Нейронные сети

И.И. Приезжев

2019

www.ivanplab.ru

Содержание

Введение

Нейронные сети — это эффективный инструмент для решения множество задач интеллектуального анализа данных. Список направлений решаемых задач: распознавание образов, прогноз переменных во времени или пространстве, классификация объектов.
Распознавание образов является основой в реализации компьютерного зрения для распознавания человеческих лиц и других визуальных объектов по фото и видео материалам, распознаванию рукописных и печатных знаков, распознавании человеческой речи, при переводе с одного языка на другой, играх и множества аналогичных задач. Эти задачи широко применяются в робототехнике и при создании интеллектуальных интерфейсов компьютера и человека.
Прогноз переменных во времени и пространстве является очень важной задачей при интеллектуальном анализе данных и широко применяется в различных отраслях знаний и промышленности, например, для контроля во времени сложных производственных технологических процессов. Такой прогноз применяется для анализа и прогноза финансовых показателей и разработки торговых роботов. Пространственный прогноз применяется в различных отраслях знаний, например, в геологических отраслях для прогноза положения потенциальных рудных точек или прогноза геологического потенциала нефтегазовой продуктивности геологических пластов.
Задачу классификации многомерных объектов можно рассматривать как часть общей задачи распознавания образов, но эта задача имеет очень важное самостоятельное значение для решения большого количества практических задач в различных отраслях знаний.
В настоящее время возникла новая волна интереса к нейронным сетям (первая волна имела место в 60 годах в связи с изобретением нейронных сетей, вторая волна - в 90-00 годах в связи с изобретением метода обратного распространения ошибки для обучения. Новая волна возникла в связи с появлением новых стабильных способов обучения многослойных глубоких нейронных сетей с высокой степенью свободы на основе приемов генерализации данных первых слоев. Также широко используются алгоритмы обучения на основе генетических алгоритмов и их стабилизации на основе методов регуляризации Тихонова. Кроме этого, наблюдается рост коммерческих решений на основе глубоких нейронных сетей.

Основные элементы нейронных сетей

Нейронные сети, или более точное название «искусственные нейронные сети», являются попыткой разработки математического аналога биологических нейронных сетей как сетей нервных клеток с основным элементом в виде клеточного нейрона (см рис. 1).

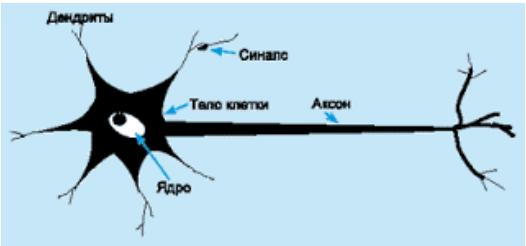

Рис. 1. Схема основных элементов клеточного нейрона.

Основным элементом искусственных нейронных сетей является математический нейрон как многовходовый нелинейный сумматор с одним выходом. Схематически можно изобразить такой нейрон, как показано на рисунке 2.

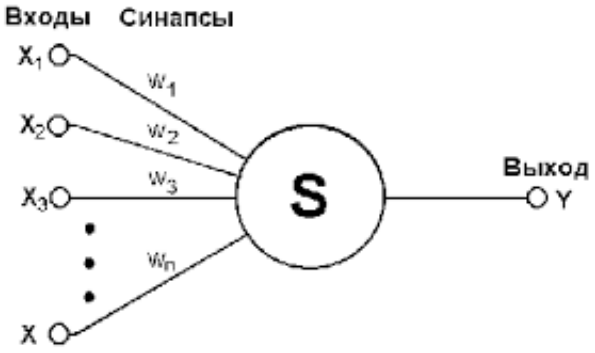

Рис. 2. Схема математического нейрона как нелинейный сумматор S нескольких входных данных и одного выхода.

Математическую формулу нейрона можно выразить как нелинейный сумматор S (интегральный оператор) с помощью уравнения 1.

$$y = f(\sum_{i=0}^{n} x_i\, w_i) \qquad (1)$$

где n – количество входов нейрона, x_i – значения входов нейрона, w_i – весовые коэффициенты, f()– нелинейная функция активации, y - выходное значение нейрона.

Активационная функция предназначена для включения (усиления влияния) или выключения (подавления влияния) отдельных входов нейрона обычно имеют вид сглаженной ступенчатой функции, как показано на рисунке 3. Если применить ступенчатую функцию (релейную функцию), то некоторые входы в нейрон будут полностью выключены или включены. Если применить линейную функцию, то нейрон превращается в линейный оператор типа множественная регрессия. В последнем случае есть возможность получить весовые коэффициенты в процессе обучения путем решения системы линейных уравнений на основе массива с известными входами и соответствующими им выходами.

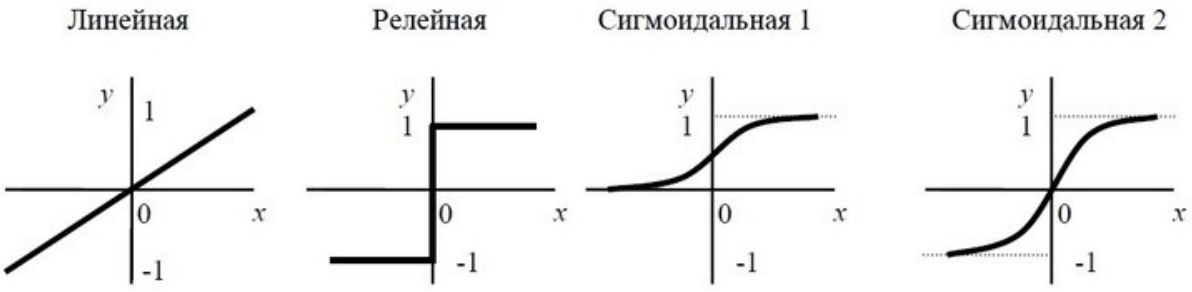

Рис. 3. Графики различных активационных функций.

Для обученного нейрона весовые коэффициенты и вид активационной функции являются константами данного нейрона и не меняются в процессе вычислений, но они должны быть определены в процессе обучения.

В качестве входа нейрона понимается некий набор чисел или вектор, описывающий изучаемый объект. Если подавать на вход нейрона последовательно несколько векторов, то каждому из них будет соответствовать уникальный выход. Если эти вектора описывают изменение изучаемого объекта во времени (например, ежедневное изменение курса разных валют в рублях), в этом случае их можно назвать сигналом поданного на вход нейрона. Так как мы всегда подразумеваем подачу разных векторов на вход нейрона, то удобно называть такой поток сигналом.

В общем виде нейронную сеть образуют хаотически соединенные нейроны, когда выходы одних нейронов являются входами других нейронов с возможными обратными связями, когда сигнал полученный на выходе некоторого нейрона может быть подан на него как один из возможных входных сигналов, в том числе, когда этот выходной сигнал пройдет через несколько других нейронов. Для таких нейронных сетей общие алгоритмы обучения не разработаны и в настоящее время имеется только несколько частных случаев, для которых имеются теоретические исследования и реализации.

Нейронные сети без обратных связей можно представить в виде многослойной конструкции, как показано на рисунке 4. Такие нейронные сети имеют особое название - персептроны.

Все нейроны организованы в скрытых слоях между входом и выходными нейронами. Входы иногда называют входным слоем или просто входами. Выходных нейронов может быть только один или больше в соответствии с количеством выходов нейронной сети.

