МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет компьютерных наук

Кафедра цифровых технологий

Нейронные сети в машинном обучении

Курсовая работа

Направление: 02.03.01 Математика и компьютерные науки

Профиль: Распределенные системы и искусственный интеллект

Зав. Кафедрой _____ С. Д. Кургалин, д. ф.-м. н., профессор

Обучающийся _____ А. М. Гузенко, 3 курс

Руководитель _____ А. Ф. Клинских, д. ф.-м. н., профессор
подпись *расшифровка подписи, ученая степень, ученое звание*

Воронеж 2021

Содержание

Введение

Обучающиеся машины

Интерес к машинному обучению необычайно возрос за последние 10 лет, благодаря увеличению вычислительных мощностей и объема данных, в том числе и общедоступных в сети интернет. Вы можете видеть машинное обучение в программах высших учебных заведений по компьютерным наукам, и на страницах журналах по типу «Wall Street Journal» каждый день. Машинное обучение – это применение алгоритмов, которые извлекают информацию из исходных данных и представляют ее в виде той или иной модели. Эта модель в будущем используется для вывода других данных, которые в модели отсутствуют.

Нейронные сети - один из типов моделей машинного обучения, они существуют уже по меньшей мере 50 лет. Фундаментальной единицей нейронной сети является узел, который является приблизительным аналогом биологического нейрона в мозге млекопитающих. Связь между узлами так же смоделирована на примере биологического мозга, ровно, как и эволюция с течением времени (обучение с «учителем»).

В середине 1980-ых и начале 1990-ых много важных открытий было произведено в области архитектуры нейронных сетей. Однако, большое количество времени и данных требовалось для хороших результатов, что замедляло их практическое внедрение, в следствие чего интерес угас. В начале 2000-ых вычислительные мощности росли экспоненциально, и индустрия увидела «кембрийский взрыв» в области методов вычислений, которые до этого были невозможны. Глубокое обучение, которое появилось благодаря столь стремительному развитию, показало себя как уверенного конкурента в области машинного обучения.

Интерес не угасает и до сегодняшнего дня, глубокое обучение можно найти в любом закоулке машинного обучения.

Как машины могут учиться?

Для определения того, как машины могут обучаться мы должны определить, что мы понимаем под «обучением». В быту, когда мы говорим “обучение”, мы имеем в виду приобретение знаний посредством изучения, опыта или в процессе обучения с учителем. Сместив немного наш фокус, мы можем считать, что машинное обучение – это применение алгоритмов для получения структурного описания из данных-примеров. Компьютер обучается информационным структурам, которые предоставляют исходные данные. Структурное описание – другое название моделей, которые мы получаем из данных и которые мы можем использовать в дальнейшем для предсказания неизвестных данных.

Структурные описания, или модели, могут иметь разные форму, например:

- Дерево решений
- Линейная регрессия
- Весы связей в нейронных сетях

Каждая модель имеет разный путь обработки имеющихся данных и предсказания неизвестны данных. Дерево решений создает набор правил в форме дерева, а линейная регрессия набор параметров, представляющих входные данные.

В нейронных сетях имеется так называемый вектор параметров, представляющий веса связей между узлами в сети.

Артур Самуэль пионер в области искусственного интеллекта (ИИ), работающий, в IBM и Стэнфордском Университете, давал определение машинному обучению следующим образом: «Область

исследований, которая наделяет компьютер возможностью обучаться без явного программирования». Самуэль разработал программу, которая могла играть в шашки и изменяла свою стратегию по мере обучения, она ассоциировала позиции и вероятность победы в игре.

Это фундаментальная схема поиска паттернов, которые ведут к победе или поражению, с последующим анализом и подкреплением успешных паттернов, данная схема по сей день лежит в основе машинного обучения и ИИ.

Концепция машин, которые могут обучаться сами для достижения целей поражала нас десятилетиями. Возможно, лучше всего об этом выразились дедушки современного ИИ, Стюардом Расселом и Питером Норвигом: «Как это возможно, чтобы медленный, маленький мозг, не важно, биологический или электронный, воспринимать, понимать, предсказывать и манипулировать миром, который намного больше и сложнее его самого?».

Эта цитата обращает нас к мысли о том, что концепция обучения была позаимствована у процессов и алгоритмов, открытых в природе. На рисунке 1 наглядно представлено взаимоотношение между искусственным интеллектом, машинным обучения и глубоким обучением.

Рис. 1. Взаимоотношение между ИИ, машинным обучением и глубоким обучением.

Основы глубокого обучения

Используя доступные мне материалы, в данной курсовой работе я разберу математическую составляющую нейронных сетей и на практическом примере рассмотрю применение их на аппроксимации просто функции.

Нейронная сеть прямого распространения

Линейные модели для регрессии и классификации базируются на линейных комбинациях фиксированных нелинейных базисных функций $\phi_j(x)$ и имеют форму

$$y(x, w) = f\left(\sum_{j=1}^{M} \omega_j \phi_j(x)\right) \tag{2.1}$$

где $f(\cdot)$ нелинейная функция активации в случае классификации и идентично в случае регрессии. Наша цель расширить данную модель создав базисную функцию $\phi_j(x)$, которая будет зависеть от параметров и потом позволим этим параметрам изменяться вместе с коэффициентами $\{\omega_j\}$. Конечно, существует много способов построения параметрической нелинейной базисной функции. Нейронные сети используют базисные функции формы, подобной (2.1), поэтому каждая базисная функция является нелинейной функцией для линейных входных данных, где коэффициенты в линейной комбинации являются адаптивными параметрами.

Это приводит нас к базовой нейронной сети, которая может быть описана как серия функциональных преобразований. В начале мы строим M линейных комбинаций входных переменных $x_1, \ldots, x_D$ в форме

$$a_j = \sum_{i=1}^{D} \omega_{ji}^{(1)} x_i + \omega_{j0}^{(1)} \tag{2.2}$$

где $j = 1, \ldots, M$ и верхний индекс (1) означает, что данные параметры находятся на «первом» уровне нейронной сети. Мы будем ссылаться на параметр $\omega_{ji}^{(1)}$ как на вес, а на параметр $\omega_{j0}^{(1)}$ как на погрешность. Переменные a_j известны как «активатор». Потом каждая из них

преобразуется с помощью дифференцируемой нелинейной «функции активации» h(·), что дает нам

$$z_j = h(a_j) \tag{2.3}$$

Переменные z_j соответствуют результатам в базисной функции (2.1), которые в контексте нейронных сетей, называются «скрытым узлом». Нелинейная функция $h(\cdot)$ обычно являются сигмоидальными функциями, такими как логическая сигмоидная или «tanh» функция. Учитывая (2.1) эти значения снова линейно комбинируются, чтобы получить выходной слой активации.

