# ЛАБОРАТОРНАЯ РАБОТА № 3

# СОЗДАНИЕ НЕЙРОННОЙ СЕТИ С ПРЯМОЙ ПЕРЕДАЧЕЙ ИНФОРМАЦИИ. АЛГОРИТМЫ ОБУЧЕНИЯ НЕЙРОННЫХ СЕТЕЙ.

**Цель работы:** Освоение методики создания нейронной сети с прямой передачей данных. Освоение разнообразных алгоритмов обучения нейронных сетей и моделирование их с помощью предоставляемых библиотек.

## 3.1 Указания по подготовке к лабораторной работе

### 3.1.1 Нейронная сеть с прямой передачей информации. Различные алгоритмы обучения

При решении с помощью нейронных сетей прикладных задач необходимо собрать достаточный и представительный объем данных для того, чтобы обучить нейронную сеть решению таких задач. Обучающий набор данных - это набор наблюдений, содержащих признаки изучаемого объекта. Первый вопрос, какие признаки использовать и сколько и какие наблюдения надо провести.

Выбор признаков, по крайней мере первоначальный, осуществляется эвристически на основе имеющегося опыта, который может подсказать, какие признаки являются наиболее важными. Сначала следует включить все признаки, которые, по мнению аналитиков или экспертов, являются существенными, на последующих этапах это множество будет сокращено.

Нейронные сети работают с числовыми данными, взятыми, как правило, из некоторого ограниченного диапазона. Это может создать проблемы, если значения наблюдений выходят за пределы этого диапазона или пропущены.

Вопрос о том, сколько нужно иметь наблюдений для обучения сети, часто оказывается непростым. Известен ряд эвристических правил, которые устанавливают связь между количеством необходимых наблюдений и размерами сети. Простейшее из них гласит, что количество наблюдений должно быть в 10 раз больше числа связей в сети. На самом деле это число зависит от сложности того отображения, которое должна воспроизводить нейронная сеть. С ростом числа используемых признаков количество наблюдений возрастает по нелинейному закону, так что уже при довольно небольшом числе признаков, скажем 50, может потребоваться огромное число наблюдений. Эта проблема носит название "проклятие размерности".

Для большинства реальных задач бывает достаточным нескольких сотен или тысяч наблюдений. Для сложных задач может потребоваться большее количество, однако очень редко встречаются задачи, где требуется менее 100 наблюдений. Если данных мало, то сеть не имеет достаточной информации для обучения, и лучшее, что можно в этом случае сделать, - это попробовать подогнать к данным некоторую линейную модель.

### 3.1.2 Нейронная сеть прямой передачи информации

*Синтаксис*

```
net = newff(minmax, size, transf=None)
```

*Описание*

Функция **newff** предназначена для создания многослойных нейронных сетей прямой передачи информации с заданными функциями обучения и настройки, которые используют метод обратного распространения ошибки.

*Входные аргументы:*

**`minmax`** – массив (list) размера $ci \times 2$ минимальных и максимальных входных значений для $ci$ векторов входа; может быть сформирован функцией neurolab.tool.minmax() по готовой обучающей выборке.

**`size`** – массив (list), задающий количество нейронов в каждом слое;

**`transf -`** массив (list) функций активации каждого слоя. По умолчанию TanSig.

*Выходные аргументы:*

net – объект класса Net многослойной нейронной сети.

С внутренними аргументами класса с пояснениями можно ознакомиться, введя команду `print net.__doc__`, а с внутренними характеристиками конкретной сети – `net.__dict__`.

Свойства сети:

Функциями активации (**`transf`**) могут быть любые дифференцируемые функции, например TanSig, LogSig, Purelin и другие. Подробнее **`help(neurolab.trans)`**.

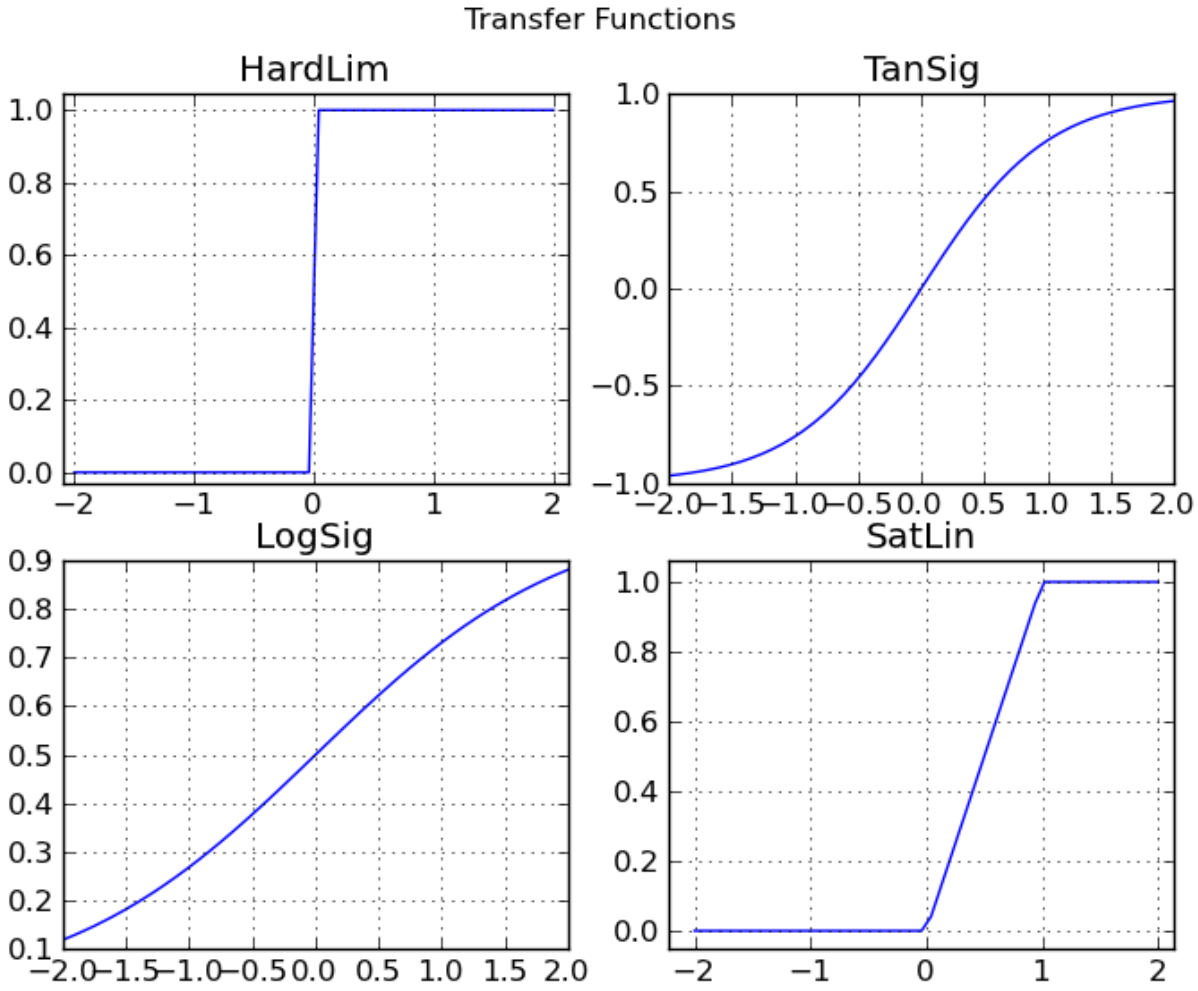


Рисунок 3.1 — Некоторые передаточные функции нейронов (по горизонтали каждого графика отложен входной сигнал, по вертикали выходной)

Обучающими функциями (**`trainf`**) могут быть любые функции, реализующие метод обратного распространения ошибки: train_gd, train_gdx, train_rprop и другие. Подробнее **`help(neurolab.train)`**.

