Алгоритм обратного распространения ошибки

Роман Малашин

2 июня 2025 г.

Содержание

1 Введение

Обучение нейронной сети заключается в поиске таких параметров (весов связей), которые обеспечивают минимальную ошибку на обучающей выборке. Эта оптимизация осуществляется с помощью алгоритма градиентного спуска (а точнее его модификаций). В этом случае сеть инициализируется случайными весами, которые итеративно изменяются в направлении градиента функции ошибки. Для применения алгоритма градиентного спуска, очевидно, требуется знать совокупность частных производных всех весов (градиент).

Именно для этого был разработан алгоритм обратного распространения ошибки (АОРО), который позволяет рассчитывать необходимые частные производные функции потерь (функции ошибки) по каждому весу связи w_{ij} и смещению b_i в нейронной сети. АОРО состоит из двух этапов:

1. Прямой проход. На этом этапе нейронная сеть получает входные данные, и генерирует предсказание относительно них.

2. Обратный проход. На этом этапе выход сравнивается с эталонным значением, и считается рассогласование в соответствии с выбранной функцией ошибки. Ошибка верхнего слоя может быть преобразована в производные по весам связи, а также распространена ниже в сеть для получения производных весов более низких слоев. Именно с этим связано название алгоритма.

Именно второй этап и дает название АОРО и является ключевым для его понимания. В данном методическом указании приводится математическое описание алгоритма обратного распространения ошибки, а также способ векторизации вычислений в этом алгоритме, что позволяет ускорить вычисления с использованием современных центральных и графических процессоров.

АОРО является наиболее важными из алгоритмов автоматического дифференцирования и был разработан много лет назад, но при этом долгое время оставался неизвестным для широкого круга исследователей, пока в 1986 [1] году он не был в очередной раз переизобретен[1]. В этом случае удалось с помощью него добиться многообещающих результатов и с тех пор АОРО является основой для обучения наиболее успешных нейронных сетей, а с 2012 года все прорывные решения в области искусственного интеллекта так или иначе используют этот алгоритм.

Современные библиотеки глубокого обучения, такие как Pytorch [2][2], основаны на алгоритмах автоматического дифференцирования, включая и АОРО. При использовании таких библиотек для разработки более-менее стандартных архитектур нейронных сетей и при решении типовых задач достаточно реализовывать только прямой проход нейронной сети - обратный проход делается автоматически причем, как правило, очень эффективно. Достаточно посчитать ошибку на выходе нейронной сети и дать библиотеке указание выполнить обратный проход.

Например, с использованием библиотеки Pytorch фрагмент кода, отвечающий за один шаг градиентного спуска, может выглядеть таким образом:

```
# получение пары <входные данные-эталонная метка> (x, t)
...

# обнуление производных, рассчитанных на прошлом шаге
optimizer.zero_grad()

# прямой проход нейронной сети

y = model(x)
```

[1] С учетом того, что первые нейронные сети были реализованы еще в начале 1960-х, время, которое потребовалось, чтобы алгоритм обучения многослойных сетей стал широко использоваться, может вызывать вопросы, т.к. он основан на правиле взятия производной сложной функции, которое было известно математикам уже в 18 веке. Получается, профессорам потребовалось 40 лет, чтобы вспомнить правило взятия производной сложной функции, которому учат на первом курсе технического вуза.

[2] Аналогичные возможности имеют и другие библиотеки для обучения нейронных сетей

```
# рассчет ошибки
loss = criterion(y,t)

# вызов алгоритма обратного распространения ошибки
loss.backward()

# применение производных для изменения весов нейронной сети
optimizer.step()
```

Во фрагменте кода выше loss - это переменная, в которой хранится значение ошибки, рассчитанной на одном или нескольких парах вход-эталон. Функция backward выполняет обратный проход и считает частные производные для каждого веса в нейронной сети. Значение градиента весов используются в алгоритме оптимизации, который в простейшем случае реализует градиентный спуск.

Ниже будет по сути подробно рассмотрено, как работает функция backward[3]: какая математическая основа используется, и какие шаги выполняются, чтобы каждому свободному параметру модели (model) было сопоставлено значение частной производной[4].

В связи с этим, может возникнуть вопрос: "Зачем надо понимать, как работает алгоритм обратного распространения ошибки, если реализовывать его не потребуется?". Вопрос является резонным. Например, многие программисты могут иметь весьма смутные представления о том, как работают компиляторы и интерпретаторы, но это не мешает им вполне успешно писать исполняемый код с их использованием, поскольку реализация программы на языке высокого уровня не требует знания о том, как именно текст программы преобразуется в исполняемые машинные инструкции. И, действительно, множество людей успешно обучают нейронные сети для решения прикладных задач и не имеют представления, как именно реализуется расчет градиентов, на основании которых происходит обучение. Помимо очевидного замечания, что знание деталей определяют квалификацию специалиста, а также, что знание ядра наиболее успешных алгоритмов искусственного интеллекта является ценным самим по себе, можно структурировать случаи востребованности знаний об АОРО на практике:

1. Подобно случаю с использованием компилятора "в слепую обучение архитектур нейронных сетей без понимания работы АОРО может вестись вполне успешно до тех пор, пока задачи, базы данных, являются стандартными, и в них достаточно просто вносить минимальные изменения "интуитивно". Однако, при возникновении сложностей обучения, когда задача отклоняется от стереотипной, понять причины, например, возникающего расхождения обучения без понимания работы АОРО, зачастую бывает невозможно.

[3]При этом за backward, разумеется, скрывается и множество других не раскрываемых нюансов, обеспечивающих высокую эффективность и гибкость расчета градиентов для разных архитектур нейронных сетей, активационных функций, функций ошибки, которые выходят за рамки этого небольшого материала. Однако базовые принципы сохраняются.

[4]Все вместе частные производные задают градиент - направление в пространстве весов наискорейшего уменьшения значения ошибки

2. При создании новых слоев в нейронных сетях к ним необходимо реализовать и обратный проход. Таким образом, для создания эффективных новых дифференцируемых моделей машинного обучения[5] понимание алгоритма обратного распространения ошибки является обязательным. Почти все новые архитектурные решения так или иначе интерпретируются с точки зрения повышения эффективности обратного прохода. Например, в архитектуре Resnet [3] остаточные связи были предложены, чтобы преодолеть проблему размывающегося к первым слоям градиента. Эффективное (быстрое) обучение с помощью АОРО было обеспечено и в блоке самовнимания [4], который позволил преодолеть многие недостатки рекуррентных нейронных сетей за счет эффективной векторизации и распараллеливания вычислений. Подход к векторизации рассмотрен в разделе 6.

