СВЕРТОЧНЫЕ СЕТИ НА PYTHON

Методические указания
к выполнению практических работ
для студентов 2-го курса магистратуры, обучающихся по образовательным программам
направления 09.04.03 «Прикладная информатика» всех форм обучения

Воронеж 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«Воронежский государственный технический университет»

Кафедра инноватики и строительной физики имени профессора И.С. Суровцева

СВЕРТОЧНЫЕ СЕТИ НА PYTHON

Методические указания
к выполнению практических работ
для студентов 2-го курса магистратуры, обучающихся по образовательным программам
направления 09.04.03 «Прикладная информатика» всех форм обучения

Воронеж 2022

УДК 519.85(07)
ББК 22.18.я73

Составители:
П. А. Головинский, А.С. Тарасова, А. О. Шаталова, Э.И. Еникеев.

СВЕРТОЧНЫЕ СЕТИ НА PYTHON: методические указания к выполнению практических работ по дисциплине «Глубокое обучение с Python» для студентов 2-го курса магистратуры, обучающихся по образовательной программе «Технологии искусственного интеллекта» направления 09.04.03 «Прикладная информатика» /ФГБОУ ВО «Воронежский государственный технический университет»; сост.: П.А. Головинский, А.С. Тарасова, А.О. Шаталова, Э.И. Еникеев. - Воронеж, 2022. – 37 с.

Содержат комплекс требований к содержанию, порядку выполнения и оформлению практических работ студентов магистрантов в соответствии с перечнем общекультурных, общепрофессиональных и профессиональных компетенций, осваиваемых в процессе обучения. Описан алгоритм реализации автокодировщика в виде многосвязной нейросети для удаления шума в изображении цифр для самостоятельного выполнения по дисциплине «Глубокое обучение с Python».

Предназначены для студентов, обучающихся по образовательной программе «Технологии искусственного интеллекта» направления 09.04.03 «Прикладная информатика» всех форм обучения.

Ил. 10. Табл.1. Библиогр.: 8 назв.

УДК 519.85(07)
ББК 22.18.я73

Печатается по решению редакционно-издательского совета Воронежского государственного технического университета

***Рецензент** – С. А. Баркалов, д.т.н., профессор кафедры управления строительством Воронежского государственного технического университета*

ВВЕДЕНИЕ

Сверточные нейронные сети (convolutional neural networks, CNN) являются распространенным классом искусственных нейронных сетей. Их архитектура в современной форме была предложена группой Яна Лекуна в 1988 году. Сама идея появилась раньше и была стимулирована исследованиями в области нейрофизиологии. Современные технологии сверточных нейросетей хорошо развиты и имеют большую область применения. Сверточные нейронные сети относятся к искусственным нейронным сетям глубокого обучения, **представляющего раздел** машинного обучения, нацеленный на обработку больших наборов входных данных.

Из известных типов архитектуры искусственны нейронных сетей, сверточные сети лучше других обрабатывают многомерную (тензорную) информацию с целью распознавания или улучшения данных. Основной сферой применения сверточных нейронных сетей является обработка изображений. Принцип их работы аналогичен работе зрительной коры головного мозга. Отметим, что зрительная кора является частью коры больших полушарий головного мозга и сосредоточена по большей части в затылочной области, выполняя функции обработки визуальной информации.

1. МАТЕМАТИЧЕСКАЯ ОПЕРАЦИЯ СВЕРТКИ

Операция свертки является одним из важных понятий математического анализа. Свертка используется в различных областях науки и техники. В нейросетях она помогает распределять обрабатываемые данные, и при этом находить их общие параметры. Свертку принято обозначать звездочкой ($*$). В математическом анализе она применяется к двум функциям $\boldsymbol{f}(\boldsymbol{x})$ и $\boldsymbol{g}(\boldsymbol{x})$. Результатом ее применения является функция взаимной корреляции:

$$h(x) = f(x) * g\,(x). \qquad (1)$$

Для дискретных функций с областью определения x от 1 до n свертка вычисляется как сумма

$$h(x) = (f * g)(x) = f(1)g(x-1) + f(2)g(x-2) + \cdots + f(n)g(x-n)\,. \quad (2)$$

В дискретном случае свертка является суммой произведений функции f от смещения аргумента $k \in [1, n]$ по оси x на отраженную и смещенную на тот же коэффициент k функцию $g(x)$. Если функции определены на промежутке $[1, n]$, то k принимает все значения от 1 до n.

Свертку можно интерпретировать как степень схожести функции $f(x)$ с отраженной и сдвинутой копией функции $g(x)$. В общем виде для дискретных функций с областью определения от $-\infty$ до $+\infty$, свертку можно выразить как:

$$(f * g)(x) = \sum_{k=-\infty}^{+\infty} f\,(k)g\,(x-k). \qquad (3)$$

Для непрерывных функций она определяется как интеграл от произведения двух функций после того, как вторая реверсируется и смещается:

$$(f * g)(x) = \int_{-\infty}^{+\infty} f\,(k)g\,(x-k)\,dk. \qquad (4)$$

Понятие свертки обобщается для функций, определенных на произвольных измеримых пространствах и может рассматриваться как особый вид интегрального преобразования.

Слой нейросети, в котором выполняется операции свертки, называется сверточным слоем. В отличие от обычных нейронных сетей, сверточные нейронные сети используют корреляцию между смежными входами данных.

