_2	1040	=	10000010000_2	=	$2d(5)$
256	=	100000000_2	2080	=	100000100000_2	=	$4d(5)$
512	=	1000000000_2	2600	=	101000101000_2	=	$5d(5)$
1024	=	10000000000_2	4097	=	1000000000001_2	=	$d(6)$
2048	=	100000000000_2	8194	=	10000000000010_2	=	$2d(6)$
4096	=	1000000000000_2	16388	=	100000000000100_2	=	$4d(6)$
8192	=	10000000000000_2	24582	=	110000000000110_2	=	$6d(6)$
16384	=	100000000000000_2	27182	=	110101000101110_2		

Рисунок 4 – Вычисление числа 27182 через 23 сложения

2.4 Выводы

Алгоритмы Брауэра и Яо являются приближёнными. В работе рассматриваются полиномы, меньшие 100-й степени. Поэтому, соответственно, длина АЦ может находиться в пределах от 4 до 13ти, что вносит сильные колебания по времени аппроксимации полинома. Так, например, в зависимости от промежуточной степени, время обучения нейронных сетей для АЦ длины 13 может быть в 3 раза больше, чем для АЦ длины 4, так как алгоритм обучения будет вызываться 13 и 4 раза соответственно.

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

В данном разделе будут рассмотрены этапы построения ИНС и подбора параметров обучения. Также будут представлены результаты тестирования обучения НС на полиномах. Будет проведено сравнение производительности разных подходов к вычислению полинома (стандартного и с использованием АЦ для вычисления степени), также будут рассмотрены степени вида 2^x и проведено сравнение стандартного алгоритма с алгоритмом замены переменных.

3.1 Описание работы с исходными данными в НС

Имея набор данных – входные и выходные данные некоторой неизвестной функции, предполагается, что существует неизвестная базовая функция $f(x)$, которая согласованно отображает входные данные в выходные в целевом домене и приводит к исходному набору данных. Используя алгоритмы обучения НС, требуется аппроксимировать эту базовую функцию, то есть получить функцию $g(x)$, близкую к базовой.

При обучении с учителем набор данных состоит из входов и выходов (аргументов и значений некоторой функции), а алгоритм контролируемого обучения изучает, как лучше всего сопоставить параметры входных данных с параметрами выходных данных.

Можно рассматривать это отображение как управляемое математической функцией, называемой функцией отображение, и именно эту функцию алгоритм контролируемого обучения стремится наилучшим образом аппроксимировать.

Нейронные сети являются примером алгоритма обучения с учителем и стремятся приблизить функцию, представленную вашими данными. Это достигается путем вычисления ошибки между прогнозируемыми выходными

данными и ожидаемыми выходными данными и минимизации этой ошибки в процессе обучения.

Истинная функция, которая сопоставляет входы с выходами, неизвестна называется целевой функцией. Это цель процесса обучения, функция, которую нужно аппроксимировать, используя только доступные данные. Если бы целевая функция была бы известна, не было бы смысла ее приближать, т.е. не понадобился бы алгоритм машинного обучения с учителем. Следовательно, аппроксимация функции – полезный инструмент только в том случае, если основная целевая функция отображения неизвестна.

3.2 Подготовка данных для обучения НС

3.2.1 Перемешивание данных

В машинном обучении часто нужно перемешивать данные. Например, если нужно провести разделение на train / test и данные были заранее отсортированы по категориям (в случае используемых входных данных так и есть, так как функции имеют определённые промежутки возрастания или убывания), можно закончить обучение только на половине классов, что повлекло бы ухудшение качества обучения. Равномерное перемешивание гарантирует, что каждый элемент имеет одинаковый шанс встретиться в любой позиции. Поэтому перед подачей в НС данные перемешиваются случайным образом.

3.2.2 Нормировка данных

Перемешивание входных данных позволяет избавиться от последовательностей монотонного возрастания и убывания функции по оси абсцисс, но по оси ординат всё ещё имеется несколько диапазонов значений. Для того, чтобы НС не обучалась выборочно лучше или хуже на

определённых диапазонах, требуется нормировка. В данной работе выбрана линейная нормировка данных в пределах от 0 до 1, которая записывается как

$$x_k = \frac{x_k - x_{\min}}{x_{\max} - x_{\min}},$$

где x_k – k-е наблюдение, $x_{\min}$ – минимальное значение набора, $x_{\max}$ – максимальное значение набора.

Совокупность перемешивания и нормировки позволяет подготовить входные данные к обучению в НС, а НС в свою очередь, корректно обучиться на этих данных.

3.3 Метод обратного распространения ошибки

В машинном обучении метод обратно распространения ошибки – широко используемый алгоритм для обучения нейронных сетей с прямой связью. При подборе нейронной сети обратное распространение вычисляет градиент функции потерь относительно весов сети для одного примера ввода-вывода, и делает это эффективно, в отличие от наивного прямого вычисления градиента по каждому весу в отдельности. Эта эффективность делает возможным использование градиентных методов для обучения многослойных сетей, обновления весов для минимизации потерь; обычно используются градиентный спуск или такие варианты, как стохастический градиентный спуск. Алгоритм обратного распространения работает путем вычисления градиента функции потерь по отношению к каждому весу по правилу цепочки, вычисления градиента по одному слою за раз, итерации назад от последнего уровня, чтобы избежать избыточных вычислений промежуточных членов в цепном правиле; это пример динамического программирования.

Ошибка – это метрика, которая высчитывается как разница между параметром на входе сети и истинным. Её обозначают как $E(W)$, здесь W – веса определённого слоя. Метод обратного распространения заключается в

изменении весов в противоположную сторону от направления градиента функции ошибки в точке W, обозначающимся как $\nabla E(W)$.

Градиент ошибки записывается как

$$\nabla E(W) = \left[\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_k} \right],$$

а производная функции ошибки как

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial s_j} \frac{\partial s_j}{\partial w_{ij}},$$

где w_{ij} – вес ребра между нейронами i и j, y_j – выход i, s_j – состояние j.

$\frac{\partial y_j}{\partial s_j}$ – это производная функции активации по состоянию нейрона.

$\frac{\partial s_j}{\partial w_{ij}}$ – это производная состояния нейрона по весу связи нейронов.

$\frac{\partial E}{\partial y_j}$ – определяется как значение функции ошибки на нейроне выходного слоя, а на внутреннем слое определяется по следующей формуле:

$$\frac{\partial E}{\partial y_i} = \frac{\partial y_i}{\partial s_i} \cdot \sum_j \frac{\partial E}{\partial y_j} w_{ij}.$$

Это и есть обратное распространение ошибки.

Теперь можно записать общий алгоритм градиентного метода обучения.

Алгоритм градиентного метода обратного распространения ошибки

Инициализация весов W_0

Задание параметров конфигурации p

Задание граничного значения ошибки $E_{\max}$ или эпох обучения t_{ep}

пока $E(W_t) > E_{\max}$ **или** $t < t_{ep}$

$t \leftarrow t + 1$

Вычисление ошибки $E(W_{t-1})$

Вычисление градиента $\nabla E_t \leftarrow \nabla E(W_{t-1})$

Вычисление изменения весов $\Delta W_t = F(\nabla E_t, \Delta W_{t-1}, W_{t-1} p,)$

$W_t \leftarrow W_{t-1} - \Delta W_t$

3.4 Функции оптимизации

Метод обратного распространения ошибки используется в таких функциях оптимизации как стохастический градиентный спуск (SGD) и метод адаптивной оценки моментов (Adam). В этом пункте будут рассмотрены оба этих метода.