3. Понимание особенностей АОРО полезно для поиска путей преодоления некоторых проблем искусственных интеллектуальных систем, основанных на нейронных сетях. Например, остро стоит проблема катастрофического забывания - при обучении нейронной сети новой задаче, знания о старой стираются. Это ограничивает разработку интеллектуальных агентов, функционирующих в открытых средах. Для преодоления этой проблемы авторы [5] использует наблюдение, что градиент весов слоя представляет собой значения активаций нейронов, взвешенных значением ошибки (формула (43)), что позволяет им разработать алгоритм преобразования градиента на основании активаций нейронов таким образом, чтобы градиент весов при решении задачи t наименьшим образом сказывался на значения активаций необходимых для решения предыдущих задач $t-1, t-2, \cdots$. Предложенное в [5] решение является чрезвычайно эффективным; существует множество вариантов его улучшения.

4. С учетом успеха АОРО разработка альтернативных алгоритмов обучения больших моделей машинного обучения (и нейронных сетей, в частности) всегда ведется, исходя из понимания особенностей АОРО. Так, например, корректировка весов первого слоя в нейронной сети с помощью АОРО должна делаться с учетом ошибок[6] всех слоев, располагающихся выше (формула (26)). Для обучения слоя нейронной сети необходимо знание состояния прочих слоев, что ведет к необходимости синхронизации, усложняет в том числе и распараллеливание вычислений. Например, алгоритм forward-forward недавно предложенный Хинтоном, такого недостатка не имеет [6] [7].

Таким образом, можно резюмировать, что знание АОРО является практически полезным для разработчика нейронных сетей, сетей искусственного интеллекта. С учетом того, что ИИ является сквозной технологией[8], можно уверенно говорить, что в настоящее время знание алгоритма АОРО должно быть полезным для всех технических (и не только) специалистов в области информационных технологий.

[5] К классу таких моделей относятся и традиционные искусственные нейронные сети

[6] В данном случае ошибка - это конкретное число (δ)), вычисляемое с помощью АОРО

[7] Правда, он уступает в эффективности алгоритму обратного распространения ошибки

[8] Например, можно утверждать следующее: "Любая нетривиальная программа может быть потенциально улучшена за счет использования методов машинного обучения".

2 Обозначения

Рассмотрим как работает АОРО на фрагменте нейронной сети, представленной на рисунке 1.

На рисунке для удобства один нейрон представлен в виде двух связанных окружностей обозначающих активность нейрона до и после применения активационной функции.

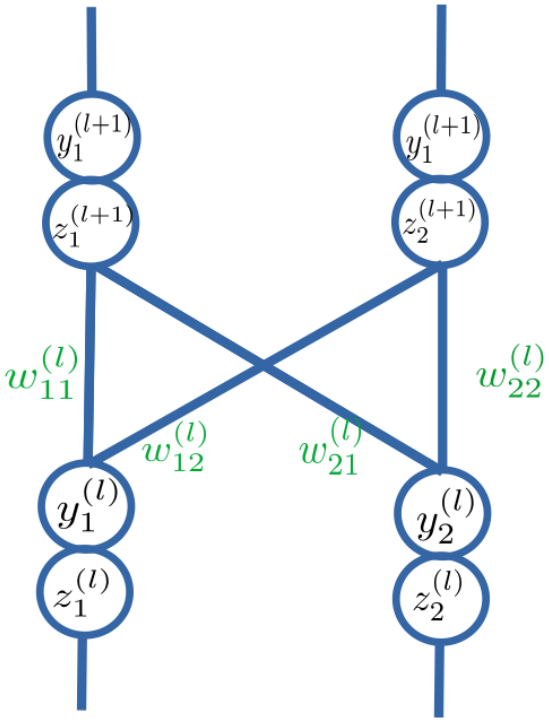

Рис. 1: Фрагмент нейронной сети с двумя нейронами в скрытом слое. На рисунке не обозначены смещения $b_k^{(t)}$.

Далее все математические обозначения будут сделаны в соответствии со следующим соглашением:

- жирным будут выделены векторы и матрицы (например $\mathbf{z}_i^{(l)}$ или $\mathbf{Z}_i^{(l)}$), без выделения будут обозначены скаляры (например, $z_i^{(l)}$, $F_h^{(l)}$ или δ_i);
- матрицы, если это не оговаривается отдельно, будут обозначаться заглавными символами ($\mathbf{Y}^{(l)}$), а вектора - строчными ($\mathbf{y}^{(l)}$);
- в верхнем надстрочном индексе в скобках указывается номер слоя (в обозначениях выше это слой l).

На рисунке 1:

- $y_i^{(l)}$ обозначает активность i -го нейрона слоя l;

- $z_i^{(l)}$ - вход нейрона i (активность нейрона до применения активационной функции);
- $w_{ij}^{(l)}$ - вес связи между нейроном i слоя l и нейроном j слоя $l+1$.

Уравнения, приведенные ниже, могут на первый взгляд показаться громоздкими и сложными для человека, который не привык работать с математическими формулами. Однако громоздкость связана в основном с наличием верхних и нижних индексов из-за необходимости уточнять для каждой переменной номер слоя, а также индекс веса или нейрона. Если с обозначениями разобраться, то предлагаемые ниже записи требуют лишь базовые знания об операциях над матрицами и правил взятия производных. Большое количество формул связано с желанием подробнее рассмотреть каждую необходимую операцию, чтобы упростить понимание материала.

3 Прямой проход

Вход нейрона слоя $l+1$ рассчитывается как взвешенная сумма активностей нейронов предыдущего слоя с учетом смещения:

$$z_i^{(l+1)} = \sum_{j=1}^{N} y_j^{(l)} \cdot w_{ji}^{(l)} + b_i^{(l)}, \tag{1}$$

где $b_i^{(l)}$ - смещение (bias) i -го нейрона слоя l.

На рисунке 1 смещения $b_i^{(l)}$ и $b_i^{(l-1)}$ не обозначены, поскольку они не влияют на расчет производных по весам $w_{ji}^{(l)}$. Расчет производных по смещениям рассмотрен в разделе 5.

Активация нейрона $y_i^{(l+1)}$ рассчитывается с использованием активационной функции f:

$$y_i^{(l+1)} = f(z_i^{(l+1)}). \tag{2}$$

Выбор активационной функции может быть различным в зависимости от использованной архитектуры. Одни из наиболее популярных активационных функций - это ReLU и сигмоида. В случае сигмоидальной функции активации:

$$y_i^{(l)} = f(z_i^{(l)}) = \frac{1}{1 + e^{-z_i^{(l)}}}, \tag{3}$$

а в случае ReLU:

$$y_i^{(l)} = f(z_i^{(l)}) = \max(0, z_i^{(l)}). \tag{4}$$

Заметим, что к моменту выполнения обратного прохода все значения $y_i^{(l)}$, $z_i^{(l)}$, $w_{ij}^{(l)}$, b_i известны для всех i, j, l.