$$a_k = \sum_{j=1}^{M} \omega_{kj}^{(2)} z_j + \omega_{k0}^{(2)} \tag{2.4}$$

где $k = 1, \dots, K$ и K общее число результатов. Это преобразование соответствует второму уровню нейронной сети, и снова $\omega_{k0}^{(2)}$ является параметром погрешности. Наконец, выходной слой активации преобразуется, используя соответствующую функцию активации, чтобы дать набор выводов нейронной сети y_k. Выбор функции активации основывается на природе данных и предполагаемым распределением искомых данных и следует тем же соображениям, что и для линейных моделей. Таким образом, для стандартных регрессионных задач функция активации идентична, значит $y_k = a_k$. Аналогичным образом, для нескольких задач бинарной классификации, каждый выходной слой активации преобразуется, используя логическую сигмоидную функцию так, что

$$y_k = \sigma(a_k) \tag{2.5}$$

где

$$\sigma(a_k) = \frac{1}{1 + exp(-a)} \tag{2.6}$$

Мы можем объединить различные этапы для того, чтобы получить общую сетевую функцию, которая для сигмоидной функции слоя активации имеет вид

$$y_k(x, w) = \sigma\left(\sum_{j=1}^{M} \omega_{kj}^{(2)}\, h\left(\sum_{i=1}^{D} \omega_{ji}^{(1)}\, x_i + \omega_{j0}^{(1)}\right) + \omega_{k0}^{(2)}\right) \tag{2.7}$$

где набор всех параметров веса и погрешности сгруппирован в один вектор w. Таким образом нейронная сеть — это нелинейная функция из множества входных $\{x_i\}$ и выходных $\{y_k\}$ значений, регулируемых вектором w, вектором регулируемых параметров.

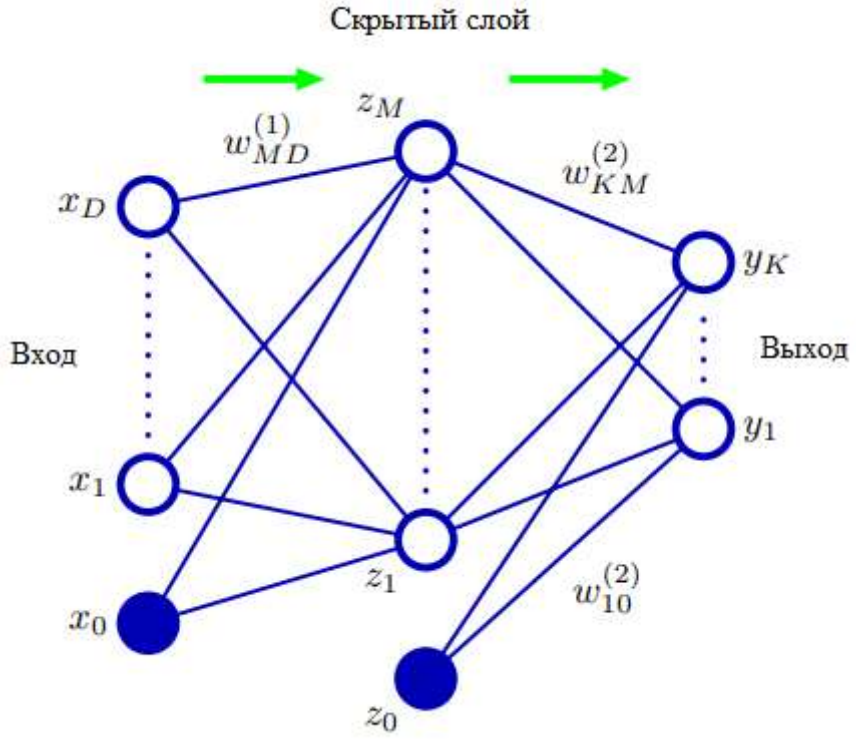

Рис. 2. Сетевая диаграмма для двухслойной нейронной сети (2.7)

Процесс расчета (2.7) может быть интерпретирована как прямое распространение данных через нейронную сеть.

Параметры погрешности в (2.2) могут быть включены во множество весовых параметров при помощи определения дополнительной входной переменной x_0, определим ее значение как $x_o = 1$, тогда (2.2) примет вид

$$a_j = \sum_{i=0}^{D} \omega_{ji}^{(1)} x_i \tag{2.8}$$

Мы можем аналогичным образом превратить веса второго слоя в погрешности второго слоя, так что общая сетевая функция примет вид

$$y_k(x, w) = \sigma\left(\sum_{j=0}^{M} \omega_{kj}^{(2)} h\left(\sum_{i=0}^{D} \omega_{ji}^{(1)} x_i\right)\right) \tag{2.9}$$

Как мы могли видеть на рисунке 2, модель нейронной сети содержит два этапа обработки данных, каждая из которых напоминает «персептрон» и по этой причине нейронная сеть известна как многослойный персептрон (MLP или multilayer perceptron). Ключевое различие с персептроном это то, что MLP использует непрерывные сигмоидальные нелинейности в скрытом слое, в то время как персептрон использует кусочно-постоянные нелинейности. Это значит, что MLP дифференцируется по отношению к параметрам нейронной сети и это свойство имеет решающую роль в обучении нейронной сети.

Если функции активации скрытого слоя в нейронной сети приняты как линейные, то для каждой нейронной сети мы всегда можем найти эквивалентную нейронную сеть без скрытого слоя. Это следует из того факта, что композиция последовательных линейных

преобразований и есть, само по себе, линейной преобразование. Однако, если число скрытых узлов меньше, чем количество узлов ввода и вывода, то преобразования этой нейронной сети не являются наиболее общими возможными линейными преобразованиями ввода и вывода, так как данные теряются в уменьшении размерности скрытых узлов.

Архитектура нейронной сети, показанная на рисунке 2, является наиболее часто используемой на практике. Однако ее легко обобщить, рассматривая дополнительные слои обработки, каждая из которых состоит из взвешенной линейной комбинации вида (2.4) с последующим поэлементным преобразованием при помощи нелинейной функции активации. Обратим внимание, что существует некоторая путаница в терминологии, по отношению к подсчету количества слоев в таких сетях. Таким образом сеть на рисунке 2 может быть описана как трехуровневая сеть (которая считает количество узлов в слоях и трактует входные данные как узлы). Мы рекомендуем терминологию, в которой нейронная сеть на рисунке 2 называется двухуровневой нейронной сетью, так как количество слоев адаптивных весов важнее для определения свойств сети.

