Приложение № 2
к распоряжению РТУ МИРЭА
от__ _____№_____

Титульный лист материалов по дисциплине
(заполняется по каждому виду учебного материала)

ДИСЦИЛИНА Проектирование и обучение нейронных сетей
(полное наименование дисциплины без сокращений)

ИНСТИТУТ Информационые технологии

КАФЕДРА Вычислительная техника
полное наименование кафедры

ВИД УЧЕБНОГО МАТЕРИАЛА Лекция
(в соответствии с пп.1-11)

ПРЕПОДАВАТЕЛЬ Сорокин Алексей Борисович
(фамилия, имя, отчество)

СЕМЕСТР 7 семестр, 2023/2024
(указать семестр обучения, учебный год)

14. ЛЕКЦИЯ. Реализация сверточных сетей

Выбор максимального значения из соседних

В классическом сверточном слое, кроме линейной свертки и следующей за ней нелинейности, есть и еще одна операция: субдискретизация (pooling – по-русски ее иногда называют еще операцией «подвыборки», от альтернативного английского термина subsampling; встречается и слово «пулинг»)

Смысл субдискретизации прост: в сверточных сетях обычно исходят из предположения, что наличие или отсутствие того или иного признака гораздо важнее, чем его точные координаты. Например, при распознавании лиц сверточной сетью гораздо важнее понять, есть на фотографии лицо и с какого конкретно пиксела оно начинается и в каком заканчивается. Поэтому обобщают выделяемые признаки, потеряв часть информации об их местоположении, но зато сократив размерность.

Обычно в качестве операции субдискретизации к каждой локальной группе нейронов применяется операция взятия максимума (max-pooling). Иногда встречаются и другие операции субдискретизации например взятие среднего, а не максимума. Однако именно максимум встречается на практике чаще всего и для большинства практических задач дает хорошие результаты [5].

$$x_{i,j}^{l+1} = \max_{-d \le a \le d, -d \le b \le d} z_{i+a,j+b}^{l}$$

Здесь d – это размер окна субдискретизации. Как правило, шаг субдискретизации и размер окна совпадают, то есть получаемая на вход матрица делится на непересекающиеся окна, в каждом из которых выбирается максимум; для $d = 2$ эта ситуация проиллюстрирована на рис.17, в.

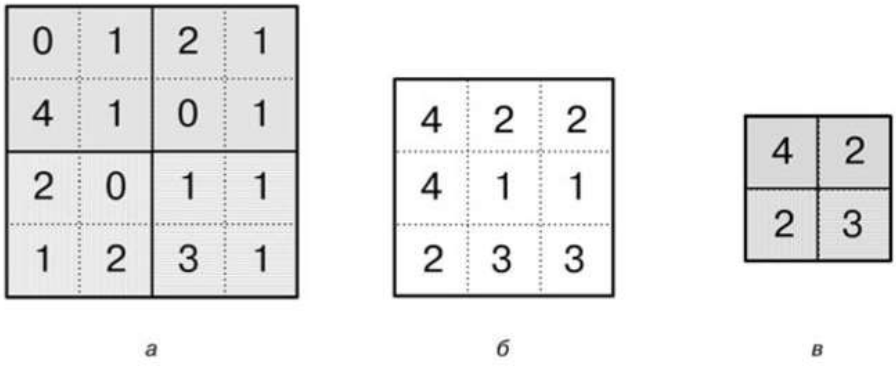

Рис. 17. Пример субдискретизации с окном размера 2×2:
а – исходная матрица; б – матрица после субдискретизации с шагом 1;
в – матрица после субдискретизации с шагом 2.

Хотя в результате субдискретизации действительно теряется часть информации, сеть становится более устойчивой к небольшим трансформациям изображения в роде сдвига или поворота.

Субдискретизация позволяет сделать представление приблизительно инвариантным относительно малых параллельных переносов входа. Инвариантность относительно параллельного переноса означает, что если сдвинуть вход на небольшую величину, то значения большинства подвергнутых пулингу выходов не изменятся (рис. 18). Инвариантность физической величины означает её неизменность при изменении физических условий или по отношению к некоторым преобразованиям [3].

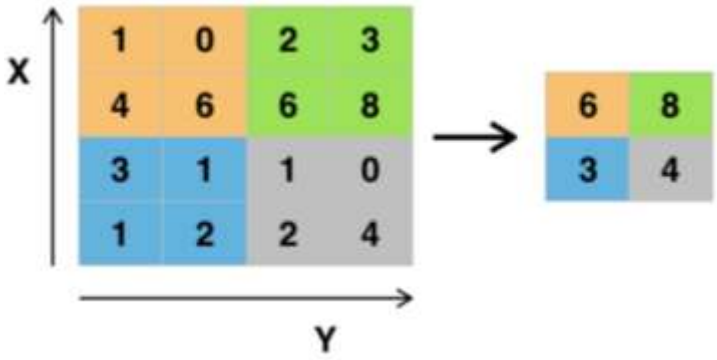

Рис. 18. Пример max-пулинга

Локальная инвариантность относительно параллельного переноса полезна, если нас больше интересует сам факт существования некоторого признака, а не его точное местонахождение (рис.19) [2].