Предположим, что мы используем некоторую функцию ошибки

$$\mathcal{L} = f(\mathbf{y}^L, \mathbf{t}), \tag{5}$$

где $\mathbf{y}^L$ - вектор откликов нейронов последнего слоя с номером L, а $\mathbf{t}$ - эталонные значения.

Выбор функции ошибки зависит от решаемой задачи. Одна из наиболее популярных среднеквадратичная функция ошибки:

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{N}(t_i - y_i^{(L)})^2, \tag{6}$$

где t_i - эталонные значения для текущего примера, N - это количество нейронов в слое L[9].

Для использования алгоритма градиентного спуска необходимо рассчитать производные функции ошибки по всем весам $\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}}$ и смещениям $\frac{\partial \mathcal{L}}{\partial b_i^{(l)}}$ нейронной сети. Именно для этого и применяется алгоритм обратного распространения ошибки.

4 Производные весов (weights)

Для расчета частных производных по весам алгоритм обратного распространения ошибки реализует многократное использование правила взятия производной сложной функции:

$$\boxed{\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g}\cdot\frac{\partial g}{\partial x}}, \tag{7}$$

где $\frac{\partial f}{\partial g}$ обозначает частную производную функции f по её аргументу g, а $\frac{\partial g}{\partial x}$ — частную производную функции g по x.

Правило взятие производной сложной функции на английском языке[10] называется chain rule, что можно перевести как "цепное правило"или "правило цепи". Английское название очень удачно отражает суть: функции f и x связываются между с помощью дополнительного звена-посредника - функции g. Применяя это правило много раз можно добавлять произвольно большое количество звеньев $g_1, g_2, \cdots$, что позволяет получать "цепи"произвольной длины.

Рассмотрим производную по весу $\frac{\partial \mathcal{L}}{\partial w_{21}^{(l)}}$ для сети с рисунка 1. Для ее расчета применим правило взятия производной сложной функции:

$$\frac{\partial \mathcal{L}}{\partial w_{21}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}}\cdot\frac{\partial z_1^{(l+1)}}{\partial w_{21}^{(l)}}. \tag{8}$$

В этом случае в правиле взятия производной сложной функции (7) в качестве функции f выступает функция ошибки $\mathcal{L}$, а в качестве функции g - функция $z_1^{(l+1)}$. Применим правило взятия производной сложной функции еще раз уже для $\frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}}$. В качестве g используем $y_1^{(l+1)}$:

$$\frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}} = \frac{\partial \mathcal{L}}{\partial y_1^{(l+1)}}\cdot\frac{\partial y_1^{(l+1)}}{\partial z_1^{(l+1)}} \tag{9}$$

[9]Обратим внимание, что каллиграфической $\mathcal{L}$ мы обозначаем функцию ошибки (loss – по-английски потеря), а классической L – количество слоев (layer – по-английски слой).

[10]Французы, насколько известно автору, называют это правило в честь первооткрывателя правилом Лейбница.

а значит:

$$\frac{\partial \mathcal{L}}{\partial w_{21}^{(l)}} = \frac{\partial \mathcal{L}}{\partial y_1^{(l+1)}} \cdot \frac{\partial y_1^{(l+1)}}{\partial z_1^{(l+1)}} \cdot \frac{\partial z_1^{(l+1)}}{\partial w_{21}^{(l)}}. \tag{10}$$

Визуально можно представить себе компоненты, необходимые для составления формулы (10), рассмотрев путь связывающий вес w_{21} с производной ошибки активации верхнего слоя. На рисунке 2 этот путь выделен красным цветом.

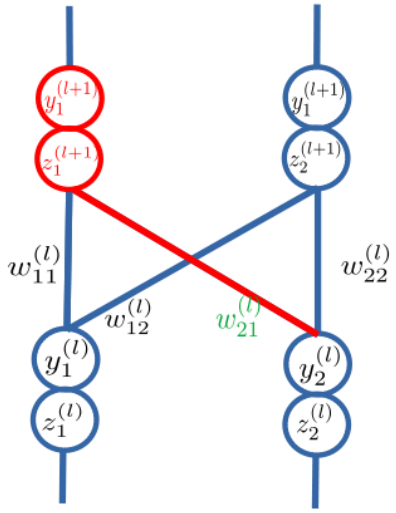

Рис. 2: Расчет производной ошибки по весу w_{21}.

В формуле (10) можно посчитать значение:

$$\frac{\partial z_1^{(l+1)}}{\partial w_{21}^{(l)}} = \frac{\partial \sum_{j=1}^{N} y_j^{(l)} \cdot w_{ij}^{(l)} + b_i^{(l)}}{\partial w_{21}^{(l)}} = y_2^{(l)}, \tag{11}$$

Теперь нужно понять, каким образом, формируются значения частных производных $\frac{\partial \mathcal{L}}{\partial y_1^{(l+1)}}$ и $\frac{\partial y_1^{(l+1)}}{\partial z_1^{(l+1)}}$ со слоя $l+1$.

Предположим, что слой $l+1$ является последним. Пусть используется среднеквадратичная функция ошибки[11]:

$$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{N} (t_i - y_i^{(l+1)})^2, \tag{12}$$

где t_i - эталонные значения для текущего примера.

В этом случае обычно не используют активационную функцию, т.е. $y_i^{(l+1)} = z_i^{(l+1)}$, тогда

$$\frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}} = \frac{\partial \mathcal{L}}{\partial y_1^{(l+1)}} = -(t_1 - y_1^{(l+1)}). \tag{13}$$

[11]Может быть выбранная и другая функции ошибки, но все они выбираются так, чтобы по ним можно было рассчитать аналитическую производную

Переобозначим:

$$\delta_1^{(l+1)} = \frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}}. \tag{14}$$

Значения $\delta_1^{(l+1)}$ называются ошибками нейронов слоя $l+1$ и если слой $l+1$ является последним, то они вычисляются непосредственно по формуле (13).

Рассмотрим как меняется запись уравнения (10) частной производной веса $w_{21}^{(l)}$ с использованием ошибок δ_i.