Входной вектор подается на входы всех нейронов в первом скрытом слое и все выходы этих нейронов подаются на входы следующего слоя и так далее до выходного слоя.

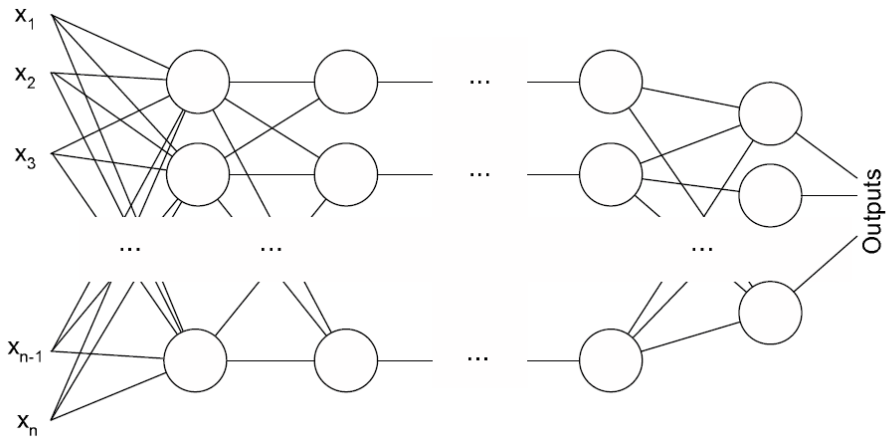

Рис. 4. Многослойный персептрон.

Если использовать линейную активационную функцию, имеется только один выходной нейрон и нет ни одного скрытого слоя, то в этом случае мы имеем простой оператор линейной множественной регрессии.

Теоретические основы нейронных сетей

Математически работу нейронной сети можно определить как нелинейное отображение F[.] многомерных входных векторов **X** в одномерную функцию y (см уравнение 2).

$$y = F[\boldsymbol{X}] \qquad (2)$$

Без ограничения общности, на выходе нейронной сети может быть многомерный выходной вектор **Y**. В последнем случае такую сеть можно рассматривать как композицию нескольких нейронных сетей с одномерной выходной функцией.

Задача представления функции многих переменных в виде суперпозиции функций меньшего числа переменных была определена Давидом Гильбертом как одна из 32 наиболее важных математических задач в 1900 году. Тринадцатую проблему Гильберт формулировал следующим образом, что невозможно представить трехмерную функцию в виде суперпозиции двухмерных функций.

Данная задача была решена в процессе научной дискуссии А. Н. Колмогорова и В. И. Арнольда, которая показана в цепочке их публикаций (Колмогоров, 1956; Колмогоров, 1957), (Арнольд, 1957). Задачу для трехмерной задачи решил В. И. Арнольд в 1957 и затем более общая теорема существования отображения многомерной функции была доказана А. Н. Колмогоровым в фундаментальной работе 1957 г. «Теорема о представлении непрерывных функций нескольких переменных в виде суперпозиций непрерывных функций одного переменного и сложения».

В теореме доказывалось, что при любом целом n >= 2, многомерная непрерывная действительная функция $y = f(x_1, x_2, x_3, \dots, x_n)$, определенная на n-мерном единичном кубе $\mathbf{E^n}$ может быть **точно** выражена в виде двух вложенных сумм.

$$f(x_1, x_2, x_3, \dots, x_n) = \sum_{q=1}^{2n+1} \chi_q \left[\sum_{p=1}^{n} \varphi_{pq}(x_p)\right] \qquad (3)$$

где $\varphi_{pq}(x_p)$ –действительные функции, определенные на отрезке [0,1], непрерывно возрастающего типа, что определенно как замечание в теореме Колмогорова (1957).
$\chi_q()$ – непрерывные действительные функции, также определенные на отрезке [0,1].

Выражение (3) можно представить в виде двухслойной нейронной сети, где внутреннее суммирование есть входной слой и внешнее суммирование есть скрытый слой с количеством нейронов равным 2n+1. Функции $\varphi_{pq}(x_p)$ и $\chi_q()$ можно считать активационными функциями неизвестного вида. $\sum_{q=1}^{2n+1} \chi_q(\xi)$

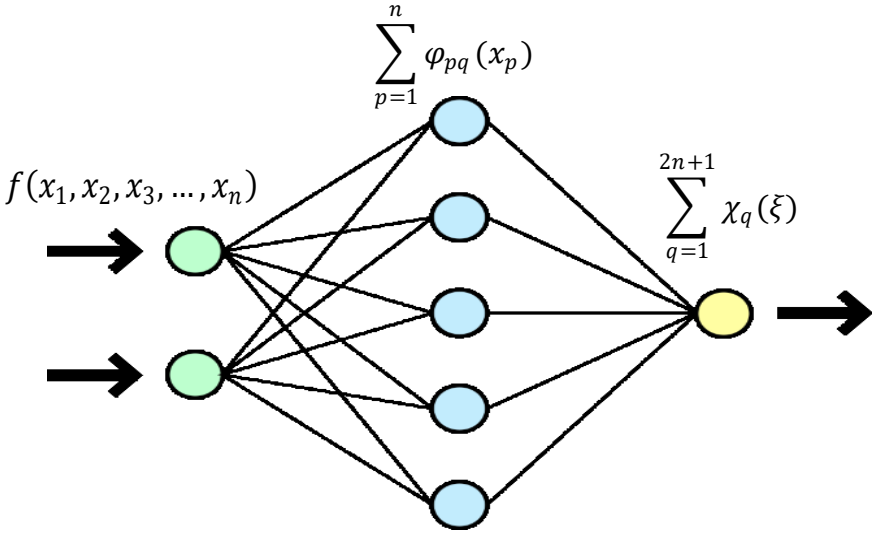

Рис. 5. Двухслойная нейронная сеть Колмогорова с двумя входами согласно уравнению 3.

Из теоремы Колмогорова известно только, что функции $\varphi_{pq}(x_p)$ монотонно возрастающие и не зависят от вида входной функции, а определяются только его размерностью.

В более поздних исследованиях (Lorentz, 1966 ; Sprecher, 1965), показано что вид функции $\varphi_{pq}(x_p)$ может быть одного вида для каждого нейрона скрытого слоя, с неким коэффициентом $\lambda_q \varphi_q(x_p)$.

Существует множество попыток прямого вычисления этих функций или в использовании однородного набора активационных функция типа сигмоида для двухслойных сетей типа (3) (Hecht-Nielsen, 1987).
В 1989 году была доказана теорема о приблизительной аппроксимации функций нескольких переменных с помощью двухслойной нейронной сети с активационной функцией в виде сигмоида σ (*Hornick et al, 1989; Cybenko, 1989; Funahashi* , 1989) - *Существуют такое число H, набор чисел ω_{ip}, u_i и набор чисел v_i, что функция*

$$f'(x_1, x_2, x_3, \dots, x_n) = \sum_{i=1}^{H} v_i \sigma[\sum_{p=1}^{n} \omega_{ip} x_p + u_i]$$

приближает данную функцию $f(x_1, x_2, x_3, \dots, x_n)$ с погрешностью не более ε на всей области определения.

Обучение и нестабильность нейронных сетей

Нейронная сеть может использоваться как нелинейный прогнозный оператор, который широко используется в промышленности для прогнозных построений (Roth and Tarantola, 1994; Schultz et al., 1994; Russell et al., 1997; Hampson et al., 2001; Veeken et al., 2009; Priezzhev et al., 2014).

