Оглавление

Содержание

Глава 3

Сверточные нейронные сети

Рассмотренные до сих пор NN были полносвязными. То есть они обладают тем свойством, что все линейные блоки в слое связаны со всеми линейными блоками в следующем слое. Однако вовсе не требуется, чтобы NN имели именно эту форму. Мы, конечно, можем представить себе прямой проход, при котором линейный блок передает свой выходной сигнал только некоторым блокам следующего слоя. Немного сложнее, но не чрезмерно, увидеть, что, скажем, Tensorflow, если он знает, какие блоки связаны с какими, может правильно вычислить производные веса на обратном проходе.

Один частный случай частично связанных NN — это *сверточные нейронные сети* (convolutional neural network). Сверточные NN особенно часто применяются в компьютерном зрении, поэтому мы продолжаем обсуждение набора данных Mnist.

Одноуровневая полносвязная NN Mnist учится связывать определенные интенсивности света в определенных позициях на изображении с определенными цифрами, например высокие значения в позиции (8, 14) с числом 1. Но это явно не то, как работают люди. Фотографирование цифр в более яркой комнате может добавить 10 к каждому значению пикселя, но это мало повлияет на нашу классификацию, поскольку при распознавании сцены важны различия в значениях пикселей, а не их абсолютные значения. Кроме того, различия имеют смысл только между близкими значениями. Предположим, вы находитесь в маленькой комнате, освещенной одной лампочкой в одном углу комнаты. То, что мы воспринимаем как светло пятно, скажем, на некоторых обоях в противоположном конце комнаты, может фактически отражать меньше фотонов, чем "темное" пятно рядом с лампой. Для определения того, что происходит на сцене, важны локальные различия интенсивности света с акцентом на "местные" и "различия". Естественно, исследователи компьютерного зрения вполне осведомлены об этом, и стандартным ответом на эти факты было почти повсеместное применение сверточных методов.[1]

---

[1] В этом обсуждении термин "свертка" употребляется так, как в глубоком обучении. Это близко, но не совсем то же самое, что и его использование в математике, где глубокая свертка называется *взаимной корреляцией* (cross-correlation).

3.1. Фильтры, шаги и дополнения

Для наших целей *сверточный фильтр* (convolutional filter) (называемый также *сверточным ядром* (convolutional kernel)) представляет собой (как правило, небольшой) массив чисел. Если мы имеем дело с черно-белым изображением, то это двумерный массив. Набор данных Mnist — черно-белый, так что это все, что нам здесь нужно. Если бы у нас был цвет, потребовался бы трехмерный массив — или, что эквивалентно, три двумерных массива — по одному для длин волн красного, синего и зеленого (RBG) света, из которых можно создать все цвета. На данный момент мы игнорируем сложности цвета. Мы вернемся к ним позже.

Рассмотрим сверточный фильтр, показанный на рис. 3.1. Для *свертки* (convolve) фильтра с фрагментом изображения мы берем скалярное произведение фильтра и фрагмент изображения равного размера. Следует помнить, что скалярное произведение двух векторов подразумевает попарное умножение соответствующих элементов векторов и суммирование произведения, чтобы получить одно число. Здесь мы обобщаем это понятие для массивов двух или более измерений, поэтому умножаем соответствующие элементы массивов и затем суммируем все произведения.

1,0	1,0	1,0	1,0
1,0	1,0	1,0	1,0
–1,0	–1,0	–1,0	–1,0
–1,0	–1,0	–1,0	–1,0

Рис. 3.1. Простой фильтр для определения горизонтальной линии

Более формально мы считаем ядро свертки функцией, *функцией ядра* (kernel function). Мы получаем значение V этой функции в позиции x, y на изображении I следующим образом:

$$V(x,y) = (I \cdot K)(x,y) = \sum_m \sum_n I(x+m, y+n) K(m,n) \tag{3.1}$$

То есть формально свертка — это операция (здесь она представлена центральной точкой), которая получает две функции, I и K, и возвращает третью функцию, которая выполняет операцию справа. Для наших обычных целей мы можем пропустить формальное определение и просто перейти к операциям с правой стороны. Кроме того, мы обычно думаем о точке x, y как о середине фрагмента (или рядом с серединой), над которым мы работаем, поэтому для ядра $4 * 4$, показанного выше, и m, и n могут варьироваться от –2 до +1 включительно.

0,0	0,0	0,0	0,0	0,0	0,0
0,0	2,0	2,0	2,0	0,0	0,0
0,0	2,0	2,0	2,0	0,0	0,0
0,0	2,0	2,0	2,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0

Рис. 3.2. Изображение небольшого квадрата

Давайте свернем фильтр на рис. 3.1 с правой нижней частью простого изображения квадрата, показанного на рис. 3.2. Два нижних ряда фильтра перекрываются нулями на изображении. Но четыре верхних левых элемента фильтра перекрываются квадратом 2,0, поэтому значение фильтра в этом фрагменте равно 8. Естественно, если бы все значения пикселей были равны нулю, значение применения фильтра тоже было бы равно нулю. Но если бы весь фрагмент изображения состоял из всех десяток, он все равно был бы нулевым. На самом деле нетрудно увидеть, что этот фильтр имеет самые высокие значения для фрагментов с горизонтальной линией, проходящей через середину фрагмента, идущей от высоких значений сверху и значений ниже снизу. Дело, конечно, в том, что фильтры можно сделать такими, чтобы они были чувствительны к изменениям интенсивности света, а не к их абсолютным значениям, и поскольку фильтры обычно намного меньше, чем полные изображения, они концентрируются на локальных изменениях. Мы можем, конечно, спроектировать ядро фильтра, которое имеет высокие значения для фрагментов изображения с прямыми линиями, идущими от верхнего левого угла к нижнему правому или как угодно.