Исходно уравнение частной производной функции ошибки по весу w_{21} выглядело так:

$$\frac{\partial \mathcal{L}}{\partial w_{21}^{(l)}} = \frac{\partial \mathcal{L}}{\partial y_1^{(l+1)}} \cdot \frac{\partial y_1^{(l+1)}}{\partial z_1^{(l+1)}} \cdot \frac{\partial z_1^{(l+1)}}{\partial w_{21}^{(l)}}. \tag{15}$$

С учетом того, что частная производная входа по весу равняется y_2 (уравнение (11)), то после замены получаем следующую запись:

$$\frac{\partial \mathcal{L}}{\partial w_{21}^{(l)}} = \delta_1^{(l+1)} \cdot y_2^{(l)}, \tag{16}$$

Обобщая формулу на любой вес, мы получаем:

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_j^{(l+1)} \cdot y_i^{(l)},} \tag{17}$$

где

$$\delta_j^{(l+1)} = -(t_j - y_j^{(l+1)}). \tag{18}$$

Заметим еще раз, что в случае если $l+1$ последний слой активаций, то значения $t_1, y_1^{(l+1)}$ известны, и мы можем вычислить $\frac{\partial \mathcal{L}}{\partial w_{21}^{(l)}}$ непосредственно.

Рассмотрим теперь случай, когда слой $l+1$ является промежуточным. Оказывается, что в этом случае формула (17) останется неизменной, но изменится расчет $\delta_j^{(l+1)}$. Например, рассчитаем значения $\frac{\partial \mathcal{L}}{\partial z_2^{(l)}} = \delta_2^{(l)}$ для слоя l сети, представленной на рисунке 1. На рисунке 3 выделены пути воздействия на значения $z_2^{(l)}$, которые должны быть учтены в формуле

С использованием правила взятия производной сложной функции аналогично тому, как это делалось до этого, получаем:

$$\frac{\partial \mathcal{L}}{\partial z_2^{(l)}} = \frac{\partial \mathcal{L}}{\partial y_2^{(l)}} \cdot \frac{\partial y_2^{(l)}}{\partial z_2^{(l)}}. \tag{19}$$

В данном случае необходимо заметить, что $y_2^{(l)}$ влияет на вход сразу двух нейронов следующего слоя $z_1^{(l+1)}$ и $z_2^{(l+1)}$(рисунок 3). Таким образом, существует сразу две функции, для которых необходимо использовать правило взятия сложных производных:

$$\frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}} \cdot \frac{\partial z_1^{(l+1)}}{\partial y_2^{(l)}} \cdot \frac{\partial y_2^{(l)}}{\partial z_2^{(l)}}, \tag{20}$$

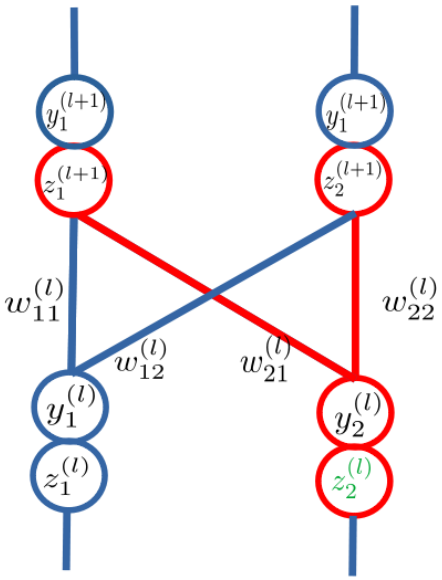

Рис. 3: Расчет производной по выходу второго нейрона слоя l.

$$\frac{\partial \mathcal{L}}{\partial z_2^{(l+1)}} \cdot \frac{\partial z_2^{(l+1)}}{\partial y_2^{(l)}} \cdot \frac{\partial y_2^{(l)}}{\partial z_2^{(l)}} \tag{21}$$

Сложение двух этих производных и определяет значение

$$\frac{\partial y_2^{(l)}}{\partial z_2^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}} \cdot \frac{\partial z_1^{(l+1)}}{\partial y_2^{(l)}} + \frac{\partial \mathcal{L}}{\partial z_2^{(l+1)}} \cdot \frac{\partial z_2^{(l+1)}}{\partial y_2^{(l)}}, \tag{22}$$

а сама формула (19) приобретает вид:

$$\frac{\partial \mathcal{L}}{\partial z_2^{(l)}} = \left(\sum_{k=1}^{2} \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial y_2^{(l)}} \right) \cdot \frac{\partial y_2^{(l)}}{\partial z_2^{(l)}}. \tag{23}$$

С учетом того, что

$$\frac{\partial z_k^{(l+1)}}{\partial y_2^{(l)}} = \frac{\partial \left(\sum_{j=1}^{2} y_j^{(l)} \cdot w_{jk}^{(l)} + b_k^{(l)} \right)}{\partial y_2^{(l)}} = w_{2k}^{(l)}, \tag{24}$$

получаем следующую формулу:

$$\frac{\partial \mathcal{L}}{\partial z_2^{(l)}} = \left(\sum_{k}^{2} \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \cdot w_{2k}^{(l)} \right) \cdot \frac{\partial y_2^{(l)}}{\partial z_1^{(l)}} = \left(\sum_{k=1}^{2} \delta_k^{(l+1)} \cdot w_{2k}^{(l)} \right) \cdot \frac{\partial y_2^{(l)}}{\partial z_2^{(l)}} \tag{25}$$

Обобщая формулу для произвольного $z_j^{(l)}$, а также с учетом того, скобками можно пренебречь, получаем:

$$\frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \boxed{\delta_j^{(l)} = \sum_{k}^{N^{(l+1)}} \delta_k^{(l+1)} \cdot w_{jk}^{(l)} \cdot \frac{\partial y_j^{(l)}}{\partial z_j^{(l)}},} \tag{26}$$

где $N^{(l+1)}$ - это количество нейронов в слое $l+1$.

Таким образом для расчета $\delta_j^{(l)}$ необходимо знать значение $\delta_k^{(l+1)}$ для всех k. Производная активационной функции относительно входа $\frac{\partial y_j^{(l)}}{\partial z_j^{(l)}}$ определяется видом активационной функции f используемой в слое l.

Рассмотрим производные разных активационных функций:

1. Для сигмоидальной функции $y_j^{(l)} = \frac{1}{1+e^{-z_j^{(l)}}}$:

$$\frac{\partial y_j^{(l)}}{\partial z_j^{(l)}} = y_j^{(l)} \cdot (1 - y_j^{(l)}) \tag{27}$$

2. Для RELU функции $y_j^{(l)} = \max(0, z_j^{(l)})$ производная равна:

$$\frac{\partial y_j^{(l)}}{\partial z_j^{(l)}} = \begin{cases} 1, & \text{если } z_j^{(l)} > 0 \\ 0, & \text{если } z_j^{(l)} \leq 0 \end{cases} \tag{28}$$

3. Для Leaky RELU функции $\max(0.01x, x)$ производная равна:

$$\frac{\partial y_j^{(l)}}{\partial z_j^{(l)}} = \begin{cases} 1, & \text{если } z_j^{(l)} > 0 \\ 0.01, & \text{если } z_j^{(l)} \leq 0 \end{cases} \tag{29}$$

Как видим, во всех случаях мы можем рассчитать значения производных активационных функций на основе известных из прямого прохода значений $z_j^{(l)}$, либо $y_j^{(l)}$(в случае с сигмоидальной функцией).