При использовании нейронных сетей всегда имеется два этапа. Первый этап — это обучение нейронных сетей, когда определяются все ее коэффициенты при заданной ее конфигурации (геометрии связей) - количество входов, количество скрытых слоев, количество нейронов в каждом скрытом слое и количество выходов. Также должен быть определен тип активационной функции, обычно один для всех нейронов как однородный тип сетей. Для обучения используется обучающее множество в виде массива обучающих пар, входной вектор и соответствующим им выходной вектор. В процессе обучения необходимо минимизировать объектную функцию, обычно в виде суммы квадратов ошибок – как квадрат разницы вычисленного выхода с текущими весовыми коэффициентами и заданными выходами в обучающих парах.

Оператор прогноза в виде нейронной сети имеет несколько существенных преимуществ и недостатков, описанных в таблице 1 (Schultz et al., 1994; Bishop, 1995; Girosi et al., 1995).

Недостатки нейронных сетей	**Достоинства нейронных сетей**
Эффект переобучения (overlearning), который проявляется в виде хорошей аппроксимации данных обучения и нестабильной работы оператора в процессе прогнозирования. Природа этого явления объясняется наличием большой степени свободы оператора прогноза при	Оператор прогноза с большой степенью свободы (нелинейности) может быть использован в сложных случаях, когда детерминистическое соотношение между зависимыми (прогнозными) переменными и независимыми переменными неизвестно.

отсутствии существенных связей между зависимыми (прогнозными) переменными и независимыми переменными.	
Эффект экстраполяции проявляется при использовании ограниченного диапазона данных при обучении. Если пытаться выполнять прогноз за этими пределами, то он может быть не верен.	Позволяет управлять нелинейностью (степенью свободы) с помощью количества скрытых слоев, количество нейронов в этих слоях и типа функции активации на основе современных методов построения глубоких нейронных сетей (Deep neural network).

Таблица 1. *Недостатки и преимущества использования нейронных сетей в качестве оператора прогноза.*

В общем случае, задачу обучения нейронных сетей можно отнести к типичным некорректным задачам – многозначности и неустойчивости решений, а также к возможному отсутствия решения. Например, если количество обучающих пар меньше, чем количество неизвестных весовых коэффициентов, то возникает возможность многозначности такого решения. Если обучающие пары имеют сильную корреляцию и все обучающие пары могут быть вычислены через другие, путем линейных операций, то может вообще не существовать такой нейронной сети для такого обучающего множества. Поэтому мы рекомендуем всегда применять методы регуляризации по А. Н. Тихонову, специально предназначенных для таких задач. В общем случае регуляризация по А. Н. Тихонову представляется в виде добавления к объектной функции дополнительного члена – суммы квадратов всей весовых коэффициентов как показано в формуле (4) (для одного выхода нейронной сети). Коэффициент регуляризации альфа определяет степень «гладкости», что не позволяет быть весовым коэффициентам слишком большими, и это стабилизирует решение.

$$F = \sum_k^M (y_k - y^r{}_k)^2 + \alpha \sum_i^K w_i{}^2 \rightarrow min \qquad (4)$$

Где F – объектная функция для минимизации,

M – количество обучающих пар,

y_k – выходное значение в k- ой обучающей паре,

$y^r{}_k$- вычисленное значение в k- ой обучающей паре

K – количество неизвестных весовых коэффициентов w_i,

α – коэффициент регуляризации Тихонова.

Обратное распространение ошибки

Как показано выше, обучение нейронной сети - это очень сложная задача, так как такая задача относится к некорректным задачам, ввиду ее неустойчивости и неоднозначности.

Если свести нейронную сеть к виду одного нейрона (много входов и один выход), то задача обучения может быть решена достаточно легко, например, на основе хорошо проработанной теории градиентных решений. Активационная функция в этом случае должна иметь возможность быть дифференцирована аналитически. Это является причиной

выбора очень простых функций в качестве активационных в нейронных сетях. Обычно это функция сигмоида или гиперболический тангенс. Одним из способов решения – градиентный итерационный процесс (Метод Ньютона), который в общем случае выглядит следующим образом.

$$w_i^{\,l} = w_i^{\,l-1} + \eta \frac{\partial E}{\partial w} \qquad (5)$$

Где l номер итерации

η – коэффициент релаксации (скорости сходимости)

$\frac{\partial E}{\partial w}$ - градиент изменения функции ошибки E на текущей итерации (может быть выражен через аналитически дифференцированной активационной функции или рассчитан численно).

Вся проблема заключается в том, что ошибка вычислений известна только для выходных нейронов, а для промежуточных нейронов, в сложной многослойной сети эта ошибка неизвестна.

Преимущество алгоритма обратного распространения ошибки заключается в том, что он позволяет выполнять оценку ошибки для каждого промежуточного нейрона на основе ее обратного пропорционального распределения по произведениям весов и значений, известных при вычислении сумм для каждого нейрона при прямых вычислениях. Схема таких вычислений показана на рис 5.

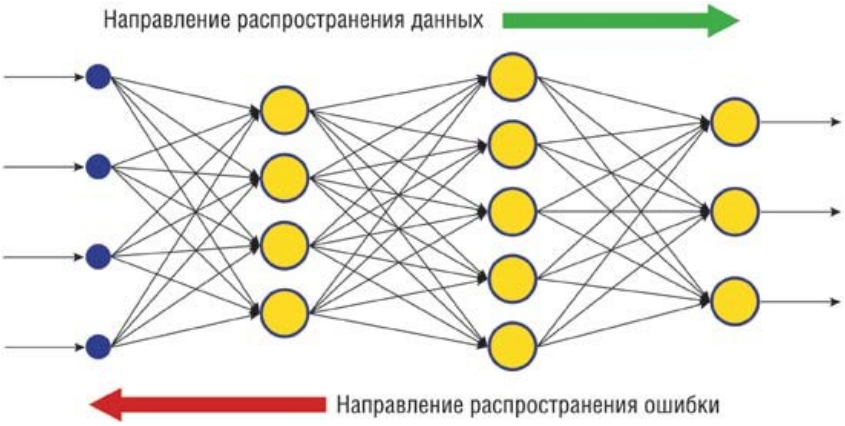

Рис. 5. Схема прямых вычислений нейронной сети и обратного распространения ошибки.

Есть варианты метода обратного распространения ошибки со стохастическим изменением весов

$$w_i^{\,l} = w_i^{\,l-1} + \eta \frac{\partial E}{\partial w} + \xi_i \qquad (6)$$

где ξ_i стохастическая составляющая в виде маленького случайного числа.

Генетические алгоритмы и гибридная схема

Чтобы построить нейронную сеть, как правило, используются два основных метода обучения. В первую очередь - это алгоритм обратного распространения ошибки, который является типичным градиентным методом (Bishop, 1991; Schultz et al., 1994; Bishop, 1995).

Второй способ обучения нейронной сети использует генетические алгоритмы (Whitley et al., 1990; Bishop and Bushnell, 1993; Veeken et al., 2009). Преимущества и недостатки этих методом обучения показана в таблице 2.