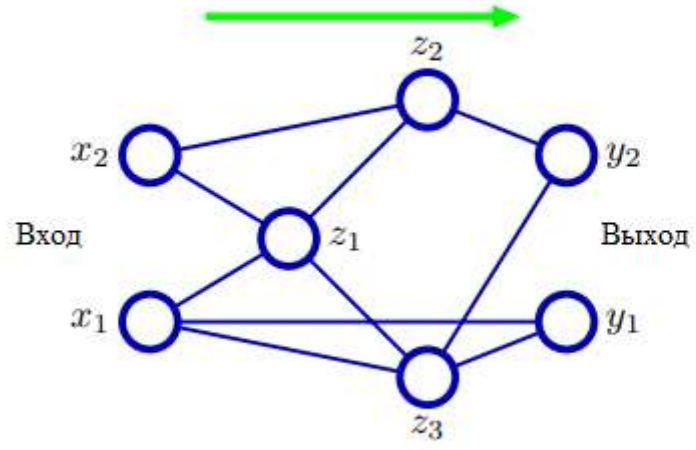

Рис. 3. Пример нейронной сети с прямым распространением

Другим обобщением сетевой архитектуры является включение соединения «пропускного слоя», каждое из которых связано с соответствующим адаптивным параметром. Для реализации, в двухслойной нейронной сети они будут идти напрямую от входным данным к выходным данным. В принципе, нейронная сеть с сигмоидными спрятанными узлами могут всегда может имитировать соединения пропускного слоя (для ограниченных входные значений) используя достаточно небольшой первый слой весов, что во всем рабочем диапазоне скрытые узлы являются линейными, а затем компенсируется большим значением весов скрытого слоя для выходных данных. На практике, однако, возможно эффективно включать пропускной слой явно.

Кроме того, сеть может быть разрозненной и не все возможные соединения присутствуют.

Поскольку существует прямое соответствие между диаграммой нейронной сети и ее математической функцией, мы можем разработать общие сетевые отображения, за счет рассмотрения более сложных диаграмм нейронных сетей. Однако они должны быть ограничены архитектурой с прямым распространением, другими словами, для тех, у кого нет замкнутых направленных цикло, нужно убедиться, что выходные данные являются детерминированными функциями входных данных. Это продемонстрировано на простом примере на рисунке 3. Каждый (скрытый или входной) узел в нейронной сети вычисляется как

$$z_k = h\left(\sum_j \omega_{kj}\, z_j\right) \tag{2.10}$$

где сумма проходит по всем узлам, которые отправляют соединения с узлом k (и параметр погрешности включен в суммирование). Для заданного набора значений, примененных к входным данным

нейронной сети, последовательное применение (2.10) позволяет активировать все узлы в нейронной сети, подлежащие расчету, включая выходные слои.

Нейронные сети называют универсальными аппроксиматорами. Для примера, двухслойная нейронная сеть с линейными выходными данными может равномерно аппроксимировать любую непрерывную функцию на компактной входной области с произвольной точностью при условии, что нейронная сеть имеет достаточное количество скрытых узлов. Этот результат справедлив для широкого для широкого спектра функций активации скрытых узлов, за исключением полиномиальных. Хотя такие теоремы обнадеживают, ключевая проблема заключается в том, как найти подходящие значения параметров из множества тренировочных данных.

Способность двухуровневой сети моделировать широкий спектр функций изображена на рисунке 4.