Выпишем отдельно итоговые формулы (30) и (26) для расчета производных функции ошибки по весам и ошибок для скрытых слоев:

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_j^{(l+1)} \cdot y_i^{(l)},} \tag{30}$$

$$\frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \boxed{\delta_j^{(l)} = \sum_k^{N^{(l+1)}} \delta_k^{(l+1)} \cdot w_{jk}^{(l)} \cdot \frac{\partial y_j^{(l)}}{\partial z_j^{(l)}}.} \tag{31}$$

Для последнего слоя L, значения $\delta_i^{(L)}$ определяются на основании эталонных данных в зависимости от функции ошибки. Например, для среднеквадратичной ошибки (формула (13)):

$$\delta_j^{(L)} = -(t_j - y_j^{(L)}). \tag{32}$$

Таким образом начиная с последнего слоя можно рассчитать производные по всем весам вплоть до первого, распространяя значения ошибок все дальше вглубь сети.

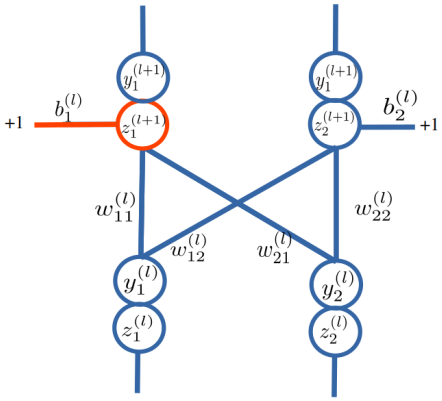

Рис. 4: Расчет производной по смещению $b_1^{(l)}$

5 Производные смещения (bias)

Рассмотрим производную ошибки по смещению $b_1^{(l)}$. На рисунке 4 представлена сеть с рисунка 1, включающая смещения.

Производная функции ошибки по смещению $b_1^{(l)}$ вычисляется по формуле:

$$\frac{\partial \mathcal{L}}{\partial b_1^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}} \cdot \frac{\partial z_1^{(l+1)}}{\partial b_1^{(l)}}. \tag{33}$$

Как видно, из рисунка 4, действительно, на $b_1^{(l)}$ влияют пути, обязательно проходящие через $z_i^{(l)}$. При этом в соответствии с выбранным выше обозначением:

$$\frac{\partial \mathcal{L}}{\partial z_1^{(l+1)}} = \delta_1^{(l+1)} \tag{34}$$

С учетом того, что

$$\frac{\partial z_1^{(l+1)}}{\partial b_1^{(l)}} = \frac{\partial \left(\sum_{j=1}^{N} y_j^{(l)} \cdot w_{j1}^{(l)} + b_1^{(l)} \right)}{\partial b_1^{(l)}} = 1 \tag{35}$$

из 33 получаем следующую формулу:

$$\frac{\partial \mathcal{L}}{\partial b_1^{(l)}} = \delta_1^{(l+1)}. \tag{36}$$

Обобщая получаем:

$$\boxed{\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l+1)}}. \tag{37}$$

6 Векторная форма

Векторные вычисления выполняются значительно быстрее с использованием современных центральных и графических процессоров. Такие операции как

транспонирование, умножение матриц и поэлементное умножение (произведение Адамара) выполняются параллельно, что позволяет значительно ускорить обучение за счет использования оптимизированных операций линейной алгебры на GPU или других высокопроизводительных вычислительных устройствах. Поэтому векторизация вычислений является важным шагом при реализации алгоритма обратного распространения ошибки.

Рассмотрим как полученные ранее выражения могут быть представлены в виде операций с векторами и матрицами. Веса связей в полносвязной сети представляются в виде матрицы естественным образом. Размер весовой матрицы $\mathbf{W}^{(l)} = [w_{ji}]$ равен $N^{(l+1)} \times N^{(l)}$, где $N^{(l)}$ - количество нейронов в слое l, а $N^{(l+1)}$ - количество нейронов в слое $l+1$[12].

Тогда отклик слоя l можно записать в виде вектора-столбца:

$$\mathbf{y}^{(l)} = [y_j^{(l)}]. \tag{38}$$

Размер этого вектора $N^{(l)} \times 1$. Вектор смещения слоя $l+1$ также можно записать в виде вектора-столбца:

$$\mathbf{b}^{(l+1)} = [b_j^{(l+1)}]. \tag{39}$$

Размер этого вектора $N^{(l+1)} \times 1$. Тогда отклик слоя $l+1$ можно рассчитать по формуле:

$$\boxed{\mathbf{z}^{(l+1)} = \mathbf{W}^{(l)}\mathbf{y}^{(l)} + \mathbf{b}^{(l+1)}.} \tag{40}$$

На рисунке 5а представлена визуализация этих вычислений.

Также можно объединить в вектор-столбец значения ошибок $\delta_j^{(l+1)}$

$$\boldsymbol{\delta}^{(l+1)} = [\delta_j^{(l+1)}] \tag{41}$$

Размер этого вектора $N^{(l+1)} \times 1$.

Теперь рассмотрим векторную форму записи формулы (30).

В скалярном виде формула выглядит так:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_j^{(l+1)} \cdot y_i^{(l)}, \tag{42}$$

а в векторном виде матрица частных производных по весам будет иметь размерность $N^{(l+1)} \times N^{(l)}$, , совпадающую с размерностью матрицы весов $\mathbf{W}$, и будет вычисляться по формуле:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boxed{\nabla \mathbf{W}^{(l)} = \boldsymbol{\delta}^{(l+1)} \cdot {\mathbf{y}^{(l)}}^T,} \tag{43}$$

где $\mathbf{y}^{(l)}$ - вектор активаций слоя l, символ ^{T} обозначает транспонирование. На рисунке 5б представлена визуализация вычислений по формуле (43).

Векторизируем формулу (26), которая в скалярном виде выглядит так:

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \sum_k^2 \delta_k^{(l+1)} \cdot w_{jk}^{(l)} \cdot \frac{\partial y_j^{(l)}}{\partial z_j^{(l)}}. \tag{44}$$

[12]Такой размер объясняется тем, что как было указано выше w_{ij} обозначает вес связи от нейрона i слоя l к нейрону j слоя $l+1$. Часто могут использовать обозначение, когда индексы инвертированы (i - индекс нейрона слоя $l+1$), в этом случае размер матрицы весов будет $N^{(l)} \times N^{(l+1)}$.