Критерием качества обучения (**errorf**) может быть любая дифференцируемая функция. В **neurolab.error** представлены функции, основанные на модуле ошибки (MAE, SAE) и на её квадрате (MSE, SSE). Функции MAE, MSE определяют среднюю ошибку (абсолютную и квадратичную соответственно), а SAE и SSE — их сумму по всем наблюдениям. Между суммой ошибок и средней ошибкой нет никакой разницы в эффективности, но сравнивать обычно удобнее средние ошибки.

Пример:

```
>>> # создание ИНС с 2 входами, 1 выходом и 1 скрытым слоем из 3-х нейронов
>>> net = lb.net.newff( [ [-0.5, 0.5],[-0.5, 0.5] ], [3,1], [nl.trans.TanSig,
nl.trans.PureLin])
>>> net.ci # число входов
2
>>> net.co # число выходов
1
>>> len(net.layers) # число слоёв (выходной тоже считается)
2
>>> net.__dict__
{'ci': 2,
 'co': 1,
# схема соединения слоёв
 'connect': [[-1], [ 0], [1]],
# критерий качества обучения
 'errorf': <neurolab.error.SSE instance at 0x058C50A8>,
# текущие значения входа
'inp': array([  0.,  0.]),
# диапазон входных значений каждого входа
'inp_minmax': array([[- 0.5,  0.5],
       [- 0.5,  0.5]]),
# тип нейронов каждого слоя
'layers': [<neurolab.layer.Perceptron object at 0x058E20D0>,
           <neurolab.layer.Perceptron object at 0x058E23D0>],
# текущие значения выхода
'out': array([  0.]),
# диапазон выходных значений
 'out_minmax': array([[-1., 1.]]),
# алгоритм обучения
 'trainf': <neurolab.core.Trainer object at 0x058A77D0>}
```

### 3.1.3 Методы обучения

Как только начальные веса и смещения нейронов установлены (разные способы начальной инициализации см. в neurolab.init), сеть готова для того, чтобы начать процедуру ее обучения. Сеть может быть обучена решению различных прикладных задач — аппроксимации функций, идентификации и управления объектами, распознавания образов, классификации объектов и т. п. Процесс обучения требует набора примеров ее желаемого поведения — входов $p$ и желаемых (целевых) выходов $t$; во время этого процесса веса и смещения настраиваются так, чтобы минимизировать некоторый функционал ошибки (функцию потерь). По умолчанию в качестве такого функционала для сетей с прямой передачей сигналов принимается среднеквадратичная ошибка между векторами выхода $a$ и $t$. Ниже обсуждается несколько методов обучения для сетей с прямой передачей сигналов.

При обучении сети рассчитывается некоторый функционал, характеризующий качество обучения:

$$J = \frac{1}{2}\sum_{q=1}^{Q}\sum_{i=1}^{S^M}\left(t_i^q - a_i^{qS^M}\right)^2, \tag{3.1}$$

где $J$ — *квадратичная* функция потерь (SSE); $Q$ — объем выборки; $M$ - число слоев сети; $q$ - номер выборки; $S^M$ - число нейронов выходного слоя; $a^q = [a_i^{qM}]$ - вектор сигнала на выходе сети; $t^q = [a_i^q]$ - вектор желаемых (целевых) значений сигнала на выходе сети для выборки с номером $q$.

Есть и другие методы вычисления потерь (некоторые примеры в **`neurolab.error)`**. Главная задача любой функции потерь — оценить несоответствие результата работы сети и обучающего ответа для каждого наблюдения и для выборки в целом. Ошибки не могут компенсировать друг друга, поэтому все оценки знаконезависимы (для этого применяется модуль или квадрат разницы)

Затем с помощью того или иного метода обучения определяются значения настраиваемых параметров (весов и смещений) сети, которые обеспечивают минимальное значение функционала ошибки. Таким образом, задача обучения сети сводится к задаче оптимизации в многомерном пространстве (размерность равна количеству параметров сети). Задача оптимизации давно известна и хорошо изучена, большинство численных методов оптимизации применимы (и применяются) в качестве алгоритмов обучения нейронных сетей.

Большинство методов обучения многослойных сетей основано на вычислении градиента функционала ошибки по настраиваемым параметрам. Рекомендуется повторить теоретический материал градиентных методов оптимизации перед продолжением работы.

**Обучение однослойной сети**

Наиболее просто градиент функции ошибки вычисляется для однослойных нейронных сетей. В этом случае $M = 1$ и выражение для функционала принимает вид:

$$J = \frac{1}{2}\sum_{q=1}^{Q}\sum_{i=1}^{S^1}\left(t_i^q - a_i^{qS^1}\right)^2 = \frac{1}{2}\sum_{q=1}^{Q}\sum_{i=1}^{S^1}\left(t_i^q - f(n_i^q)\right)^2, \quad i = 1,\ldots S, \tag{3.2}$$

где $f(n_i^q)$ — функция активации; $n_i^q = \sum_{j=0}^{R} w_{ij}\, p_j^q$ — сигнал на входе функции активации для $i$-го нейрона; $p^q = [p_i^q]$ — вектор входного сигнала; $R$ — число элементов вектора входа; $S$ — число нейронов в слое; $w_{ij}$ — весовые коэффициенты сети.

Включим вектор смещения в состав матрицы весов $W = [w_{ij}],\ i = 1,\ldots,S,\ j = 1,\ldots,R$, а вектор входа дополним элементом, равным 1.

Применяя правило дифференцирования сложной функции, вычислим градиент функционала ошибки, предполагая при этом, что функция активации дифференцируема:

$$J = -\sum_{q=1}^{Q}\left(t_i^q - f(n_i^q)\right)\frac{\partial(f(n_i^q))}{\partial w_{ij}} = -\sum_{q=1}^{Q}\left(t_i^q - f(n_i^q)\right)f'(n_i^q)\,p_j^q. \tag{3.3}$$

Введем обозначение

$$\Delta_i^q = (t_i^q - f(n_i^q))\, f'(n_i^q) = (t_i^q - a_i^q)\, f'(n_i^q),\ i = 1,\ldots,S, \qquad (3.4)$$

и преобразуем выражение (3.3) следующим образом:

$$\frac{\partial J}{\partial w_{ij}} = -\sum_{q=1}^{Q} \Delta_i^q\, f'(n_i^q)\, p_j^q,\ i = 1,\ldots,S. \qquad (3.5)$$

Полученные выражения упрощаются, если сеть линейна. Поскольку для такой сети выполняется соотношение $a_i^q = n_i^q$, то справедливо условие $f'(n_i^q) = 1$. В этом случае выражение (3.3) принимает уже знакомый вид:

$$\frac{\partial J}{\partial w_{ij}} = (t_i^q - a_i^q)\, p_j^q,\ i = 1,\ldots,S,\ j = 0,\ldots,R. \qquad (3.6)$$

Выражение (3.6) положено в основу алгоритма Уидроу-Хоффа, применяемого для обучения линейных нейронных сетей.