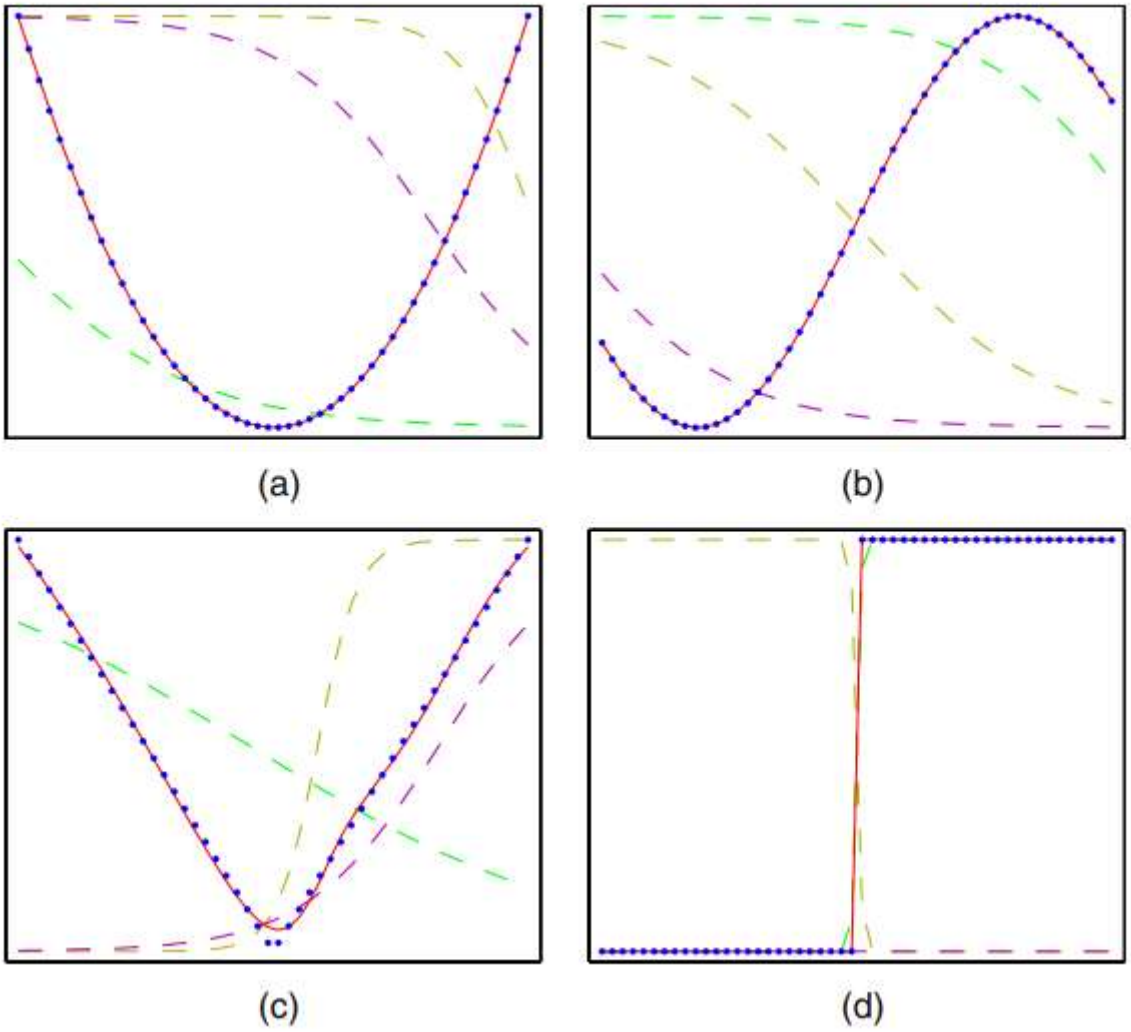

Рис. 4. Иллюстрация возможностей многослойного персептрона для аппроксимации четырех различных функций (a) $f(x) = x^2$ (b) $f(x) = sin(x)$ (c) $f(x) = |x|$ (d) $f(x) = H(x)$, где $H(x)$ – ступенчатая функция Хэвисайда. В каждом примере $N = 50$ точки данных показаны синими точками, были выбраны равномерно в интервале $(-1,1)$ и соответствующие вычисленные значения $f(x)$. Эти точки в дальнейшем используются для обучения двухуровневой сети, имеющей скрытые блоки с функциями активации «tanh» и линейными выходными блоками. Результирующие сетевые функции показаны красными кривыми, а выходы трех скрытых блоков показаны тремя пунктирными кривыми.

Обучение сети

До сих пор мы рассматривали нейронные сети как общий класс параметрических нелинейных функций из вектора x (входных параметров) в вектор y (выходных параметров). Простой подход к проблеме определения параметров нейронной сети состоит в том, чтобы минимизировать функцию ошибки суммы квадратов. Учитывая обучающий набор, состоящий из входных параметров вектора x_n, где $n = 1, \dots, N$ вместе с соответствующим набором целевых векторов t_n, мы минимизируем функцию ошибки.

$$E(w) = \frac{1}{2}\sum_{n=1}^{N} \left|\left|y(x_n, w) - t_n\right|\right|^2 \tag{3.1}$$

Однако мы можем предоставить гораздо более общий взгляд на обучение нейронной сети, в первую очередь дав вероятностную интерпретацию выходным данным сети. Нам это даст более четкую мотивацию, как для выбора нелинейности выходных узлов, так и для выбора функции ошибки. Мы начнем с обсуждения проблем регрессии, а пока рассмотрим единственную целевую переменную t, которая может принимать любое реальное значение. Мы предполагаем, что t имеет Гауссово распределение с зависимыми от x средним значением, которое задается выходными значениями нейронной сети, так что

$$p(t|x, w) = N(t|y(x, w), \beta^{-1} \tag{3.2}$$

где β точность (обратная дисперсии) Гауссовского шума. Для условного распределения (3.2) достаточно в качестве функции активации выходного устройства взять тождество, так как такая сеть может аппроксимировать любую непрерывную функцию от x до y.

Учитывая набор данных из N независимых, одинаково распределенных наблюдений $X = \{x_1, \dots, x_N\}$, вместе с соответствующими целевыми значениями $t = \{t_1, \dots, t_N\}$, мы можем построить соответствующую функцию правдоподобия

$$p(t|X, w, \beta) = \prod_{n=1}^{N} p(t_n, x_n, w, \beta) \tag{3.3}$$

Взяв отрицательный логарифм, мы получим

$$\frac{\beta}{2}\sum_{n=1}^{N}\{y(x_n, w) - t_n\}^2 - \frac{N}{2} ln\, \beta + \frac{N}{2} ln(2\pi) \tag{3.4}$$

с помощью которых мы можем узнать параметры w и β. Начнем с рассмотрения w. Максимизация функции правдоподобия эквивалентная минимизации функции ошибки суммы квадратов, определяемая как

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, w) - t_n\}^2 \tag{3.5}$$

где отброшены аддитивные и мультипликативные константы. Значение w находится минимизацией $E(w)$, которое будет обозначено w_{ML}, так как оно соответствует решению с максимальным правдоподобием. На практике нелинейность функции нейронной сети $y(x_n, w)$ приводит к «не выпуклости» ошибки $E(w)$, и поэтому на практике локальные максимумы правдоподобия могут быть найдены путем соответствия локальным минимумам функции ошибки.

Найдя w_{ML}, значение β может быть найдено за счет минимизации отрицательной логарифмической вероятности, в результате получим

$$\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n=1}^{N}\{y(x_n, w_{ML}) - t_n\}^2 \tag{3.6}$$

Обратим внимание, что это можно рассчитать после завершения итеративной оптимизации, необходимой для поиска w_{ML}. Если у нас есть несколько целевых переменных и мы предполагаем, что они независимые, обусловленные x и w с общей точностью шума β, то условное распределение целевых значений определяется выражением

$$p(t|x, w) = N(t|y(x, w), \beta^{-1}I) \tag{3.7}$$

Следуя тому же аргументу, что и для одной целевой переменной, мы видим, что максимальные весы правдоподобия определяются путем минимизации функции ошибок суммы квадратов (3.1). Тогда точность шума определяется выражением

$$\frac{1}{\beta_{ML}} = \frac{1}{NK}\sum_{n=1}^{N}\left||y(x_n, w_{ML}) - t_n|\right|^2 \tag{3.8}$$

где K – количество целевых переменных. От предположения о независимости можно отказаться при помощи немного более сложной задачи оптимизации.

Мы можем рассматривать нейронную сеть, как имеющую на выходе функцию активации, которая тождественна, так что $y_k = a_k$. Соответствующая функция ошибок суммы квадратов имеет свойство

$$\frac{\partial E}{\partial a_k} = y_k - t_k \tag{3.9}$$

которое мы будем использовать при обсуждении обратного распространения ошибок в разделе 4. Обратное распространение ошибки.