3.4.1 SGD

Стохастический градиентный спуск - это итерационный метод оптимизации целевой функции с подходящими свойствами гладкости. Его можно рассматривать как стохастическое приближение оптимизации градиентного спуска, поскольку оно заменяет фактический градиент (вычисленный из всего набора данных) его оценкой (вычисленной из случайно выбранного подмножества данных). Это снижает вычислительную нагрузку, особенно в задачах оптимизации большой размерности, обеспечивая более быстрые итерации для более низкой скорости сходимости. [12] Хотя основная идея стохастической аппроксимации восходит к алгоритму Роббинса – Монро 1950-х годов, стохастический градиентный спуск стал важным методом оптимизации в машинном обучении [13].

Этот метод принимает в качестве параметра скорость обучения a. Изменение параметров вычисляется по формуле

$$\Delta W_t = \alpha \nabla E_t .$$

Основным преимуществом SGD является его эффективность, которая в основном линейна по количеству обучающих примеров.

3.4.2 Adam

Метод адаптивной оценки моментов использует средние значения как градиентов, так и вторых моментов градиентов. Учитывая параметры $\omega^{(t)}$ и функцию потерь $L^{(t)}$, где t индексирует текущую операцию обучения (индексируется с 0), обновление параметров Adam производится как:

$$m^{(t+1)}{}_{\omega} \leftarrow \beta_1 m^{(t)}{}_{\omega} + (1-\beta_1)\nabla_{\omega} L^{(t)},$$

$$v^{(t+1)}{}_{\omega} \leftarrow \beta_2 v^{(t)}{}_{\omega} + (1-\beta_2)\left(\nabla_{\omega} L^{(t)}\right)^2,$$

$$\hat{m}_{\omega} = \frac{m^{(t+1)}{}_{\omega}}{1-\beta^{t+1}{}_1}\ldots,$$

$$\hat{v}_{\omega} = \frac{v^{(t+1)}{}_{\omega}}{1-\beta^{t+1}{}_2},$$

$$\omega^{(t+1)} \leftarrow \omega^{(t)} - \eta \frac{\hat{m}_{\omega}}{\sqrt{\hat{v}_{\omega}} + \varepsilon},$$

где ε это небольшой скаляр (например, 10^{-8}), используемый для предотвращения деления на 0, β_1 и β_2 – факторы затухания для градиентов и секундные моменты градиентов соответственно. Возведение в квадрат и извлечение корня выполняется поэлементно.

Метод Adam работает более медленно из-за большего количества вычислений, чем в методе SGD, тем не менее, Adam сходится быстрее метода SGD [14].

3.5 Функция потерь

Градиентные алгоритмы оптимизации требуют определения функции потерь (или функции ошибки). Это непосредственно функция, которую будут минимизировать алгоритмы оптимизации. В работе за функцию потерь была взята ошибка MSE, также называемая среднеквадратичной ошибкой. Она записывается как

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2,$$

где n – длина вектора предсказанных значений, y – вектор значений предсказываемой переменной, а $\hat{y}$ – вектор предсказанных значений.

3.6 Функция активации ReLU

В контексте ИНС функция активации ReLU (Rectified Linear Unit), также называемая функцией выпрямления, является функцией активации, определённой как положительная часть её аргумента. Она записывается как

$$f\left(x\right) = x^{+} = \max\left(0, x\right),$$

где x – вход нейрона.

Эта функция активации начала проявляться в контексте извлечения визуальных признаков в иерархических нейронных сетях, начиная с конца 1960-х годов [15] [16]. Позже утверждалось, что у этого есть сильные биологические мотивы и математическое обоснование. [17] [18] В 2011 году было обнаружено, что она позволяет лучше обучать глубокие сети [19] по сравнению с широко используемыми функциями активации, например, логистической сигмоидой и ее более практичным [20] аналогом, гиперболическим тангенсом.

3.7 Алгоритм аппроксимации полинома с помощью НС

Код программы представлен в приложении А. Программа реализована на языке Python 3.9, среда выполнения кода – Google Colab. Для создания нейронной сети была использована библиотека Keras Tensorflow, которая позволяет создать последовательную (Sequential) модель НС, добавить в неё определённое количество слоёв, настроить функцию оптимизации, функцию потерь, количество эпох обучения и размер пакета.

В работе в НС были протестированы обе функции оптимизации (Adam и SGD), но выбор был остановлен на функции Adam, так как она обеспечивает более быструю сходимость (то есть для аппроксимации требуется меньшее количество эпох). Также, дополнительно, был реализован коллбек, который сохраняет лучшую модель, основываясь на значении функции потерь. Также был использован полиномиальный спад скорости обучения в зависимости от количества шагов обучения (начальное значение – 0.0001, конечное значение – 0.00001). Размер пакеты был выставлен в значение 131072, которое является степенью 2. Настолько большой размер пакета был использован для того, чтобы исключить попадание в локальный минимум при обучении, что вызвало бы остановку прогресса обучения на определённом промежуточном этапе и не позволило бы корректно аппроксимировать функцию.

В начале работы программы задаётся полином, который разбивается на составляющие (степени и коэффициенты). Для некоторого вектора x аргументов полинома генерируются значения полиномиальной функции, оба эти параметра вместе со степенью полинома передаются в функцию, осуществляющую обучение НС.

Архитектура НС строится в соответствии с архитектурой, представленной в разделе 1 работы в пункте 1.6.3. Такая архитектура обеспечивает экспоненциально возрастающую точность аппроксимации. Количество слоёв зависит от теоретически вычисленного по формуле (2) количества. Для корректного обучения количество слоёв ограничено сверху значением 30, что теоретически позволительно по формуле (2). Количество эпох линейно зависит от значения формулы (2).

Так как в общем случае доподлинно неизвестна функция, которую требуется аппроксимировать, вектор x аргументов функции остаётся неизменным, а значения функции генерируются заново случайным образом (в соответствии с 3м параметром алгоритма обучения НС). Затем происходит

перемешивание и нормировка входных данных. Векторы-строки приводятся к векторам-столбцам, затем происходит обучение НС. Модель, для которой функция потерь минимальна, становится моделью, на основе которой будут предсказываться значения аппроксимированной функции.

Получив обученную модель, алгоритм осуществляет предсказание по ней значений аппроксимированной функции. С помощью метрики MSE вычисляется ошибка между истинными значениями функции и предсказанными, строится наглядный график истинной и аппроксимированной функции. Пример такого графика для функции x^3 на интервале $[-50,50]$ представлен на рис. 5.