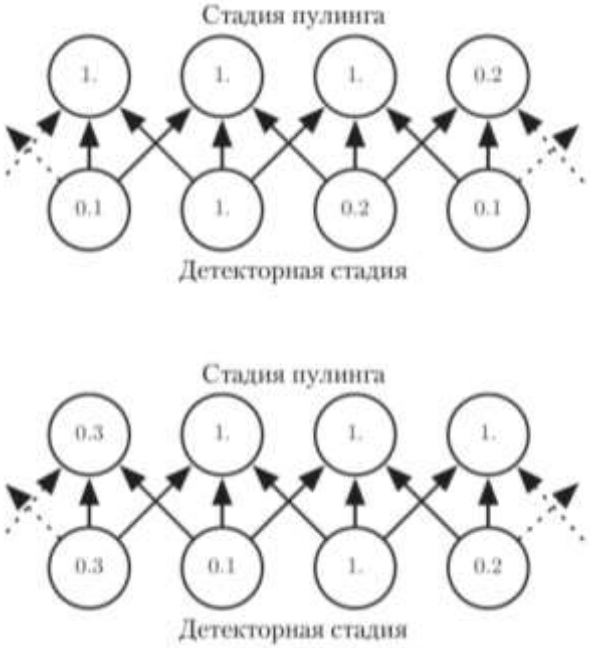

Рис. 19. Max-пулинг привносит инвариантность.

Вверху на рис.19 изображена середина выходного слоя сверточной сети. В нижней строке показаны выходы нелинейности, а в верхней – выходы max-пулинга с шагом в один пиксель между областями пулинга, каждая из которых имеет ширину три пикселя. Ввнизу. Та же самая сеть, сдвинутая вправо на один пиксель. В нижней строке изменились все значения, а в верхней – только

половина, потому что блоки max-пулинга чувствительны лишь к максимальному значению в своей окрестности, а не точному положению этого значения.

Когда определяется, присутствует в изображении лицо, не важно положение глаз с точностью до пикселя, нужно только знать, есть ли глаз слева и глаз справа. В других ситуациях важнее сохранить местоположение признака. Например, если ищется угловая точка, образованная пересечением двух границ, ориентированных определенным образом, то необходимо сохранить положение границ настолько точно, чтобы можно было проверить, пересекаются ли они.

Пулинг можно рассматривать как добавление бесконечно сильного априорного предположения, что обучаемая слоем функция должна быть инвариантна (неизменна) к малым параллельным переносам. Если это предположение правильно, то оно может существенно улучшить статистическую эффективность сети [3].

Пулинг по пространственным областям порождает инвариантность к параллельным переносам, но если он производится по выходам сверток с различными параметрами, то признаки могут обучиться, к каким преобразованиям стать инвариантными (рис.20).

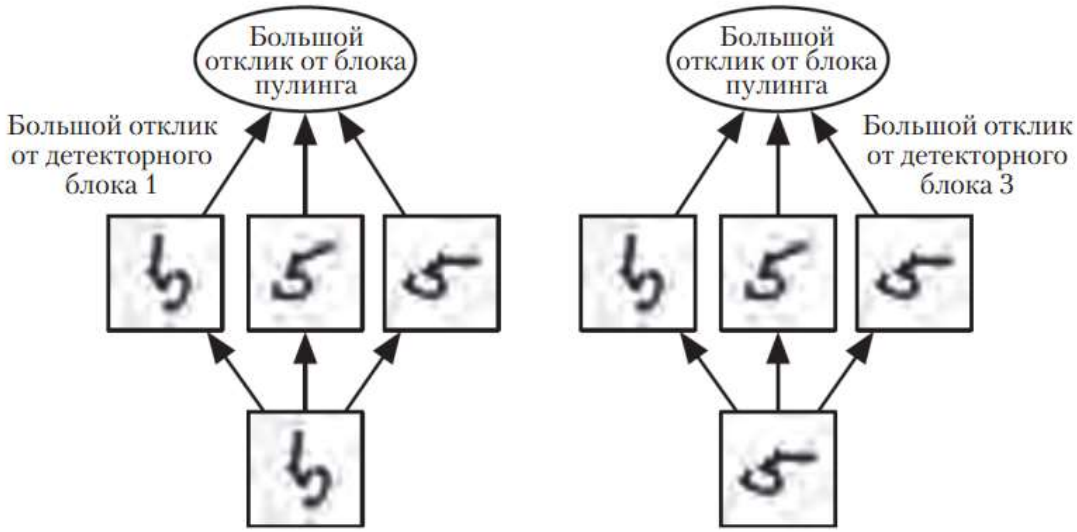

Рис. 20. Пример обученной инвариантности

Блок, выполняющий пулинг по нескольким признакам, обученным с разными параметрами, может обучиться инвариантности к преобразованиям входа. На рис. 20 видно набор из трех обученных фильтров и блок max-пулинга, который обучился инвариантности к вращению. Все три фильтра предназначены для распознавания рукописной цифры 5. Каждый фильтр настроен на свою ориентацию пятерки. Если на входе появляется цифра 5, то соответствующий фильтр распознает ее, что даст большой отклик на детекторный блок. Тогда блок max-пулинга даст большой отклик независимо от того, какой детекторный блок

был активирован. На рис. 20 показано, как сеть обрабатывает два разных входа, активирующих разные детекторные блоки. В обоих случаях выход блока пулинга примерно одинаков. Max-пулинг по пространственной области обладает естественной инвариантностью к параллельным переносам; такой многоканальный подход необходим только для обучения другим преобразованиям.