В приведенном выше обсуждении мы представили фильтр так, как если бы он был разработан программистом, чтобы выделить особый вид изображения, и действительно, это было сделано до появления глубокой сверточной фильтрации. Однако что делает подходы глубокого обучения особенными, так это то, что *значения фильтра являются параметрами NN — они изучаются во время обратного прохода*. В нашей текущей дискуссии о том, как работает свертка, это лучше игнорировать, и мы продолжаем представлять наши фильтры “уже готовыми” до следующего раздела.

Помимо свертки фильтра с *фрагментом* (patch) изображения, мы говорим о свертке фильтра с *изображением* (image). Это подразумевает применение фильтра ко многим фрагментам изображения. Обычно у нас есть много разных фильтров, и цель каждого фильтра — подобрать определенный признак в изображении. Сделав это, мы можем затем передать все значения признаков в один или несколько полносвязных слоев, в softmax и, следовательно, в функцию потерь. Эта архитектура показана на рис. 3.3. Здесь мы представляем слой

сверточного фильтра в виде объемного прямоугольника (поскольку банк фильтров является (как минимум) трехмерным тензором, высота и ширина, а также количество различных фильтров).

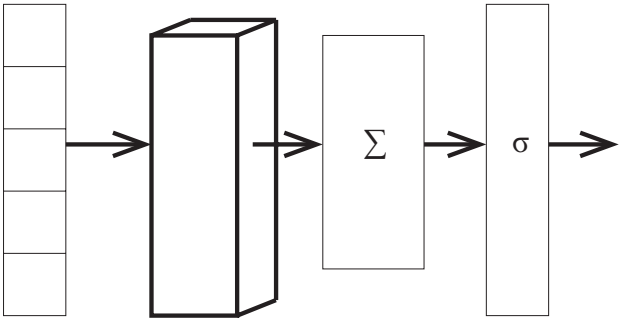

Рис. 3.3. Архитектура распознавания изображений с фильтрами свертки

Обратите внимание на преднамеренную неопределенность, описанную выше, когда мы сказали, что сворачиваем фильтр со "многими" фрагментами изображения. Точнее, мы сначала определим *шаг* (stride) — расстояние между двумя применениями фильтра. Скажем, два шага означают, что мы применяем фильтр к каждому следующему пикселю. Чтобы стать еще более конкретными, мы говорим как о горизонтальном шаге, s_h, так и о вертикальном шаге, s_v. Идея в том, что при прохождении изображения мы применяем фильтр после каждых s_h пикселей. Достигнув конца строки, мы спускаемся вертикально на s_v строк и повторяем процесс. Применяя фильтр на шаге 2, мы по-прежнему применяем фильтр ко всем пикселям в области. Единственное, на что влияет шаг, — это на место, где именно фильтр применяется в следующий раз.

Далее мы определяем, что подразумеваем под "концом строки" при применении фильтра. Это осуществляется за счет указания *дополнения* (padding) для свертки. TF допускает два возможных дополнения: *Valid* (допустимое) и *Same* (одинаковое). После свертки фильтров с определенным участком изображения мы перемещаем s_h вправо. Существует три варианта: а) мы не находимся вблизи границы изображения, поэтому продолжаем работать над этой строкой, б) крайний слева пиксель для следующей свертки выходит за край изображения и в) крайний слева пиксель, который видит фильтр, находится в изображении, но крайний справа располагается за концом изображения. Дополнение *Same* останавливает в случае *б*, *Valid* прекращает в случае *в*. Например, на рис. 3.4 показана ситуация, когда изображение имеет ширину 28 пикселей, фильтр имеет ширину 4 пикселя и высоту 2 пикселя, а шаг равен 1. При дополнении *Valid* мы останавливаемся после пикселя 24 при отсчете с нуля. Это потому, что наш шаг приведет нас к пикселю 25, а для размещения фильтра шириной 4 потребуется 29-й пиксель, которого не существует. Дополнения

Same будут продолжать свертку до тех пор, пока не встретится пиксель 27. Естественно, мы делаем тот же выбор в вертикальном направлении, когда достигаем нижней части изображения.

0	1		23	24	25	26	27
.	.	3,2	3,1	2,5	2,0	0	0
.	.	3,2	3,1	2,5	2,0	0	0
.	.	3,2	3,1	2,5	2,0	0	0
.	.	3,2	3,1	2,5	2,0	0	0
.	.	3,2	3,1	2,5	2,0	0	0

Рис. 3.4. Конец строки при дополнениях Valid и Same

Решение о том, где остановиться, называется *дополнением* (padding) потому, что при горизонтальном движении с дополнением *Same* к моменту остановки мы должны использовать “мнимые” пиксели. Левая сторона фильтра находится внутри границы изображения, а правая — нет. В TF мнимые пиксели имеют значение “нуль”. Таким образом, при дополнении *Same* нужно дополнить границу изображения мнимыми пикселями. При дополнении *Valid* фактическое дополнение почти никогда не требуется, поскольку мы прекращаем свертку до того, как какая-либо часть фильтра выйдет за границы изображения. Когда требуется дополнение (при дополнении *Same*), оно применяется ко всем краям как можно более одинаково.