Теперь рассмотрим случай бинарной классификации, в которой у нас есть одна целевая переменная t, такая что $t = 1$ обозначает класс C_1, а $t = 0$ обозначает класс C_2. Мы рассмотрим нейронную сеть, имеющую один выход, который описан функцией активации в виду логической сигмоиды.

$$y = \sigma(a) \equiv \frac{1}{1 + exp(-a)} \tag{3.10}$$

тогда $0 \leq y(x, w) \leq 1$. Мы можем интерпретировать $y(x, w)$ как условную вероятность $p(C_1|x)$, где $p(C_2, x)$ задается уравнением $1 - y(x, w)$. Условное распределение целевых переменных при входных данных является распределением Бернулли в виде

$$p(t|x, w) = y(x, w)^t\{1 - y(x, w)\}^{1-t} \tag{3.11}$$

Если мы рассматриваем обучающий набор независимых наблюдений, тогда функция ошибки, которая дается отрицательным логарифмом функции правдоподобия, будет кросс-энтропийной функцией ошибок формы

$$E(w) = -\sum_{n=1}^{N}\{t_n \, ln \, y_n + (1 - t_n) \, ln(1 - y_n)\} \tag{3.12}$$

где y_n обозначает $y(x_n, w)$. Отметим, что нет аналога шумовой точности β, так как предполагается, что целевые значения правильно помечены. Однако модель легко расширяется, чтобы учесть ошибки маркировки. Simard et al. (2003) обнаружил, что использование кросс-энтропийной функции ошибок вместо суммы квадратов для задачи классификации приводит к более быстрому обучению, а так же к лучшему обобщению.

Если у нас есть K отдельных двоичных классификаций для выполнения, то мы можем использовать нейронную сеть, имеющую K выходов, каждая из которых представлена логической сигмоидной функцией активации. С каждой выходом связана метка двоичного класса $t_k \in \{0,1\}$, где $k = 1, \dots, K$. Если мы предположим, что метки классов независимы, учитывая входной вектор, то условное распределение целевых данных будет иметь вид

$$p(t|x,w) = \prod_{k=1}^{K} y_k(x,w)^{t_k} \left[1 - y_k(x,w)\right]^{1-t_k} \tag{3.13}$$

Отрицательный логарифм соответствующей функции правдоподобия дает следующую функцию ошибок

$$E(w) = -\sum_{n=1}^{N}\sum_{k=1}^{K}\{t_{nk}\, ln\, y_{nk} + (1 - t_{nk})\, ln(1 - y_{nk})\} \tag{3.14}$$

где y_{nk} обозначает $y_k(x_n, w)$. Снова, производная функций ошибок со спектром активации для конкретных выходных узлов принимает вид (3.9), как и в случае регрессии.

Предположим, что мы используем стандартную двухуровневую нейросеть показанную на рис. 2. Мы видим, что весовые параметры в первом слое сети распределяются между различными выходами, тогда как в линейной модели каждая классификация решается независимо. Первый уровень сети может рассматриваться как выполнение нелинейного извлечения признаков, и совместное использование извлечения между различными выходными данными может сэкономить на вычислениях, а также может привести к улучшению обобщения.

Наконец, мы рассмотрим стандартную задачу много классовой классификации, в которой каждый вход относится к одному из K взаимоисключающих классов. Бинарные целевые переменные $t_k \in \{0,1\}$ имеет схему кодирования 1 из K с указанием класса, а выходы нейронной сети интерпретируются как $y_k(x,w) = p(t_k = 1|x)$, в итоге имею следующую функцию ошибки

$$E(w) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn}\, ln\, y_k\,(x_n, w) \tag{3.15}$$

Мы видим, что функция активации узлов выхода, соответствующая канонической ссылке, задается функцией «SoftMax».

$$y_k(x,w) = \frac{exp\big(a_k(x,w)\big)}{\sum_j exp\left(a_j(x,w)\right)} \tag{3.16}$$

что удовлетворяет $0 \le y_k \le 1$ и $\sum_k y_k = 1$. Отметим, что $y_k(x,w)$ не изменяются, если ко всем $a_k(x,w)$ добавляется константа, в результате чего функция ошибок остается постоянной для некоторых направлений в пространстве весов. Это вырождение снимается, если к функции ошибок добавить соответствующий член регуляризации.

Еще раз, производная функции ошибок относительно активации для конкретного выходного узла принимает знакомую форму (3.9).

Таким образом, существует естественный выбор как функции активации узла вывода, так и функции ошибки согласования, в зависимости от типа решаемой проблемы. Для регрессии мы используем линейные выходы и ошибку суммы квадратов, для (нескольких независимых) двоичных классификаций мы используем логистические сигмоидные выходы (функции активации для этих выходов) и функцию кросс-энтропийной ошибки, а для много классовой классификации мы для выходов используем «SoftMax», с соответствующей мультиклассовой кросс-энтропийной функцией

ошибки. Для задач классификации, включающих два класса, мы можем использовать один логистический сигмоидный выход, или, как альтернатива, мы можем использовать нейронную сеть с двумя выходами, каждая из которых имеет функцию активации «SoftMax».

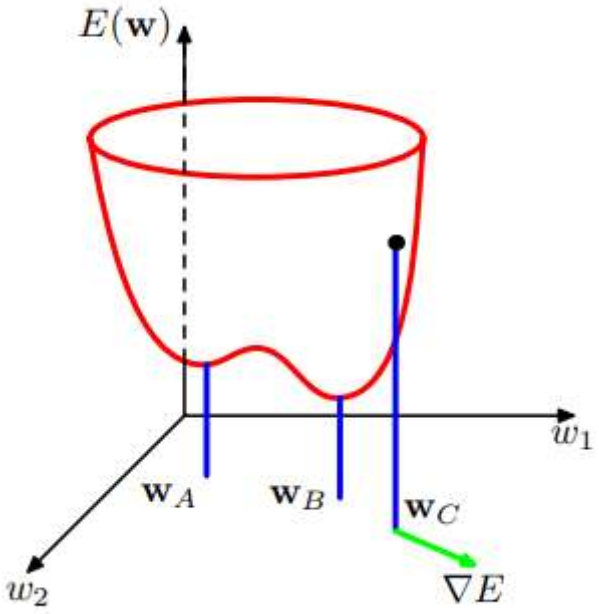

Рис. 5. Геометрический вид функции ошибки $E(w)$ как поверхности, расположенной над пространством весов. Точка w_A – это локальный минимум, а w_B – глобальный минимум. В любой точке w_C локальный градиент ошибки задается вектором ∇E.

Обратное распространение ошибки

Наша цель в этом разделе – найти эффективный метод расчета градиента функции ошибок $E(w)$ для нейронной сети с прямой связью. Мы покажем, что это может быть достигнуто с помощью схемы передачи локальных сообщений, в которой информация пересылается поочередно вперед и назад по сети и известна как «Распространение ошибок» или проще «Обратное распространение».