Поскольку пулинг агрегирует отклики по целой окрестности, количество блоков пулинга можно сделать меньшим, чем количество детекторных блоков, если агрегировать статистику по областям, отстоящим друг от друга на $k > 1$ пикселей. Пример приведен на рис. 21.

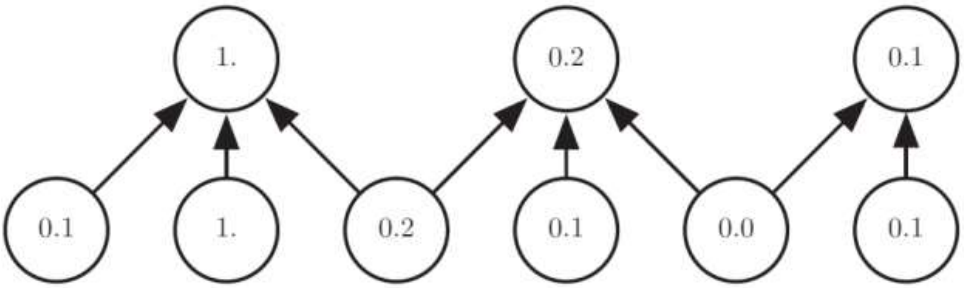

Рис. 21. Пулинг с понижающей передискретизацией.

Здесь max-пулинг используется с пулом ширины 3 и шагом 2 между пулами. В результате размер представления уменьшается вдвое, что снижает вычислительную и статистическую нагрузки на следующий слой. Отметим, что размер самой правой области пулинга меньше остальных, но ее все равно необходимо включить, если мы не хотим игнорировать некоторые детекторные блоки [2].

Тем самым повышается вычислительная эффективность сети, поскольку следующему слою предстоит обработать примерно в k раз меньше входов. Если число параметров в следующем слое – функция от размера входа (например, когда следующий слой полносвязный и основан на умножении матриц), то уменьшение размера входа также может повысить статистическую эффективность и уменьшить требования к объему памяти для хранения параметров.

В большинстве задач пулинг необходим для обработки входов переменного размера. Например, если классифицировать изображения разного размера, то код слоя классификации должен иметь фиксированный размер. Обычно это достигается за счет варьирования величины шага между областями пулинга, так чтобы слой классификации всегда получал одинаковый объем сводной статистики независимо от размера входа. Так, можно определить финальный слой пулинга в сети, так чтобы он выводил четыре сводных

статистических показателя, по одному на каждый квадрант изображения, вне зависимости от размера самого изображения.

Обучение

Таким образом стандартный слой сверточной сети (рис.22) состоит из трех компонентов [1]:

- свертка в виде линейного отображения, выделяющая локальные признаки;
- нелинейная функция, примененная покомпонентно к результатам свертки;
- субдискретизация, которая обычно сокращает геометрический размер получающихся тензоров.

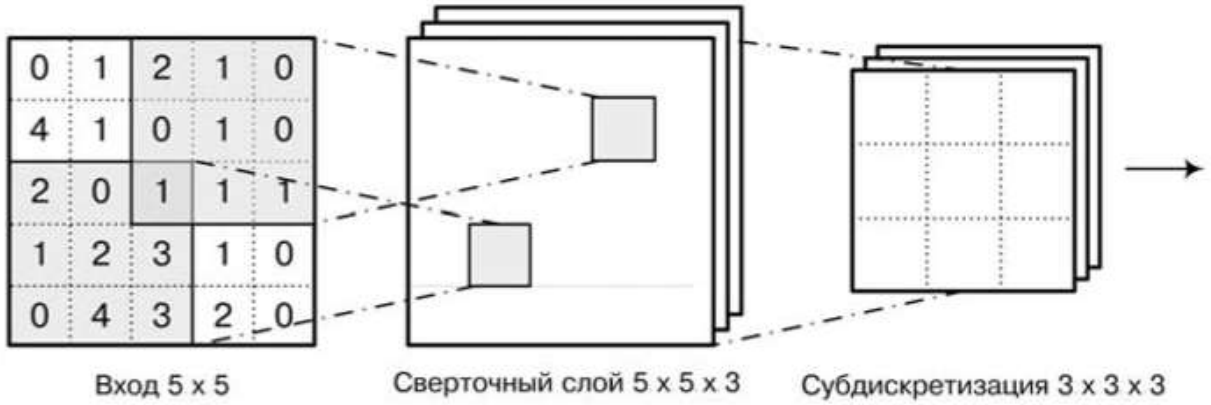

Рис. 22. Схема одного слоя сверточной сети: свертка, за которой следует субдискретизация

По сравнению с «картинкой» на входе размерность тензора увеличилась: сверточная сеть обычно обучает сразу несколько карт признаков на каждом слое (на рис. 22 таких карт три).

Теперь необходимо обучить сверточную сеть. Предположим, что необходимо оптимизировать некоторую функцию ошибки E, при этом уже известно значения на выходах нашего сверточного слоя. Чтобы провести итерацию обучения, нужно понять, как через них выражаются значения градиентов функции ошибки от весов. Через функцию взятия максимума ошибка проходит без изменений, слой субдискретизации ничего не обучает. Однако он делает проходящие по графу вычисления градиенты разреженными, ведь из всех элементов окна субдискретизации $z_{i,j}^{l}$ частная производная $\partial E / \partial x_{i,j}^{l+1}$ относится только к одному, максимальному, а остальные получат нулевой градиент, и на этом их обучение можнj законченным.