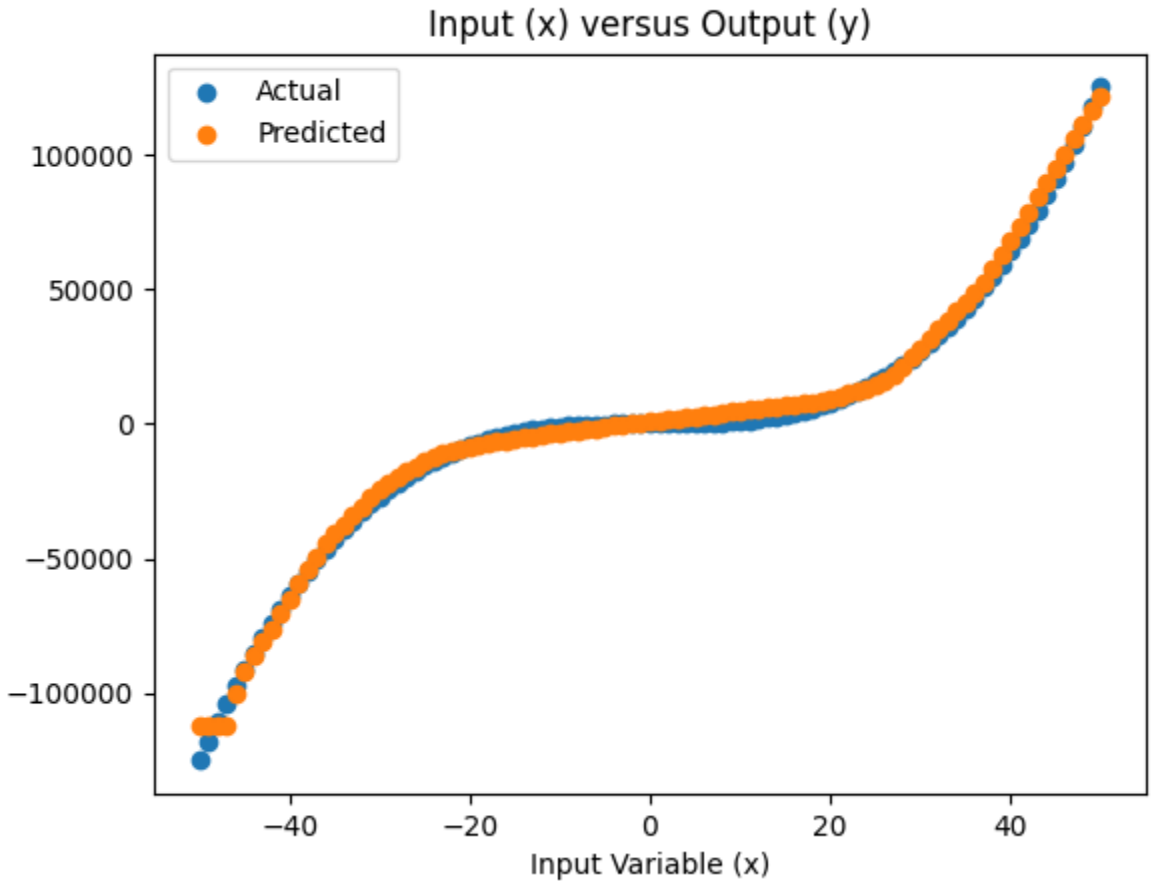

Рисунок 5 – Аппроксимация функции x^3 на интервале $[-50,50]$

Из-за ограничений сверху реализации целочисленного int в Python, далее аппроксимация производилась на интервале $[-1,1]$.

3.8 Аппроксимация функций вида x^{2^k}

Перед применением АЦ для разложения степеней, можно сравнить эффективность стандартного алгоритма и алгоритма замены переменных. Под стандартным алгоритмом подразумевается алгоритм, подающий для обучения в НС сразу функцию степени 2^k без её разложения. Под алгоритмом замены переменных подразумевается такой алгоритм, который последовательно производит замену $z = x^2$, то есть заменяет квадрат на 1-ю степень. Таким образом, если имеется функция x^{2^3}, сначала производится замена $z = x^2$, которая аппроксимируется НС как функция 2-й степени. Затем, производится замена $z_1 = z^2$ и также аппроксимируется НС как функция 2-й степени, и 3-я аналогично замена $z_2 = z_1^2$. В то же время, стандартный алгоритм в примере сразу подаст на аппроксимацию функцию x^{2^3} степени 8. Таким образом, требуется сравнить время работы стандартного алгоритма и алгоритма замены переменных, например, для функции x^{2^6}.

На рис. 6-11 представлены результаты аппроксимации функций x^2 - x^{2^6} для алгоритма замены переменных.