Методы обучения нейронных сетей	Недостатки	Достоинства
Обратного распространения ошибки (градиентный метод)	Сходимость к ближайшему к начальному приближению локальному минимуму решение существенно зависит от начального приближения для коэффициентов нейронной сети, и это затрудняет поиск глобального минимума	Высокая скорость сходимости.
Генетический алгоритм	Низкая скорость сходимости	Более высокая вероятность найти глобальный минимум целевой функции, так как анализируется большое количество вариантов обученной нейронной сети.

Таблица 2. *Недостатки и преимущества методов обучения нейронных сетей.*

Одним из общепризнанных подходов стабилизации решения для этих методов обучения нейронных сетей является регуляризация по Тихонову (Girosi et al., 1995; Bishop, 1995). При использовании регуляризации по Тихонову возникает необходимость применения дополнительного члена для целевой функции, которая, как правило, может быть суммой квадратов для всех коэффициентов в операторе прогнозирования, что не позволяет этому коэффициенту быть большим. (Tikhonov and Arsenin, 1977). Однако этот путь не решает проблемы единственности. Она остаётся «неуправляемой». Модификация стабилизации решения с управлением направления поиска оптимума обеспечивается на основе введения в стабилизирующий функционал соответствующего члена. Тем самым обеспечивается выделение единственного оптимума, соответствующего заданному критерию, определяемому введенным членом (Кобрунов, 1994).

Сочетание генетических алгоритмов и градиентных методов (Mishra and Debroy, 2006) в настоящее время считается перспективным направлением для построения стабильной нейронной сети.

Ранее мы предложили (Kobrunov and Priezzhev, 2015; Kobrunov and Priezzhev, 2016) использовать сочетание генетического алгоритма и градиентного метода для построения нелинейного оператора нейронной сети работает в соответствии с нижеприведенной процедурой:

1. Создать первоначальное поколение нейронных сетей как совокупность объектов (несколько вариантов коэффициентов прогнозирующего оператора на основе случайной функции.
2. Сделать выборку небольшого количества лучших объектов, т.е. вариантов нейронных сетей с наименьшей объектной функцией. (Этап селекции генетического алгоритма.)
3. Создать следующее поколение нейронных сетей на основе выбранных объектов в соответствии со следующими правилами:

- Кросс обмен: обмен коэффициентами между двумя случайно выбранными объектами.

- Мутация: Добавление небольших значений в коэффициенты нейронных сетей
- Применение одной итерации градиентного метода в соответствии с данными предыдущих итераций объектами.

Повторение шагов 2 и 3 до тех пор, пока объектная функция не станет достаточно маленькой или не будет достигнуто максимальное количество итераций.

Так как оператор нейронной сети может быть построен в соответствии с генетическим алгоритмом в сочетании с управляемым градиентным методом, то в данном случае имеются следующие основные преимущества:

- Использование стохастических измененных начальных значений для метода управляемого градиента может ускорить процесс итерации.
- Использование оптимального коэффициента релаксации управляемого градиентного метода максимально ускоряет сходимость итераций.
- Нет каких-либо ограничений для внутренней структуры и сложности нейронной сети, потому что расчет градиента и другие компоненты управляемого градиентного метода могут быть выполнены численно.
- Управляемый градиентный метод позволяет использовать априорную информацию наиболее эффективным способом для нахождения глобального решения с более высокой вероятностью.

Предложенная процедура позволяет избежать многих недостатков классических генетических алгоритмом, особенно их низкой скорости сходимости. Предложенная процедура также позволяет избежать многих недостатков градиентных методов, таких как высокий уровень зависимости от начальных значений и высокий уровень ошибок во время выполнения заключительной итерации.

Основной класс Neuron

Ниже приводится текст класса `Neuron.`
Данный класс включает определение и сохранение значения весовых коэффициентов текущих (weight) и для предыдущей итерации (parent_weight). Последние нужны в процессе обучения для определения градиента весовых коэффициентов в процессе обучения. В процессе создания объектов данного класса весовые коэффициенты инициализируются случайными числами. Также в процессе создания объектов определяется тип активационной функции. Функция OutputDSum нужна только для алгоритма обратного распространения ошибки и позволяет вычислять производную функцию от ранее вычисленной суммы произведения входных данных и весовых коэффициентов. Предназначение остальных элементов очевидно.
Атрибуте `[Serializable]` здесь и далее необходим для сохранения данных классов.

```
[Serializable]
public class Neuron
    {
        private List<float> weight;
        private List<float> parent_weight;
        private int numWeights;
        private float sum = 0f;
        private ActivationType activationFunction = ActivationType.Sigmoid;
        public Neuron(int nWeights, ActivationType activationFun)
        {
```

```
    activationFunction = activationFun;
    numWeights = nWeights;
    Random rnd = new Random();
    weight = new List<float>();
    parent_weight = new List<float>();
    for (int i = 0; i < numWeights; i++)
    {
        double r = 2.0 * rnd.NextDouble() - 1.0;
        double r1 = 2.0 * rnd.NextDouble() - 1.0;
        weight.Add((float)r);
        parent_weight.Add((float)r1);
    }
}

public float this[int i]
{
    get { return weight[i]; }
    set { weight[i] = value; }
}
public float Calculate(List<float> input)
{
    sum = 0;
    for (int i = 0; i < numWeights; i++)
    {
        float inp = input[i];
        if (inp != 0)
            sum += weight[i] * inp;
    }
    sum /= numWeights;
    return (ActivationFun(sum));
}
public float ActivationFun(float x)
{
  if (activationFunction == ActivationType.Sigmoid)
     return (float)(1 / (1 + Math.Exp(-x)));
  else if (activationFunction == ActivationType.Linear)
     return x;
  else if (activationFunction == ActivationType.HyperbolicTangent)
     return (float)((Math.Exp(x) - Math.Exp(-x)) / (Math.Exp(x) + Math.Exp(-x)));
  else if (activationFunction == ActivationType.PiaceWizeLinear)
     return x <= -0.5f ? 0 : x > -0.5f && x < 0.5f ? x + 0.5f : 1f;
  return 1f;
}

public float OutputDSum
{
    get { return OutputD(sum); }
}

public float OutputD(float x)
{
    if (activationFunction == ActivationType.Sigmoid)
    {
        float y = ActivationFun(x);
        return (y * (1 - y));
    }
    else if (activationFunction == ActivationType.Linear)
    {
        return 1f;
    }
    else if (activationFunction == ActivationType.HyperbolicTangent)
    {
        return (float)(4 / Math.Pow(Math.Exp(x) + Math.Exp(-x), 2));
    }
    else if (activationFunction == ActivationType.PiaceWizeLinear)
    {
        return x <= -0.5f ? 0 : x > -0.5f && x < 0.5f ? 1f : 0f;
    }
    return 0f;
}
public int Num_Inputs
{
    get { return numWeights; }
}
public void setWeight(List<float> w)
{
    for (int i = 0; i < numWeights; i++)
```

```
            {
                parent_weight[i] = weight[i];
                weight[i] = w[i];
            }
        }
        public List<float> getWeight()
        {
            return weight;
        }
        public List<float> Parent_Weight
        {
            get { return parent_weight; }
        }

    }
```