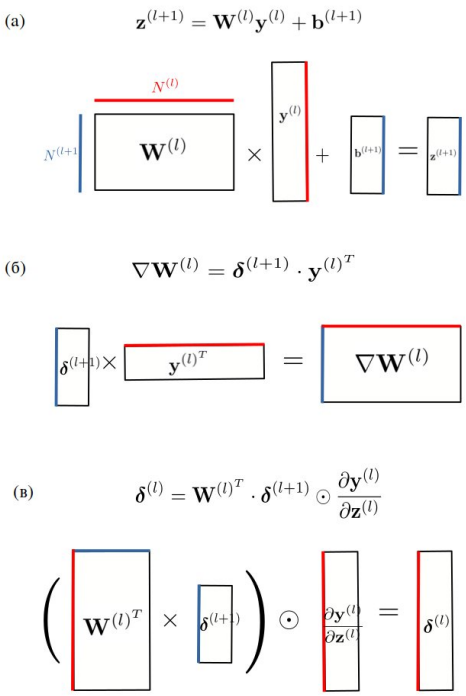

Рис. 5: Векторная форма для вычислений прямого и обратного прохода АОРО

а в векторном виде она принимает вид:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} = \boxed{\boldsymbol{\delta}^{(l)} = \mathbf{W}^{(l)^T} \cdot \boldsymbol{\delta}^{(l+1)} \odot \frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{z}^{(l)}},} \tag{45}$$

где $\mathbf{z}^{(l)}$ - вектор входов слоя l, символ ^{T} обозначает транспонирование, а $\odot$ - поэлементное умножение (произведение Адамара). Визуализация вычислений по формуле (45) представлена на рисунке 5в.

Нетрудно убедиться, что каждый элемент матрицы $\nabla \mathbf{W}^{(l)}$ будет вычислен в соответствии с формулой (30), а каждый элемент вектора $\boldsymbol{\delta}^{(l)}$ вычислен в соответствии с формулой (26)[13].

Векторный расчет градиента смещения является тривиальным, поскольку:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}. \tag{46}$$

Градиент нейронной сети определяется как совокупность всех частных производных ошибки по весам и смещениям всех слоев нейронной сети:

$$\nabla \mathbf{W} = < \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}}, \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}}, \cdots > \tag{47}$$

$$\nabla \mathbf{b} = < \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(2)}}, \cdots > \tag{48}$$

Градиентный спуск использует эти значения, чтобы обновлять веса нейронной сети на каждом шаге обучения t:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha \nabla \mathbf{W}, \tag{49}$$

$$\mathbf{b}_{t+1} = \mathbf{b}_t - \alpha \nabla \mathbf{b}, \tag{50}$$

где α - это скорость (шаг) обучения. Как видно, в этом случае и для корректировки весов все операции выполняются над векторами и матрицами.

7 Векторная форма для обработки пакета примеров

Оптимизация вычислений в mini-batch пакетах позволяет еще больше повысить эффективность обучения нейронных сетей. В этом случае вместо обработки одного примера за раз, mini-batch обработка позволяет обрабатывать множество примеров параллельно, что значительно сокращает время обучения за счет дополнительной векторизации операций. Кроме того, как правило, использование сразу нескольких примеров для расчета градиентов оправдано и с точки зрения улучшения свойств сходимости алгоритма оптимизации.

Для mini-batch пакета размером M, отклик слоя $l+1$ для всех примеров в пакете рассчитывается как:

$$\mathbf{Z}^{(l+1)} = \mathbf{W}^{(l)} \mathbf{Y}^{(l)} + \mathbf{B}^{(l)}, \tag{51}$$

[13]Это можно сделать самостоятельно

где $\mathbf{Y}^{(l)}$ теперь является матрицей размером $N^{(l)} \times M$, содержащей векторы активаций всех примеров в пакете, а $\mathbf{B}^{(l+1)}$ получается копированием вектора $\mathbf{b}^l$ M раз.

Иллюстрация к формулам вычислений прямого и обратного прохода для пакета примеров приведена на рисунке 6.

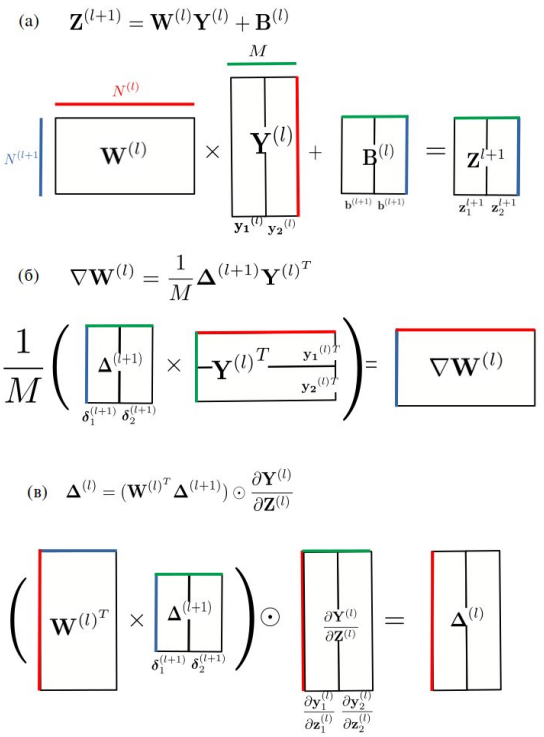

Рис. 6: Визуализация векторной формы вычислений прямого и обратного прохода для пакетной обработки примеров (M=2)

Матрица частных производных по весам $\nabla\mathbf{W}^{(l)}$ для всего пакета вычисляется как среднее значение градиентов по всем примерам в пакете:

$$\nabla\mathbf{W}^{(l)} = \frac{1}{M}\boldsymbol{\Delta}^{(l+1)}\mathbf{Y}^{(l)T}, \tag{52}$$

где $\boldsymbol{\Delta}^{(l+1)}$ является матрицей, содержащей вектора ошибок слоя $l+1$ для всех примеров в пакете:

$$\boldsymbol{\Delta}^{(l+1)} = \begin{bmatrix} | & | & \cdots & | \\ \boldsymbol{\delta}_1^{(l+1)} & \boldsymbol{\delta}_2^{(l+1)} & \cdots & \boldsymbol{\delta}_M^{(l+1)} \\ | & | & \cdots & | \end{bmatrix}. \tag{53}$$