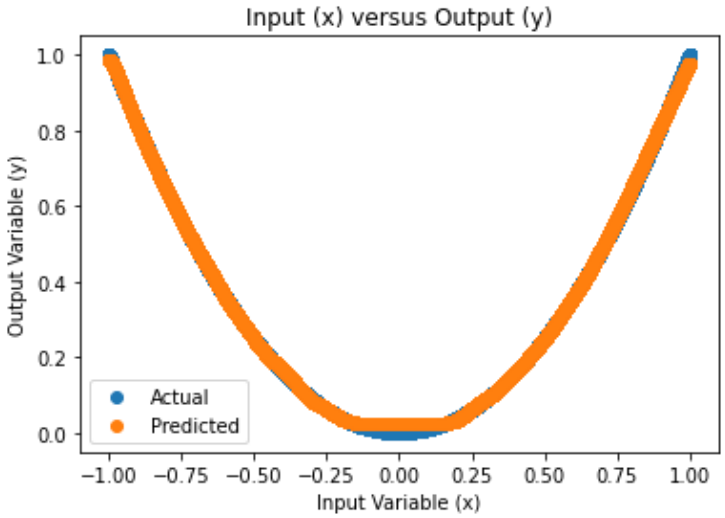

Рисунок 6 – Аппроксимация функции x^2

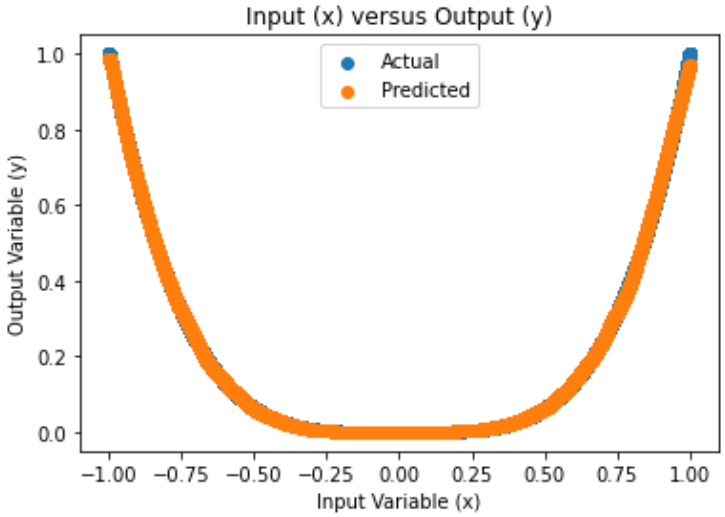

Рисунок 7 – Аппроксимация функции x^{2^2}

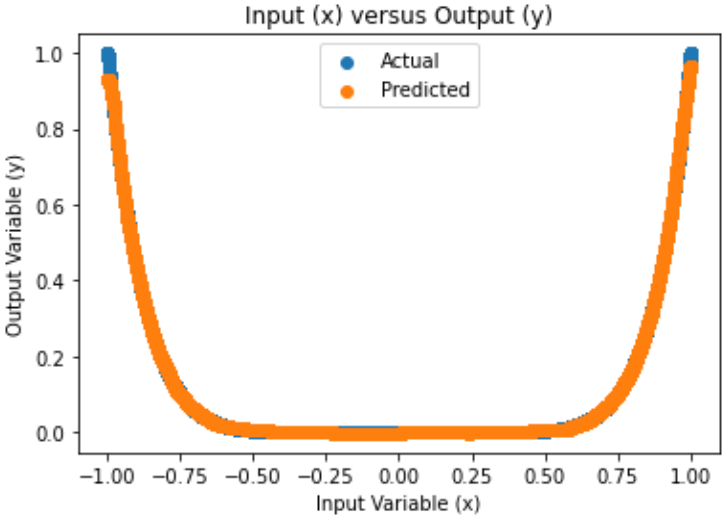

Рисунок 8 – Аппроксимация функции x^{2^3}

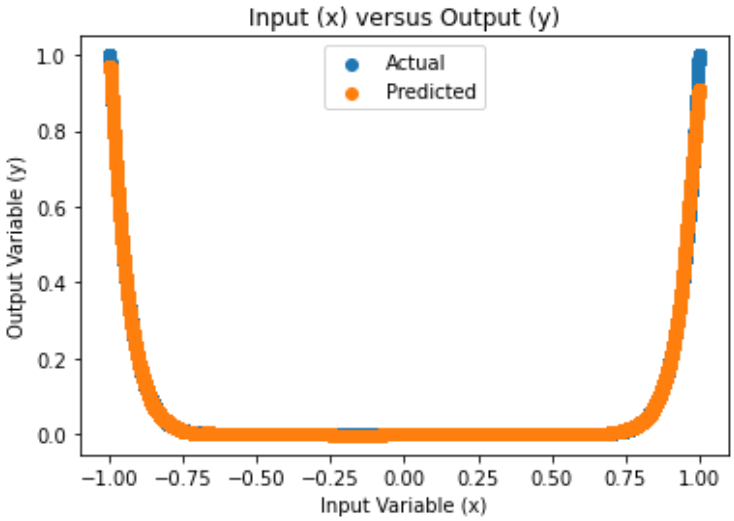

Рисунок 9 – Аппроксимация функции x^{2^4}

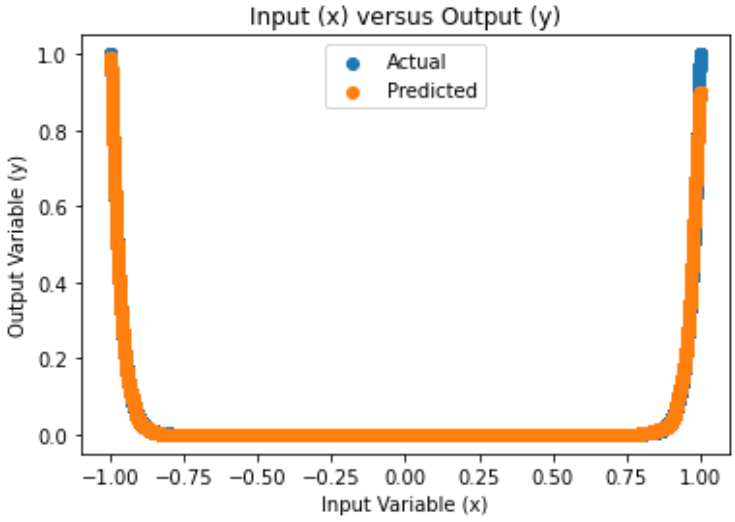

Рисунок 10 – Аппроксимация функции x^{2^5}

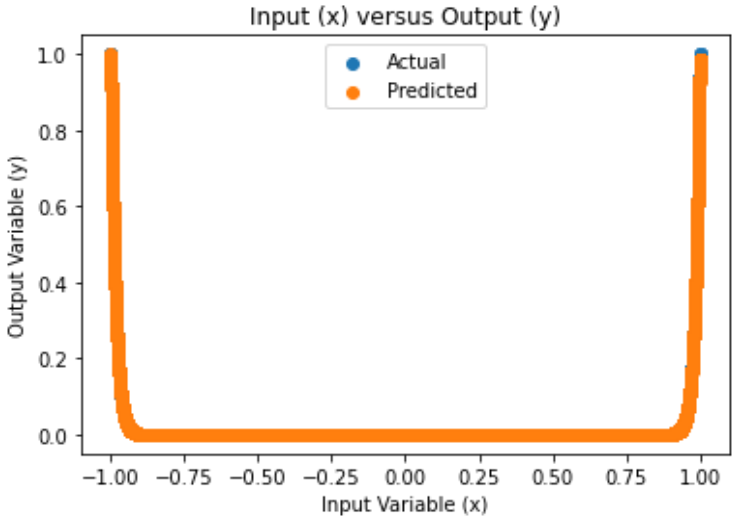

Рисунок 11 – Аппроксимация функции x^{2^6}

На рис. 12 представлены результаты аппроксимации функции x^{2^6} стандартным алгоритмом.

По рис. 11 и 12 видно, что функция x^{2^6} аппроксимировалась лучше стандартным алгоритмом, чем алгоритмом замены переменных. Это можно объяснить большим количеством эпох в стандартном алгоритме.

Стандартный алгоритм отработал за 22025 секунд, алгоритм замены переменных отработал за 1924 секунды. Таким образом, можно сделать вывод о том, что гораздо эффективнее раскладывать степень полинома перед обучением НС. Это приводит к реализации алгоритмов АЦ.

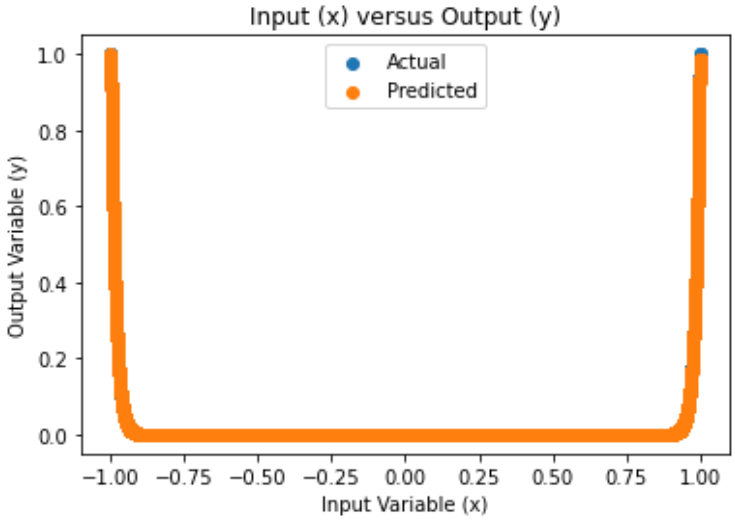

Рисунок 12 – Аппроксимация функции x^{2^6} стандартным алгоритмом

В следующем подпункте, учитывая предыдущий вывод об эффективности алгоритмов, раскладывающих степень, были исследованы эффективности алгоритмов Брауэра и Яо в сравнении со стандартным.

3.8.1 Сравнение эффективности приближенных алгоритмов вычисления АЦ в сравнении со стандартным.

В работе были реализованы алгоритмы Брауэра и Яо. Код программы представлен в приложении А. Принцип их работы описан в разделе 2 в пунктах 2.2 и 2.3. Тестирование скорости работы алгоритмов будет производиться на полиноме $15x^{51}+9x^2+34$. Степень данного полинома позволяет наглядно продемонстрировать работу каждого алгоритма.

Так, число из числа 51 алгоритм Брауэра при $k=3$ строит АЦ $1,2,3,4,5,6,7,12,24,48,51$, другими словами $51=6\cdot 2^3+3$. АЦ алгоритма Яо выглядит как $1,2,3,6,8,16,24,32,40,48,51$, другими словами, $51=d(1)+2d(2)+3d(3)+4d(4)+5d(5)+6d(6)+7d(7)=6\cdot 8+3\cdot 1$. Для степеней 2 и 0 аппроксимация происходит за одну операцию.

На рис. 13-21 представлены результаты работы алгоритма Брауэра, на рис. 22-30 представлены результаты работы алгоритма Яо, на рис. 31 представлены результаты работы стандартного алгоритма.