Такой тип обработки хорошо подходит для данных с сеточной топологией. Примерами таких данных могут быть:

– временные ряды, которые можно рассматривать как одномерный массив примеров, выбираемых через регулярные промежутки времени;

– цифровые изображения, рассматриваемые как двумерные сетки пикселей.

В приложениях машинного обучения входом обычно является многомерный массив данных, а ядром (второй функцией свертки) является многомерный массив параметров, адаптированных алгоритмом обучения. Будем называть эти массивы тензорами, поскольку тензор – объект линейной алгебры, линейно преобразующий элементы одного линейного пространства в элементы другого. На практике формальную сумму от $-\infty$ до $+\infty$ можно заменить суммированием по конечному числу элементов массива. Часто используют свертки сразу по нескольким осям. Например, если входом является двумерное изображение I, то ядро K тоже должно быть двумерным:

$$S(i,j) = (I * K)(i,j) = \sum_{i=1} \sum_n I(m,n)K(i-m, j-n). \quad (5)$$

Операция двумерной свертки коммутативна, поэтому формулу (5) можно записать в виде:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \quad (6)$$

Как правило, формулу (6) проще реализовать в библиотеке машинного обучения, поскольку диапазон допустимых значений m и n меньше, чем диапазон i и j. В реализации нейронных сетей коммутативность обычно роли не играет. Вместо этого во многих библиотеках реализована родственная функция – перекрестная корреляция, т.е. та же свертка, только без отражения ядра:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \quad (7)$$

В машинном обучении алгоритм ставит найденные значения ядра в правильную позицию, поэтому если алгоритм основан на свертке с отражением ядра, то он обучит ядро, отраженное относительно того, которое обучено алгоритмом со сверткой без отражения [5].

2. МАТРИЧНЫЕ ВЫЧИСЛЕНИЯ СВЕРТКИ

На рис. 1 графически представлен пример операции свертки без отражения ядра на двумерном входном тензоре $X_{5\times5}$, с шагом равным единице. В качестве ядра выступает матрица $W_{3\times3}$. В результате получается двумерный тензор $S_{3\times3}$ (карта признаков) как результат свертки матриц:

$$S = X * W\,, \tag{8}$$

$$S = \begin{pmatrix} x_{11} & \cdots & x_{15} \\ \vdots & \ddots & \vdots \\ x_{51} & \cdots & x_{55} \end{pmatrix} * \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix} = \begin{pmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{pmatrix}. \tag{9}$$

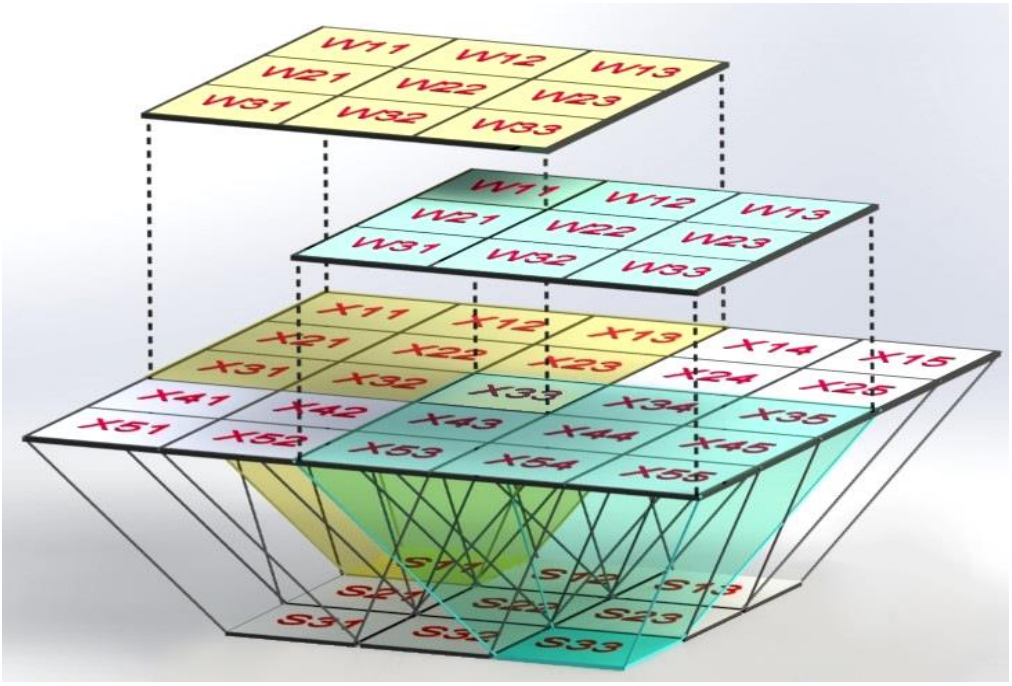


Рис. 1. Двухмерная свертка

Ядро свертки накладывается на входной тензор с левого верхнего угла, согласно нумерации, и двигается в нижний правый угол. Таким образом, при вычислении по формуле (9) произошло девять наложений с размещением центра на элементах $x_{22} : x_{44}$. В частности, первый элемент s_{11} выходного тензора, полученный наложением ядра по центру элемента x_{22}, имеет вид:

$$s_{11} = x_{11}w_{11} + x_{12}w_{12} + x_{13}w_{13} + x_{21}w_{21} + x_{22}w_{22} + x_{23}w_{23} + x_{31}w_{31} + \\ + x_{32}w_{32} + x_{33}w_{33}. \tag{10}$$

Одним из популярных направлений использования сверточной операции является фильтрация изображений. Например, для повышения резкости изображения используется ядро свертки (фильтр):

$$W = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}. \tag{11}$$

Допустим, что входной тензор – это монохромное изображение зеленого цвета, и нужно повысить его резкость. Приведя в соответствие каждой

интенсивности цвета, используемой в изображении, определенное числовое значение, получим входной тензор X (см. рис. 2 а, б).