Линейные сети могут быть обучены и без использования итерационных методов, а путем решения следующей системы линейных уравнений:

$$\sum_{j=0}^{R} w_{ij}\, p_j^q = t_i^q,\ i = 1,\ldots,S,\ \ q = 1,\ldots,Q, \qquad (3.7)$$

или в векторной форме записи:

$$Wp = t,\ W = [w_{ij}],\ p = [p_j^q], t = [t_i^q], i = 1,\ldots,S,\ j = 0,\ldots,R,\ q = 1,\ldots,Q. \qquad (3.8)$$

Если число неизвестных системы (3.7) равно числу уравнений, то такая система может быть решена, например, методом исключения Гаусса с выбором главного элемента. Если же число уравнений превышает число неизвестных, то решение ищется с использованием метода наименьших квадратов.

Важно понимать, что в случае нелинейной функции активации нейрона, рельеф ошибки значительно усложняется, как это показано на рисунке 3.2. Поверхность уже не может считаться унимодальной, и доступны только численные методы поиска минимума.

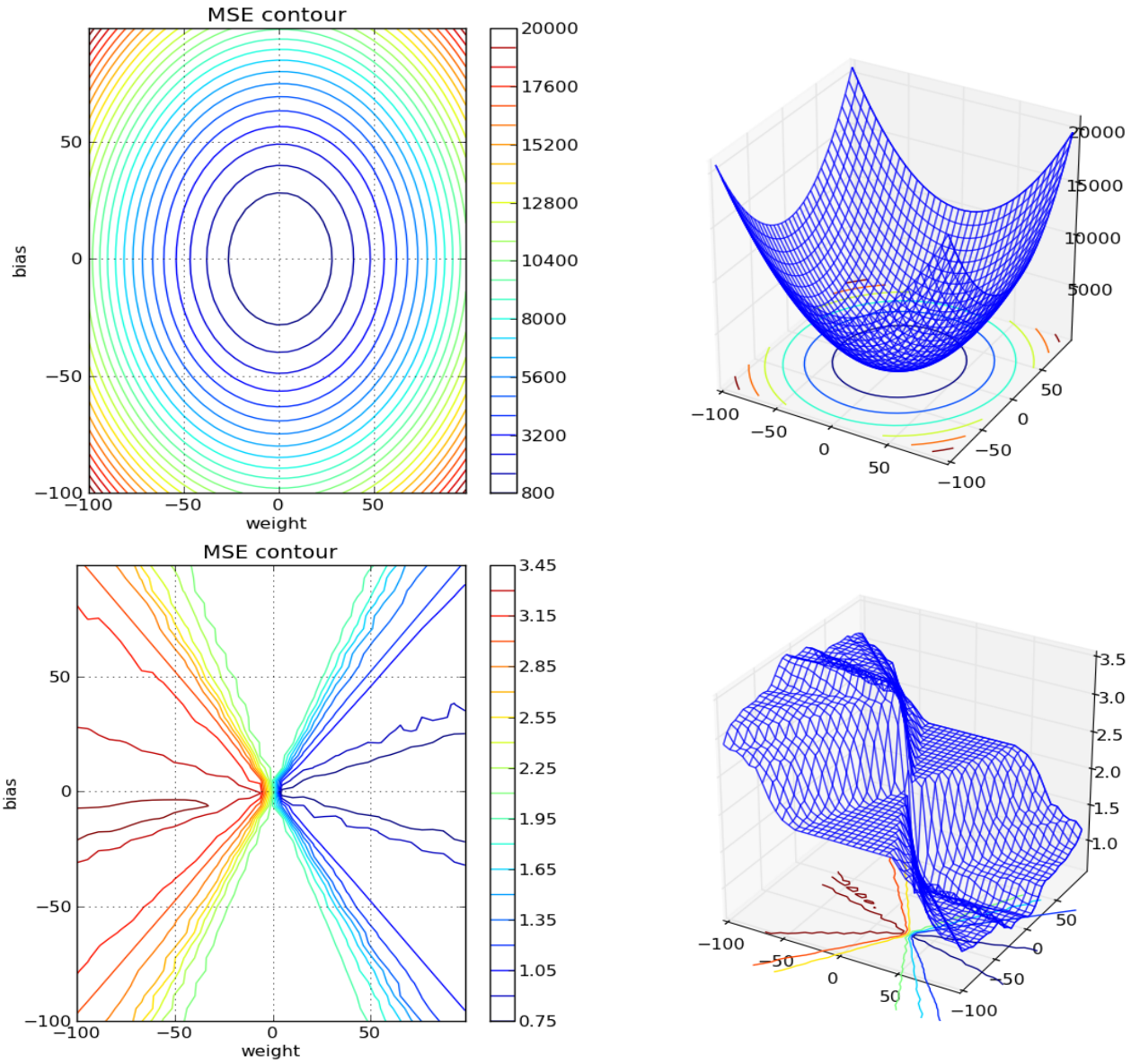


Рис 3.2а — Поверхность ошибки однонейронной сети с линейной функцией активации (вверху) и сигмоидальной для одинаковой задачи предсказания сигнала.

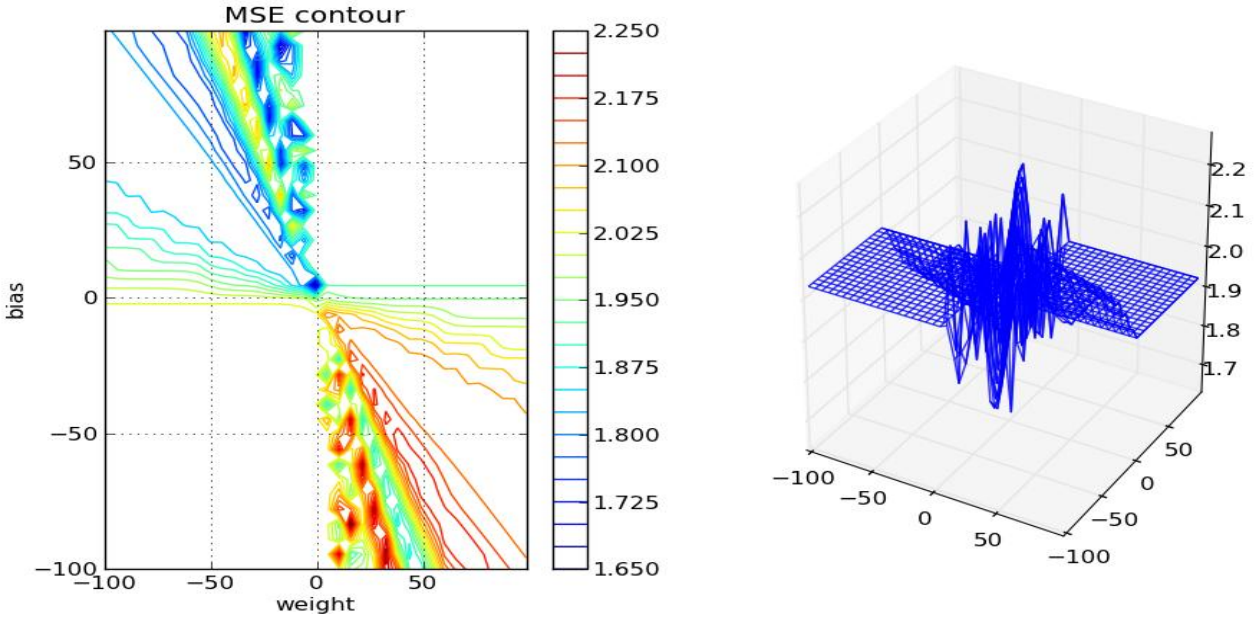


Рис 3.2б — Поверхность ошибки однонейронной сети для задачи аппроксимации

**Обучение многослойной сети**

Архитектура многослойной сети существенно зависит от решаемой задачи. Для линейных нейронных сетей может быть установлена связь между суммарным количеством весов и смещений с длиной обучающей последовательности. Для других типов сетей число слоев и нейронов в слое часто определяется опытом, интуицией проектировщика и эвристическими правилами.