Пропуская через нелинейность получится [4]:

$$\frac{\partial E}{\partial y_{i,j}^l} = \frac{\partial E}{\partial z_{i,j}^l}\frac{\partial z_{i,j}^l}{\partial y_{i,j}^l} = \frac{\partial E}{\partial z_{i,j}^l} h \quad (y_{i,j}^l)$$

На сверточном уровне появляются веса, которые нужно обучить. Некоторая сложность здесь состоит в том, что все веса делятся, и каждый участвует во всех выходах; так что сумма получится достаточно большая:

$$\frac{\partial E}{\partial \omega_{a,b}^l} = \sum_i \sum_j \frac{\partial E}{\partial y_{i,j}^l}\frac{\partial y_{i,j}^l}{\partial \omega_{a,b}^l} = \sum_i \sum_j \frac{\partial E}{\partial z_{i+a,j+b}^{l-1}}$$

где индексы i и j пробегают все элементы картинки на промежуточном слое $y_{i,j}^l$, то есть после свертки, но до субдискретизации. Для полноты картины осталось только пропустить градиенты на предыдущий слой.

$$\frac{\partial E}{\partial x_{i,j}^l} = \sum_i \sum_j \frac{\partial E}{\partial y_{i-a,j-b}^l}\frac{\partial y_{i-a,j-b}^l}{\partial x_{i,j}^l} = \sum_i \sum_j \frac{\partial E}{\partial y_{i-a,j-b}^l}\omega_{a,b}$$

Таким образом, адаптируется процедура обратного распространения ошибки, она же обратный проход по графу вычислений, для сверточного слоя. Обратный проход для свертки оказался очень похож на, опять же, свертку с теми же весами $\omega_{a,b}$, только вместо $i + a$ и $j + b$ теперь $i - a$ и $j - b$. В случае, когда изображение дополняется нулями по необходимости и размерности сохраняются, обратный проход можно считать в точности такой же сверткой, что и в прямом проходе, только с развернутыми осями.

Для построения глубокой сети из таких слоев необходимо использовать выход очередного слоя как вход для следующего, а разные карты признаков будут служить каналами. Размер слоя за счет субдискретизации будет постепенно сокращаться, и в конце концов последние слои сети смогут «окинуть взглядом» весь вход, а не только маленькое окошечко из него.

Минимальный размер, при котором окно называемым фильтром имеет центр и выражает такие отношения, как «слева»/«справа» или «сверху»/«снизу», – это размер 3×3. Именно такой размер фильтров используется в большинстве современных сверточных архитектур. Обычно, в фильтрах, упоминается только две размерности, отвечающие за «рецептивное поле» фильтра. Но на самом деле фильтр задается четырехмерным тензором, последние две размерности которого обозначают число каналов предшествующего и текущего слоя. Так, например, при работе с цветными изображениями на вход получается три слоя, передающие соответственно красный, зеленый и синий цвета. Когда в первом сверточном слое «фильтр размера 5×5», это значит,что в первом слое есть несколько наборов весов, переводящих тензор размером $5 \times 5 \times 3$

(«параллелепипед весов») в скаляр. Фильтры размером 1×1 – это просто линейные преобразования из входных каналов в выходные с последующей нелинейностью [5].

Теперь, познакомившись со всеми видами сверток, мы можем привести полноценный пример, который упрощенно, но вполне адекватно отражает то, что происходит при реальной обработке изображений сверточными сетями (рис. 23).

Рис. 22. Пример выделения признаков на двух сверточных слоях.

Поясним рис. 22 [1]:

- каждая карта признаков первого слоя выделяет некий признак в каждом окне исходного изображения; в частности, первая карта признаков «ищет» диагональную линию из единичек, точнее, из пикселов высокой интенсивности (это значит, что она сильнее всего активируется, когда пикселы на диагонали присутствуют), вторая просто активируется на закрашенное окно изображения (чем больше пикселов с высокой интенсивностью, тем лучше), а третья аналогична первой иищет другую диагональ;
- нелинейность в сверточном слое и слой субдискретизациимы в этом примере решили пропустить;
- карта признаков на втором слое (для простоты одна) пытается найти на картинке крестик из двух диагональных линий; для этого она объединяет признаки, выделенные на первом слое, то есть свертка второго слоя – это одномерная свертка по каналам (признакам), а не по окнам в изображении; вот что свертка второго слоя «хочет увидеть» в том или ином наборе признаков:

– как можно более ярко выраженную диагональную линию из левого верхнего в правый нижний угол, то есть сильно активированный первый признак первого слоя;

– ярко выраженную линию из левого нижнего в правый верхний угол, то есть сильно активированный третий признак первого слов;

– и как можно меньше других активированных пикселов, то есть суммарная активация, выраженная во втором признаке первого слоя, играет для этой свертки в минус;

- в результате такая сеть действительно находит крестик и на исходной картинке; обратите внимание на разницу между двумя выделенными на рис. 22 окнами – в одном действительно получается крестик, а в другом кроме крестика есть еще много «мусора», и за счет второго признака нейрон, отвечающий за крестик, получает отрицательную активацию;
- а если бы субдискретизация была нетривиальной, она находила бы ещё и «псевдокрестики», в которых диагональные линии находятся рядом друг с другом, а не обязательно в одном и том же окне.