Несложно убедиться, что каждый элемент матрицы $\nabla\mathbf{W}^{(l)}$ является усредненным значением частной производной по весу w_{ij} в соответствии с урав-

нением частной производной (30):

$$\nabla \mathbf{W}_{ij}^{(l)} = \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{1}{M} \sum_{k=1}^{M} \delta_{kj}^{(l+1)} \cdot y_{ki}^{(l)}. \tag{54}$$

Градиент по смещениям $\nabla \mathbf{B}^{(l)}$ вычисляется как усредненная ошибка по всем примерам в пакете:

$$\nabla \mathbf{B}^{(l)} = \frac{1}{M} \sum_{k=1}^{M} \boldsymbol{\delta}_{k}^{(l+1)} \tag{55}$$

Расчет ошибок $\boldsymbol{\Delta}^{(l)}$ для каждого примера основывается на ошибках всех примеров следующего слоя $\boldsymbol{\Delta}^{(l+1)}$. Для перехода от $\boldsymbol{\Delta}^{(l+1)}$ к $\boldsymbol{\Delta}^{(l)}$, используется следующая формула:

$$\boldsymbol{\Delta}^{(l)} = (\mathbf{W}^{(l)^T} \boldsymbol{\Delta}^{(l+1)}) \odot \frac{\partial \mathbf{Y}^{(l)}}{\partial \mathbf{Z}^{(l)}} \tag{56}$$

где $\frac{\partial \mathbf{Y}^{(l)}}{\partial \mathbf{Z}^{(l)}}$ матрица производных функции активации нейронов слоя l каждого из M примеров, а $\odot$ обозначает поэлементное умножение (произведение Адамара). Можно убедиться, что и в этом случае столбцы матрицы $\Delta^{(l+1)}$ будут рассчитаны в соответствии с (45)[14].

Важно отметить, что размер mini-batch пакета M выбирается таким образом, чтобы обеспечить хороший баланс между эффективностью обучения и качеством получаемой модели. Слишком большой размер пакета может привести к ухудшению качества обучения из-за снижения стохастичности градиентного спуска (шум является источником регуляризации), в то время как слишком маленький размер пакета увеличивает время обучения и может приводить к расхождению обучения из-за малой точности оценки градиента по малому количеству примеров.

8 Кросс-энтропия и softmax

8.1 Softmax

Softmax - это групповая активационная функция, которая преобразует вектор вещественных чисел (соответствующий активациям группы нейронов) в вектор вероятностей, удовлетворяющих условию нормировки (рисунок 7).

Softmax часто используется в качестве последнего слоя в многоклассовой задаче классификации, где каждый элемент вектора представляет вероятность

Формально, для входного вектора $\mathbf{z}$ размером K, функция softmax определяется как:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}. \tag{57}$$

Значение элементов вектора $\mathbf{z}$ называют логитами. Логиты представляют собой необработанные оценки классов, полученные от модели. Чем больше

[14]Это можно сделать самостоятельно

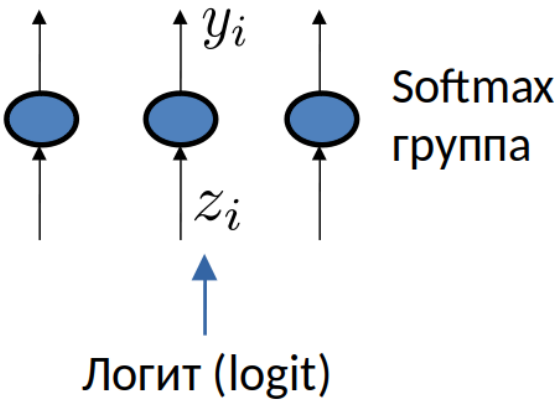

Рис. 7: Визуализация функции softmax

логит, тем больше вероятность, что объект принадлежит к классу. Однако, логиты не являются вероятностями, поскольку они могут принимать отрицательные значения и не нормированы. Функция softmax преобразует логиты в вероятности, которые удобно использовать для принятия решений в задачах классификации.

Рассмотрим основные свойства функции softmax:

- Все элементы вектора $\text{softmax}(\mathbf{z})$ находятся в интервале $[0, 1]$.
- Сумма всех элементов вектора $\text{softmax}(\mathbf{z})$ равна 1, что соответствует условию нормировки вероятностей.

Функция называется softmax, поскольку она усиливает различия между элементами вектора, т.е. увеличивает вероятность наибольшего элемента и уменьшает вероятности остальных элементов. Т.е. она выполняет операцию, напоминающую операцию взятия максимума (а точнее обозначения места максимума), однако при этом является дифференцируемой (существует производная для всех элементов вектора). Строго говоря, правильно название для softmax должно было бы быть softargmax. Для того, чтобы увидеть каким образом функция softmax усиливает различия между элементами вектора рассмотрим логарифм от функции softmax:

$$\log(\text{softmax}(\mathbf{z})_i) = z_i - \log\left(\sum_{j=1}^{K} e^{z_j}\right). \tag{58}$$

Как видно, логарифм от функции softmax представляет собой разность между логитом и логарифмом суммы экспонент логитов. При этом заметим, что $\log\left(\sum_{j=1}^{K} e^{z_j}\right) \approx \max_j z_j$, поскольку экспонента усиливает различия между элементами вектора. Например, для вектора логитов $\mathbf{z} = [1, 2, 3]$ значение $\log\left(\sum_{j=1}^{K} e^{z_j}\right) \approx 3.408$, что близко к максимальному значению в векторе $\mathbf{z}$. Связано это с тем, что экспонента растет очень быстро, и подавляющий вклад в сумму вносит максимальный элемент вектора логитов.

$$\text{softmax}([1, 2, 3]) = [0.09, 0.244, 0.665]. \tag{59}$$

Строчку выше можно интерпретировать так: "максимум среди элементов [1,2,3] с вероятностью 66% находится в третьей позиции.

Взятие экспоненты от больших чисел может привести к нарушению численной стабильности. Для решения этой проблемы можно воспользоваться следующими свойствами:

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} + c), \tag{60}$$

где c — произвольная константа. В качестве c можно взять минус максимальное значение вектора логитов, поскольку в этом случае все элементы вектора логитов становятся отрицательными, и экспонента будет возвращать числа, близкие к нулю. Это позволяет избежать переполнения при вычислении экспоненты от больших чисел:

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} - \max_i z_i). \tag{61}$$

8.2 Кросс-энтропия

Кросс-энтропия (cross entropy) — это понятие из теории информации, используемое в различных областях, включая машинное обучение. Она используется для измерения различий между двумя вероятностными распределениями. Она часто применяется для измерения потерь в задачах классификации, где нужно оценить, насколько хорошо предсказанные вероятностные распределения соответствуют истинным распределениям.