Обучение сети включает несколько шагов:

- выбор начальной конфигурации сети с использованием, например, следующего эвристического правила: количество нейронов промежуточного слоя определяется половиной суммарного количества входов и выходов;
- проведение ряда экспериментов с различными конфигурациями сети и выбор той, которая дает минимальное значение функционала ошибки;
- если качество обучения недостаточно, следует увеличить число нейронов слоя или количество слоев;
- если наблюдается явление переобучения, следует уменьшить число нейронов в слое или удалить один или несколько слоев.

Нейронные сети, предназначенные для решения практических задач, могут содержать до нескольких тысяч настраиваемых параметров, поэтому вычисление градиента может потребовать весьма больших затрат вычислительных ресурсов. С учетом специфики многослойных нейронных сетей для них разработаны специальные методы расчета градиента, среди которых следует выделить метод обратного распространения ошибки.

*Метод обратного распространения ошибки*

Термин "обратное распространение" относится к процессу, с помощью которого могут быть вычислены **производные функционала ошибки по параметрам сети**. Этот процесс может использоваться в сочетании с различными стратегиями оптимизации. Существует много вариантов и самого алгоритма обратного распространения. Обратимся к одному из них.

Рассмотрим выражение для градиента критерия качества по весовым коэффициентам для выходного слоя $M$.

$$\frac{\partial J}{\partial w_{ij}^M} = \frac{\partial}{\partial w_{ij}^M}\left(\frac{1}{2}\sum_{q=1}^{Q}\sum_{k=1}^{S^M}\left(t_k^q - a_k^{qS^M}\right)^2\right) = -\sum_{q=1}^{Q}\sum_{k=1}^{S^M}\left(t_k^q - a_k^{qM}\right)\frac{\partial a_k^{qM}}{\partial w_{ij}^M},\ i = 1,\ldots,S^M,\ j = 0,\ldots,S^{M-1}, \quad (3.9)$$

где $S^M$ - число нейронов в слое; $a_k^{qM}$ - $k$-й элемент вектора выхода слоя $M$ для элемента выборки с номером $q$.

Правило функционирования слоя $M$:

$$a_k^{qM} = f_M\left(\sum_{l=0}^{S^{M-1}} w_{kl}^M a_l^{q(M-1)}\right),\ m = 1,\ldots,S^M. \quad (3.10)$$

Из уравнения (3.8) следует

$$\frac{\partial a_k^{qM}}{\partial w_{ij}^M} = \begin{cases} 0, & k \neq i \\ f'(n_i^{qM}) a_j^{q(M-1)}, & k = i, \ i = 1,\ldots,S^{M-1}, \ j = 0,\ldots,S^{M-1}. \end{cases} \quad (3.11)$$

После подстановки (3.11) в (3.9) имеем:

$$\frac{\partial J}{\partial w_{ij}^M} = -\sum_{q=1}^{Q} (t_i^q - a_i^{qM}) f_k'(n_i^{qM}) a_i^{q(M-1)} .$$

Если обозначить

$$\Delta_i^{qM} = (t_i^{qM} - a_i^{qM}) f_M'(n_i^{qM}, \ i = 1,\ldots,S^M), \quad (3.12)$$

то получим

$$\frac{\partial J}{\partial w_{ij}^M} = -\sum \Delta_i^{qM} a_i^{q(M-1)}; \ \ i = 1,\ldots,S^M, \ \ j = 1,\ldots,S^{M-1} \quad (3.13)$$

Перейдем к выводу соотношений для настройки весов $w_{ij}^{M-1}$ слоя $M-1$

$$\begin{aligned} \frac{\partial J}{\partial w_{ij}^{M-1}} &= -\sum_{q=1}^{Q}\sum_{k=1}^{S^M} (t_k^q - a_k^{qM}) f_M'(n_k^{qM}) \frac{\partial n_k^{qM}}{\partial a_i^{q(M-1)}} \frac{\partial a_k^{q(M-1)}}{\partial w_i^{M-1}} a_i^{q(M-1)} = \\ &= -\sum_{q=1}^{Q}\sum_{k=1}^{S^M} (t_k^q - a_k^{qM}) f_M'(n_k^{qM}) w_k^M \frac{\partial n_k^{qM}}{\partial a_i^{q(M-1)}} f_{M-1}'(n_i^{q(M-1)}) a_j^{q(M-1)} = \\ &= -\sum \Delta_i^{q(M-1)} a_j^{q(M-2)}, \end{aligned} \quad (3.14)$$

где

$$\Delta_i^{q(M-1)} = \sum_{k=1}^{S^M} (t_k^q - a_k^{qM}) f_M'(n_k^{qM}) w_k^M \ f_{M-1}'(n_i^{q(M-1)}) = \left( \sum_{k=1}^{S^M} \Delta_k^{qM} \right) f_{M-1}'(n_i^{q(M-1)}), \ \ i = 1,\ldots,S^{M-1}.$$

Для слоев $M-2, M-3,\ldots,1$ вычисление частных производных критерия $J$ по элементам матриц весовых коэффициентов выполняется аналогично. В итоге получаем следующую общую формулу:

$$\frac{\partial J}{\partial w_{ij}^r} = -\sum_{q=1}^{Q} \Delta_i^{q(r-1)} a_j^{q(r-1)}, \ \ r = 1,\ldots,M, \ \ i = 1,\ldots,S^r, \ \ j = 0,\ldots,S^{r-1}, \quad (3.15)$$

где $r$ - номер слоя

$$\Delta_i^{qr} = \left( \sum_{k=1}^{S^{r+1}} \Delta_k^{q(r+1)} w_{ki}^{r+1} \right) f_r'(n_i^{qr}), \quad r = 1, \ldots, M-1,$$

$$\Delta_i^{qM} = \left( t_i^q - a_i^{qM} \right) f_M'(n_i^{qM}), \; i = 1, \ldots, S^M .$$

На рис. 3.3 представлена схема вычислений, соответствующая выражению (3.15).