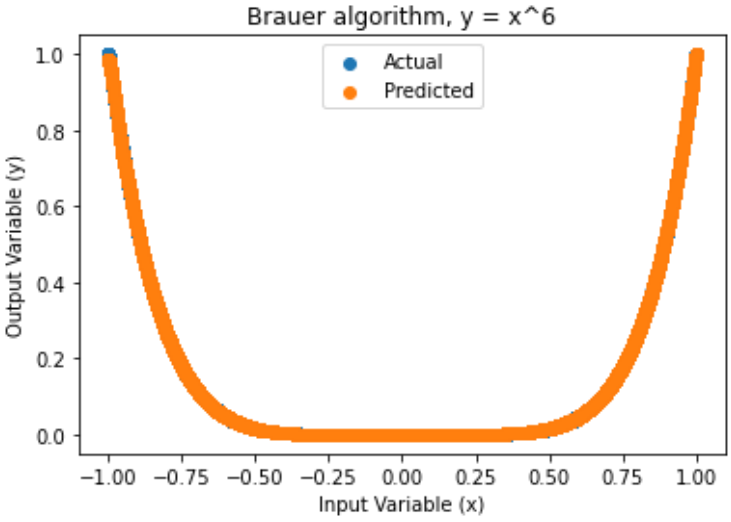

Рисунок 13 – Аппроксимация функции x^6, алгоритм Брауэра

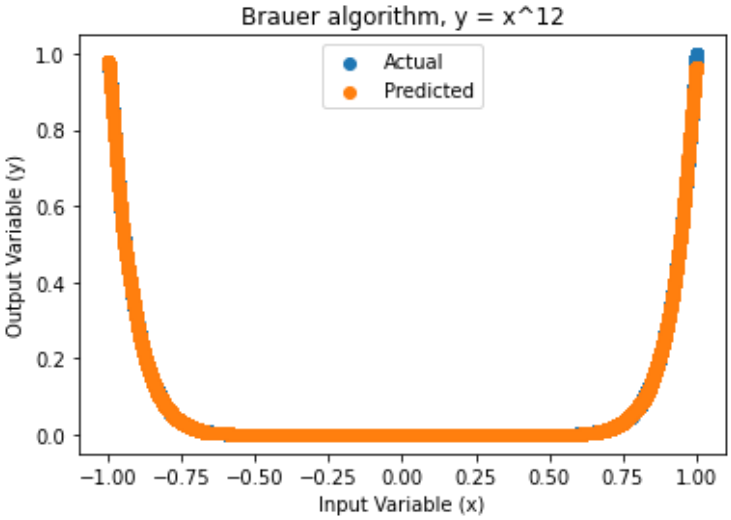

Рисунок 14 – Аппроксимация функции x^{12}, алгоритм Брауэра

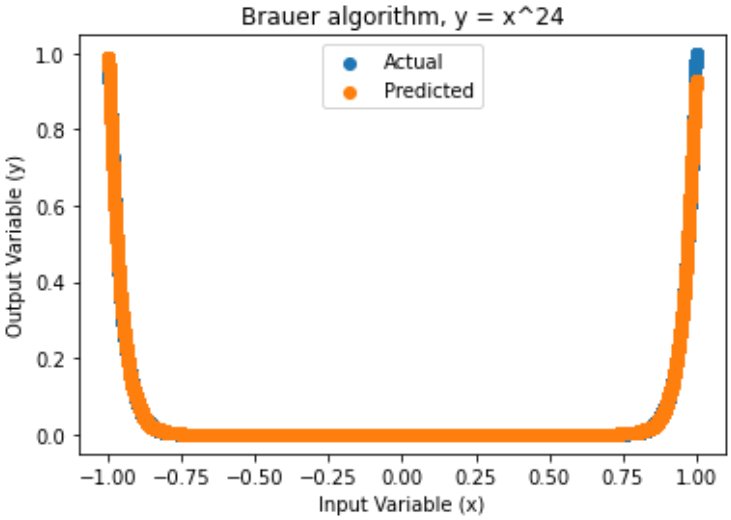

Рисунок 15 – Аппроксимация функции x^{24}, алгоритм Брауэра

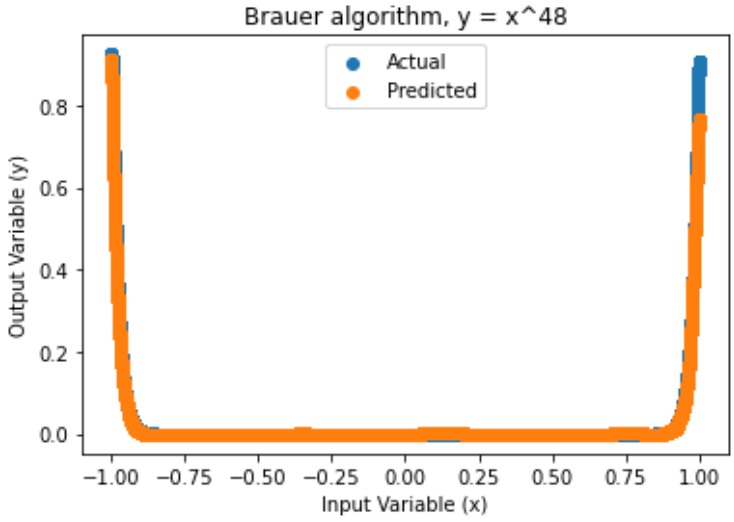

Рисунок 16 – Аппроксимация функции x^{48}, алгоритм Брауэра

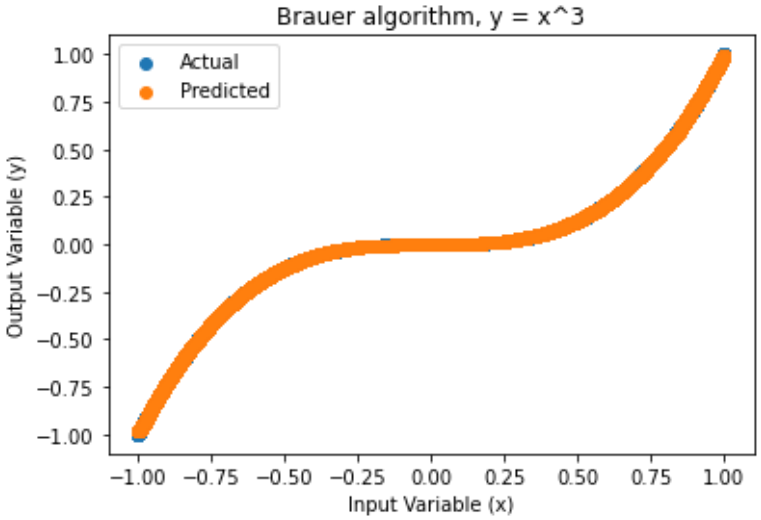

Рисунок 17 – Аппроксимация функции x^{3}, алгоритм Брауэра

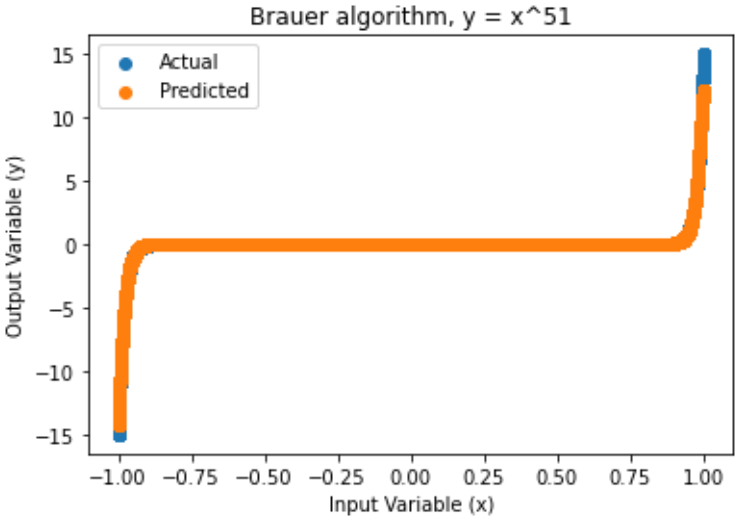

Рисунок 18 – Аппроксимация функции x^{51}, алгоритм Брауэра

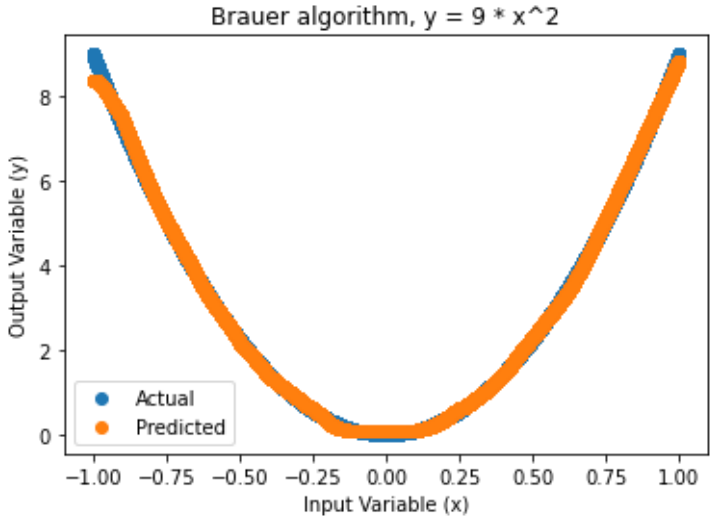

Рисунок 19 – Аппроксимация функции $9x^2$, алгоритм Брауэра

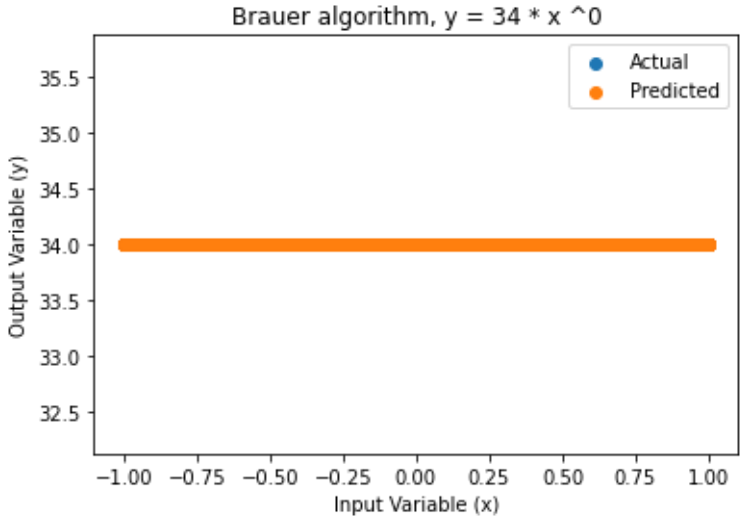

Рисунок 20 – Аппроксимация функции $34x^0$, алгоритм Брауэра

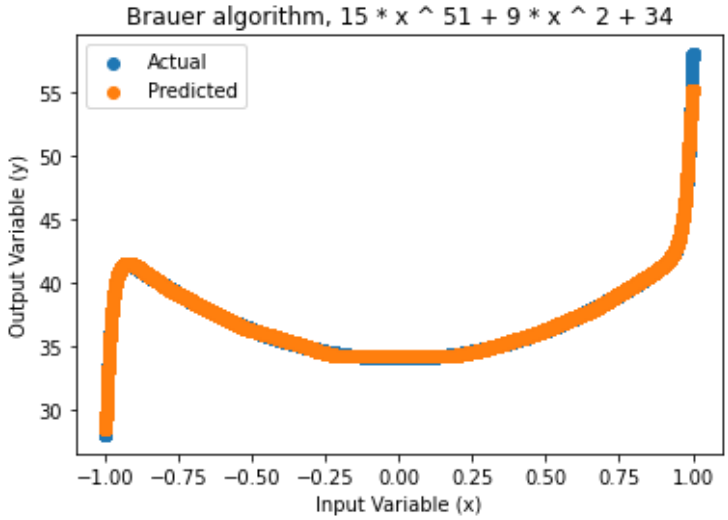

Рисунок 21 – Аппроксимация функции $15x^{51} + 9x^2 + 34$, алгоритм Брауэра

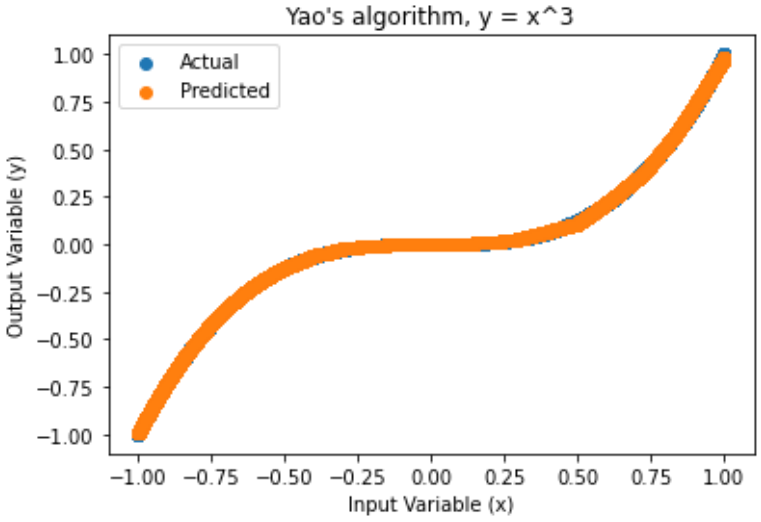

Рисунок 22 – Аппроксимация функции x^3, алгоритм Яо

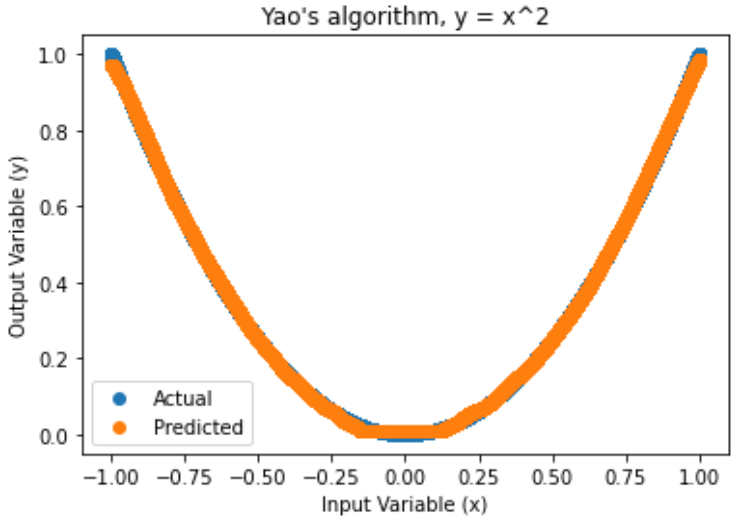

Рисунок 23 – Аппроксимация функции x^2, алгоритм Яо

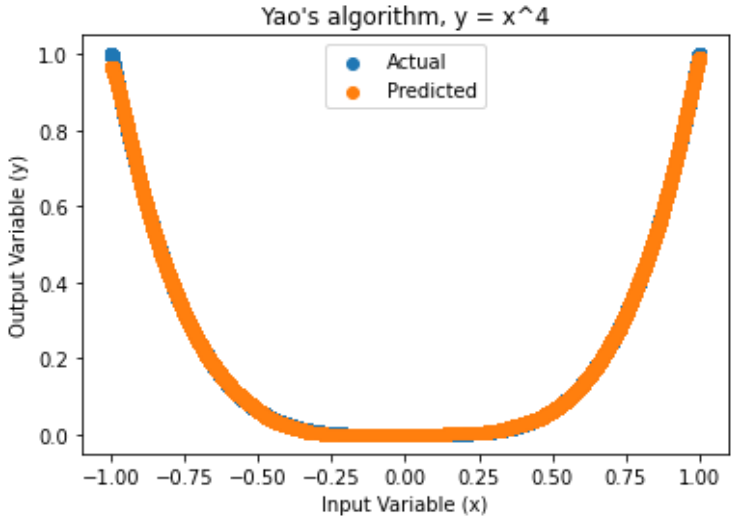

Рисунок 24 – Аппроксимация функции x^4, алгоритм Яо

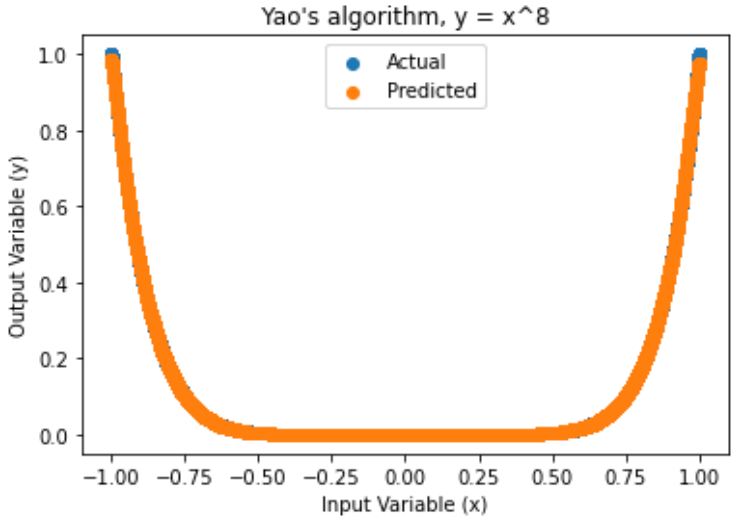

Рисунок 25 – Аппроксимация функции x^8, алгоритм Яо

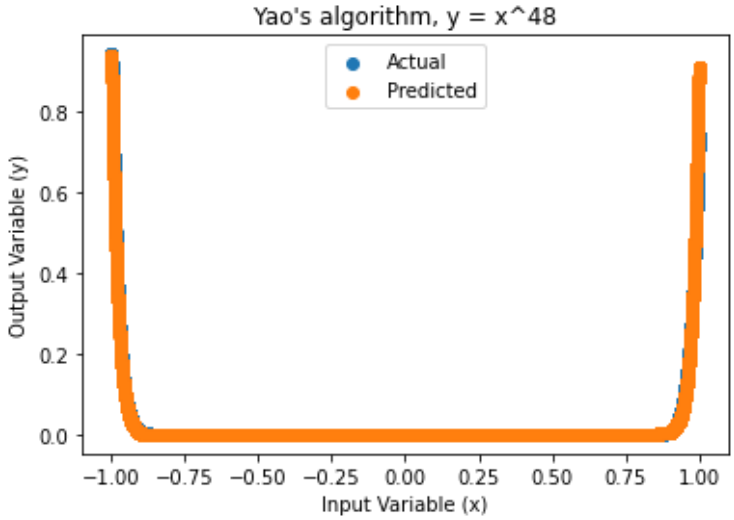

Рисунок 26 – Аппроксимация функции x^{48}, алгоритм Яо

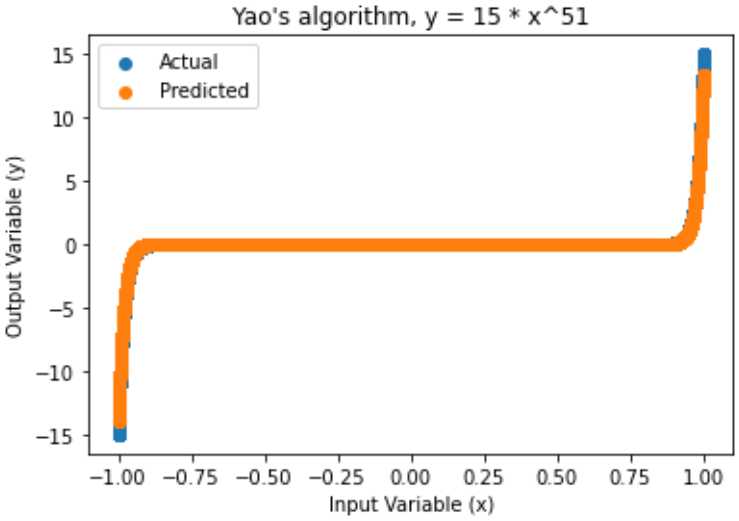

Рисунок 27 – Аппроксимация функции $15x^{51}$, алгоритм Яо

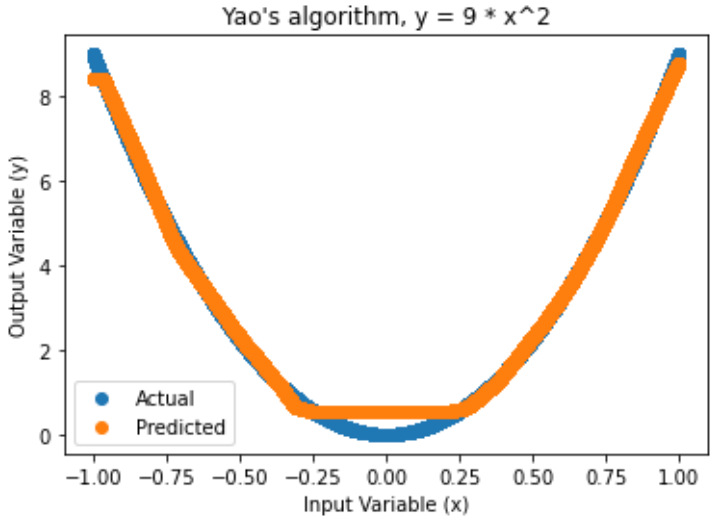

Рисунок 28 – Аппроксимация функции $9x^2$, алгоритм Яо

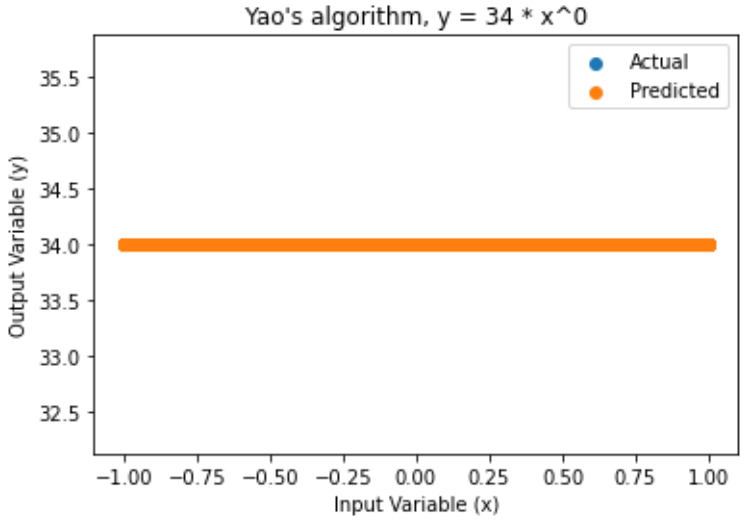

Рисунок 29 – Аппроксимация функции $34x^0$, алгоритм Яо

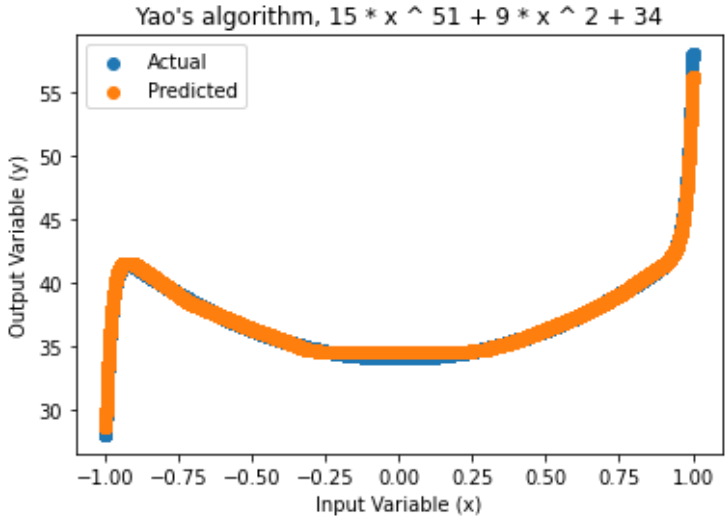

Рисунок 30 – Аппроксимация функции $15x^{51} + 9x^2 + 34$, алгоритм Яо

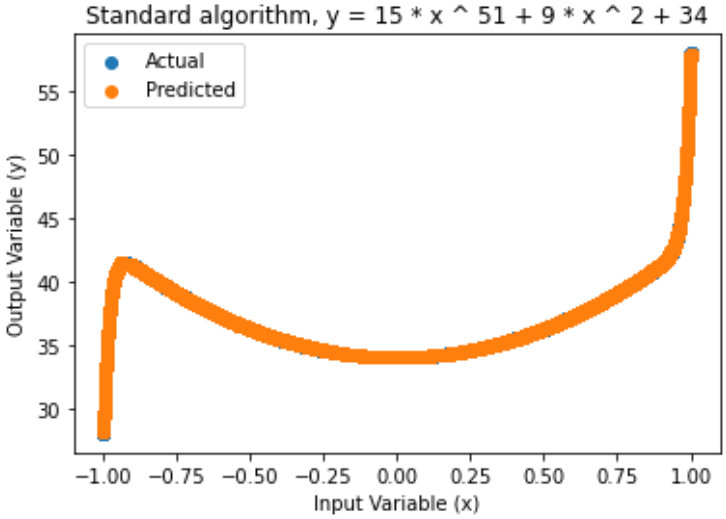

Рисунок 30 – Аппроксимация функции $15x^{51}+9x^{2}+34$, стандартный алгоритм