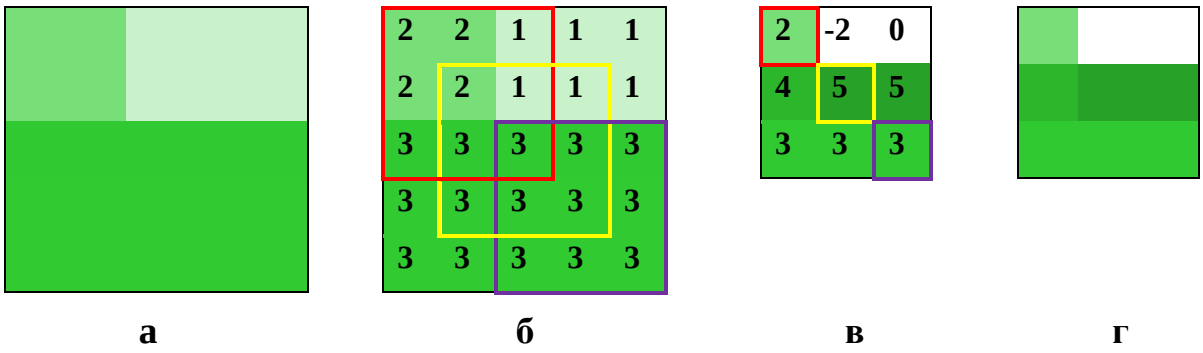


Рис.2. Свертка с повышением контрастности: а) исходное изображение; б) входной тензор; в) выходной тензор; г) изображение после свертки

Шаг сверточной операции может принимать и другие значения, отличные от единицы. При шаге равном двум, после x_{22} центрами наложения сверточного ядра станут точки x_{24} и x_{42} по горизонтали и вертикали соответственно. Таким образом, для шага свертки (2, 2) выходной тензор S, из примера рис. 2, будет состоять из четырех элементов. В некоторых задачах шаг свертки, при обработке двухмерного тензора, по горизонтали и вертикали может отличаться. Также возможно добавление нулевых элементов по всему периметру входного тензора, такая свертка называется сверткой с отступом. Большой отступ в свертке может использоваться для лучшей обработки краев, если они обладают повышенной информативностью.

При увеличении изображения требуется повышение его дискретности. В таком случае применяется транспонированная (растягивающая) свертка. Она используется, например, в семантической сегментации для автокодировщика. Сначала извлекаются карты характеристик с уменьшение размера изображения в кодировщике, а затем восстанавливается исходный размер изображения в декодере. Целью процедуры является улучшение исходного изображения за счет корректировки пикселей.

У транспонированной свертки может появляться такой недостаток, как эффект шахматной доски. Он возникает из-за участия разного количества элементов входного тензора в формировании элементов выходного тензора. Те элементы из выходного тензора, которые имеют больше связей с основными элементами входного тензора, а не с нулевыми элементами, получают большее значение, чем соседние, имеющие большое количество соединений с нулевыми элементами. Как результат, в выходных данных образуется «рябь». Большое количество таких бракованных участков появляется, когда размер сверточного ядра не делится нацело на шаг. В результате происходит неравномерное

перекрытие наложений ядра. Фактически, неравномерно перекрывающаяся область имеет тенденцию быть более экстремальной в двух измерениях, так как там два шаблона умножаются вместе, а неравномерность возводится в квадрат. Для уменьшения таких «бракованных» участков желательно взять шаг свертки равный единице, или такой, которому кратно сверточное ядро.

В свертках, рассмотренных выше, размерность ядра тождественна его дальности обзора. Существует тип свертки, у которой увеличение дальности обзора ядра может достигаться без увеличения количества его параметров (значащих элементов). Такую свертку называют дилатационной (растяженной). Достигается она за счет вставки нулевых элементов (пробелов) не во входной тензор, как описывалось ранее, а в сверточное ядро, которое при этом как бы раздувается. Идея такой свертки заключается в наложении ядра на большую область входного тензора, чем размерность самого ядра. В стандартной реализации идет чередование через один основных и нулевых элементов.