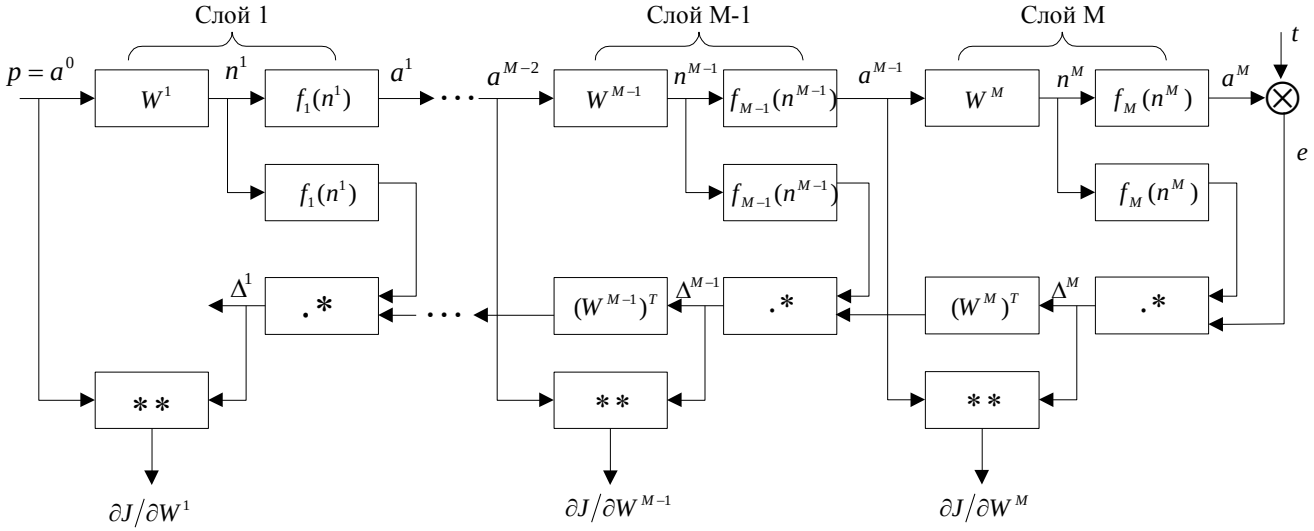


Рис. 3.3 Схема обратного распространения ошибки

На этой схеме символом * обозначена операция поэлементного умножения векторов, а символом ** - умножение вектора $\Delta$ на $a^T$; символ, обозначающий номер элемента выборки, для краткости опущен.

*Общая характеристика методов обучения*

Методы, используемые при обучении нейронных сетей, во многом аналогичны методам определения экстремума (численной оптимизации) функции нескольких переменных. В свою очередь, последние делятся на 3 категории - методы нулевого, первого и второго порядка.

Все алгоритмы численной оптимизации действуют итеративно, на каждом шаге перенося точку внимания $x_k$ по направлению к глобальному минимуму. Общая схема описывается следующей процедурой:

$$\vec{x}_k = \vec{x}_{k-1} + \alpha_k \vec{s}_k$$

В рамках выполнения этой процедуры, необходимо решить три задачи: направление спуска $\vec{s}_k$, скорость спуска $\alpha_k$ и условие останова $k_{\max}$.

Проблема останова для нейронных сетей неактуальна, т.к. заведомо известно, что минимум ошибки равен нулю (или наперёд заданному порогу точности $\varepsilon \geq 0$).

За скорость спуска отвечает настраиваемый в большинстве алгоритмов параметр скорости обучения (*learning rate, lr*), или же некоторая стратегия его последовательного уменьшения, обычно опирающаяся на длину выборки.

Наконец, для выбора направления спуска используется различная информация о локальном характере функции. Именно исходя из обширности этой информации, выделяют три порядка алгоритмов оптимизации:

- точечные или нулевого порядка (используется только информация о текущем значении функции);
- градиентные или первого порядка (вычисляется градиент (вектор частных производных) функции в данной точке);
- квазиньютоновы или второго порядка (вычисляется матрица Гессе (матрица вторых частных производных) или её приближения).

Остановимся на некоторых методах подробнее.

*Методы оптимизации*

В методах *нулевого порядка* для нахождения экстремума используется только информация о значениях функции в заданных точках. К таковым, относятся, например, метод покоординатного спуска, метод золотого сечения, метод чисел Фибоначчи. На практике такие методы практически не используются, в нейронных сетях не применимы.

В методах *первого порядка* используется *градиент функционала ошибки* по каждому из настраиваемых параметров.

$$x_{k+1} = x_k - \alpha_k g_k, \tag{3.16}$$

где $x_k$ - вектор параметров, $\alpha_k$ - параметр скорости обучения, $g_k$ - градиент функционала ошибки $\nabla f(x_k)$, соответствующие итерации с номером $k$.

Процедура 3.16 описывает итеративный алгоритм, определяющий вектор экстремума *x*. На каждом шаге предполагаемым экстремум сдвигается в направлении, противоположном градиенту (антиградиент) со скоростью $a$, то есть по направлению кратчайшего спуска по поверхности функционала ошибки (пример поверхности ошибки приведён на рисунках 2.4, 2.5, 3.2). Если реализуется движение в этом направлении, то ошибка будет уменьшаться. Последовательность таких шагов в конце концов приведет к значениям настраиваемых параметров, обеспечивающим минимум функционала.

В зависимости от принятого алгоритма параметр скорости обучения может быть постоянным или переменным. Правильный выбор этого параметра зависит от конкретной задачи и обычно осуществляется опытным путем; в случае переменного параметра его значение уменьшается по мере приближения к минимуму функционала.

Именно процедура 3.16 обобщает рассмотренные ранее алгоритм Уидроу-Хоффа (для случая одной переменной) и дельта-правило обучения персептрона (для случая дискретной оптимизации).

В алгоритмах *сопряженного градиента* поиск минимума выполняется вдоль *сопряженных* направлений, что обеспечивает обычно более быструю сходимость, чем при наискорейшем спуске. Все алгоритмы сопряженных градиентов на первой итерации начинают движение в направлении антиградиента

$$p_0 = -g_0 \tag{3.17}$$

Тогда направление следующего движения определяется так, чтобы оно было сопряжено с предыдущим. Соответствующее выражение для нового направления движения является комбинацией (взвешенной суммой) нового направления наискорейшего спуска и предыдущего направления:

$$p_k = -g_k + \beta_k p_{k-1}. \tag{3.18}$$

Здесь $p_k$ - направление движения, $g_k$ - градиент функционала ошибки, $\beta_k$ - коэффициент сопряжения на итерации с номером $k$. Коэффициент $\beta$ играет очень

важную роль, т.к. аккумулирует в себе информацию о предыдущих направлениях спуска. Существует ряд методик для его вычисления. Для квадратичной функции оптимальным является

$$\beta_k = \frac{\|g_k\|^2}{\|g_{k-1}\|^2}.$$

Для более высоких размерностей может потребовать вычисление обратной матрицы Гессе, что делает алгоритм промежуточным между методами первого и второго порядка.

Когда направление спуска определено, то новое значение вектора настраиваемых параметров $x_{k+1}$ вычисляется по формуле

$$x_{k+1} = x_k - \alpha p_k \tag{3.19}$$

*Методы второго порядка* требуют знания вторых производных функционала ошибки. К методам второго порядка относится метод Ньютона. Основной шаг метода Ньютона определяется по формуле

$$x_{k+1} = x_k - H_k^{-1} g_k, \tag{3.20}$$

где $x_k$ - вектор значений параметров на $k$-я итерации; $H$ - матрица вторых частных производных целевой функции, или матрица Гессе; $g_k$ - вектор градиента на $k$-й итерации.

Во многих случаях метод Ньютона сходится быстрее, чем методы сопряженного градиента, но требует больших затрат из-за вычисления гессиана, а также может быть численно неустойчив из-за вычисления обратной матрицы. Для того чтобы избежать вычисления матрицы Гессе, предлагаются различные способы ее замены приближенными выражениями, что порождает так называемые *квазиньютоновы алгоритмы* (алгоритм метода секущих плоскостей OSS, алгоритм LM Левенберга – Марквардта и т.п.).

### 3.1.4 Алгоритмы обучения

Алгоритмы обучения, как правило, функционируют пошагово; и эти шаги принято называть *эпохами* или *циклами* (не путать с итерациями). На каждом цикле на вход сети последовательно подаются все элементы обучающей последовательности, затем вычисляются выходные значения сети, сравниваются с целевыми и вычисляется функционал ошибки. Значения функционала, а также его градиента используются для корректировки весов и смещений, после чего все действия повторяются. Процесс обучения прекращается, когда выполнено определенное количество циклов либо когда ошибка достигнет некоторого малого значения или перестанет уменьшаться.

