§ 2.4. Метод градиентного спуска

В примере 2.2 нам повезло: дизъюнкция оказалась функцией, положительные значения которой можно линейно отделить от отрицательных. Но что делать, если нужно реализовать функцию, для которой это невозможно? Если нельзя увеличить сложность нейронной сети (а мы пока останемся в рамках одного перцептрона), то точного попадания в функцию не достичь.

Нужно найти перцептрон, который бы в каком-то смысле *минимизировал* ошибку. Сначала немного обобщим стоящую перед нами задачу: откажемся от идеи лимита срабатывания и предположим, что мы просто вычисляем перцептроном функцию

$$o(x_0, \ldots, x_n) = \sum_0^n w_i x_i,$$

а нужно нам как можно лучше приблизить функцию $f(x_0, \ldots, x_n)$, которая на тестовых примерах x^j, $j = 1..m$, задана значениями $f(x_0^j, \ldots, x_n^j) = t_j$.

В качестве меры ошибки мы возьмём среднеквадратичное отклонение от целевых значений:

$$E(w_0, \ldots, w_n) = \frac{1}{2} \sum_{j=1}^m (t_j - o(x_0^j, \ldots, x_n^j))^2.$$

Эта мера широко используется в статистике; она как нельзя лучше подходит и для нашей задачи (разговор о том, *почему*, выходит за рамки этой лекции).

Цель — минимизировать функцию E на пространстве возможных весов $\{w_i\}$. График E представляет собой параболическую поверхность, и у неё должен быть один-единственный минимум. А вопрос о том, как исправлять веса так, чтобы двигаться в сторону этого минимума, давным-давно решён в математическом анализе. Для этого нужно двигаться в направлении, противоположном *градиенту* — вектору, вдоль которого производная максимальна. Градиент вычисляется следующим образом:

$$\nabla E(w_0, \ldots, w_n) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right].$$

Таким образом, чтобы определить, как подправлять веса, мы должны вычислить градиент и отнять вектор какой-нибудь наперёд заданной длины (да, это та самая η) от имеющегося вектора весов:

$$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i}.$$

Чтобы реализовать это программно, нужно научиться дифференцировать функцию E; к счастью, это совсем несложно:

GradientDescent$(\eta, \{x_i^j, t^j\}_{i=1,j=1}^{n,m})$

1. Инициализировать $\{w_i\}_{i=0}^n$ маленькими случайными значениями.
2. Повторить NUMBER_OF_STEPS раз:
   а) Для всех i от 1 до n $\Delta w_i = 0$.
   б) Для всех j от 1 до m:
      (i) Для всех i от 1 до n
      $$\Delta w_i = \Delta w_i + \eta \left(t^j - \sum_0^n w_i x_i^j \right) x_i^j.$$
   в) $w_i = w_i + \Delta w_i$.
3. Выдать значения $w_0, w_1, \ldots, w_n$.

Рис. 2.4. Алгоритм градиентного спуска для одного перцептрона.

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{j=1}^m \frac{\partial}{\partial w_i} \left(t^j - \sum_0^n w_i x_i^j \right)^2 = \sum_{j=1}^m \left(t^j - \sum_0^n w_i x_i^j \right) (-x_i^j).$$

Это значит, что нам нужно подправлять веса после каждого тестового примера так:

$$w_i \leftarrow w_i + \eta \sum_j \left(t^j - \sum_0^n w_i x_i^j \right) x_i^j.$$

Новый, изменённый алгоритм показан на рис. 2.4. Правда, нужно внести и другие изменения. Во-первых, мы больше не можем рассчитывать на то, что в какой-то момент достигнем идеальной гармонии с исходными данными, и нам нужно научиться останавливаться в какой-то момент. В качестве условия для остановки здесь принято банальное повторение алгоритма достаточное количество раз. Другое изменение — в том, что если оставлять η постоянным, то на каком-то этапе вектор весов перестанет приближаться к искомому минимуму, а начнёт его «перепрыгивать» на каждой итерации, то в одну сторону, то в другую. Поэтому η нужно уменьшать со временем. Записывать это в алгоритм — излишнее захламление, и мы реализуем эту особенность в тексте собственно программы (строки 8–9), а в алгоритме для ясности не будем.