Математически кросс-энтропия между двумя распределениями вероятностей P и Q по одному и тому же пространству событий определяется как: $H(P, Q) = -\sum_x P(x) \log(Q(x))$ где суммирование ведётся по всем возможным событиям x, $P(x)$ — истинная вероятность события x, а $Q(x)$ — предсказанная вероятность события x.

В контексте машинного обучения, если рассмотреть задачу классификации, P обычно представляет собой истинное распределение меток классов (обычно в форме one-hot кодировки, где истинный класс имеет вероятность 1 (рисунок (8)), а все остальные — 0), а Q — предсказанное распределение, полученное с помощью модели после применения функции softmax к выходным данным модели.

Пусть $\mathbf{t}$ - эталонный one-hot вектор, $\mathbf{y}$ - вектор после использования softmax. Тогда значение кроссэнтропийной функции потерь:

$$\mathcal{L}_{CE} = -\sum_{j=1}^{K} t_j \log y_j. \tag{62}$$

8.3 Производные кросс-энтропии и softmax

Если применить кроссэнтропийную функцию потерь $\mathcal{L}_{CE}$ к вектору активации $\mathbf{y} = \text{softmax}(\mathbf{z})$, то придется брать производные сперва с учетом взятия экспоненты, а затем с использованием логарифма. Обе операции (логарифм и взятие экспоненты) по отдельности очень сильно преобразуют сигналы, причем экспонента чувствительна к шумам, а логарифм наоборот избыточно подавляет низкие значения. В результате производные будут

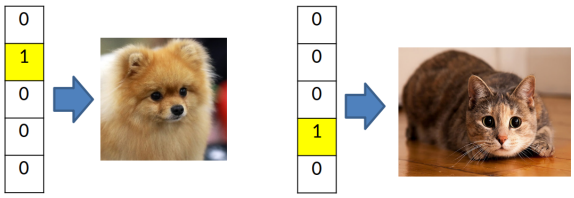

Рис. 8: Использования эталонного one-hot вектора для представления меток классов

посчитаны с большой погрешностью. Вместо этого можно заметить, что операции являются обратными и упростить выражение:

$$\mathcal{L}_{CE} = -\sum_{j=1}^{K} t_j \log\left(\frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}\right). \tag{63}$$

Сперва можно использовать свойства логарифма дроби:

$$\mathcal{L}_{CE} = -\sum_{j=1}^{K} t_j \cdot \log\frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} = -\sum_{j=1}^{K} t_j \cdot \underbrace{\log e^{z_j}}_{=z_j} + \sum_{j=1}^{K} t_j \cdot \log\sum_{k=1}^{K} e^{z_k}. \tag{64}$$

Далее, поскольку $\log e^{z_j} = z_j$:

$$\mathcal{L}_{CE} = -\sum_{j=1}^{K} t_j \cdot z_j + \left(\sum_{j=1}^{K} t_j\right) \cdot \log\sum_{k=1}^{K} e^{z_k}. \tag{65}$$

При расчете производной по элементу вектора z_i будем учитывать, что в соответствии с правилом взятия производной сложной функции:

$$\frac{\partial \log\sum_k e^{z_k}}{\partial z_i} = \frac{1}{\sum_k e^{z_k}} \cdot e^{z_i}, \tag{66}$$

Таким образом, производная кросс-энтропии по логиту z_i:

$$\frac{\partial \mathcal{L}_{CE}}{\partial z_i} = -t_i + \left(\sum_j t_j\right) \frac{e^{z_i}}{\sum_k e^{z_k}} = -t_i + \frac{e^{z_i}}{\sum_k e^{z_k}} \underbrace{\sum_j t_j}_{=1} \tag{67}$$

С учетом того, что $\sum_i t_i = 1$ и уравнения softmax (57) получаем

$$\boxed{\frac{\partial \mathcal{L}_{CE}}{\partial z_i} = y_i - t_i.} \tag{68}$$

Уравнение (68) может быть получено при использовании аналитического вывода при учете сразу двух операций (softmax и кросс-энтропия), однако при использовании АОРО напрямую (независимо для двух операций) точность оценки производных будет меньше, т.к. выражения будут другими.

Однако используя значения полученные с использованием выражения (68) в качестве ошибки δ_i можно использовать АОРО для скрытых слоев без изменения. Библиотеки автоматического дифференцирование стараются оптимизировать типовые сочетания функций, чтобы повысить эффективность АОРО. Заметим, что векторизация выражения (68) является тривиальной.

8.4 Softmax с температурой*

Склонность функции softmax к сильному предпочтению максимума в некоторых приложениях может быть избыточной, т.к. часто требуется, чтобы распределение обладало достаточно высокой энтропией. Для контроля этого часто используют параметр температуры T:

$$\text{softmax}_T(\mathbf{z})_i = \frac{e^{z_i/T}}{\sum_{j=1}^{K} e^{z_j/T}}. \tag{69}$$

Увеличение температуры $T \to \infty$ повышает энтропию выхода, распределение стремится к равномерному:

$$\text{softmax}_{T=100}([1, 2, 3]) = [0.33, 0.33, 0.34], \tag{70}$$

а при $T \to 0$ к жесткому максимуму:

$$\text{softmax}_{T=0.1}([1, 2, 3]) = [2 \cdot 10^{-9}, 4 \cdot 10^{-5}, 0.99995]. \tag{71}$$

Температура позволяет выбирать требуемые характеристики функции итогового распределения.

9 Заключение

В данном методическом пособии подробно рассматривался алгоритм обратного распространения ошибки. Сперва был приведен вывод формул для скалярных величин весов полносвязных сетей. АОРО не получил бы большого распространения, если бы не существовало возможности эффективного распараллеливания вычислений. В связи с этим, скалярные выражения получены так, чтобы их легко было представить в виде операций над матрицами и векторами. Вопрос векторизации формул был подробно рассмотрен в разделе 6. В разделе 8 были рассмотрены операции softmax и кросс-энтропии, на примере которых было показано, что для отдельных функций существуют более эффективные способы подсчета производных чем при использовании АОРО напрямую. Более общо проблему получения производных рассматривает область автоматического дифференцирования. Несмотря на отдельные модификации, АОРО остается основным способом подсчета производных в библиотеках глубокого обучения.

Список литературы

[1] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. nature, 323(6088):533–536, 1986.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32, 2019.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[5] Gobinda Saha, Isha Garg, and Kaushik Roy. Gradient projection memory for continual learning. In International Conference on Learning Representations, 2020.

[6] Geoffrey Hinton. The forward-forward algorithm: Some preliminary investigations. arXiv preprint arXiv:2212.13345, 2022.