При такой формализации задачи обучения предполагаются известными желаемые (целевые) реакции сети на входные сигналы, что ассоциируется с присутствием учителя, а поэтому такой процесс обучения называют *обучением с учителем.* Для некоторых типов нейронных сетей задание целевого сигнала не требуется, и в этом случае процесс обучения называют *обучением без учителя.*

Ниже для обозначения алгоритмов используются их англоязычные сокращения, ассоциирующиеся с названиями алгоритмов в Neural Network Toolbox и, соответственно, в библиотеке Neurolab.

**Градиентные алгоритмы обучения**

*Алгоритм GD*

Алгоритм GD, или алгоритм градиентного спуска, используется для такой корректировки весов и смещений, чтобы минимизировать функционал ошибки, т.е. обеспечить движение по поверхности функционала в направлении, противоположном градиенту функционала по настраиваемым параметрам.

Рассмотрим двухслойную нейронную сеть прямой передачи информации с сигмоидальным и линейным слоями для обучения ее на основе метода обратного распространения ошибки (рис. 3.3):

```
import neurolab as nl
net = nl.net.newff([ [-1,2],[0,5] ], [3,1], transf=[nl.trans.TanSig(),
nl.trans.PureLin()])
```

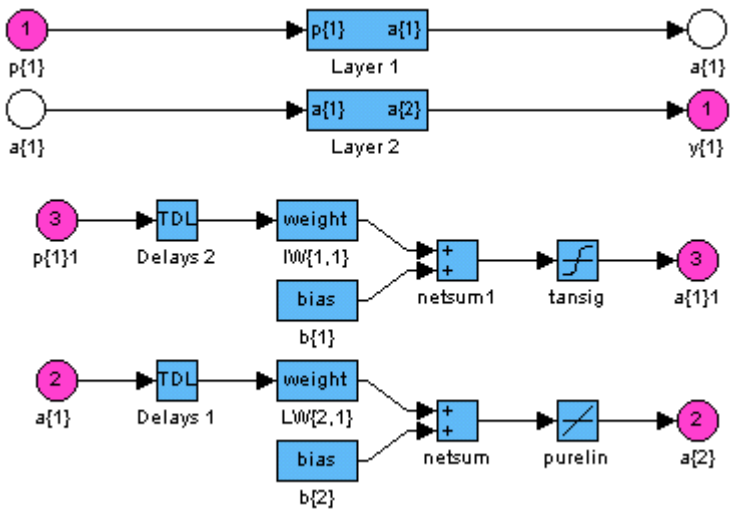


Рис. 3.3 — Схема нейронной сети

Назначим алгоритм обучения:

```
net.trainf = nl.train.train_gd
```

Зададим обучающие множества. Для этого простого примера множество входов и целей определим следующим образом:

```
input = array([[-1, -1, 2, 2],[0, 5, 0, 5]])
target = array([-1, -1, 1, 1])
```

Поскольку используется последовательный способ обучения, необходимо транспонировать вектора:

```
input = input.reshape(4,2)
target = target.reshape(4,1)
```

Теперь выполним обучение сети с помощью функции train.

Функция train_gd характеризуется следующими параметрами:

```
>>> nl.train.train_gd?
epochs:    int (default 500)
                Number of train epochs
show:      int (default 100)
                Print period
```

```
goal:     float (default 0.01)
               The goal of train
lr:       float (defaults 0.01)
               learning rate
```

Функция возвращает величину ошибки:

```
error = net.train(input, target, epochs=500, show=15, goal=0.01)
```

Используя аргументы функции, мы установили параметры настройки: входной и обучающий сигнал, максимум эпох обучения, периодичность вывода результатов и целевую величину ошибки, при достижении которой обучение можно завершать досрочно.

После выполнения этой функции, мы получим настроенную сеть net и значения ошибки на каждой эпохе в переменной error.

Кроме того, с алгоритмами градиентного спуска связан параметр скорости настройки **lr**. Текущие приращения весов и смещений сети определяются умножением этого параметра на вектор градиента. Чем больше значение параметра, тем больше приращение на текущей итерации. Если параметр скорости настройки выбран слишком большим, алгоритм может стать неустойчивым; если параметр слишком мал, то алгоритм может потребовать длительного счета. По умолчанию lr = 0.01. Поэксперементируйте с ним.

Чтобы проверить качество обучения, после окончания обучения смоделируем сеть:

```
out = net.sim(input)
```

Хотя при таких размерах векторов можно оценить результат непосредственно взглянув на значения переменных out и target, для наглядности всё же выведем на экран график ошибки и графики целей и результатов обучения:

```
subplot(211)
plot(error)
xlabel('Epoch number')
ylabel('Error')

subplot(212)
plot(target, '-', out, '--o')
legend(['train target', 'net output'])
show()
```

На рис. 1.3 приведен график изменения ошибки в зависимости от числа выполненных циклов обучения, а так же наглядное сравнение исходной и аппроксимированной функции.

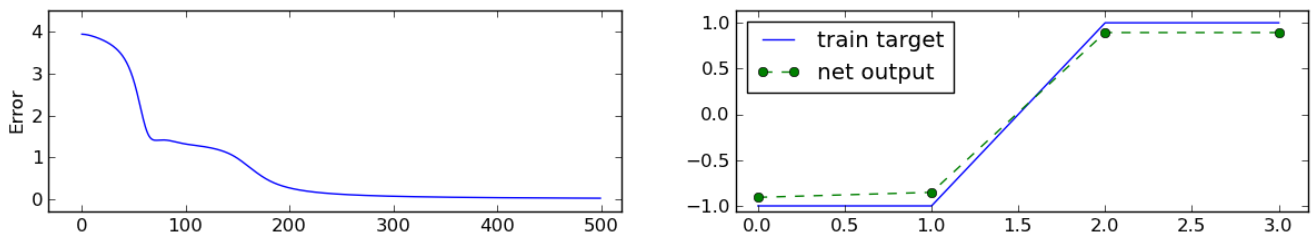


Рис. 3.4

*Алгоритм GDM*

Алгоритм GDM, или алгоритм градиентного спуска с возмущением, предназначен для настройки и обучения сетей прямой передачи. Этот алгоритм позволяет преодолевать локальные неровности поверхности ошибки и не останавливаться в локальных минимумах. С учетом возмущения метод обратного распространения ошибки реализует следующее соотношение для приращения вектора настраиваемых параметров:

$$\Delta w_k = mc\Delta w_{k-1} + (1-mc)\ lr\ g_k, \tag{1.21}$$

где $\Delta w_k$ - приращение вектора весов; $mc$ - параметр возмущения; $lr$ - параметр скорости обучения; $g_k$ - вектор градиента функционала ошибки на $k$-й итерации.

Если параметр возмущения равен 0, то изменение вектора настраиваемых параметров определяется только градиентом, если параметр равен 1, то текущее приращение равно предшествующему как по величине, так и по направлению. В более продвинутой версии этого алгоритма, train_gdx возможно вручную устанавливать параметр mc, задающий величину случайного возмущения.

Вновь рассмотрим двухслойную нейронную сеть прямой передачи сигнала с сигмоидальным и линейным слоями (см. рис. 1.2)

```
    import neurolab as nl
net = nl.net.newff([ [-1,2],[0,5] ], [3,1], transf=[nl.trans.TanSig(),
nl.trans.PureLin()])


net.trainf = nl.train.train_gdm

input = array([[-1, -1, 2, 2],[ 0, 5,  0, 5]])
target = array([-1, -1, 1, 1])

input = input.reshape(4,2)
target = target.reshape(4,1)

error = net.train(input, target, epochs=300, show=15, goal= 0.0001, lr
=  0.2)
```

На рис. 1.4 приведен график изменения ошибки обучения в зависимости от числа выполненных циклов.