Поскольку это понадобится нам позже, мы дадим количество сверток фрагментов, которые мы применяем горизонтально для дополнения *Same*:

$$\lceil i_h / s_h \rceil \tag{3.2}$$

Здесь $\lceil x \rceil$ — это *функция потолка* (ceiling function). Она возвращает наименьшее целое число $\geq x$. Чтобы убедиться в необходимости функции потолка, рассмотрим случай, когда изображение имеет нечетное количество пикселей в ширину, скажем, пять, а шаг — два. Сначала фильтр применяется к фрагменту 0–2 в горизонтальном направлении. Затем он перемещается на две позиции вправо и применяется к фрагменту 2–4. Когда мы доберемся до позиции 4, фильтр следует применить к фрагменту 4–6. Поскольку ширина равна 5, позиция 6 отсутствует. Однако для дополнения *Same* мы добавляем нуль в конец строки, чтобы фильтр работал в позициях 4–6, а общее количество применений равно 4. Если дополнение *Same* не добавляет дополнительные нули, вышеприведенное уравнение будет иметь *функцию пола* (floor function), а не потолка.

Естественно, те же самые рассуждения применимы и в вертикальном направлении, давая нам $\lceil i_v / s_v \rceil$.

Для дополнения *Valid* количество по горизонтали составляет

$$\left\lfloor \left(i_h - f_h + 1\right) / s_h \right\rfloor \tag{3.3}$$

Если вы не видите это последнее уравнение сразу, сначала убедитесь, что вы видите, что $i_h - f_h$ — это то, как часто вы можете сдвигаться (до того, как закончится свободное пространство), если шаг равен единице. Но количество применений составляет один плюс количество смещений.

Несмотря на использование мнимых пикселей, дополнение *Same* довольно популярно, поскольку в сочетании с одним шагом оно обладает тем свойством, что размер вывода такой же, как и у исходного изображения. Зачастую мы объединяем множество слоев свертки, каждый из которых становится входом для следующего слоя. Независимо от того, каков размер шага, дополнение *Valid* всегда имеет вывод, который меньше, чем ввод. При повторяющихся слоях свертки результат постепенно уменьшается.

Прежде чем перейти к реальному коду, обсудим, как свертка влияет на то, как мы представляем изображения. Сердцем сверточного NN в TF является двумерная *функция свертки* (convolution function)

```
tf.nn.conv2d(input, filters, strides, padding)
```

плюс необязательные именованные аргументы, которые мы здесь игнорируем. `2d` в имени указывает, что мы сворачиваем изображение. (Существуют также версии `1d` и `3d`, которые сворачивают одномерные объекты, такие как аудиосигнал, или трехмерные, такие как видеоклип.) Как и следовало ожидать, первый аргумент — это размер партии отдельных изображений. До сих пор мы рассматривали отдельное изображение как двумерный массив чисел, в котором каждое число — это интенсивность света. Если вы укажете тот факт, что у нас есть размер партии, входом будет трехмерный тензор.

Но `tt.nn.conv2d` требует, чтобы отдельные изображения были трехмерными объектами и последним измерением был вектор *каналов* (channel). Как упоминалось ранее, нормальные цветные изображения имеют три канала — по одному для красного, синего и зеленого (RBG). Отныне, обсуждая изображения, мы все еще говорим о двумерном массиве пикселей, но каждый пиксель представляет собой список интенсивностей. Этот список содержит одно значение для черно-белых изображений и три значения для цветных.

То же самое верно для фильтров свертки. Фильтр $m * n$ совпадает с размером $m \times n$ пикселей, но теперь и пиксели, и фильтр могут иметь несколько каналов. В данном случае мы создадим фильтр, чтобы найти горизонтальные края бутылок кетчупа на рис. 3.5. Первый сверху ряд фильтра активируется наиболее

высоко, когда входной свет интенсивен только для красного и менее интенсивен для синего и зеленого. Следующие два ряда хотят меньше красного (поэтому есть некоторый контраст) и больше синего и зеленого.

```
( 1, –1, –1)   ( 1, –1, –1)   ( 1, –1, –1)   ( 1, –1, –1)
(–1,  1,  1)   (–1,  1,  1)   (–1,  1,  1)   (–1,  1,  1)
(–1,  1,  1)   (–1,  1,  1)   (–1,  1,  1)   (–1,  1,  1)
```