Алгоритм Брауэра отработал за 3190 секунд, алгоритм Яо за 3158 секунд, а стандартный алгоритм за 10112 секунд. Таким образом, самым быстрым оказался алгоритм Яо, а самым медленным (более чем в 3 раза) – стандартный алгоритм.

3.9 Выводы

Представленный полином позволяет наглядно продемонстрировать на практике преимущества использования АЦ при вычислении степеней. Так, при анализе функций вида x^{2^k}, прирост производительности по сравнению со стандартным алгоритмом составил 20000 секунд, что в 11 раз больше, чем метод замены переменных. В случае алгоритмов построения АЦ, алгоритм Яо оказался быстрее алгоритма Брауэра за счёт обработки в среднем полиномов меньших степеней, но они оба, в среднем в 3 раза быстрее стандартного алгоритма.

4. ОБЕСПЕЧЕНИЕ КАЧЕСТВА РАЗРАБОТКИ, ПРОДУКЦИИ, ПРОГРАММНОГО ПРОДУКТА

ЗАКЛЮЧЕНИЕ

В данной работе были проанализированы теоретические выкладки по теме построения архитектуры НС и найдено их практическое приложение к построению реальных НС на их основе. Так, в теории, описанной в 1-м разделе, не учитывалось то, что, например, что набор обучающих данных технически ограничен, то есть нельзя обучить НС на бесконечном наборе обучающих данных. Также не учитывалось время обучения. Так, на практике даже после использования ограниченного набора обучающих данных на конечном количестве эпох обучения, суммарное время обучения НС составило около 6 часов, то есть, в этом плане, всё ещё остается пространство для улучшения. Тем не менее, реализованная в работе НС является оптимальной в том плане, что удовлетворяет теории и обеспечивает качественную аппроксимацию полиномов за конечное время. То, что НС удовлетворяет теории подтверждается вхождением абсолютной разности реальных результатов и предсказанных НС в интервал заранее заданного ε.