Листинг 2.2. Обучение перцептрона методом градиентного спуска на языке Python.

```
1   def PerceptronGradientDescent(eta0,x,NUMBER_OF_STEPS):
        import random
        eta,w,deltaw=eta0,[],[]
        for i in xrange(len(x[0])):
```

```
        w.append((random.randrange(-5,5))/50.0)
        deltaw.append(0)
    for i in xrange(NUMBER_OF_STEPS):
        if ((NUMBER_OF_STEPS > 100) and (i % (NUMBER_OF_STEPS/5))==0):
            eta = eta/2.0
        for i in xrange(len(w)): deltaw[i]=0
        for xj in x:
            t,o,curx=xj[0],0,[1]+xj[1:len(xj)]
            for i in xrange(len(w)): o+=w[i]*curx[i]
            for i in xrange(len(w)): deltaw[i]+=eta*(t-o)*curx[i]
        for i in xrange(len(w)): w[i]+=deltaw[i]
    return w
```

Пример 2.3. Обучение перцептрона методом градиентного спуска.

Попытаемся для примера выразить перцептроном линейную форму $x_1 - x_2 + x_3$. Для этого вызовем процедуру из листинга 2.4 так[3]:

```
print PerceptronGradientDescent(0.1,[[1,1,1,1],[1,0,-1,0],[1,1,0,0],
                                [1,0,0,1],[0,0,1,1],[2,1,0,1]],1000)
```

Тогда на выходе мы получим вектор, похожий на

$$[w_0, w_1, w_2, w_3] = [0.00078384114974986135,\\ 0.99962820707427769, -0.99942100811538592, 0.99902715041783252],$$

что уже является очень хорошим приближением к искомой линейной форме.

На рис. 2.5 изображена стохастическая модификация алгоритма градиентного спуска. Этот алгоритм может преодолевать локальные минимумы (у рассматривавшегося нами до сих пор параболоида локальных минимумов быть не может, но в более сложных случаях...). Его отличие в том, что он не собирает всю информацию со всех тестовых примеров, а модифицирует веса сразу же, после каждого примера.

§ 2.5. Нелинейные перцептроны. Сигмоид.

Теперь, когда мы разобрались с обучением одного перцептрона, нужно научиться обучать (простите за тавтологию) целые сети перцептронов. Однако для этого нам придётся снова видоизменить понятие перцептрона. Дело в том, что те линейные

[3] Обращаем внимание читателей на то, что вызывать процедуры в Python нужно обязательно в одну строчку — язык чувствителен к табуляциям и переводам строки. Мы разбиваем текст вызова на строки только для того, чтобы он помещался на странице.

`GradientDescent`$(\eta, \{x_i^j, t^j\}_{i=1,j=1}^{n,m})$

1. Инициализировать $\{w_i\}_{i=0}^{n}$ маленькими случайными значениями.
2. Повторить `NUMBER_OF_STEPS` раз:
   a) Для всех j от 1 до m:
      (i) Для всех i от 1 до n

$$w_i = w_i + \eta \left(t^j - \sum_0^n w_i x_i^j \right) x_i^j.$$

3. Выдать значения $w_0, w_1, \ldots, w_n$.

Рис. 2.5. Алгоритм стохастического градиентного спуска.

перцептроны без лимита активации, для которых у нас уже есть отличный метод градиентного спуска, для сетей не годятся. Суперпозиции линейных функций снова линейны, и сеть любой глубины можно было бы заменить одним-единственным перцептроном. Лучше обстоят дела с перцептронами с лимитом активации: там уже переход от глубины 1 к глубине 2 приносит реализацию всех булевских функций. Но с ними другая проблема: функция, которую они реализуют, лишь кусочно-гладкая, и требующий дифференцируемости метод градиентного спуска не сработает.

Решение, разумеется, простое: нужно «сгладить» линейный результат суммирования с весами w_i, использовать гладкую монотонную, но не линейную функцию в качестве выхода перцептрона. Для этого обычно используют *сигмоид* — функцию вида

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Применять мы её будем после того, как подсчитаем скалярное произведение $\sum_i w_i x_i$. То есть общая формула работы перцептрона такова:

$$o(x_1, \ldots, x_n) = \frac{1}{1 + e^{-\sum_i w_i x_i}}.$$

Сигмоид обладает важным преимуществом: от него легко считать производную. Верна формула: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Поэтому правило изменения веса, хотя и будет посложнее, чем в линейном случае, но всё же не запредельно сложным; в случае одного перцептрона (на одном тестовом примере) оно будет выглядеть так:

$$w_i \leftarrow w_i + \eta o(x)(1 - o(x))(t(x) - o(x))x_i,$$

где $t(x)$ — целевое значение интересующей нас функции. Впрочем, все эти сложности мы вводили не для того, чтобы ограничиваться одним перцептроном.