Архитектуры сверточных сетей

Одной из самых популярных глубоких сверточных архитектур (представленная в 2014 году) является модель, которую принято называть VGG. Название происходит от того, что эта модель была разработана в Оксфордском университете в группе визуальной геометрии (Visual Geometry Group). VGG – это сразу две конфигурации сверточных сетей, на 16 и 19 слоев. Основным нововведением стала идея использовать фильтры размером 3×3 с единичным шагом свертки вместо использовавшихся в лучших моделях предыдущих лет сверток с фильтрами 7×7 с шагом 2 и 11×11 с шагом 4 [1]. Причем это хорошо аргументированное предложение:

Во-первых, рецептивное поле трех подряд идущих сверточных слоев размером 3×3 имеет размер 7×7, в то время как весов у них будет всего 27, против 49 в фильтре 7×7. Аналогично обстоит дело с фильтрами 11×11. Это значит, что VGG может стать более глубокой, то есть содержать больше слоев, при этом одновременно уменьшая общее число весов. Конечно между соответствующими сверточными слоями недолжно быть слоев субдискретизации. Во-вторых, наличие дополнительной нелинейности между слоями позволяет увеличить «разрешающую способность» по сравнению с единственным слоем с большей сверткой. Этот же аргумент можно использовать как мотивацию для того, чтобы ввести в сеть свертки размером 1×1; такие слои

тоже позволяют добавить дополнительную нелинейность в сеть, не меняя размер рецептивного поля. Схема одной из VGG-сетей показана на рис.23 [2].

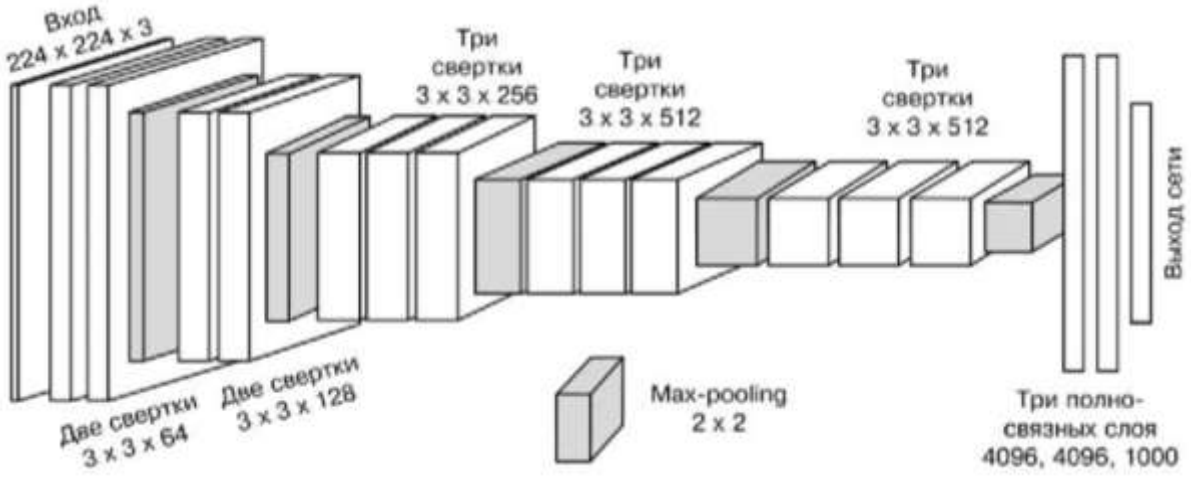

Рис. 23.Схема сети VGG-16

На рис. 23 воплощены три общих принципа [3]:

во-первых, как по две-три свертки 3×3 следуют друг за другом без субдискретизации;

во-вторых, как число карт признаков постепенно растет наболее глубоких уровнях сети;

в-третьих, как в конце полученные признаки окончательно «сплющиваются» в одномерный вектор и на нем работают последние, уже полносвязные слои. Все это стандартные методы создания архитектур глубоких сверточных сетей, и если разрабатывать свою архитектуру, стоит следовать этим общим принципам.

Следующая важная сверточная архитектура – это архитектура Inception (начало) [1]. Она была разработана в Google и появилась практически одновременно с VGG. Авторы вдохновились идеями использовать в качестве строительных блоков для глубоких сверточных сетей непросто последовательность «свертка—нелинейность—субдискретизация», как это обычно делается, а более сложными конструкциями.

В Inception такой компонент «собирают» из небольших сверточных конструкций. Так что в сети развивается идея «вложенной» архитектуры. «Строительными блоками» Inception являются модули, комбинирующие свертки размером 1×1, 3×3 и 5×5, а также max-pooling субдискретизацию. Каждый блок представляет собой объединение четырех «маленьких» сетей, выходы которых объединяются в выходные каналы и передаются в наследующий слой. Выбор набора сверток и субдискретизации обусловлен скорее удобством, чем необходимостью, и при желании можно использовать другие конфигурации.

Использование сверточных слоев 1×1 не столько в качестве дополнительной нелинейности, сколько для понижения размерности между слоями. Свертки 3 × 3 и тем более 5 × 5 между слоями с большим числом каналов (в Inception-модулях каналов может быть вплоть до 1024), оказываются крайне ресурсоемкими, несмотря на малые размеры отдельно взятых фильтров. Фильтры 1×1 могут помочь сократить число каналов, прежде чем подавать их на фильтры большего размера. Эта идея отражена на рис. 24, а, на котором показана структура одного блока из исходной работы [1]. На рис. 24, б представлена очень общая и высокоуровневая схема всей сети: она начинается с двух «обычных» сверточных слоев, а затем идут 11 Inception-модулей, дважды перемежаемых субдискретизацией, которая понижает размерность; после этого сеть завершается традиционными полносвязными слоями, дающими уже собственно выход классификатора.