При свертке двухмерного тензора с квадратным ядром доступна процедура разделения сверточной операции на два этапа: свертку по высоте и ширине. Свертка с двумя такими отдельными операциями называется пространственно-отделимой. Например, свертку с ядром размера 3×3, можно заменить на две свертки размерами 3×1 и 1×3, которые выполняются последовательно. Здесь в первом варианте мы имеем 9 параметров (весов) ядра, а во втором – 6, т.е. 3 параметра от первой операций и 3 от второй. Обычная свертка для входного тензора 5×5 (см. рис.1), состоит из 9-ти наложений ядра по 9 элементов каждое, и 81 раз используется операция умножения. Теперь для первого ядра (вектора-столбца) получим по 3 наложения на каждую из 5-ти строк, в каждом из которых происходит умножение по 3-м элементам. Итого 45 раз использовалась операция умножения, и на выходе имеем матрицу 3×5, а для второго ядра – по 3 наложения на каждую из 3-х строк, в каждом из которых происходит умножение по 3-м элементам. Здесь 27 раз использовалась операция умножения и на выходе имеем матрицу 3×3. В общей сложности, с помощью пространственно-отделимой свертки вместо 81-й операции умножения мы использовали 72. При большом размере входного тензора, такое преимущество может иметь значение. Заметим, что свертка по высоте всегда выполняется первой. Так, при умножении вектора-столбца X на вектор-строку Y с таким же количеством элементов n мы получим квадратную матрицу размерностью $n \times n$. Например, если:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad Y = (y_1 \quad y_2 \quad y_3), \tag{12}$$

то

$$X \cdot Y = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \cdot (y_1 \quad y_2 \quad y_3) = \begin{pmatrix} x_1 y_1 & x_1 y_2 & x_1 y_3 \\ x_2 y_1 & x_2 y_2 & x_2 y_3 \\ x_3 y_1 & x_3 y_2 & x_3 y_3 \end{pmatrix}. \quad (13)$$

При умножении вектора-строки на вектор-столбец мы получим число, равное сумме поэлементных произведений исходных векторов:

$$Y \cdot X = (y_1 \quad y_2 \quad y_3) \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = y_1 x_1 + y_2 x_2 + y_3 x_3. \quad (14)$$

Свертки, которые рассматривались выше, являлись двумерными и одноканальными. Даже последний вариант, где ядро представляется вектором столбцом и вектором строкой, – это тоже две 2-мерные свертки. В процессе их выполнения ядро «скользит» по входному тензору (плоскости) в 2-х направлениях и умножается на входные данные, так что выходные данные преобразуются в новый двумерный тензор. Для двумерных входных данных чаще всего используют ядра 3×3 или 1×1, реже 5×5 или 7×7, но могут быть и другие размеры. Как правило, для существующих архитектур нейронных сетей эти размерности наиболее используемые.

Можно сказать, что размерность ядра не должна превышать размерности входного тензора, и ядро должно охватывать все направления в структуре входных данных. Далее количество направлений в структуре данных тензора будет обозначаться как его мерность, а количество позиций по каждому направлению структуры данных, как его размерность.

Двумерный входной тензор $n \times m$ может принимать различные размеры, где $n, m \geq 2$ и $n,\ m \in N$. Это может быть, к примеру, и 5×8, или 900×700 и т.д. Если n или m взять равным единице, то получится ряд чисел или одномерный тензор, который используется для обработки временных рядов с помощью одномерной свертки, как в примере о передаче сигнала.

Входные данные для изображения не всегда поступают в виде одной двумерной матрицы. Если это цветное изображение в системе RGB, то на входе будет 3 матрицы, называемые входными каналами, каждая из которых передает интенсивность одного из 3-х цветов: синий, красный, зеленый. Эти матрицы будут сопоставлены друг с другом порядком пикселей (см. рис.3). Здесь прослеживается аналогия с процессом ретинотопического картирования в зрительной системе человека.

Для свертки многоканальных 2-мерных данных необходимо использовать 2-мерные ядра в количестве равном числу каналов. Совокупность этих ядер называется фильтром. В зависимости от количества каналов, их может быть и 2 и 10, а операция свертки будет считаться двумерной, так как у фильтра только два направления движения (вверх – вниз, вправо – влево). По 3-му направлению (между каналами цвета) фильтр не перемещается, так как каждый

канал взаимодействует со своим ядром. Таким образом, свертка цветного изображения в системе RGB, в силу 2-мерного перемещения фильтра, является 2-мерной на 3-мерных входных данных. Результаты вычислений от разных ядер в одном фильтре могут объединяться на выходе согласно их пиксельной структуре в 2-мерный выходной тензор.

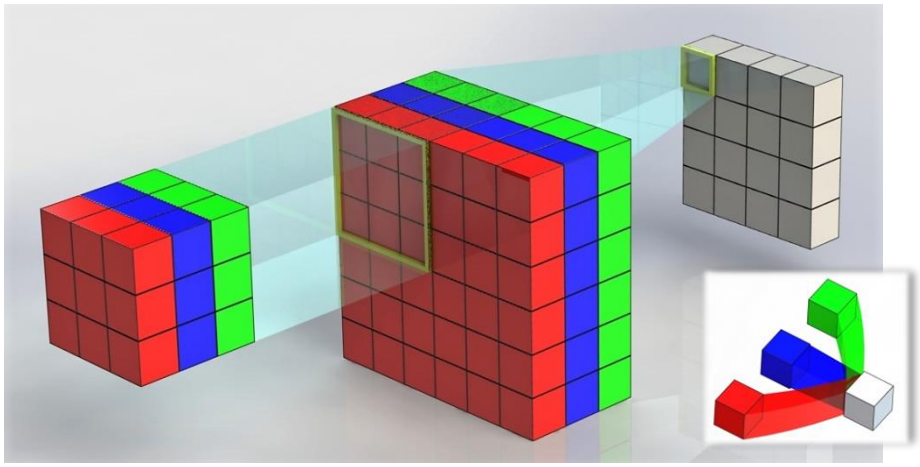


Рис.3. Многоканальная двухмерная свертка с ядром (3, 3, 3) и шагом (1, 1)

3. СВЕРТОЧНАЯ НЕЙРОННАЯ СЕТЬ ДЛЯ ВОССТАНОВЛЕНИЯ ДАННЫХ

В этом разделе рассмотрим автокодировщик [6] для восстановления изображений цифр, т.е. для устранения шумов (помех), не входящих в обучающую выборку. Автокодировщик реализуется с использованием библиотек PyTorch и Keras на языке программирования Python 3.9.