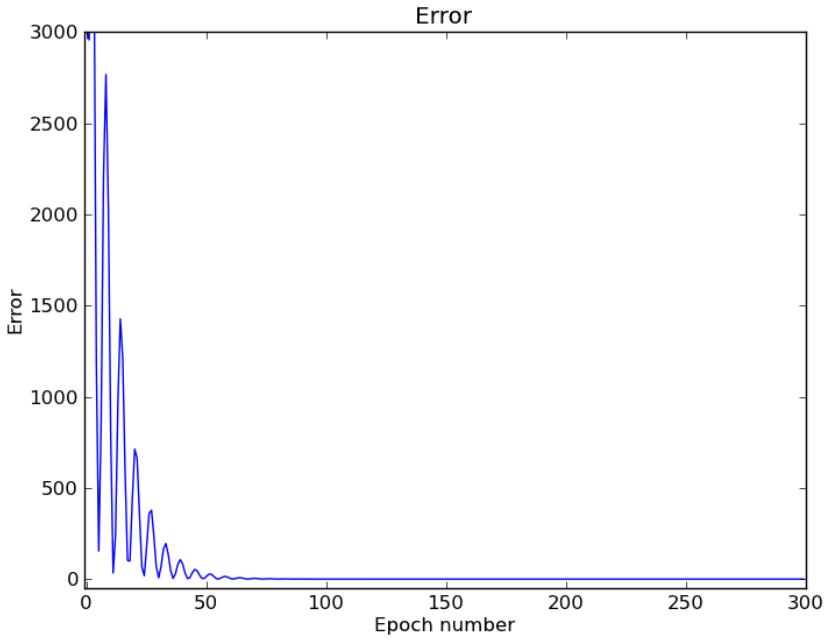


Рис. 3.5

```
>>> out = net.sim(input)
>>> out.transpose()
array([[-1.3334608 ,  0.84995557,  0.85000169,  0.85000169]])
```

Поскольку начальные веса и смещения инициализируются случайным образом, графики ошибок на рис. 1.3 и 1.4 будут отличаться от одной реализации к другой.

Практика применения описанных выше алгоритмов градиентного спуска показывает, что эти алгоритмы слишком медленны для решения реальных задач. Ниже обсуждаются алгоритмы группового обучения, которые сходятся в десятки и сотни раз быстрее. Ниже представлены 2 разновидности таких алгоритмов: один основан на стратегии выбора параметра скорости настройки и реализован в виде алгоритма GDA, другой - на стратегии выбора шага с помощью порогового алгоритма обратного распространения ошибки и реализован в виде алгоритма Rprop.

*Алгоритм GDA*

Алгоритм GDA, или алгоритм градиентного спуска с выбором параметра скорости настройки, использует эвристическую стратегию изменения этого параметра в процессе обучения.

Эта стратегия заключается в следующем. Вычисляются выход и погрешность инициализированной нейронной сети. Затем на каждом цикле обучения вычисляются новые значения настраиваемых параметров и новые значения выходов и погрешностей. Если отношение нового значения погрешности к прежнему превышает величину max_perf_inc (по умолчанию 1.04), то новые значения настраиваемых параметров во внимание не принимаются. При этом параметр скорости настройки уменьшается с коэффициентом lr_dec (по умолчанию 0.7). Если новая погрешность меньше прежней, то параметр скорости настройки увеличивается с коэффициентом lr_inc (по умолчанию 1.05).

Эта стратегия способствует увеличению скорости и сокращению длительности обучения.

Алгоритм GDA в сочетании с алгоритмом GD определяет функцию обучения train_gda, а в сочетании с алгоритмом GDM - функцию обучения train_gdx.

Вновь обратимся к той же нейронной сети (см. рис. 3.3), но будем использовать функцию обучения train_gda:

net.trainf = nl.train.train_gda

```
>>> out = net.sim(input)
>>> out.transpose()
array([[-0.9925826 , -1.00074703,  1.00106152,  1.00106152]])
```

Сравните этот и предыдущий алгоритмы по характеристикам устойчивости, скорости обучения, величине ошибки.

*Алгоритм Rprop*

Алгоритм Rprop, или пороговый алгоритм обратного распространения ошибки, реализует следующую эвристическую стратегию изменения шага приращения параметров для многослойных нейронных сетей.

Многослойные сети обычно используют сигмоидальные функции активации в скрытых слоях. Эти функции относятся к классу функций со сжимающим отображением, поскольку они отображают бесконечный диапазон значений аргумента в конечный диапазон значений функции. Сигмоидальные функции характеризуются тем, что их наклон приближается к нулю, когда значения входа нейрона существенно возрастают. Следствием этого является то, что при использовании метода наискорейшего спуска величина градиента становится малой и приводит к малым изменениям настраиваемых параметров, даже если они далеки от оптимальных значений.

Цель порогового алгоритма обратного распространения ошибки Rprop (Resilient propagation) состоит в том, чтобы повысить чувствительность метода при больших значениях входа функции активации. В этом случае вместо значений самих производных используется только их знак.

Значение приращения для каждого настраиваемого параметра увеличивается с коэффициентом rate_inc (по умолчанию 1.2) всякий раз, когда производная функционала ошибки по данному параметру сохраняет знак для двух последовательных итераций. Значение приращения уменьшается с коэффициентом rate_dec (по умолчанию 0.5) всякий раз, когда производная функционала ошибки по данному параметру изменяет знак по сравнению с предыдущей итерацией. Если производная равна 0, то приращение остается неизменным. Поскольку по умолчанию коэффициент увеличения приращения составляет 20 %, а коэффициент уменьшения - 50 %, то в случае попеременного увеличения и уменьшения общая тенденция будет направлена на уменьшение шага изменения параметра. Если параметр от итерации к итерации изменяется в одном направлении, то шаг изменения будет постоянно возрастать.

Алгоритм Rprop определяет функцию обучения train_rprop. Вновь обратимся к сети, показанной на рис. 3.3, но будем использовать функцию обучения train_rprop (параметры по умолчанию).

```
>>> out = net.sim(input)
>>> print out.transpose()
[[-1.06424878, -1.01712411,  1.08247963,  1.08247963]]
```

Нетрудно заметить, что количество циклов обучения по сравнению с алгоритмом GDA сократилось практически 10 раз, а ошибка уменьшилась.

**Алгоритмы метода сопряженных градиентов**

Основной алгоритм обратного распространения ошибки корректирует настраиваемые параметры в направлении наискорейшего уменьшения функционала ошибки. Но такое направление далеко не всегда является самым благоприятным направлением, чтобы за возможно малое число шагов обеспечить сходимость к минимуму функционала. Существуют направления движения, двигаясь по которым можно определить искомый минимум гораздо быстрее. В частности, это могут быть так называемые сопряженные направления, а соответствующий метод оптимизации - это метод сопряженных градиентов.

Если в обучающих алгоритмах градиентного спуска, управление сходимостью осуществляется с помощью параметра скорости настройки, то в алгоритмах метода сопряженных градиентов размер шага корректируется на каждой итерации. Для определения размера шага вдоль сопряженного направления выполняются специальные одномерные процедуры поиска минимума.