Класс Layer

Ниже приводится текст класса `Layer`.
Здесь выполняется определение, определение весов и вычисление всех нейронов, входящих в данный слой.

```
[Serializable]
public class Layer
    {
        private List<Neuron> neurons;
        private int numOuts;
        private int num_weights;
        private int numInputs;
        List<float> result;
        public Layer(int nOuts, int nInputs, ActivationType activation)
        {
            numInputs = nInputs;
            numOuts = nOuts;
            num_weights = 0;
            neurons = new List<Neuron>();
            for (int i = 0; i < numOuts; i++)
            {
                neurons.Add(new Neuron(numInputs, activation));
                num_weights += numInputs;
            }
        }

        public Neuron this[int i]
        {
            get { return neurons[i]; }
        }
        public List<float> Calculate(List<float> input)
        {
            result = new List<float>();
            for (int i = 0; i < numOuts; i++)
            {
                result.Add(neurons[i].Calculate(input));
            }
            return result;
        }

        public List<float> LastOutput
        {
            get { return result; }
        }

        public int setWeight(List<float> w)
        {
            int num = 0;
            foreach (Neuron n in this.neurons)
            {
                int ni = n.Num_Inputs;
                var arr = w.GetRange(num, ni);
                n.setWeight(arr);
                num += ni;
            }
            return num;
        }
```

```
        public List<float> getWeight()
        {
            List<float> arr = new List<float>();
            foreach (Neuron n in this.neurons)
            {
                arr.AddRange(n.getWeight());
            }
            return arr;
        }
        public int Num_weights
        {
            get { return num_weights; }
        }
        public int NumOuts
        {
            get { return numOuts; }
        }
        public int NumInputs
        {
            get { return numInputs; }
        }
    }
```

Класс NeuralNetwork

Это основной класс, который определяет нейронные сети. Для определения геометрии нейронных сетей необходимо определить количество входов и количество нейронов в каждом скрытом слое. Последний элемент в списке слоев определяет количество выходов или количество нейронов в выходном слое.
Другие элементы данного класса позволяют вычистить, получить или определить все коэффициенты из внешнего источника.
Элементы enum определяют тип вычисления ошибки, тип активационной функции, тип обучения нейронной сети.

```
    [Serializable]
    public enum ErrorType
    {
        L2,
        L1,
       Label
    };
    [Serializable]
    public enum ActivationType
    {
        Sigmoid,
        Linear,
        HyperbolicTangent,
        PiaceWizeLinear
    };
    [Serializable]
    public enum LearningType
    {
        Genetic,
        Gradient,
        Hibrid
    };
    [Serializable]
    public class NeuralNetwork
    {
        private List<Layer> layers;
        private int numLayers;
        private int nnSize;
        private int nnOuts;
        private int numInputs;
        private ActivationType activationFunction = ActivationType.Sigmoid;

        public NeuralNetwork(int nInputs, List<int> nLayers)
        {
```

```
            numInputs = nInputs;
            numLayers = nLayers.Count;
            nnSize = 0;
            layers = new List<Layer>();
            for (int i = 0; i < numLayers; i++)
            {
                if (i == 0) layers.Add(new Layer(nLayers[i], numInputs, activationFunction));
                else layers.Add(new Layer(nLayers[i], nLayers[i - 1], activationFunction));
                nnSize += layers[layers.Count - 1].Num_weights;
            }
            nnOuts = nLayers[numLayers - 1];
        }
        public NeuralNetwork(NeuralNetwork other)
        {
            numInputs = other.numInputs;
            numLayers = other.numLayers;
            nnSize = 0;
            layers = new List<Layer>();
            for (int i = 0; i < numLayers; i++)
            {
                layers.Add(new Layer(other.layers[i].NumOuts, other.layers[i].NumInputs,
activationFunction));

                nnSize += other.layers[i].Num_weights;
            }
            nnOuts = other.Num_Outputs;

        }
        public ActivationType ActivationFunction
        {
            get { return activationFunction; }
            set { activationFunction = value; }
        }



        public int NumInputs
        {
            get { return numInputs; }
        }
        public Layer this[int i]
        {
            get { return layers[i]; }
        }

        public List<float> Calculate(List<float> input)
        {
            List<float> inpt = new List<float>(input);

            List<float> outs = null;
            for (int i = 0; i < numLayers; i++)
            {
                outs = layers[i].Calculate(inpt.ToList());
                inpt = new List<float>(outs);
            }
            return outs;
        }
        public int SizeNN()
        {
            return nnSize;
        }
        public void SetNN(float[] w)
        {
            int num = 0;
            for (int i = 0; i < numLayers; i++)
            {
                int size = layers[i].Num_weights;
                var arr = w.ToList().GetRange(num, size);
                num += layers[i].setWeight(arr);
            }
        }
        public float[] GetNN()
        {
            List<float> weights = new List<float>();
            for (int i = 0; i < numLayers; i++)
            {
                weights.AddRange(layers[i].getWeight());
```

```
            }
            return weights.ToArray();
        }
        public int Num_Outputs
        {
            get { return nnOuts; }
        }
        public List<Layer> Layers
        {
            get { return layers; }
        }
        public int NumLayers
        {
            get { return numLayers; }
        }
    }
```

Класс Learning

Данный класс используется для обучения нейронных сетей. При инициализации (new Learning()) необходимо определить ряд параметров обучения, такие как:
MaxIter- максимальное количество итераций обучения,
LearningType- определяет тип обучения:

Genetic – для обучения будет использоваться генетические алгоритмы,
Gradient – для обучения будет использоваться градиентный метод,
Hibrid – для обучения будет использоваться гибридное сочетание генетического алгоритма и градиентного метода.

ErrorType- тип вычисления ошибки:

L2 - квадратическая ошибка,
L1 – абсолютное значение разницы,
Label – специальная ошибка для распознавания рукописных символов.

PopulationSize – размер поколения решений для генетического алгоритма.

Для старта процесса обучения нужно обратится к функции Learn с массивами для обучения **input** и **output**, которые должны содержать пары входных и выходных векторов для обучения и иметь одинаковый размер. Каждому вектору в массиве **input** соответствует вектор в массиве **output.**
Для вывода графиков ошибки в зависимости от номера итерации нужно определить графическое окно Chart chart1.