Автокодировщик (автоэнкодер) сверточного типа, представляет собой искусственную нейронную сеть прямого распространения для восстановления входного сигнала. Такая искусственная нейронная сеть является моделью обучения с самоконтролем, и реализует процедуру сжатия входных данных во внутреннем слое сети.

Сверточной тип слоев дает возможность учета пространственных взаимосвязей данных (в рассматриваемом случае в двух направлениях, по высоте и ширине). В силу уменьшения числа связей в сверточных искусственных нейронных сетях, возрастает скорость обработки данных.

Автокодировщик сначала кодирует входной сигнал в некоторое скрытое состояние, размерность которого, как правило, меньше размерности входного сигнала, а затем, декодирует данные в новое состояние выходного сигнала. Сигнал в выходном слое сети должен являться, по тем или иным критериям, улучшенной версией входного сигнала. Сеть автокодировщика должна учиться отбирать наиболее важные признаки в данных, чтобы минимизировать ошибки

в сигнале из-за потерь при кодировании. Структурная схема автокодировкщика показана на рис. 4.

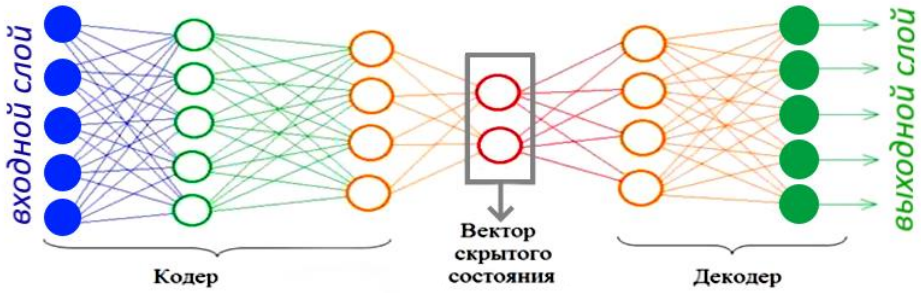


Рис. 4. Обобщенная схема автокодировщика

В простейшем автокодировщике с линейной функцией активации, изображенном схематически на рис. 5, значение функции, передаваемое от кодировщика к декодировщику, обозначено как h. Кодировщик выполняет стандартную операцию скалярного произведения, а декодировщик выполняет обратную операцию, т.е. разворачивает полученное значение в 2-мерный вектор:

$$h = x_1 * w_{11} + x_2 * w_{12}\,, \qquad y_1 = w_{21} * h, \qquad y_2 = w_{22} * h\ . \tag{15}$$

Вычисления подобного автокодировщика, принимающего значения на входах 2 и 6, при весах входного слоя равных 2 и 1, и весах выходного слоя равных 0,7 и 0,3 соответственно, показаны на рис. 5.

Автокодировщик с линейной функцией активации воспроизводит только линейные зависимости, т.е. он может сформировать модель только в виде линии или гиперплоскости (многомерный случай). Если добавить в кодировщик и декодировщик хотя бы по одному слою с нелинейной функцией активации (сигмоидальны, кусочно-линейные – ReLu и т.п.), то они смогут воспроизводить произвольные зависимости.

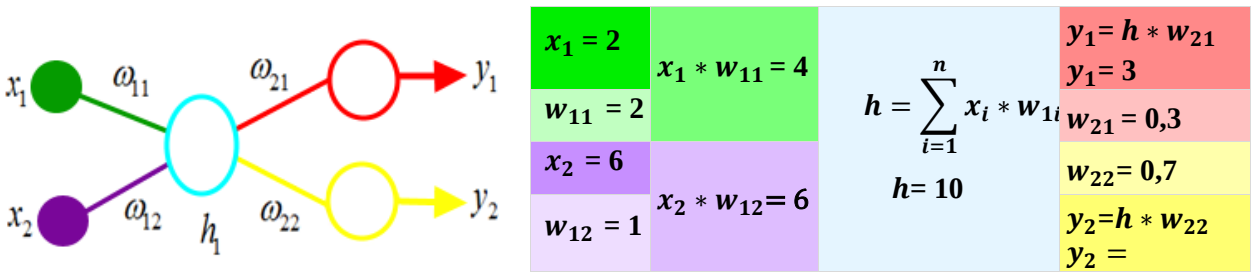


Рис. 5. Пример простейшего автокодировщика с линейной функцией активации и схема его расчета

В качестве примера реализации сверточного автокодировщика для очистки изображений, была выбрана архитектура, принимающая на входе пакеты по 32 входных тензора. Каждый тензор имеет размер 28×28 пикселей и один канал серого цвета. Группировка тензоров в пакеты носит случайный характер. Достоинством пакетного метода, помимо увеличения скорости обучения, является оптимизация системы одновременно под выборку из нескольких различных входных сигналов. Это исключает адаптацию сети к специфическому виду сигнала, и обучение не будет признано законченным, пока полученный результат не позволит обрабатывать все сигналы из пакета с наименьшей погрешностью. Пакетная обработка повышает готовность сети воспринимать сигналы, которые отсутствовали в учебном наборе данных.