Рис. 3.5. Простой фильтр для определения горизонтальной линии кетчупа

На рис. 3.6 показан простой пример TF применения функции небольшой свертки к маленькому изображению. Как отмечалось выше, первый вход в `conv2D` — это четырехмерный тензор, где `I` — константа. В комментарии перед объявлением `I` мы демонстрируем, что она будет выглядеть как простой двумерный массив, без дополнительных измерений, добавляемых размером партии (здесь — единица) и размер канала (снова единица). Второй аргумент — это четырехмерный тензор фильтров, здесь `W` также с комментарием, показывающим двухмерную версию, на этот раз без дополнительных измерений количества каналов и количества фильтров (по одному на каждый). Затем следует вызов `conv2D` с горизонтальными и вертикальными шагами, оба как единица и с дополнением *Valid*. Глядя на результат, мы видим, что это четырехмерность (размер партии (1), высота (2), ширина (2), каналы (1)). Высота и ширина значительно уменьшены по сравнению с размером изображения, как и следует ожидать, когда мы используем дополнение *Valid*, а также фильтр достаточно активен (со значением 6), опять же, как и следовало ожидать, поскольку он предназначен для обнаружения вертикальных[2] линий, которые являются именно тем, что появляется на изображении.

```
ii = [[ [[0],[0],[2],[2]],
        [[0],[0],[2],[2]],
        [[0],[0],[2],[2]],
        [[0],[0],[2],[2]] ]]
''' ((0 0 2 2)
     (0 0 2 2)
     (0 0 2 2)
     (0 0 2 2))'''
I = tf.constant(ii, tf.float32)

ww = [ [[-1]],[[-1]],[[1]]],
       [[-1]],[[-1]],[[1]]],
       [[-1]],[[-1]],[[1]]] ]
```

[2] Вероятно, имелось в виду “горизонтальных”. — *Примеч. ред.*

```
'''((-1 -1 1)
    (-1 -1 1)
    (-1 -1 1))'''
W = tf.constant(ww, tf.float32)

C = tf.nn.conv2d( I, W, strides=[1, 1, 1, 1], padding='VALID')
sess = tf.Session()
print sess.run(C)
'''[ [[ 6.] [ 0.]]
     [[ 6.] [ 0.]]]]'''
```

Рис. 3.6. Простое упражнение с использованием `conv2D`

3.2. Простой пример свертки TF

Теперь мы выполним упражнение по превращению прямой программы TF Mnist из главы 2 в сверточную модель NN. Код, который мы создаем, приведен на рис 3.7.

```
1 image = tf.reshape(img, [100, 28, 28, 1])
2    #Превращает img в четырехмерный тензор
3 flts=tf.Variable(tf.truncated_normal([4,4,1,4],stddev=0.1))
4    #Создает параметры для фильтров
5 convOut = tf.nn.conv2d(image, flts, [1, 2, 2, 1], "Same")
6    #Создает граф для свертки
7 convOut= tf.nn.relu(convOut)
8    #Не забудьте добавить нелинейность
9 convOut=tf.reshape(convOut,[100, 784])
10   #Назад к 100 одномерным векторам изображений
11 prbs = tf.nn.softmax(tf.matmul(convOut, W) + b)
```

Рис. 3.7. Основной код, необходимый для преобразования рис. 2.2 в сверточную NN

Как уже отмечалось, ключевым вызовом функции TF является `tf.nn.conv2d`. На рис. 3.7 мы видим в строке 5

```
convOut = tf.nn.conv2d(image, flts, [1, 2, 2, 1], "Same")
```

Рассмотрим каждый аргумент по очереди. Как обсуждалось только что, `image` является четырехмерным тензором — в данном случае вектором трехмерных изображений. Мы выбираем размер партии равным 100, поэтому `tf.nn.conv2d` ожидает 100 трехмерных изображений. Функции, которые читают данные в главе 2, читают векторы одномерных изображений (длиной 784), поэтому строка 1 на рис. 3.7

```
image = tf.reshape(img,[100, 28, 28, 1])
```

преобразует вход в форму [100, 28, 28, 1], где последняя единица указывает, что у нас есть только один входной канал. `tf.reshape` работает почти так же, как `reshape` Numpy.

Следующий аргумент `tf.nn.conv2d` в строке 5 — это указатель на фильтры, которые будут использоваться. Это тоже четырехмерный тензор, в данном случае — формы

[*высота*, *ширина*, *каналы*, *количество*]

Параметры фильтра создаются в строке 3. Мы выбрали фильтры 4×4 ([4,4]), каждый пиксель имеет один канал ([4,4,1]), и мы выбрали четыре фильтра ([4,4,1,4]). Обратите внимание, что высота и ширина фильтра, а также количество создаваемых нами фильтров являются гиперпараметрами. Количество каналов (в данном случае — 1) определяется количеством каналов в изображении, а потому является фиксированным. Очень важно, что мы наконец-то сделали то, что обещали в начале: строка 3 создает значения фильтра в качестве параметров модели NN (с начальными значениями, равными нулю и стандартным отклонением 0,1), поэтому они запоминаются моделью.

Аргумент *шага* (stride) в `tf.nn.conv2d` представляет собой список из четырех целых чисел, указывающих размер шага в каждом из четырех измерений ввода. В строке 5 мы видим, что выбраны шаги 1, 2, 2 и 1. На практике первыми и последними почти всегда являются 1. Во всяком случае, трудно представить себе случай, когда они не будет 1. В конце концов, первое измерение — это отдельные трехмерное изображения в пакете. Если бы шаг по этому измерению равнялся двум, мы бы пропустили все остальные изображения! Столь же странно, что если бы последний шаг был больше единицы, скажем 2, и у нас было три цветовых канала, то мы смотрели бы только на красный и синий свет, пропуская зеленый. Таким образом, типичные значения для шага будут (1, 1, 1, 1) или, если мы хотим свертывать только каждый второй фрагмент изображения в горизонтальном и вертикальном направлениях, (1, 2, 2, 1). Вот почему вы часто видите в обсуждениях инструкций `tf.nn.conv2d`, что первый и последний шаги должны быть единицей.