CrossValidation – процент обучающей выборки, используемой для валидации, и которая не используется при обучении. Разделение обучающей выборки на используемую (`inLearn`, `outLearn`) и неиспользуемую (`inCross`, `outcross`) для обучения части выполняется случайным образом динамически в процессе итераций. Если обучающая выборка содержит значительное количество пар, которые сильно коррелируют между собой, то для ускорения обучения рекомендуется большие значения CrossValidation>50%.

```
public class Learning
    {
        protected static Random rnd = new Random();
        private int populationSize = 50;
        private int maxIter = 500;
        private int mutationProcent = 75;
        private float mutationAmp = 1f;
        private int selectionSize = 10;
```

```
private float alpha = 0.5f;
private List<Genome> population;
private float crossValidation = 10;
private NeuralNetwork nn;
private LearningType learningType = LearningType.Gradient;
private ErrorType errorType = ErrorType.L2;

public NeuralNetwork NN
{
    set { nn = value; }
}
public LearningType LearningType
{
    get { return learningType; }
    set { learningType = value; }
}
public ErrorType ErrorType
{
    get { return errorType; }
    set { errorType = value; }
}
public int MaxIter
{
    set { maxIter = value; }
}
public int PopulationSize
{
    set { populationSize = value; }
}
public int SelectionSize
{
    set { selectionSize = value; }
}
public float CrossValidation
{
    set { crossValidation = value; }
}
public float Alpha
{
    set { alpha = value; }
}
protected float MutationValue
{
    get
    {
        double r = 2.0 * rnd.NextDouble() - 1.0;
        return ((float)r * mutationAmp);
    }
}
protected bool Mute
{
    get { return (rnd.Next(100) < mutationProcent); }
}
protected int RandSelectionIndex
{
    get { return rnd.Next(selectionSize); }
}
protected Genome Muted_Genome
{
    get
    {
        Genome muted = new Genome(nn);
        muted.Init(nn);
        for (int i = 0; i < muted.N_Genes; i++)
            muted[i] = MutationValue;
        return muted;
    }
}
protected Genome CrossOver()
{
    Random rnd1 = new Random();
    Random rnd2 = new Random();
    Genome rez = new Genome(nn);
    var i1 = rnd1.Next(selectionSize);
    var i2 = rnd1.Next(selectionSize);
    Genome gen1 = population[i1];
    Genome gen2 = population[i2];
```

```
            int i = 0;
            while (i < gen1.N_Genes)
            {
                if (rnd1.Next(100) >= 50) rez[i] = gen1[i];
                else rez[i] = gen2[i];
                if (rnd1.Next(100) < mutationProcent)
                {
                    var val = (2.0f * (float)rnd2.NextDouble() - 1.0f) * mutationAmp;
                    rez[i] += val;
                }
                i++;
            }
            return rez;
        }

        protected Genome BackAllocationCorrection(int i1, List<float[]> input, List<float[]> output)
        {
            NeuralNetwork nn1 = new NeuralNetwork(nn);
            population[i1].setWeights(nn1);
            var err = new float[nn1.Num_Outputs];
            float[][] tmp = new float[nn1.NumLayers][];
            for (int ll = 0; ll < nn1.NumLayers; ll++) tmp[ll] = new float[nn1[ll].NumOuts];
            for (int im = 0; im < input.Count; im++)
            {
                var nout = nn1.Calculate(input[im].ToList());
                for (int j = 0; j < nout.Count; j++) err[j] = output[im][j] - nout[j];
                int layer = nn1.NumLayers - 1;
                for (int j = 0; j < nn1[layer].NumOuts; j++)
                {
                    tmp[layer][j] = nn1[layer][j].OutputDSum * err[j];
                }
                for (layer--; layer >= 0; layer--)
                {
                    for (int j = 0; j < nn1[layer].NumOuts; j++)
                    {
                        var sum = 0f;
                        for (int i = 0; i < nn1[layer + 1].NumOuts; i++) sum += tmp[layer + 1][i] *
nn1[layer + 1][i][j];
                        tmp[layer][j] = nn1[layer][j].OutputDSum * sum;
                    }
                }

                for (layer = 0; layer < nn1.NumLayers; layer++)
                {
                    List<float> cur = new List<float>(); ;
                    if (layer == 0) cur = input[im].ToList();
                    else cur = nn1[layer - 1].LastOutput;
                    for (int j = 0; j < nn1[layer].NumOuts; j++)
                    {
                        for (int i = 0; i < nn1[layer][j].Num_Inputs; i++)
                        {
                            var deltaWeight = alpha * cur[i] * tmp[layer][j];
                            nn1[layer][j][i] += deltaWeight;
                        }
                    }
                }
            }

            Genome result = new Genome(nn1);

            float[] f1 = nn1.GetNN();
            result.SetGenes(f1);
            return result;
        }

        private void makeNewGeneration(List<float[]> input, List<float[]> output)
        {
            Genome[] result = new Genome[population.Count];
            result[0]=population[0];
            System.Threading.Tasks.Parallel.For(1, population.Count, (int index) =>
            {


                if (learningType == LearningType.Gradient)
                {
                    result[index] = (BackAllocationCorrection(index, input, output));
                }
```

```
            else if (learningType == LearningType.Genetic)
            {
                result[index] = (CrossOver());//
            }
            else if (learningType == LearningType.Hibrid)
            {
                if (index % 2 == 0)
                    result[index] = (BackAllocationCorrection(index, input, output));
                else
                    result[index] = (CrossOver());
            }
        });

        population = result.ToList();
    }

    private void CalculateError(List<float[]> input, List<float[]> output)
    {

        //foreach (Genome ind in population) // без параллельных вычислений
        //{
        //    CalculateErrorOneGenom(ind, input, output);
        //}

        System.Threading.Tasks.Parallel.For(0, population.Count, (int ii) =>
        {
            CalculateErrorOneGenom((Genome)population[ii], input, output);
        });

    }

    private void CalculateErrorOneGenom(Genome ind, List<float[]> input, List<float[]> output)
    {
        NeuralNetwork nn1 = new NeuralNetwork(nn);
        ind.setWeights(nn1);

        int nouts = nn1.Num_Outputs;
        int n = input.Count;
        if (ErrorType == ErrorType.Label)
        {
            int error = 0;
            for (int im = 0; im < n; im++)
            {
                var rez = nn1.Calculate(input[im].ToList());
                float max = float.MinValue;
                int imax = 0;
                for (int i = 0; i < nouts; i++)
                {
                    if (max < rez[i])
                    {
                        max = rez[i];
                        imax = i;
                    }
                }
                if (output[im][imax] != 1f) error++;
            }
            ind.Error = ((float)error / n);
        }
        else if (ErrorType == ErrorType.L2)
        {
            double error = 0;
            for (int im = 0; im < n; im++)
            {
                var rez = nn1.Calculate(input[im].ToList());
                for (int i = 0; i < nouts; i++)
                {
                    error += Math.Pow(output[im][i] - rez[i], 2);
                }
            }
            ind.Error = ((float)Math.Sqrt(error / n));
        }
        else if (ErrorType == ErrorType.L1)
        {
            double error = 0;
            for (int im = 0; im < n; im++)
            {
                var rez = nn1.Calculate(input[im].ToList());
```

```
                for (int i = 0; i < nouts; i++)
                {
                    error += Math.Abs(output[im][i] - rez[i]);
                }
            }
            ind.Error = ((float)error / n);
        }
    }

public void Learn(List<float[]> input, List<float[]> output,  Chart chart1)
    {

        population = new List<Genome>();


        chart1.ChartAreas.Clear();
        ChartArea chartArea = new ChartArea() { Name = "ChartArea" };
        chartArea.AxisX.MajorGrid.LineWidth = 0;
        chartArea.AxisY.MajorGrid.LineWidth = 0;
        chartArea.AxisY.LabelStyle.Format = "{0.000}";
        chartArea.AxisY.Title = "Error";
        chartArea.AxisY.Minimum = 0;
        chartArea.AxisY.Maximum = 0;
        chartArea.AxisX.LabelStyle.Format = "{0}";
        chartArea.AxisX.Title = "Iterations";
        chartArea.AxisX.Minimum = 0;
        chartArea.AxisX.Maximum = maxIter;
        chart1.Legends[0].Docking = Docking.Top;
        chart1.ChartAreas.Add(chartArea);
        chart1.Series.Clear();
        chart1.Series.Add("Error");
        chart1.Series["Error"].Color = Color.Red;
        chart1.Series["Error"].ChartType = SeriesChartType.Line;
        chart1.Series["Error"].BorderWidth = 1;
        chart1.Series.Add("ErrorQC");
        chart1.Series["ErrorQC"].Color = Color.Blue;
        chart1.Series["ErrorQC"].ChartType = SeriesChartType.Line;
        chart1.Series["ErrorQC"].BorderWidth = 1;

        for (int i = 0; i < populationSize; i++)
            population.Add(Muted_Genome);
        population[0].Init(nn);

        Genome genomeQC = Muted_Genome;
        Genome genomeQCmin = Muted_Genome;
        float minGenome = float.MaxValue;
        List<float[]> inCross = new List<float[]>();
            List<float[]> outCross = new List<float[]>();
            List<float[]> inLearn = new List<float[]>();
            List<float[]> outLearn = new List<float[]>();
        int iter = 0;
        int iterWithout = 0;
        List<float[]> alphaStat = new List<float[]>();
        do
        {

            int n_ins = input.Count;

            int n_CV = (int)(n_ins * (100 - crossValidation) / 100);
            inLearn.Clear();
            outLearn.Clear();
            inCross.Clear();
            outCross.Clear();

            for (int i = 0; i < n_ins; i++)
            {
                if (rnd.Next(n_ins) < n_CV)
                {
                    inLearn.Add(input[i]);
                    outLearn.Add(output[i]);
                }
                else
                {
                    inCross.Add(input[i]);
                    outCross.Add(output[i]);
                }
            }
```

```
            // if (iter > 0)
            makeNewGeneration(inLearn, outLearn);
            CalculateError(inLearn, outLearn);

            population.Sort();

            for (int i = selectionSize; i < populationSize; i++)
                population[i] = Muted_Genome;

            var err = population[0].Error;

            genomeQC.SetGenes(population[0].Genes());
            CalculateErrorOneGenom(genomeQC, inCross, outCross);

            var errQC = genomeQC.Error;
            if (float.IsNaN(errQC)) errQC = err;
            if (errQC < minGenome)
            {
                float[] add = new float[2];
                add[0] = alpha;
                add[1] = (minGenome - errQC);

                genomeQCmin = genomeQC;
                minGenome = errQC;
                iterWithout = 0;

                alphaStat.Add(add);
            }

            alpha = iterWithout == 0 ? alpha * 1.1f : alpha * 0.9f;
            if (alpha < 0.001f) alpha = 0.5f;
            if (alpha > 1f) alpha = 0.5f;
            mutationAmp = alpha;


            if (population.Count>1)  population[1] = genomeQCmin;


            if (err >= chartArea.AxisY.Maximum)
            {
                chartArea.AxisY.Maximum = err;
            }

            if (errQC >= chartArea.AxisY.Maximum)
            {
                chartArea.AxisY.Maximum = errQC;
            }

            chart1.Series["Error"].Points.AddXY(iter, err);
            chart1.Series["ErrorQC"].Points.AddXY(iter, minGenome);
            if (iter > 1) chart1.Update();
            iter++;
            iterWithout++;

        } while (iter < maxIter);
        nn.SetNN(genomeQCmin.Genes());


        using (System.IO.StreamWriter file =
   new System.IO.StreamWriter(@"C:\Projects\alphaLines.txt"))
        {

            foreach (float[] values in alphaStat)
            {
                string str = string.Format("{0}  {1}", values[0], values[1]);
                file.WriteLine(str);
            }

            file.Close();
        }

    }
}
public class Genome : IComparable
{
    private float[] genes;
```

```
        private float err = float.NaN;

        public float Error
        {
            get { return err; }
            set { err = value; }
        }
        public float this[int index]
        {
            get { return genes[index]; }
            set { genes[index] = value; }
        }
        public int N_Genes
        {
            get { return genes.Length; }
        }

        public Genome(int size)
        {

            genes = new float[size];
            err = float.NaN;
        }

        public float[] Genes()
        {
            return (genes);
        }

        public Genome(NeuralNetwork nn)
        {
            int sizeNN = nn.SizeNN();

            genes = new float[sizeNN];
            err = float.NaN;
        }
        public void Init(NeuralNetwork nn)
        {
            genes = nn.GetNN();
        }
        public void setWeights(NeuralNetwork nn)
        {
            nn.SetNN(genes);

        }
        public float[] getWeights()
        {
            return (genes);
        }
        public void SetGenes(float[] g)
        {
            genes = g;
        }

        public int CompareTo(Object other)
        {
            if (float.IsNaN(err)) err = float.MaxValue;
            float errOther = ((Genome)other).err;
            if (float.IsNaN(errOther)) errOther = float.MaxValue;
            if (err > errOther) return 1;
            else if (err == errOther) return 0;
            else return -1;
        }
    }
```