Из различных конфигураций слоев искусственных нейронных сетей была выбрана сверточная структура автокодировщика, имеющая по 2 слоя в кодировщике и декодировщик.

В первом слое кодера (см. рис. 6) к изображениям из пакета применяется 32 одноядерных 2-мерных фильтра размера 7×7 с шагом (3, 3). Исходя из соображений эффективного расходования ресурсов памяти компьютера, в архитектурах сверточных нейронных сетей часто используют 32 или 64 фильтра. В данном случае выбрано число равное размеру пакета. Каждый из 32-х фильтров формирует свой тензор (карту признаков) с каждым из 32-х изображений. В результате свертки тензоры изображений сжимаются с 28×28 до 8×8 точек данных (или 12×12 – II вариант для обработки более сложных структур), в новом количестве 1024 –х штук. Карты признаков 1-го слоя служат для обнаружения базовых свойств изображений, такие как границы и кривые

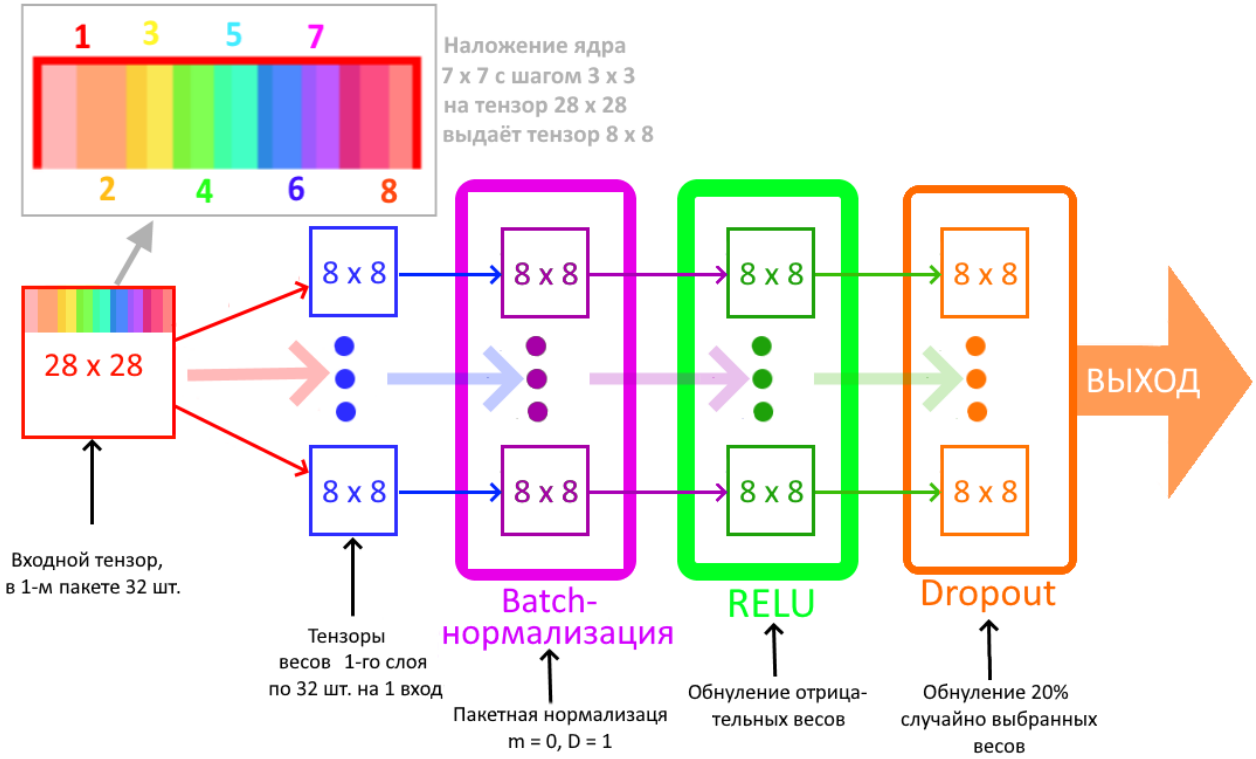


Рис. 6. Схема первого слоя кодировщика

Далее, как показано в табл. 1, к составленным тензорам применяется функция активации выпрямленной линейной единицы – RELU, которая обнуляет отрицательные элементы тензоров, а положительные оставляет без изменений, благодаря чему отсекаются ненужные детали, и происходит прореживание весов. Для ухода от переобучения добавляется дополнительное прореживание весов нейронов (dropping out) с выбранной вероятностью в 20%. Это означает, что при любых входных данных или параметрах исключенные нейроны возвращают 0, благодаря чему уменьшается резкое возрастание отдельных весов, оказывающих негативное затухающее влияние на соседние нейроны, и как следствие, способствующих созданию ложных корреляций.