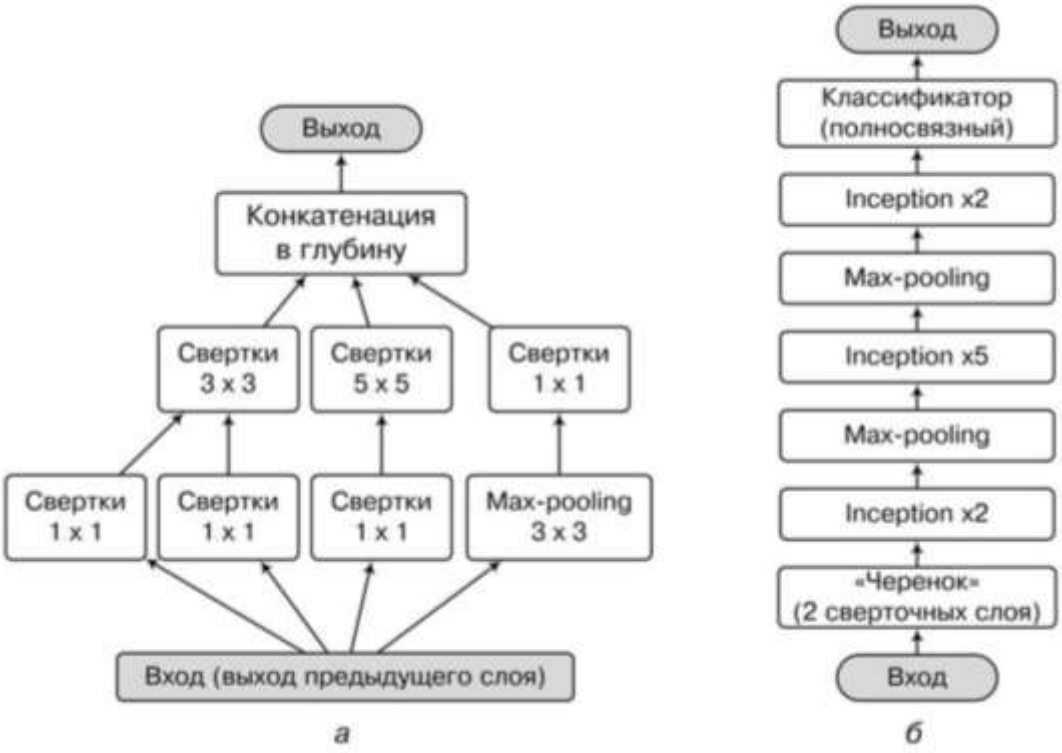

Рис. 24. Inception: а – схема одного Inception-модуля;
б –общая схема сети GoogLeNet

С учетом достаточно глубокой архитектуры сети GoogLeNet – в общей сложности она содержит порядка 100 различных слоев с общей глубиной в 22 параметризованных слоя, или 27 слоев с учетом субдискретизаций – эффективное распространение градиентов по ней вызывает сомнения. Чтобы решить эту проблему, авторы предложили добавить вспомогательные классифицирующие сети поверх некоторых промежуточных слоев. Иначе

говоря, добавляются две новые небольшие полносвязные сети, делающие предсказания на основе промежуточных признаков.

Как показала практика, стало возможно глубокие архитектуры обучать эффективно, однако те решения, к которым сходились нейронные сети большой глубины, часто оказывались хуже, чем у менее глубоких моделей. И эта «деградация» не была связана с переобучением, как можно было бы предположить. Оказалось, что это более фундаментальная проблема: с добавлением новых слоев ошибка растет не только на тестовом, но и на тренировочном множестве.

Для решения проблемы деградации команда из Microsoft Research разработала новую идею: глубокое остаточное обучение (deep residual learning), которое легло в основу сети ResNet [1]. В базовой структуре новой модели нет ничего нового: это слои, идущие последовательно друг за другом. Отдельные уровни, составные блоки сети тоже выглядят достаточно стандартно, это просто сверточные слои, обычно с дополнительной нормализацией. Разница в том, что в остаточном блоке слой из нейронов можно «обойти»: есть специальная связь между выходом предыдущего слоя $x^{(\mathrm{k})}$ и следующего слоя $x^{(\mathrm{k+1})}$, которая идет напрямую, не через вычисляющий слой. Базовый слой нейронной сети на рис. 25, а превращается в остаточный блок с обходным путем на рис. 25, б. Математически происходит очень простая вещь: когда два пути, «сложный» и «обходной», сливаются обратно, их результаты просто складываются друг с другом. И остаточный блок выражает такую функцию:

$$y^{(k)} = F(x^{(k)}) + x^{(k)},$$

где $x^{(k)}$ – входной вектор слоя k, F(x) – функция, которую вычисляет слой нейронов, а $y^{(\mathrm{k})}$ – выход остаточного блока, который потом станет входом следующего слоя $x^{(k+1)}$.