Последний аргумент, *дополнение* (padding), является строкой, равной одному из типов дополнения, которые распознает TF, например *Same*.

Вывод `conv2d` очень похож на ввод. Это тоже четырехмерный тензор, и, как и ввод, первое измерение вывода — это размер партии. Или, другими словами, выходные данные представляют собой вектор выходов свертки, по одному для каждого входного изображения. Следующие два измерения — это количество применений фильтров, за которыми следуют горизонталь и вертикаль; они могут быть определены, как в уравнениях 3.2 и 3.3. Последнее измерение выходного тензора — это количество фильтров, свернутых с изображением. Выше мы указали, что будем использовать четыре, т.е. выходная форма такова:

[*размер партии*, *размер по горизонтали*,
размер по вертикали, *количество фильтров*]

В нашем случае это будет (100, 14, 14, 4). Если мы думаем о выходе как об "изображении", то на входе будет 28×28 с одним каналом, но на выходе будет (14×14) и 4 канала. Это означает, что в обоих случаях входное изображение представлено 784 числами. Мы выбрали это намеренно, чтобы сохранить сходство с главой 2, но мы не обязаны были делать это. Скажем, мы могли бы выбрать 16, а не четыре разных фильтра, и в этом случае у нас было бы изображение, представленное $(14 * 14 * 16 = 3136)$ числами.

В строке 11 мы подаем эти 784 значения в полносвязный слой, который создает логиты для каждого изображения, которые, в свою очередь, передаются в softmax, а затем мы вычисляем кросс-энтропийную потерю (на рис. 3.7 не показана), и у нас есть очень простая сверточная NN для Mnist. Код имеет общую форму, показанную на рис. 2.2. Кроме того, строка 7 выше помещает нелинейность между выходом свертки и входом полносвязного слоя. Это важно. Как было показано ранее, без нелинейных функций активации между линейными блоками улучшения не происходит.

Производительность этой программы значительно выше, чем в главе 2, — 96% или чуть больше, в зависимости от случайной инициализации (по сравнению с 92% для версии с прямой связью). Количество параметров модели практически одинаково для двух версий. В обоих слоях с прямой связью используется 7840 весовых коэффициентов **W** и 100 смещений **b** $(784 + 10$ весовых коэффициентов в каждом блоке в полносвязном слое, умноженное на 10 единиц). Свертка добавляет четыре сверточных фильтра, каждый с $4 * 4$ весами или еще 64 параметрами. Вот почему мы устанавливаем выходной размер свертки равным 784. В нулевой аппроксимации качество NN повышается, поскольку мы даем ей больше параметров для использования. Здесь, однако, количество параметров практически не изменилось.

3.3. Многослойная свертка

Как уже упоминалось, мы можем еще больше повысить точность, перейдя от одного уровня свертки к нескольким. В этом разделе мы строим модель с двумя слоями.

Ключевым моментом многослойной свертки является то, что мы пропустили при обсуждении вывода из `tf.conv2d`: он имеет тот же формат, что и ввод. Оба являются трехмерными векторами размера партии изображений, изображения являются двумерными плюс одно дополнительное измерение для количества каналов. Таким образом, выход из одного слоя свертки может быть входом для второго слоя, и это именно то, что он делает. Когда мы говорим о заполнителе,

полученном из данных, последнее измерение — это количество цветовых каналов. Говоря о выводе conv2d, мы имеем в виду, что последнее измерение — это количество различных фильтров в слое свертки. Слово “фильтр” здесь вполне уместно. В конце концов, чтобы пропустить через объектив только синий свет, мы буквально поместили бы перед ним цветной фильтр. Таким образом, три фильтра дают нам изображения в спектрах RBG. Теперь мы получаем “изображения” в псевдоспектрах, как “горизонтальные линейно-граничные спектры”. Это будет воображаемое изображение, создаваемое фильтром, например, на рис. 3.1. Кроме того, фильтры для изображений RBG также имеют веса, связанные со всеми тремя спектрами, второй слой свертки имеет веса для каждого канала, выводимого из первого.

Мы предоставляем код для превращения NN прямой связи Minst в двухслойную модель свертки на рис. 3.8. Строки 1–4 являются повторениями первых строк на рис. 3.7, за исключением того, что в строке 2 мы увеличиваем количество фильтров в первом слое свертки до 16 (с 4 в более ранней версии). Строка 2 отвечает за создание фильтров второго сверточного слоя flts2. Обратите внимание, что мы создали 32 из них. Это отражено в строке 5, в которой значения 32 фильтров становятся 32 значениями входного канала для второго слоя свертки.

```
1 image = tf.reshape(img, [100, 28, 28, 1])
2 flts=tf.Variable(tf.normal([4, 4, 1, 16], stddev=0.1))
3 convOut = tf.nn.conv2d(image, flts, [1, 2, 2, 1], "Same")
4 convOut= tf.nn.relu(convOut)
5 flts2=tf.Variable(tf.normal([2, 2, 16, 32], stddev=0.1))
6 convOut2 = tf.nn.conv2d(convOut, flts2, [1, 2, 2, 1], "Same")
7 convOut2 = tf.reshape(convOut2, [100, 1568])
8 W = tf.Variable(tf.normal([1568,10],stddev=0.1))
9 prbs = tf.nn.softmax(tf.matmul(convOut2, W) + b)
```