Тестовый пример

Фрагмент кода для определения нейронной сети и обращения к классу обучения показан ниже.

```
…….
                List<int> Layers = new List<int>();
```

```
int numInputs = dataInput[0].Length; // количество входов
int numOutputs = dataOutput[0].Length; // количество выходов

Layers.Add(numOutputs * 10); // скрытый слой

Layers.Add(numOutputs); // выходной слой


 NeuralNetwork  NN = new NeuralNetwork(numInputs, Layers);
 NN.ActivationFunction = acType;

 Learning LN = new Learning();
 LN.NN = NN;
 LN.LearningType = LearningType.Hibrid;
 LN.CrossValidation = 80;
 LN.MaxIter = 300;
 LN.PopulationSize = 25;
 LN.SelectionSize = 5;

 LN.ErrorType = ErrorType.L2;

 LN.Learn(dataInput, dataOutput, chart1);
```

…….

Для вычислений по ранее обученной нейронной сети нужно обратится следующим образом

```
var rez = NN.Calculate(dataInput[im].ToList());
```

Получить все коэффициенты нейронной сети и ее конфигурацию для их сохранения:

```
float[] coef = NN.GetNN();
int numLayers = NN.NumLayers;

int numInputs = NN.Layers[0].NumInputs;
List<int> nLayers = new List<int>();
foreach (Layer l in NN.Layers) nLayers.Add(l.NumOuts);
```

Определить все коэффициенты из массива можно выполнить следующим образом:

```
NeuralNetwork NN = new NeuralNetwork(numInputs, Layers);
NN.SetNN(coef.ToArray());
```

Список литературы

- Bishop, C.M., 1995, Training with noise is equivalent to Tikhonov regularization: Neural Computation 7, no.1, 108–116.
- Bishop, C.M., and M.J. Bushnell, 1993, Genetic optimization of neural network architectures for color recipe prediction: Proceedings of the International Joint Conference on Neural Networks and Genetic Algorithms, Innsbruck, 719–725.
- Girosi, F., M. Jones, and T. Poggio, 1995, Regularization theory and neural networks architectures: Neural Computation 7, 219–269.
- Hampson, B., J. Schuelke, and J. Quirein, 2001, Use of multi-attribute transforms to predict log properties from seismic data: Geophysics, 66, no. 1, 3–46.