Таблица 1

Архитектура сверточного автокодировщика

Шаг	Описание
Вход	Пакет из 32-х изображений размера 28×28 пикселей
1-й слой кодера	– Сжатие в размер 8×8 из тензора 28×28, ядром 7×7 с шагом (3, 3) – Применение 32-х фильтров на одном тензоре – Применение функции RELU (отброс отрицательных чисел) – Пакетная batch-нормализация с распределением СВ: $\mu = 0$ и D = 1. – Применение 20%-го dropping out

Шаг	Описание
2-й слой кодера	– Сжатие в размер 6×6 из тензора 8×8, ядром 3×3 с шагом (1, 1) – Применение 64-х фильтров на 32-х тензорах – Применение функции RELU (с отбрасыванием отрицательных чисел) – Пакетная batch-нормализация с распределением СВ: $\mu = 0$ и D = 1. – Применение 20%-го dropping out
1-й слой декодера	– Расширение в размер 8×8 из тензора 6×6, ядром 3×3 с шагом (1, 1) – Применение 32-х фильтров на 64-х тензорах – Применение функции RELU (отброс отрицательных чисел) – Пакетная batch-нормализация с распределением СВ: $\mu = 0$ и D = 1. – Применение 20%-го dropping out
2-й слой декодера	– Расширение в размер 28×28 из тензора 8×8, ядром 7×7 с шагом (3, 3) – Применение одного фильтра на 32-х тензорах – Применение функции Sigmoid
Выход	Обработанный пакет из 32-х изображений размером 28 х 28 пикселей

На втором слое кодировщика, сверткой ядром 3×3 с шагом (1, 1) размер тензоров уменьшается с 8×8 до 6×6 точек данных (или с 12×12 до 5×5 – II вариант). Фильтры этого слоя выявляют взаимосвязи между отдаленными точками тензора. Происходит выделение 64-х признаков, которые синтезируют информацию от признаков 1-го слоя для определения целостной картины. После 2-го слоя свертки кодировщика так же применяется функция активации RELU. И 20-процентный dropping out.

Таким образом, после прохождения кодирования тензоры изображений сжимаются в размере с выделением основных взаимосвязей (6×6×64 точек данных), и в таком виде подаются на вход декодировщика.

По структуре декодировщик зеркален по отношению к кодировщику, за исключением применения после последнего (выходного) слоя вместо функции активации RELU логистической функции – sigmoid. Форма логистической кривой, имеющая высокое значение производной вблизи середины диапазона выходных значений и низкое значение производной для выходных значений вблизи их максимума и минимума, способствует успешному обучению ИНС с подобной архитектурой.

В общем виде, в первом слое декодировщика из входных 64-х тензоров для каждого изображения пакета, имеющих размер 6×6 точек данных, получается слой тензоров размера 8×8 в количестве 32-х карт признаков для каждого изображения. Далее получается однопризнаковый выходной слой (или 2-й слой декодера), размера 28×28 точек данных для каждого изображения из пакета. [1, 5, 7, 8].

4. РЕАЛИЗАЦИЯ АЛГОРИТМА

Далее представлена реализация автокодировщика в виде многосвязной нейросети для удаления шума в изображении цифр. Полный вариант программы дан в Приложении.

Вначале импортируются необходимые библиотеки и загружаются примеры цифр для обучения из набора данных Mnist:

```
import numpy as np
from keras.datasets import mnist # импортирует датасет рукописных цифр
import matplotlib.pyplot as plt
from tqdm import tqdm # отображает индикатор выполнения
from torchvision import transforms # возвращает модифицированную версию изображения (тензор)
import torch.nn as nn # создает слои нейросети
from torch.utils.data import DataLoader,Dataset # обработка, загрузка
import torch
import torch.optim as optim # методы обновления весов и смещений
from torch.autograd import Variable # автодеференциирование (оборачивает тензор)
```

Используемая библиотека PyTorch является мощным фреймворком для работы с данными. Она включает в себя инструменты для написания сверточных нейронных сетей, а также целый ряд предобученных моделей.

Данные загружаются из Keras, в количестве 60 000 образцов изображений цифр и их значений для обучения (тренировки) модели и 10 000 для ее тестирования. При этом в компоненту `y` загружаются значения цифр, а в компоненту `x` их изображения.

```
(xtrain,ytrain),(xtest,ytest) = mnist.load_data()
```

Чтобы создать модель нейросети по очистке изображений цифр от шума, потребуются «зашумленные» данные (цифры, записанные в плохом качестве). Для хранения изображений с шумом создаются массивы, заполненные нулями, размером 28x28 точек для каждого элемента из 60 000 и 10 000 соответственно.

```
traindata = np.zeros((60000,28,28))
```

```
testdata = np.zeros((10000,28,28))
```

В массив `traindata` через цикл `for` записываются данные для обучения из `xtrain`, обработанные процедурой `add_noise` с добавлением шума Гаусса для первых 30 000 образцов, и с добавлением спекл-шума («зернистости») для оставшейся половины. Таким же образом заполняем данные для тестирования в `testdata` из `xtest` через зашумление, в таком же соотношении один к одному.

```
Noises = ["gaussian", "speckle"]
    …
traindata[idx] = add_noise(xtrain[idx],
                           k = noises[noise_id])
    …
testdata[idx] = add_noise(xtest[idx],
                          k = noises[noise_id])
```

Здесь `noises` – список методов добавления шума, при индексе `noise_id` равному нулю аргумент `k` принимает значение `"gaussian"`.