Получается, что если блок в целом должен апроксимировать функцию $H(x)$, то это достигается тогда, когда $F(x)$ аппроксимирует остаток $H(x) - x$, отсюда и название остаточные сети (residual networks). В остаточном блоке обучается слой нейронов воспроизводить изменения входных значений, необходимые для получения итоговой функции.

Преимущества [1]: Во-первых , часто получается, что обучить «остаточную» функцию проще, чем исходную; Оказывается, что с помощью двухслойной нелинейной нейронной сети выучить тождественную функцию достаточно сложно; в то же время в остаточной форме от сети требуется просто заполнить все веса нулями, а «обходной путь» сделает всю работу сам.

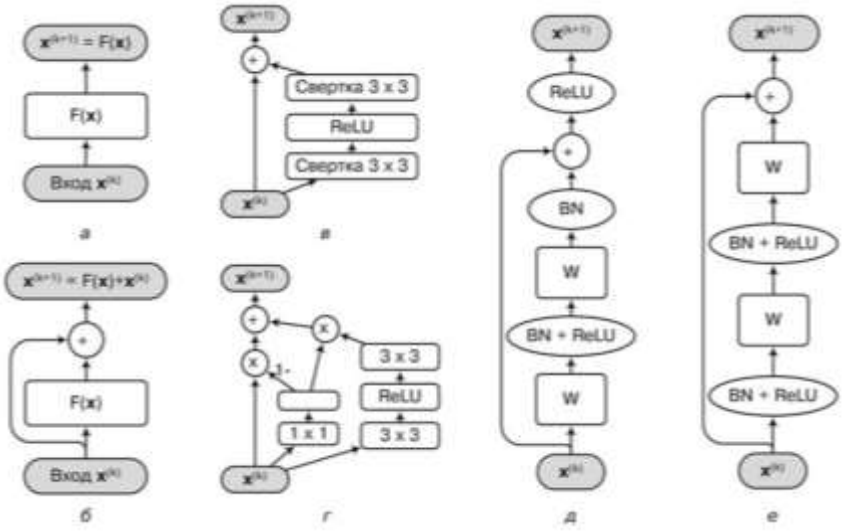

Рис. 25. Блоки сетей для остаточного обучения: а – базовый блок; б – такой же блок с остаточной связью; в – простой блок с остаточной связью; г – блок с остаточной связью, контролирующийся гейтом; д – блок с остаточной связью исходной сети ResNet; е –более поздняя модификация.

Главная же причина состоит в том, что градиент во время обратного распространения может проходить через этот блок беспрепятственно, градиенты не будут затухать, ведь всегда есть возможность пропустить градиент напрямую:

$$\frac{\partial y^{(k)}}{\partial x^{(k)}} = 1 + \frac{\partial F(x^{(k)})}{\partial x^{(k)}}$$

Это значит, что даже насыщенный и полностью обученный слой F, производные которого близки к нулю, не помешает обучению.

Примеры таких «обходных путей», которые использовались в разных вариантах сети ResNet и других исследованиях этой группы авторов, показаны на рис. 25. Проводилось подробное сравнение нескольких вариантов остаточных блоков. Остаточные блоки, которые теоретически должны быть более выразительными, на практике оказываются хуже, чем самые простые варианты.

На рис. 25, в изображен очень простой вариант остаточного блока, в котором $y^{(k)} = F(x^{(k)}) + x^{(k)}$, а на рис. 25, г – вариант посложнее:

$$y^{(k)} = \left(1 - \sigma\left(f(x^{(k)})\right)\right) x^{(k)} + \sigma\left(f(x^{(k)})\right) F(x^{(k)})$$

где f –другая функция входа, реализованная через свертки 1×1.Это значит, что $F(x^{(k)})$ и $x^{(k)}$ суммируются не с равными весами, а с весами, управляемыми дополнительным «гейтом». Казалось бы, простой вариант является частным случаем сложного: достаточно просто обучить гейты так, чтобы веса были равными, то есть чтобы всегда выполнялось $f(x^{(k)}) = 0$. Однако эксперименты в показали, что простота в данном случае важнее выразительности и важно

обеспечить максимально свободное и беспрепятственное течение градиентов. На рис. 25, д показан вариант остаточного блока, а на рис. 25, е – улучшенный вариант. Обратите внимание, что разница, по сути, только в том, что из «обходного пути» убрали ReLU-нелинейность, последнее «препятствие» на пути значений с предыдущего слоя [1].

Правда, вне которых приложениях от них отказываются ради скорости и экономии ресурсов: большая сверточная сеть с остаточными связями никак не поместится в смартфон. Если ресурсы важны, стоит посмотреть в сторону моделей, которые показывают немного более слабые результаты в собственно распознавании, но имеют при этом на порядок меньше весов; выделим, в частности, MobileNets и SqueezeNet.

В заключении отметим, что идея про гейт, управляющий остаточными блоками, как на рис. 25, г, все-таки оказалась довольно плодотворной. Именно она легла в основу так называемых магистральных сетей (highway networks), предложенных группой Юргена Шмидхубера [1]. Идея магистральных сетей именно такая, представляем $y^{(k)}$, выход слоя k, как линейную комбинацию входа этого слоя $x^{(k)}$ и результата $F(x^{(k)})$, веса которой управляются другими преобразованиями [1]:

$$y^{(k)} = C\left(x^{(k)}\right)x^{(k)} + T(x^{(k)})F(x^{(k)})$$

где C – это гейт переноса, а T – гейт преобразования; обычно комбинацию делают выпуклой, $C = 1 - T$. Магистральные сети тоже позволили обучать очень глубокие сети с сотнями уровней, а затем эта конструкция была адаптирована для рекуррентных сетей. Что здесь победит — простота или выразительность — вопрос пока открытый, но в любом случае варианты этих архитектур уже позволили сделать беспрецедентно глубокие сети, и останавливаться на достигнутом исследователи не собираются.