Все алгоритмы метода сопряженных градиентов на первой итерации начинают поиск в направлении антиградиента

$$p_0 = -g_0. \tag{1.22}$$

Когда выбрано направление, требуется определить оптимальное расстояние (шаг поиска), на величину которого следует изменить настраиваемые параметры:

$$x_{k+1} = x_k + \alpha_k p_k. \tag{1.23}$$

Затем определяется следующее направление поиска как линейная комбинация нового направления наискорейшего спуска и вектора движения в сопряженном направлении:

$$p_k = -g_k + \beta_k p_{k-1}. \tag{1.24}$$

Различные алгоритмы метода сопряженного градиента различаются способом вычисления константы $p_k$.

В Neurolab включен только один обобщённый алгоритм метода сопряжённых градиентов, заданный функцией train_cg. Более широко этот класс алгоритмов представлен в библиотеке Monte.

Результат обучения с помощью nl.train.train_cg:

```
Epoch: 30; Error:  1.2762430366;
Epoch: 40; Error:  0.0032958001424157591;
The goal of learning is reached
>>> print out.transpose()
[[-0.92487892 -0.99248247  0.97888236  0.97888236]]
```

**Квазиньютоновы алгоритмы**

*Алгоритм BFGS*

Альтернативой методу сопряженных градиентов для ускоренного обучения нейронных сетей служит метод Ньютона. Основной шаг этого метода определяется соотношением

$$x_{k+1} = x_k - H_k^{-1} g_k, \tag{1.28}$$

где $x_k$ - вектор настраиваемых параметров; $H_k$ - матрица Гессе вторых частных производных функционала ошибки по настраиваемым параметрам; $g_k$ - вектор градиента функционала ошибки. Процедуры минимизации на основе метода Ньютона, как правило, сходятся быстрее, чем те же процедуры на основе метода сопряженных градиентов. Однако вычисление матрицы Гессе - это весьма сложная и дорогостоящая в

вычислительном отношении процедура. Поэтому разработан класс алгоритмов, которые основаны на методе Ньютона, но не требуют вычисления вторых производных. Это класс *квазиньютоновых алгоритмов,* которые используют на каждой итерации некоторую приближенную оценку матрицы Гессе.

Одним из наиболее эффективных алгоритмов такого типа является алгоритм BFGS, предложенный Бройденом, Флетчером, Гольдфарбом и Шанно (Broyden, Fletcher, Goldfarb and Shanno). Этот алгоритм реализован в виде функции train_bfgs.

Вновь обратимся к сети, показанной на рис. 1.2, но будем использовать функцию обучения train_bfgs. У этой функции нет настраиваемых параметров эффективности обучения. Результат работы с параметрами по умолчанию:

```
Epoch: 15; Error:  0.00349428821406;
The goal of learning is reached
>>> print out.transpose()
[[-1.02503873, -1.07713791,  0.98565809,  0.98565809]]
```

Алгоритм BFGS требует большего количества вычислений на каждой итерации и большего объема памяти, чем алгоритмы метода градиентного спуска и сопряженных градиентов, хотя, как правило, он сходится на меньшем числе итераций. Требуется на каждой итерации хранить оценку матрицы Гессе, размер которой определяется числом настраиваемых параметров сети. Поэтому для обучения нейронных сетей больших размеров лучше использовать алгоритм Rprop или какой-либо другой алгоритм метода сопряженных градиентов. Однако для нейронных сетей небольших размеров алгоритм BFGS может оказаться эффективным.

*Алгоритм NCG*

Алгоритм OSS (One Step Secant), или одношаговый алгоритм метода секущих плоскостей, описан в работе Баттити (Battiti). В нем сделана попытка объединить идеи метода сопряженных градиентов и схемы Ньютона, поэтому в библиотек Neurolab он назван Newton Conjugate Gradient Method. Алгоритм не запоминает матрицу Гессе, полагая ее на каждой итерации равной единичной. Это позволяет определять новое направление поиска не вычисляя обратную матрицу.

Алгоритм представлен функцией train_ncg. Новых настраиваемых параметров не имеет. При настройках, аналогичных предыдущим, показывает следующие результаты:

```
Epoch: 30; Error:  0.454290655909;
Epoch: 34; Error:  0.00087973350377693786;
The goal of learning is reached
>>> print out.transpose()
[[-0.99736395, -0.95861787,  0.99552575,  0.99552575]]
```

Этот алгоритм требует меньших объемов памяти и вычислительных ресурсов на цикл по сравнению с алгоритмом BFGS, но больше, чем базовый алгоритм CG. Таким образом, алгоритм OSS может рассматриваться как некий компромисс между алгоритмами методов сопряженных градиентов и Ньютона.

### 3.2 Порядок выполнения работы

1. Повторить пройденный материал по численным методам оптимизации, алгоритмам обучения и нейронным сетям с прямой передачей информации.
2. Провести моделирование предложенных примеров.

3. Создать нейронную сеть с прямой передачей информации для аппроксимации функции, предложенной в таблице 3.1.

Таблица 3.1 – Варианты задания

| № | Сигнал |
|---|---|
| 1. | $y = \sin(3x) + \cos(5x);$ |
| 2. | $y = \cos(5x) - \sin(4x) + 0.5\cos(87x);$ |
| 3. | $y = \sin(x) + 0.25\sin(12x);$ |
| 4. | $y = \cos(6x) - \sin(5x) - \arctan(0.25x);$ |
| 5. | $y = \sin(0.5x) + \cos(4x);$ |
| 6. | $y = \cos(2x) - \sin(7x) - \tanh(2x);$ |
| 7. | $y = x\cos(2x) + \sin(5x);$ |
| 8. | $y = \cos(4x) - \sin(7x) + \arctan(2x)$ |
| 9. | $y = \cos(x^2) + \sin(2x);$ |
| 10. | $y = \cos(5x) - \sin(2x) + 0.35\sin(98x);$ |
| 11. | $y = \sin(0.5x) - 0.72\sin(4x) + 0.25\sin(35x);$ |
| 12. | $y = \sin(x\cos(2x))\,;$ |
| 13. | $y = \cos(2x) + \sin(8x) - 0.23\cos(3x);$ |
| 14. | $y = 0.75\sin(5x) - \cos(9x);$ |
| 15. | $y = \cos(2x) + \sin(10x) - 0.65\sin(45x);$ |

4. Провести обучение созданной нейронной сети несколькими алгоритмами обучения (не менее 3 из разных подвидов алгоритмов), изучить влияние настраиваемых параметров алгоритмов на эффективность обучения;
5. Провести сравнение влияния на эффективность параметров нейронной сети (количество нейронов, слоёв, функции активации и пр.)
6. Провести сравнение алгоритмов обучения нейронных сетей.

**Дополнительные задания**

1. Протестировать работу сети, разделив выборку на обучающую и тестовую. Продемонстрировать эффект переобучения и методы его преодоления. Обосновать соотношение размеров обучающей и тестовой выборки.
2. Решить задачу XOR с помощью многослойного персептрона. Построить разделяющую поверхность.
3. Реализовать компрессию с помощью многослойного персептрона.

### 3.3 Содержание отчета

Отчет должен содержать:

1. Название работы;
2. Цель работы;
3. Постановку экспериментов и выводы по каждому из них них
4. Листинг программы моделирования нейронной сети;
5. Полученные результаты в виде числовых данных и графиков;
6. Расширенные выводы по лабораторной работе.

### 3.4 Контрольные вопросы

1. Классы задач, решаемые нейронными сетями?
2. Что такое активационная функция? Типы активационных функций.

3. Топологии искусственных нейронных сетей.
4. Типы алгоритмов обучения.
5. Градиентные алгоритмы обучения.
6. Алгоритмы метода сопряженных градиентов.
7. Квазиньютоновы алгоритмы.