В стандартном подходе (без разложения степени полинома) очевидна проблема большого количества затраченного времени на обучение НС. Эта проблема была решена сначала за счёт применения алгоритма замены переменных (однако, он работает только для функций вида x^{2^k}), а затем за счёт применения алгоритмов построения АЦ, таких как алгоритм Брауэра и алгоритм Яо. Таким образом, с помощью применения данных алгоритмов удалось достичь прироста производительности в 11 раз для функций вида x^{2^k} и в 3 раза для полиномов. Тем не менее, даже при таком приросте производительности, есть возможность улучить временные результаты за счёт использования более точных алгоритмов построения АЦ (которые строят АЦ длины, близкой к минимальной), такие как алгоритмы нахождения звёздных цепочек с помощью вектора добавок. Такие алгоритмы могут оказаться полезными при вычислении полиномов степени больше 100.

Результаты данной работы можно использовать для дальнейшего анализа теории НС в плане применения её на практике. Чисто практическая польза данной работы заключается в реализации НС, осуществляющей аппроксимацию полинома (притом, что базовая функция может быть неизвестна) на основе только лишь пар аргумент-значение полинома и использовании эффективных алгоритмов построения АЦ для аппроксимации полинома.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Deep Neural Network Approximation Theory / Philipp Grohs, Dmytro Perekrestenko, Dennis Elbrachter и др. // IEEE Transactions on Information Theory. 2021, вып. (№) 5. С. 2581-2623.
2. Daniel J. Bernstein. Pippenger’s eponentation algorithm // http://cr.yp.to/papers.html#pippenger. 2002. С. 21.
3. C. Fefferman. Reconstructing a neural net from its output // Revista Matematica Iberoamericana. 1994, вып. (№) 3. С. 507-555.
4. P. Peterset and F. Voigtlaender. Optimal approximation of piecewise smooth functions using deep ReLU neural networks // Neural Networks. 2018, вып. (№) 108. С. 296-330.
5. G. Cybenko. Approximation by superpositions a sigmoidal function // Mathematics of Cintrol, Signals and Systems. 1989, вып. (№) 4. С. 303-314.
6. K. Hornik. Approximation capabilities of multilayer feedforward networks // Neural Networks. 1991, вып. (№) 2. С. 251-257.