- Kobrunov, A.I., 1978, Optimization method to inverse gravity data: Earth Physics USSR Academy of Science, 8, 73–78.
- Kobrunov, A.I., 1994, The theory of interpretation of gravity data for media of intricate structure: Academic Express, Geophysical Express 1, no. 1, 1–20.
- Kobrunov A., I. Priezzhev, 2015, Stable Nonlinear Predictive Operator Based on Neural Network, Genetic Algorithm and Controlled Gradient Method, New Orleans, 2015 SEG Annual Meeting.
- Kobrunov A., I. Priezzhev, 2016, Hybrid combination genetic algorithm and controlled gradient method to train a neural network, GEOPHYSICS, VOL. 81, NO. 4, 1–9.
- Минский, М., Пейперт, С. Персептроны = Perceptrons. — М.: Мир, 1971. — 261 с.
- Mishra, S., and T. Debroy, 2006, A genetic algorithm and gradient-descent based neural network with the predictive power of a heat and fluid flow model for welding: Welding Journal, 231–242.
- Priezzhev, I., A. Scollard, and Z. Lu, 2014, Regional production prediction technology based on gravity and magnetic data from the Eagle Ford formation, Texas, USA, Denver SEG.
- Розенблатт, Ф. Принципы нейродинамики: Перцептроны и теория механизмов мозга = Principles of Neurodynamic: Perceptrons and the Theory of Brain Mechanisms. — М.: Мир, 1965. — 480 с.
- Roth, G., and A. Tarantola, 1994, Neural networks and inversion of seismic data, Journal of Geophysical Research, 99, no. B4, 6753–6768.
- Russell, B., D. Hampson, J. Schuelke, and J. Quirein, 1997, Multi-attribute seismic analysis: The Leading Edge, 16, no. 10, 1439–1443.
- Schultz, P.S, S. Ronen, M. Hattori, and C. Corbett, 1994, Seismic-guided estimation of log properties: Part 1(a): A data-driven interpretation methodology: The Leading Edge, 13, no. 5, 305-315. Part 2 (b): Using artificial neural networks for nonlinear attribute calibration: The Leading Edge, 13, no. 6, 674–678.
- Tikhonov, A. N., and Arsenin V. Y., 1977. Solutions of ill-posed problems, V H Winston and Sons, Washington D.C.
- Veeken, P.C.H., I.I. Priezzhev, L.E. Shmaryan, Y.I. Shteyn, A.Y. Barkov, and Y.P. Ampilov, 2009, Non-linear multi-trace genetic inversion applied on seismic data across the Shtokman field (offshore northern Russia): Geophysics, 74, no. 6, 49–59.
- Whitley, D., T. Starkweather, and C. Bogart, 1990, Genetic algorithms and neural networks: Optimizing connections and connectivity: Parallel Computing 14, 347–361.
- Колмогоров А.Н. О представлении непрерывных функций нескольких переменных суперпозициями непрерывных функций меньшего числа переменных // Докл. АН СССР, том 108, с. 2, 1956.
- Колмогоров А.Н. О представлении непрерывных функций нескольких переменных в виде суперпозиций непрерывных функций одного переменного и сложения // Докл. АН СССР, том 114, с. 953-956, 1957.
- Kolmogorov A.N. On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition, American Math. Soc. Transl., 28 (1963), pp. 55-63.
- Hecht-Nielsen R. Kolmogorov's Mapping Neural Network Existence Theorem // IEEE First Annual Int. Conf. on Neural Networks, San Diego, 1987, Vol. 3, pp. 11-13.
- Арнольд В.И. // Докл. АН СССР, том 114, N 4, 1957.
- Muller B., Reinhart J. Neural Networks: an introduction, Springer-Verlag, Berlin Heidelberg, 1990.
- Widrow B., Lehr M.A. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation // Proceedings of the IEEE, vol. 78, No. 9, September, 1990, p. 1415-1442.

[Bra99] Vasco Brattka. Recursive and computable operations over topological structures.

Informatik Berichte 255, FernUniversit¨at Hagen, Fachbereich Informatik,
Hagen, July 1999. Dissertation.
[Hil00] David Hilbert. Mathematische Probleme. Nachrichten der K¨oniglichen
Gesellschaft der Wissenschaften zu G¨ottingen, pages 253{297, 1900. Vortrag,
gehalten auf dem internationalen Mathematiker-Kongre zu Paris 1900.
[Hil27] David Hilbert. ¨Uber die Gleichung neunten Grades. Mathematische Annalen,
97:243{250, 1927.
[HN87] Robert Hecht-Nielsen. Kolmogorov's mapping neural network existence theorem.
In Proceedings IEEE International Conference On Neural Networks,
volume II, pages 11{13, New York, 1987. IEEE Press.
[HN90] Robert Hecht-Nielsen. Neurocomputing. Addison-Wesley, Reading, 1990.
[Ko91] Ker-I Ko. Complexity Theory of Real Functions. Progress in Theoretical Computer
Science. Birkh¨auser, Boston, 1991.

[Kol57] A.N. Kolmogorov. On the representation of continuous functions of many variables
by superposition of continuous functions of one variable and addition.
Dokl. Akad. Nauk USSR, 114:953{956, 1957. [translated in: American Mathematical
Society Translations 28 (1963) 55{59].
[KS94] Hidefume Katsuura and David A. Sprecher. Computational aspects of Kolmogorov's
superposition theorem. Neural Networks, 7(3):455{461, 1994.
[Kur91] Vera Kurkova. 13th Hilbert's problem and neural networks. In M. Novak
and E. Pelikan, editors, Theoretical Aspects of Neurocomputing, pages 213{
216, Singapore, 1991. World Scienti

c. Symposium on Neural Networks and
Neurocomputing, NEURONET, Prague, 1990.
[Kur92] Vera Kurkova. Kolmogorov's theorem and multilayer neural networks. Neural Networks, 5:501{506, 1992.
[Lor66] G.G. Lorentz. Approximation of Functions. Athena Series, Selected Topics in Mathematics. Holt, Rinehart and Winston, Inc., New York, 1966.
[Lor76] G.G. Lorentz. The 13-th problem of Hilbert. In Felix E. Browder, editor, Mathematical developments arising from Hilbert problems, volume 28 of Proceedings
of the Symposium in Pure Mathematics of the AMS, pages 419{430, Providence,
1976. American Mathematical Society. Northern Illinois Univiversity 1974.
[NMK93] Mutsumi Nakamura, Ray Mines, and Vladik Kreinovich. Guaranteed intervals for Kolmogorov's theorem (and their possible realtion to neural networks).
Interval Computations, 3:183{199, 1993. Proceedings of the International Conference
on Numerical Analysis with Automatic Result Veri

cation (Lafayette,

LA, 1993).

[PER89] Marian B. Pour-El and J. Ian Richards. Computability in Analysis and Physics.

Perspectives in Mathematical Logic. Springer, Berlin, 1989.

[Spr65] David A. Sprecher. On the structure of continuous functions of several variables.

Transactions American Mathematical Society, 115(3):340{355, 1965.

[Spr93] David A. Sprecher. A universal mapping for Kolmogorov's superposition theorem.

Neural Networks, 6:1089{1094, 1993.

[Spr96] David A. Sprecher. A numerical implementation of Kolmogorov's superpositions.

Neural Networks, 9(5):765{772, 1996.

[Spr97] David A. Sprecher. A numerical implementation of Kolmogorov's superpositions

II. Neural Networks, 10(3):447{457, 1997.

[Wei00] Klaus Weihrauch. Computable Analysis. Springer, Berlin, 2000.

1. Hornick, Stinchcombe, White. Multilayer Feedforward Networks are Universal Approximators. Neural Networks, 1989, v. 2, № 5.
2. Cybenko. Approximation by Superpositions of a Sigmoidal Function. Mathematical Control Signals Systems, 1989, 2.
3. Funahashi. On the Approximate Realization of Continuous Mappings by Neural Networks. Neural Networks, 1989, v. 2, № 3.