Следует отметить, что термин обратное распространение используется в литературе для обозначения множества разных вещей. Например, многослойную архитектуру персептрона иногда называют сетью обратного распространения. Также термин обратного распространения используется для описания обучения многослойного персептрона с использованием градиентного спуска, применяемого к функции ошибок суммы квадратов. Для уточнения терминологии полезно более внимательно рассмотреть характер тренировочного процесса. Большинство обучающих алгоритмов включают итеративную процедуру минимизации функции ошибок, с корректировкой весов. Каждый шаг мы можем разбить на два этапа. На первом этапе, необходимо вычислить производные функции ошибок по весам. Как мы увидим позже, важный вклад обратного распространения заключается в предоставлении эффективного метода для вычисления таких производных. Так как именно на этом этапе ошибки распространяются по нейронной сети в обратном направлении, мы будем использовать термин обратное распространение специально для расчета производных. На втором этапе производные затем используются для вычисления корректировок, которые необходимо внести в веса. Самый просто из таких приемов – градиентный спуск. Важно понимать, что эти две стадии различны. Таким образом, первый этап, а именно распространение ошибок назад по сети для расчета производных, может быть применен ко многим другим типам сетей, а

не только к многослойному персептрону. Его также можно применить к функциям ошибок, отличающихся от суммы квадратов, и к вычислению других производных, таких как матрицы Якобиана и Гессе. Аналогично, второй этап регулировки веса с помощью вычисленных производных могут быть решены с использованием различных схем оптимизации, многие из которых значительно более мощные, чем простой градиентный спуск.

Расчет производных функции ошибок

Теперь мы выведем алгоритм обратного распространения ошибок для общей сети, имеющей произвольную топологию прямой связи, произвольные дифференцируемые функции активации и широкий класс функции ошибок. Затем полученные формулы будут проиллюстрированы, используя многослойную нейронную сеть, имеющей один слой сигмоидальных скрытых узлов вместе с функцией ошибки суммы квадратов.

Многие функции ошибок, представляющий практический интерес, например те, которые определяются максимумом функции правдоподобия для набора независимых и одинаково распределенных данных, содержат сумму членов, по одному для каждой точки данных в обучающем наборе, поэтому

$$E(w) = \sum_{n=1}^{N} E_n(w) \tag{4.1}$$

Здесь мы рассмотрим задачу вычисления градиента ∇E для одного такого члена в функциях ошибок. Это может быть использовано непосредственно для последовательной оптимизации, или результаты могут накапливаться по обучающей выборке в случае пакетных методов.

Рассмотрим сначала простую линейную модель, в которой выводы y_k являются линейными комбинациями x_i, так что

$$y_k = \sum_i \omega_{ki}\, x_i \tag{4.2}$$

вместе с функцией ошибок, которая для конкретного шаблона n принимает форму

$$E_n = \frac{1}{2}\sum_k (y_{nk} - t_{tk})^2 \tag{4.3}$$

где $y_{nk} = y_k(x_n, w)$. Градиент этой функции ошибок относительно веса ω_{ji} определяется выражением

$$\frac{\partial E_n}{\partial \omega_{ji}} = \big(y_{nj} - t_{nj}\big)x_{ni} \tag{4.4}$$

который может быть интерпретирован как «локальное» вычисление включающие результат «сигнала ошибки» $y_{nt} - t_{nj}$ связанного с выходным концом линии ω_{ji} и переменной x_{ni} связанного с входным концом линии.

В общей сети с прямой связью каждый узел вычисляет взвешенную сумму своих входных данных в форме

$$a_j = \sum_i \omega_{ji}\, z_i \tag{4.4}$$

где z_i – это активация узла или входа, который отправляет соединение узлом j и ω_{ji} вес, который связан этим подключением. В разделе 2 мы видели, что смещения можно включить в эту сумму, введя дополнительную единицу или вход, с фиксированной функцией активацией +1. Поэтому нам не нужно явно разбираться с предубеждениями явно. Сумма в (4.4) преобразуется нелинейной функцией активации $h(\cdot)$, чтобы дать активацию z_j узла j в виде

$$\mathrm{z_j} = \mathrm{h}\big(\mathrm{a_j}\big) \tag{4.5}$$

Обратим внимание, что одна или несколько переменных z_i в сумме (4.4) могут быть входными, и аналогично, узлы j могут быть выходными. Для каждого шаблона в обучающем наборе мы будем

предполагать, что мы предоставили соответствующий входной вектор в сеть и вычислили функции активации всех скрытых и входных узлов в сети путем последовательного применения (4.4) и (4.5). Этот процесс часто называют прямым распространением, так как его можно рассматривать как прямой поток данных через сеть.

Теперь рассмотрим расчет производной от E_n по отношению к весу ω_{ji}. Выходы различных узлов будут зависеть от конкретного входного шаблона n. Однако, чтобы не загромождать обозначения, мы будем опускать индекс n в сетевых переменных. Прежде всего отметим, что E_n зависит только от веса ω_{ji} только через суммированный вход a_j в узел j. Следовательно, мы можем применять цепное правило для частных производных, чтобы получить

$$\frac{\partial E_n}{\partial \omega_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial \omega_{ji}} \tag{4.6}$$

теперь введем полезное обозначение

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{4.7}$$

где δ ошибка по причинам, которую мы скоро увидим. Использую (4.4) мы можем написать

$$\frac{\partial a_j}{\partial \omega_{ji}} = z_i \tag{4.8}$$

подставляя (4.7) и (4.8) в (4.6), мы получим

$$\frac{\partial E_n}{\partial \omega_{ji}} = \delta_j z_i \tag{4.9}$$

Уравнение (4.9) говорит нам, что требуемая производная получается простым умножением значения δ для веса конечного узла выхода и z для веса конечного узла входа (где $z = 1$ в случае смещения). Обратим внимание, что уравнение примет ту же форму, что и для простой линейной модели, рассмотренной во втором разделе. Таким образом, чтобы рассчитать производные, нам нужно только вычислить значение

δ_j для каждого скрытого и выходного узла в сети, а затем применить в (4.9)

Как мы могли видеть для выходных узлов у нас есть

$$\delta_k = y_k - t_k \tag{4.10}$$

при условии, что мы используем каноническую ссылку в качестве функции активации в выходном узле. Чтобы рассчитать δ для скрытых узлов, мы снова используем цепное правило для частных производных.

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{4.11}$$

где сумма пробегает все узлы k, к которым узел j отправляет соединения. Расположение узлов и весов проиллюстрировано на рисунке 6. Обратим внимание, что узлы, помеченные k, могут включать другие скрытые узлы и/или узлы входа. В записи (4.11) мы используем тот факт, что вариации a_j вызываю вариации функции ошибок только через вариации переменных a_k. Если мы теперь подставим определение δ из (4.7) в (4.11) и воспользуемся формулами (4.4) и (4.5), мы получим следующую формулу обратного распространения ошибок.

$$\delta_j = h'(a_j) \sum_k \omega_{kj}\, \delta_k \tag{4.12}$$

которая говорит нам, что значение δ для конкретного скрытого узла может быть получено путем распространения δ в обратном направлении от узлов выше в сети, как и проиллюстрировано на рисунке 6. Отметим, что суммирование в (4.12) ведется по первому индексу ω_{kj} (соответствуя обратному распространению данных по сети), тогда как в уравнении прямого распространения (2.10) оно проходит по второму индексу. Поскольку мы уже знаем значения δ

для выходных узлов, отсюда следует, что, рекурсивно применяя (4.12), мы можем рассчитать δ для всех скрытых узлов в сети прямого распространения, независимо от ее топологии.