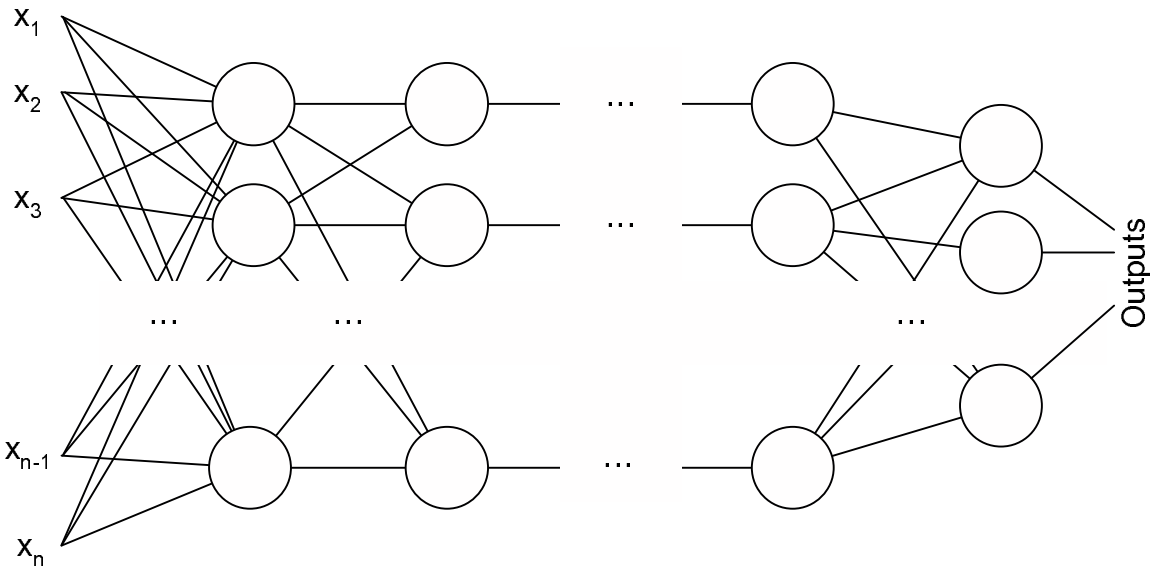

Рис. 2.6. Нейронная сеть.

§ 2.6. Алгоритм обратного распространения ошибки в нейронных сетях

Итак, отныне и впредь у нас не один перцептрон, а целая их сеть, примерно такая, как показано на рис. 2.6. У сети есть входы $x_1, \ldots, x_n$, выходы Outputs и внутренние узлы. Перенумеруем все узлы (включая входы и выходы) числами от 1 до N. Обозначим через w_{ij} вес, стоящий на ребре, соединяющем i-й и j-й узлы, а через o_i — выход i-го узла. Поскольку выходов теперь несколько, функция ошибки станет несколько сложнее. Если у нас, как и прежде, m тестовых примеров с целевыми значениями выходов $\{t_k^d\}_{,d=1..m,k\in Outputs}$, то функция ошибки выглядит так:

$$E(\{w_{ij}\}) = \frac{1}{2}\sum_{d=1}^{m}\sum_{k\in Outputs}\left(t_k^d - o_k(x_1^d, \ldots, x_n^d)\right)^2.$$

Как модифицировать веса? Мы будем реализовывать стохастический градиентный спуск, т.е. будем подправлять веса после каждого тестового примера. Как и раньше, нам нужно двигаться в сторону, противоположную градиенту, то есть добавлять к каждому весу w_{ij}

$$\Delta w_{ij} = -\eta\frac{\partial E^d}{\partial w_{ij}},$$

где

$$E^d(\{w_{ij}\}) = \frac{1}{2}\sum_{k\in Outputs}(t_k^d - o_k^d)^2.$$

Как подсчитать эту производную? Пусть сначала $j \in$ Outputs, то есть интересующий нас вес входит в перцептрон последнего уровня. Сначала отметим, что w_{ij} влияет

на выход перцептрона только как часть суммы $S_j = \sum_i w_{ij}x_{ij}$, где сумма берётся по входам j-го узла. Поэтому

$$\frac{\partial E^d}{\partial w_{ij}} = \frac{\partial E^d}{\partial S_j}\frac{\partial S_j}{\partial w_{ij}} = x_{ij}\frac{\partial E^d}{\partial S_j}.$$