7. D. E. Rummelhart, G. E. Hinton and R. J. Williams. Learning representations by back-propagating errors // Nature. 1986, вып. (№) 6088. С. 533-536.
8. Alfred Brauer. On addition chains // Bukketin of the American Mathematical Society. 1939, вып. (№) 45. С. 736-739.
9. Peter Downey, Benton Leong, Ravi Sethi. Computing sequences with additional chains // SIAM Journal on Computing. 1981, вып. (№) 10. С. 638-646.
10. Paul Erdos. Remarks on number theory III: On additional chains // Acta Arithmetica. 1960, вып. (№) 6. С. 77-81.
11. Andrew C. Yao. On the evaluation of powers // SIAM Journal on Computing. 1976, вып. (№) 5. С. 100-103.
12. Leon Bottou, Olivier Bousquet. The Tradeoffs of Large Scale Learning // MIT Press. 2012, вып. (№) 2. С. 351-368.

13. Leon Bottou. Online Algorithms and Stochastic Aprroximations // Cambridge University Press. 1998, вып. (№) 11. С. 108-119.

14. J. C. Spall. Feedback and Weighting Mechanisms for Improving Jacobian Estimates in the Adaptive Simultaneous Perturbation Algorithm // IEEE Transactions on Automatic Control. 2009, вып. (№) 6. С. 1216-1229.

15. K. Fukushima. Visual feature extraction by a multilayered network of analog threshold elements // IEEE Transactions on Systems Science and Cybernetics. 1969, вып. (№) 4. С. 322-333.

16. K. Fukushima, S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition // in Competition and cooperation in neural nets. 1982, вып. (№) 7. С. 267-285.

17. Dan Hendrycks, Kevin Gimpel. Gaussian Error Linear Units // https://arxiv.org/abs/2006.02427. 2016. С. 30.

18. R. Hahnloser, H. S. Seung. Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks // NIPS. 2001, вып. (№) 8. С. 10-20.

19. Glorot Xavier, Bordes Antoine and Bengio Yoshua. Deep sparse rectifier neural networks // AISTATS. 2011, вып. (№) 1. С. 64-80.

20. LeCun Yann, Bottou Leon. Efficient BackProp // Springer. 1998, вып. (№) 6. С. 45-61.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