Рис. 3.8. Первичный код, необходимый для преобразования рис. 2.2 в двухслойную сверточную NN

Когда мы линеаризуем эти выходные значения в строке 7, их будет (784 * 4). Помните, что мы начали с 784 пикселей и каждый слой свертки использовал шаг 2 как по горизонтали, так и по вертикали. Таким образом, полученные размеры трехмерного изображения после первой свертки были (14, 14, 16). Вторая свертка также имела шаг 2 на изображении 14 × 14 и имела 32 канала, поэтому выходной сигнал равен [100, 7, 7, 32], а линеаризованная версия одного изображения в строке 7 имеет 7 * 7 * 32 = 1568 скалярных значений, которые также являются высотой W, которая превращает эти значения изображения в 10 логитов.

Отстраняясь от деталей, давайте рассмотрим общий поток модели. Мы начинаем с изображения 28 * 28. В конце мы имеем “картинку” 7 * 7, но у нас

также есть 32 различных значения фильтра в каждой точке двумерного массива. Или, другими словами, в конце мы разделили изображение на 49 фрагментов; каждый фрагмент изначально был 4∗4 пикселя и теперь характеризуется 32 значениями фильтра. Поскольку это повышает производительность, мы вправе предположить, что эти значения говорят важные вещи о том, что происходит в соответствующем фрагменте 4∗4.

Действительно, похоже, что так и есть. Хотя, на первый взгляд, фактические значения в фильтрах могут сбивать с толку, по крайней мере на начальных уровнях изучения можно выявить некоторую логику в их "построении". На рис. 3.9 показаны веса 4∗4 для четырех из восьми фильтров свертки первого слоя, которые были изучены за один прогон кода на рис. 3.7. Вы можете потратить на них несколько секунд, чтобы разобрать, что они ищут. Для одних это получится. Другие не имеют особого смысла для меня. Однако взаимная корреляция с рис. 3.10 должна помочь. Рис. 3.10 был создан в результате вывода стандартного для нас теперь изображения семерки, фильтра с самым высоким значением для всех 14×14 точек в изображении после первого слоя свертки. Из тумана нулей довольно быстро проступает образ 7, поэтому фильтр 0 связан с областью всех нулей. Затем мы можем заметить, что правый край диагонали 7 в значительной степени соответствует всем семеркам, тогда как нижняя горизонтальная часть фрагментов на изображении соответствует единицам. Посмотрите еще раз на значения фильтра. Мне кажется, 1, 2 и 7 соответствуют результатам на рис. 3.9. С другой стороны, в фильтре 0 нет ничего, что могло бы предложить пустое значение. Однако это тоже имеет смысл. Мы использовали функцию `arg-max` от Numpy, которая возвращает позицию наибольшего числа в списке чисел. Все значения пикселей для пустых областей равны нулю, поэтому все фильтры возвращают 0. Если функция `arg-max` возвращает первое значение в случае, когда все значения равны, то это то, что мы ожидаем.

```
-0.152168  -0.366335   -0.464648  -0.531652
 0.0182653 -0.00621072 -0.306908  -0.377731
 0.482902   0.581139    0.284986   0.0330535
 0.193956   0.407183    0.325831   0.284819

 0.0407645  0.279199    0.515349   0.494845
 0.140978   0.65135     0.877393   0.762161
 0.131708   0.638992    0.413673   0.375259
 0.142061   0.293672    0.166572  -0.113099

 0.0243751  0.206352    0.0310258 -0.339092
 0.633558   0.756878    0.681229   0.243193
 0.894955   0.91901     0.745439   0.452919
 0.543136   0.519047    0.203468   0.0879601
```

```
 0.334673   0.252503   -0.339239  -0.646544
 0.360862   0.405571   -0.117221  -0.498999
 0.520955   0.532992    0.220457   0.000427301
 0.464468   0.486983    0.233783   0.101901
```

Рис. 3.9. Фильтры 0, 1, 2 и 7 из восьми фильтров, созданных за один проход двухслойной свертки NN

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 5 2 2 2 2 2 2 2 2 0 0
0 0 1 1 4 4 4 4 2 2 2 0 0
0 0 1 1 1 1 1 1 1 2 7 0 0
0 0 0 0 0 0 0 5 1 4 2 7 0 0
0 0 0 0 0 0 0 5 1 2 7 0 0 0
0 0 0 0 0 0 5 1 4 2 7 0 0 0
0 0 0 0 0 5 2 1 2 7 0 0 0 0
0 0 0 0 0 5 1 4 2 0 0 0 0 0
0 0 0 0 5 1 4 2 7 0 0 0 0 0
0 0 0 0 2 1 2 2 0 0 0 0 0 0
0 0 0 0 1 1 1 7 0 0 0 0 0 0
```

Рис. 3.10. Наиболее активная функция для всех 14 × 14 точек в слое 1 после обработки рис. 1.1

Рис. 3.11 аналогичен рис. 3.10, за исключением того, что он показывает наиболее активные фильтры на уровне 2 модели. Он менее интерпретируемый, чем уровень 1. Существуют различные аргументы в пользу того, почему это может иметь место. Мы включили его в основном потому, что первый сверточный слой иллюстраций гораздо более интерпретируемый, чем большинство, но мы не должны предполагать, что то, что мы видели для слоя 1, является типичным.