Библиотеки

В последние годы обучение нейронных сетей превратилось в высшей степени инженерную дисциплину. Появилось несколько очень удобных библиотек, которые позволяют буквально в пару строк кода построить модель нейронной сети (сколь угодно глубокой), сформировать для нее граф вычислений, автоматически подсчитать градиенты и выполнить процесс обучения, причем сделать это можно как на процессоре, так и – совершенно прозрачным образом, изменив пару строк или пару ключей при запуске – на видеокарте, что обычно ускоряет обучение в десятки раз. С каждым годом эти библиотеки становятся все удобнее, выходят новые версии существующих, а также абсолютно новые программные продукты. Есть библиотеки общего

назначения, которые могут создать любой граф вычислений, и специализированные надстройки, которые реализуют разные компоненты нейронных сетей.

Библиотеки, которые используются для реализации сверточных нейронных сетей, имеют основной интерфейс к языку программирования Python. Этот язык стал фактически стандартом в современном машинном обучении и обработке данных.

Keras (*https://keras.io*) – это фреймворк поддержки глубокого обучения для Python, обеспечивающий удобный способ создания и обучения практически любых моделей глубокого обучения. Первоначально Keras разрабатывался для исследователей с целью дать им возможность быстро проводить эксперименты. Keras обладает следующими ключевыми характеристиками [4]:

- позволяет выполнять один и тот же код на CPU или GPU;
- имеет дружественный API, упрощающий разработку прототипов моделей глубокого обучения;
- включает в себя встроенную поддержку сверточных сетей (для распознавания образов), рекуррентных сетей (для обработки последовательностей) и все возможных их комбинаций;
- включает в себя поддержку произвольных сетевых архитектур: моделей с множественными входами или выходами, совместное использование слоев, совместное использование моделей и т. д. Это означает, что Keras подходит для создания практически любых моделей глубокого обучения, от порождающих состязательных сетей до нейронной машины Тьюринга.

Фреймворк Keras распространяется на условиях свободной лицензии и может бесплатно использоваться в коммерческих проектах. Он совместим с любыми версиями Python. Keras используется в Google, Netflix, Uber, CERN, Yelp, Square и сотнях стартапов, решающих широкий круг задач.

Keras – это библиотека уровня модели, предоставляющая высокоуровневые строительные блоки для конструирования моделей глубокого обучения. Она не реализует низкоуровневые операции, такие как манипуляции с тензорами и дифференцирование, – для этого используется специализированная и оптимизированная библиотека поддержки тензоров. При этом Keras не полагается на какую-то одну библиотеку поддержки тензоров, а использует модульный подход (рис. 26); то есть к фреймворку Keras можно подключить несколько разных низкоуровневых библиотек [5].

Рис. 26. Программно-аппаратный стек поддержки глубокого обучения

В настоящее время поддерживаются три такие библиотеки: TensorFlow, Theano и Microsoft Cognitive Toolkit (CNTK). В будущем Keras, скорее всего, будет расширен еще несколькими низкоуровневыми механизмами поддержки глубокого обучения.

TensorFlow, CNTK и Theano – это одни из ведущих платформ глубокого обучения в настоящее время. Theano (*http://deeplearning.net/software/theano*) разработана в лаборатории MILA Монреальского университета, TensorFlow (*www.tensorflow.org*) разработана в Google, а CNTK (*https://github.com/Microsoft/CNTK*) разработана в Microsoft. Любой код, использующий Keras, можно запускать с любой из этих библиотек без необходимости менять что-либо в коде: можно легко переключаться между ними в процессе разработки, что часто оказывается полезно, например, если одна из библиотек показывает более высокую производительность при решении данной конкретной задачи. Рекомендуется по умолчанию использовать библиотеку TensorFlow как наиболее распространенную, масштабируемую и высококачественную [5].

Среди библиотек общего назначения, которые способны строить граф вычислений и проводить автоматическое дифференцирование, долгое время бесспорным лидером была Theano.. Однако в ноябре 2015 года Google выпустила (с открытым исходным кодом) библиотеку TensorFlow [3], предназначенную для таких же целей. TensorFlow стала вторым поколением библиотек глубокого обучения в Google. В TensorFlow реализован полный набор операций над тензорами из NumPy с поддержкой матричных вычислений над массивами разной формы и конвертирования между этими формами. Библиотеки Theano и TensorFlow на данный момент остаются двумя бесспорными лидерами в этой области, и сложно уверенно рекомендовать одну из них [5].

Используя TensorFlow (Theano или CNTK), Keras может выполнять вычисления и на CPU, и на GPU. При выполнении на CPU TensorFlow сама использует низкоуровневую библиотеку специализированных операций с тензорами, которая называется Eigen (*http://eigen.tuxfamily.org*). При выполнении на GPU TensorFlow использует оптимизированную библиотеку под названием NVIDIA CUDA DeepNeural Network (cuDNN).