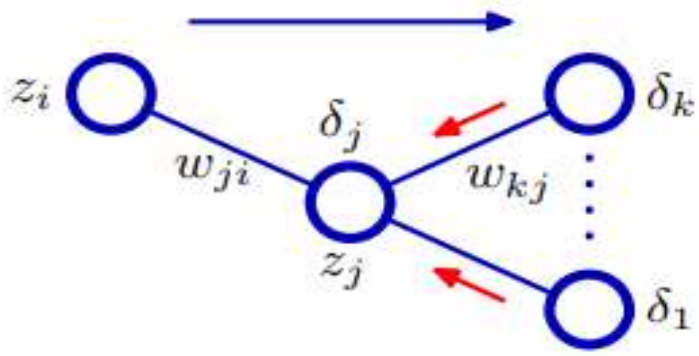

Рис. 6. Иллюстрация расчета δ_j для скрытых узлов j с помощью обратного распространения δ от единиц k, к которым блок j отправляет соединения. Синяя стрела обозначает направление данных при прямом распространении, а красные стрелки указывают на обратное распространение данных ошибки.

Таким образом процедуру обратного распространения можно резюмировать следующим образом

1. Примените входной вектор x_n к сети и распространите его вперед по сети, используя (4.4) и (4.5), чтобы найти активации всех скрытых и входных узлов.
2. Рассчитайте δ_k для всех выходных узлов, используя (4.10)
3. Выполните обратное распространение δ_k, используя (4.12), чтобы получить δ_j для каждого скрытого узла в сети.
4. Используйте (4.9) для расчета требуемых производных.

Для пакетных методов производная полной ошибки E может быть затем получена путем повторения вышеуказанных шагов для каждого шаблона в обучающем наборе и последующего суммирования через все шаблоны

$$\frac{\partial E}{\partial \omega_{ji}} = \sum_{n} \frac{\partial E_n}{\partial \omega_{ji}} \tag{4.13}$$

В приведенном выше выводе мы неявно предполагали, что каждый скрытый и входной узел в сети имеет одну и ту же функцию активации $h(\cdot)$. Однако вывод легко обобщается, что позволит различным узлам иметь индивидуальные функции активации, просто отслеживая, какая форма $h(\cdot)$ соответствует какому узлу.

Программный модуль. Практика

В примере, на практике, мы реализуем программу для аппроксимации функции $y = x^2$ при помощи нейронной сети, используя Python 3.9.2 и прилагающиеся библиотеки.

Список библиотек:

- Scikit-learn 0.24.2 – Библиотека для обработки данных.
- TensorFlow 2.5.0 – Библиотека с нейронными сетями.
- Matplotlib 3.4.2 – Библиотека для отображения графиков.
- NumPy 1.19.5 – Математическая библиотека.

Мы рассматриваем простую функцию, поэтому мы ожидаем, что небольшая нейронная сеть сможет быстро ее изучить.

Мы определим сеть с помощью библиотеки TensorFlow и воспользуемся некоторыми инструментами подготовки данных из библиотеки Scikit-learn.

В первую очередь определим набор данных.

```
x = asarray([i for i in range(-50, 51)])
y = asarray([i**2.0 for i in x])
```

Затем мы можем изменить форму данных так, чтобы входные и выходные переменные были столбцами с одним наблюдением в строке, как ожидается при использовании моделей контролируемого обучения.

```
x = x.reshape((len(x), 1))
y = y.reshape((len(y), 1))
```

Далее нам нужно будет масштабировать наши входы и выходы.

Входы будут иметь диапазон от $(-50,50)$, тогда как выходы будут иметь диапазон $-50^2, 0^2$. Большие входные и выходные данные могут сделать обучающие нейронные сети нестабильными, поэтому рекомендуется сначала масштабировать данные.

Мы можем использовать MinMaxScaler для отдельной нормализации входных и выходных значений до значений в диапазоне $(-1,1)$.

```
scale_x = MinMaxScaler()
x = scale_x.fit_transform(x)
scale_y = MinMaxScaler()
y = scale_y.fit_transform(y)
```

Теперь мы можем определить модель нейронной сети.

Путем проб и ошибок я выбрал модель с двумя скрытыми слоями и 10 узлами в каждом слое.

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(10, input_dim=1,
activation='relu', kernel_initializer='he_uniform'))
model.add(keras.layers.Dense(10, activation='relu',
kernel_initializer='he_uniform'))
model.add(keras.layers.Dense(1))
```

Мы подгоним модель, используя среднеквадратичную потерю, и воспользуемся эффективной адамовской версией стохастического градиентного спуска для оптимизации модели.

Это означает, что модель будет минимизировать среднеквадратичную ошибку между сделанными прогнозами и ожидаемыми выходными значениями y, пока она пытается аппроксимировать функцию отображения.

```
model.compile(loss='mse', optimizer='adam')
```

У нас не так много данных (около 100 строк), поэтому мы подберем модель для 500 эпох и используем небольшой размер пакеты, равный 10.

Опять же, эти значения были найдены после небольших проб и ошибок.

```
model.fit(x, y, epochs=500, batch_size=10, verbose=0)
```

После подбора мы можем рассчитать модель.

Мы сделаем прогноз для каждого примера в наборе данных и вычислим ошибку. Идеальным приближением будет 0.0. В целом это невозможно из-за шума в наблюдениях, неполных данных и сложности неизвестной базовой функции отображения.

В этом случае это возможно, потому что у нас есть все наблюдения, в данных нет шума, и лежащая в основе функция не является сложной.

Мы можем сделать прогноз.

```
yhat = model.predict(x)
```

Затем мы должны инвертировать выполненное масштабирование.

Таким образом, ошибка отображается в исходных единицах целевой переменной.

```
x_plot = scale_x.inverse_transform(x)
y_plot = scale_y.inverse_transform(y)
yhat_plot = scale_y.inverse_transform(yhat)
```

Затем мы можем вычислить и сообщить об ошибке прогноза в исходных единицах целевой переменной.