```
 0    0    0    0    0    0    0
17   11   31   17   17   16   16
 6   16   12    6    6    5    5
17   17   17    5   24    5   10
 0    0   11   26    3    5    0
 0   17   11   24    5   10    0
 0    6   24    8    5    0    0
```

Рис. 3.11. Наиболее активные функции для всех 7 × 7 точек в слое 2 после обработки рис. 1.1

3.4. Детали свертки

3.4.1. Смещения

Мы также можем иметь смещения с нашими сверточными ядрами. Мы не упоминали об этом до сих пор потому, что только в последнем примере несколько фильтров были применены к каждому фрагменту, т.е. мы указали 16 различных фильтров в строке 2 на рис. 3.8. Смещение может заставить программу придавать одному каналу фильтра больший или меньший вес, чем другому, добавляя другое значение к выходу свертки канала. Таким образом, число переменных смещения на конкретном сверточном слое равно количеству выходных каналов. Например, на рис. 3.8 мы можем добавить смещения к первому слою свертки, добавив следующее между строками 3 и 4:

```
bias = tf.Variable(tf.zeros [16])
convOut += bias
```

Широковещание неявно. Несмотря на то что `convOut` имеет форму [100, 14,1 4, 16], `bias` имеет форму [16], поэтому сложение неявно создает [100, 14, 14] его копий.

3.4.2. Слои со сверткой

В разделе 2.4.4 показано, как один стандартный компонент архитектуры NN, полносвязные слои, может быть эффективно написан с использованием `layers`. Для сверточных слоев есть эквивалентные функции

```
tf.contrib.layers.conv2d(inpt,numFlts, fltDim, strides, pad)
```

плюс необязательные именованные аргументы. Например, строки со 2 по 4 на рис. 3.8 можно заменить строкой

```
convOut = layers.conv2d(image,16, [4,4], 2,"Same") ?
```

На вывод свертки указывает `convOut`. Как и прежде, мы создаем 16 разных ядер, каждое размером 4∗4. Шагов в обоих направлениях два, и мы используем одинаковые дополнения. Это не совсем то же самое, что и версия без слоев, так как `layers.conv2d` предполагает, что вы хотите смещения, если не укажете обратное. Настаивая на отсутствии смещения, мы просто присваиваем значение соответствующему именованному аргументу `use_bias=False`.

3.4.3. Объединение

Как можно ожидать для больших изображений (например, 1000 * 1000 пикселей), уменьшение размера изображения от исходного до значения, подаваемого в полносвязный слой, за которым следует softmax в конце, является гораздо более значительным. Есть функции TF, которые могут помочь справиться с этим сокращением. Обратите внимание, что в нашей программе сокращение было потому, что шаги при свертке рассматривали лишь каждый следующий фрагмент. Вместо этого мы могли бы сделать следующее:

```
convOut = tf.nn.conv2d(image, flts, [1,1,1,1], "Same")
convOut = tf.nn.max_pool(convOut, [1,2,2,1], [1,2,2,1], "Same").
```

Эти две строки предназначены для замены строки 3 на рис. 3.8. Вместо свертки со вторым шагом мы сначала применили свертку с первым шагом. Таким образом, `convOut` имеет форму [batchSz, 28, 28, 1] — без уменьшения размера изображения. Следующая строка дает нам уменьшение размера изображения, точно равное тому, которое было получено с помощью шага два, который мы первоначально использовали.

Ключевая функция здесь, `max_pool`, находит максимальное значение для фильтра по области в изображении. Требуется четыре аргумента, три из четырех совпадают с `conv2d`. Первый — наш стандартный четырехмерный тензор изображений, третий — шаги, а последний — дополнения. В приведенном выше случае `max_pool` рассматривает `convOut`, четырехмерный вывод первой свертки. Это делается с шагами [1, 2, 2, 1]. Первый элемент списка указывает просмотреть каждое изображение размера партии, а последний — каждый канал. Две двойки указывают передвинуться на два блока перед повторением операции и делать так как по горизонтали, так и по вертикали. Второй аргумент задает размер области, в которой он должен найти максимум. Как обычно, первая и последняя единицы в значительной степени принудительны, в то время как средние две двойки указывают, что мы должны взять максимум из фрагмента 2 * 2 `convOut`.

На рис. 3.12 противопоставляются два разных способа, которыми мы можем добиться снижения четырехмерности в нашей программе Mnist, хотя здесь мы делаем это на изображении 4 * 4. (Числа придуманы.) В верхнем ряду мы применили фильтр со второго шага (дополнение *Same*) и сразу получили массив значений фильтра 2 * 2. В строке 2 мы применили фильтр с шага 1, что создает массив значений 4 * 4. Затем для каждого отдельного фрагмента 2 * 2 мы выводим наибольшее значение, которое дает нам окончательный массив в правом нижнем углу рисунка.

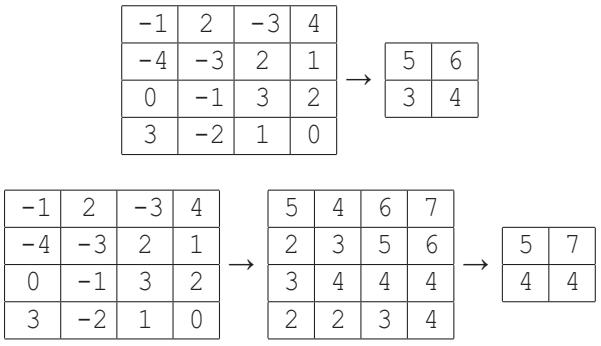

Рис. 3.12. Снижение четырехмерности с использованием `max_pool` *и без*

Прежде чем двигаться дальше, отметим, что есть также функция `avg_pool`, которая работает аналогично `max_pool`, за исключением того, что значение для пула является средним значением отдельных значений, а не максимальным.

3.5. Ссылки и что читать дальше

Вводная статья в изучение сверточных ядер с использованием NN и обратного распространения была подготовлена Яном Лекуном и другими [11], хотя гораздо более полное исследование темы принадлежит Яну Лекуну и другим [12] и является окончательной ссылкой. Часть моего образования в сверточных NN была получена из учебника Google по распознаванию цифр Mnist [13].

Если вы находите идею получить NN, способные распознавать изображения, действительно актуальной и хотите, чтобы следующий проект работал, я бы порекомендовал набор данных CFAIR 10 (Канадский институт перспективных исследований) [14]. Это также задача классификации изображений в десяти направлениях, но объекты, которые нужно распознать, более сложные (самолет, кошка, лягушка), изображения — цветные, фоны могут быть сложными, а объект, который нужно классифицировать, не отцентрирован. Размеры изображений также больше [32, 32, 3]. Набор данных можно загрузить с [15]. Общее количество изображений примерно такое же, как у Mnist, 60 000, поэтому общий импринт данных управляем. Существует также сетевое руководство от Google по созданию NN для этой задачи [16].

Если вы *действительно* амбициозны, попробуйте поработать с набором данных Imagenet Large Scale Visual Recognition Challenge (*ILSVRC*). Это намного сложнее. Существует 1000 типов изображений, в том числе такие классические, как картофель фри или картофельное пюре. За последние шесть или семь лет этот набор данных использовался серьезными исследователями компьютерного зрения на ежегодных соревнованиях. Для NN большим был год

2012, когда в конкурсе победила программа глубокого обучения *Alexnet*, т.е. впервые победила программа NN. Программа Алекса Крижевски, Ильи Суцкевера и Джеффри Хинтона [17] достигла *5 наивысших баллов* — в 15,5% случаев правильная метка не была одним из пяти лучших ответов, как было установлено оценкой программы вероятности того, что конкретная метка является правильной. Участник, занявший второе место, набрал 26,2%. С 2012 года все первые места имеют программы на базе NN.

2012	15,5
2013	11,2
2014	6,7
2015	3,6
Человек	5–10

Здесь запись “Человек” означает, что люди справляются с этой задачей в диапазоне от 5 до 10% в зависимости от обучения.

Таблицы и диаграммы с вышеприведенной информацией являются общими точками представления при объяснении влияния глубокого обучения на искусственный интеллект за последние 10 лет или около того.

3.6. Упражнения

Упражнение 3.1. а) Создайте ядро $3 * 3$, которое обнаруживает вертикальные линии на черно-белом изображении и возвращает значение 8, когда применяется к верхней левой части изображения на рис. 3.2. Оно должно возвращать нуль, если все пиксели в фрагменте имеют одинаковую интенсивность. б) Разработайте другое такое ядро.

Упражнение 3.2. В обсуждении уравнения 3.2 мы в комментарии сказали, что размер фильтра свертки не влияет на количество приложений при использовании дополнения *Same*. Объясните, как это может быть.

Упражнение 3.3. В нашем обсуждении дополнения мы говорили, что дополнение *Valid всегда* дает выходное изображение, имеющее меньшие двумерные размеры, чем входное. Строго говоря, это не так. Объясните (относительно неинтересный) случай, когда это утверждение неверно.

Упражнение 3.4. Предположим, что вход для сверточной NN является цветным изображением $32 * 32$. Мы хотим применить к нему восемь сверточных фильтров, все с формой $5 * 5$. Мы используем дополнение *Valid* и шаг 2 по вертикали и по горизонтали. а) Какова форма переменной, в которой мы храним значения фильтров? б) Какова форма вывода `tf.nn.conv2d`?

Упражнение 3.5. Объясните, что делает следующий код иначе, чем почти идентичный код в начале раздела 3.4.3?

```
convOut = tf.nn.conv2d(image, flts, [1,1,1,1], "Same")
convOut = tf.nn.maxpool(convOut, [1,2,2,1], [1,1,1,1] "Same").
```

В частности, необязательные значения `image` и `flts`, `convOut` имеют одинаковую форму, в обоих случаях? Обязательно ли им иметь одинаковые значения? Является ли один набор значений правильным подмножеством другого? Почему в каждом случае “да” или почему “нет”?

Упражнение 3.6. а) Сколько переменных создается, когда мы выполняем следующую команду `layers`?

```
layers.conv2d(image,10, [2,4], 2, "Same", use bias=False)
```

Предположим, что `image` имеет форму [100, 8, 8, 3]. Какие из этих значений формы не имеют отношения к ответу? б) Не имеет значения, сколько, если для `use_bias` установлено значение `True` (стандартно)?