Аналогично, S_j влияет на общую ошибку только в рамках выхода j-го узла o_j (напоминаем, что это выход всей сети). Поэтому

$$\frac{\partial E^d}{\partial S_j} = \frac{\partial E^d}{\partial o_j}\frac{\partial o_j}{\partial S_j} = \left(\frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in Outputs}(t_k - o_k)^2\right)\left(\frac{\partial\sigma(S_j)}{\partial S_j}\right) =$$

$$= \left(\frac{1}{2}\frac{\partial}{\partial o_j}(t_j - o_j)^2\right)(o_j(1-o_j)) = -o_j(1-o_j)(t_j - o_j).$$

Если же j-й узел — не на последнем уровне, то у него есть выходы; обозначим их через $Children(j)$. В этом случае

$$\frac{\partial E^d}{\partial S_j} = \sum_{k\in Children(j)}\frac{\partial E^d}{\partial S_k}\frac{\partial S_k}{\partial S_j},$$

и

$$\frac{\partial S_k}{\partial S_j} = \frac{\partial S_k}{\partial o_j}\frac{\partial o_j}{\partial S_j} = w_{jk}\frac{\partial o_j}{\partial S_j} = w_{jk}o_j(1-o_j).$$

Ну а $\frac{\partial E^d}{\partial S_k}$ — это в точности аналогичная поправка, но вычисленная для узла следующего уровня (будем обозначать её через δ_k — от Δ_k она отличается отсутствием множителя $-\eta x_{ij}$). Поскольку мы научились вычислять поправку для узлов последнего уровня и выражать поправку для узла более низкого уровня через поправки более высокого, можно уже писать алгоритм. Именно из-за этой особенности вычисления поправок алгоритм называется *алгоритмом обратного распространения ошибки* (backpropagation). Краткое резюме проделанной работы:

— для узла последнего уровня

$$\delta_j = -o_j(1-o_j)(t_j - o_j);$$

— для внутреннего узла сети

$$\delta_j = -o_j(1-o_j)\sum_{Outputs(j)}\delta_k w_{jk};$$

— для всех узлов

$$\Delta w_{ij} = -\eta\delta_j x_{ij}.$$

BackPropagation($\eta, \{x_i^d, t^d\}_{i=1,d=1}^{n,m}$, NUMBER_OF_STEPS)

1. Инициализировать $\{w_{ij}\}_{i,j}$ маленькими случайными значениями.
2. Повторить NUMBER_OF_STEPS раз:
   a) Для всех d от 1 до m:
      (i) Подать $\{x_i^d\}$ на вход сети и подсчитать выходы o_i каждого узла.
      (ii) Для всех $k \in$ Outputs
      $$\delta_k = o_k(1 - o_k)(t_k - o_k).$$
      (iii) Для каждого уровня l, начиная с предпоследнего:
         (A) Для каждого узла j уровня l вычислить
         $$\delta_j = o_j(1 - o_j) \sum_{k \in \mathrm{Children}(j)} w_{jk}\delta_k.$$
      (iv) Для каждого ребра сети $\{ij\}$
      $$w_{ij} = w_{ij} + \eta\delta_j x_{ij}.$$
3. Выдать значения w_{ij}.

Рис. 2.7. Алгоритм обратного распространения ошибки.

Получающийся алгоритм представлен на рис. 2.7. На вход алгоритму, кроме указанных на рис. 2.7 параметров, нужно также подавать в каком-нибудь формате структуру сети. На практике очень хорошие результаты показывают сети достаточно простой структуры, состоящие из двух уровней нейронов — скрытого уровня (hidden units) и нейронов–выходов (output units); каждый вход сети соединён со всеми скрытыми нейронами, а результат работы каждого скрытого нейрона подаётся на вход каждому из нейронов–выходов. В таком случае достаточно подавать на вход количество нейронов скрытого уровня.

§ 2.7. Упражнения и задачи

1. Постройте сеть из перцептронов глубины 2, реализующую функцию XOR.
2. Докажите, что сетью из (линейных) перцептронов глубины 2 можно реализовать любую булевскую функцию (теорема 2.1).
3. Реализуйте алгоритм стохастического градиентного спуска (рис. 2.5) на языке Python.
4. Реализуйте алгоритм обратного распространения ошибки (рис. 2.5) на языке Python.