```
print('MSE: %.3f' % mean_squared_error(y_plot, yhat_plot))
```

Наконец мы можем создать диаграмму рассеяния реального сопоставления входов с выходами и сравнить его с сопоставлением входов с предсказанными выходами и посмотреть, как аппроксимация функции сопоставления выглядит в пространстве.

```
pyplot.scatter(x_plot, yhat_plot, label='Predicted')
pyplot.title('Input (x) versus Output (y)')
pyplot.xlabel('Input Variable (x)')
pyplot.ylabel('Output Variable (y)')
pyplot.legend()
pyplot.show()
```

Полный пример кода

```
x = asarray([i for i in range(-50, 51)])
y = asarray([i**2.0 for i in x])
x = x.reshape((len(x), 1))
y = y.reshape((len(y), 1))
scale_x = MinMaxScaler()
x = scale_x.fit_transform(x)
scale_y = MinMaxScaler()
y = scale_y.fit_transform(y)
model = keras.models.Sequential()
model.add(keras.layers.Dense(10, input_dim=1,
activation='relu', kernel_initializer='he_uniform'))
model.add(keras.layers.Dense(10, activation='relu',
kernel_initializer='he_uniform'))
model.add(keras.layers.Dense(1))
model.compile(loss='mse', optimizer='adam')
model.fit(x, y, epochs=500, batch_size=10, verbose=0)
yhat = model.predict(x)
x_plot = scale_x.inverse_transform(x)
y_plot = scale_y.inverse_transform(y)
yhat_plot = scale_y.inverse_transform(yhat)
print('MSE: %.3f' % mean_squared_error(y_plot, yhat_plot))
pyplot.scatter(x_plot,y_plot, label='Actual')
pyplot.scatter(x_plot, yhat_plot, label='Predicted')
pyplot.title('Input (x) versus Output (y)')
pyplot.xlabel('Input Variable (x)')
pyplot.ylabel('Output Variable (y)')
pyplot.legend()
pyplot.show()
```

Результат выполнения на рисунке 7.

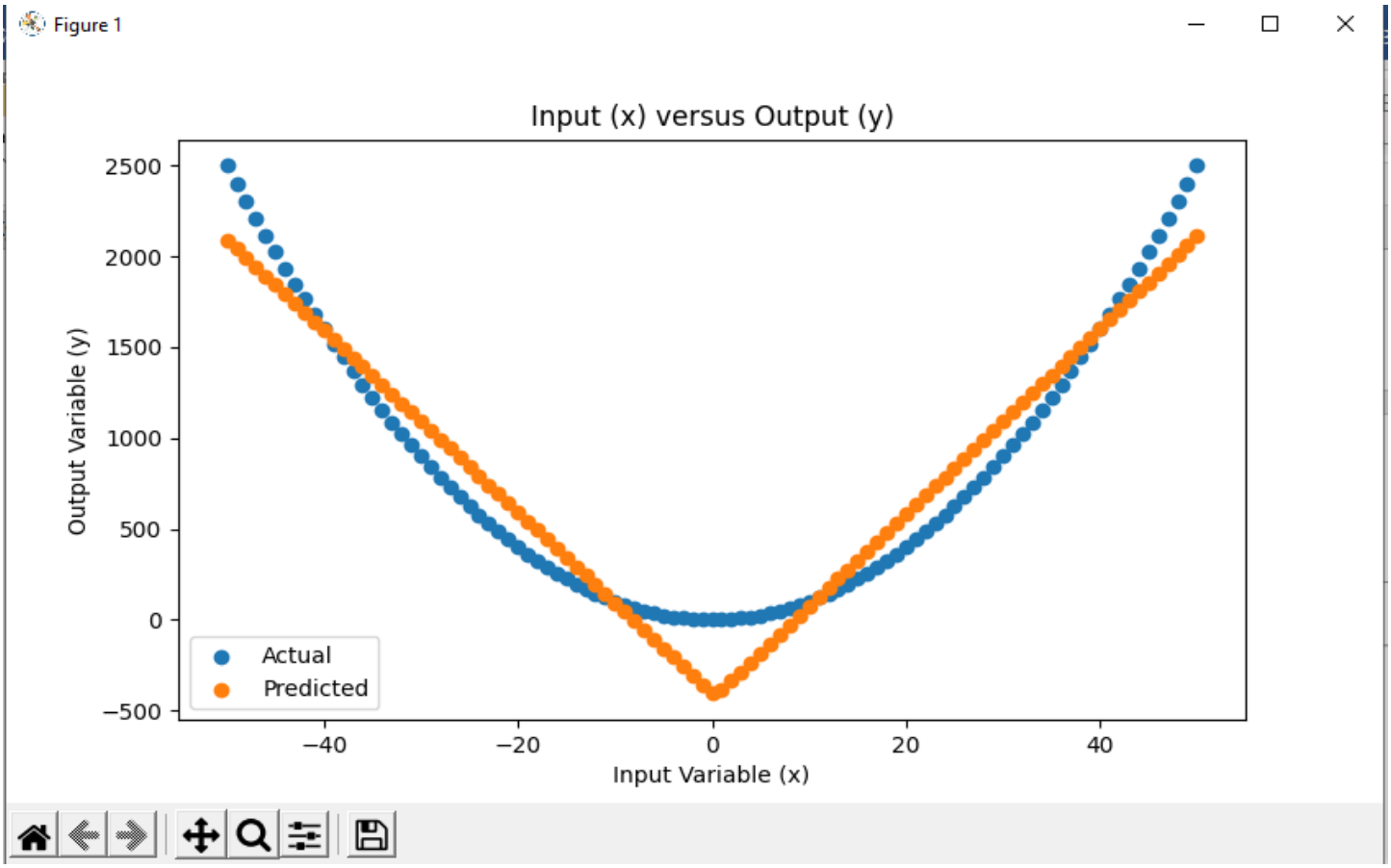

Рис.7. Результат выполнение программного модуля, где синие точки, сама функция, оранжевые точки, аппроксимация нейронной сети.

Разница между этими двумя рядами заключается в ошибки аппроксимации функции отображения. Мы видим, что приближении разумно, оно фиксирует общую форму.

Это говорит о том, что есть много способов для улучшения, например использование другой функции активации или другой сетевой архитектуры.

Литература

1. Бишоп К. М. Pattern Recognition and Machine Learning – Springer Science+Business Media, LLC, 2006. – 738с. – ISBN: 978-0387-31073-2
2. Джош Паттерсон. Deep Learning - O'Reilly Media, Inc., 2007. – 538с. – ISBN: 978-1491914250