Процедура `add_noise` является процедурой пользователя, т.е. ее код прописывается разработчиком программы самостоятельно. Эта процедура применяет к изображению (массиву чисел), получаемому через аргумент `img`, один из 2-х фильтров: шума Гаусса или спекл-шума (см. рис. 7.).

Рассмотрим ее подробнее.

```
def add_noise(img,noise_type="gaussian"): # на входе
изображение и тип обработки (по умолчанию Гаусса)

    row,col = img.shape # img.shape имеет тип tuple (кортеж),
и содержит 2 элемента (к-во строк и столбцов в img, т.е. (28,
28)), которые присваиваются переменным row и col
    img = img.astype(np.float32) # конвертируем входные
    данные изображения в 8-байтовый вещественный формат

    if noise_type == "gaussian":
        mean = -5.9 # математическое ожидание, μ
        var = 35# дисперсия, D
        sigma = var ** .5 # среднеквадратическое отклонение,
        σ,//здесь **.5 – корень 2-й степени
        noise = np.random.normal(mean, sigma, (row,col))#
    создание массива noise, размером row x col,заполненного
    случайными числами, с плавающей точкой, из одномерного
    нормального распределения Гаусса, с заданными μ и σ
        noise=noise.reshape(row,col)# контроль размеров
    массива
```

```
        img = img + noise # к каждому элементу массива img
прибавляется элемент массива nose с идентичными номерами
позиции
        return img

if noise_type == "speckle":
        noise = np.random.randn(row,col) # массив случайных
чисел, размера row x col, отличный от массива в первом
методе тем, что параметры распределения СВ заданы по
умолчанию, т.е. μ = 0 и σ =1
        noise=noise.reshape(row,col)
        img = img + img * noise # к каждому элементу массива
img прибавляется его произведение с элементом массива
nose с идентичными номерами позиции
return img
```

Фильтр шума Гаусса добавляет распределенный аддитивный белый шум, который характеризуется равномерной спектральной плотностью и нормальным распределением значения амплитуды. В общем виде фильтр Гаусса похож на средний фильтр, с той разницей, что он включает средневзвешенное значение окружающих пикселей и имеет параметр сигма, вследствие чего, он лучше сохраняет края.

В процедуре `add_noise` параметры μ и σ для фильтра Гаусса заданы разработчиком для всех изображений одинаковыми, т.е. не производится расчет математического ожидания для отдельных пикселей, по ближайшей выборке. Это пролезно в данной ситуации, т.к. фильтр используется для ухудшения изображения размытием с мелкими частыми помехами, а не для его улучшения или стилизации.

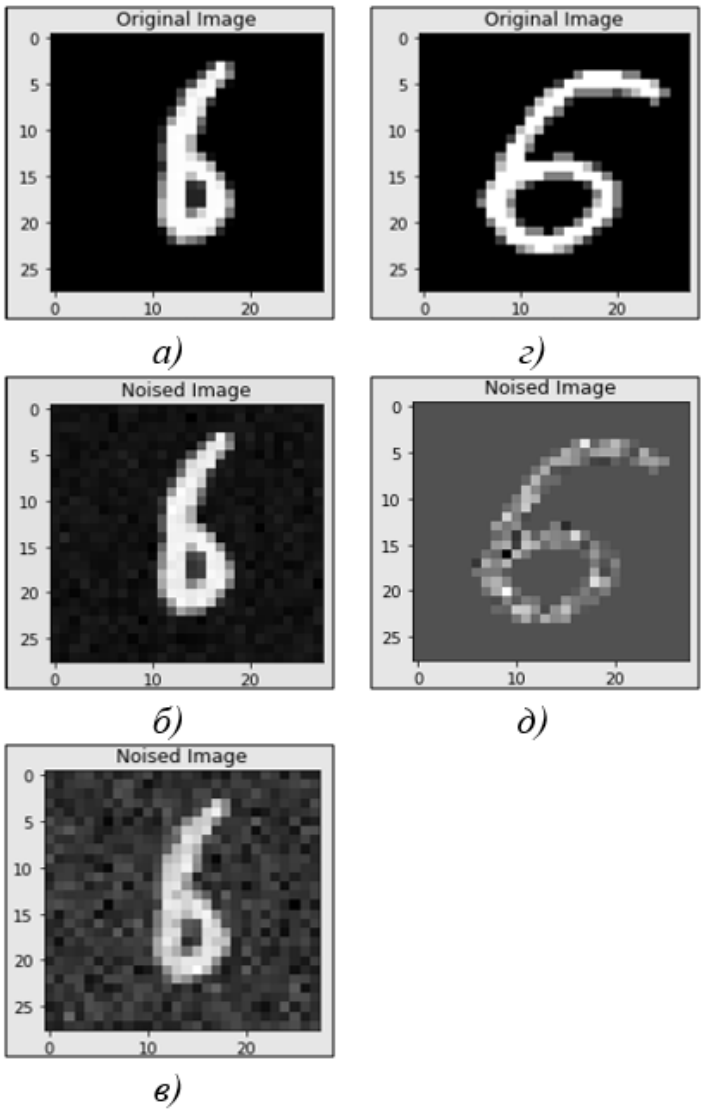


Рис. 7. Результаты фильтрации изображений библиотеки mnist а) изображение, б) и в) изображение после фильтра шума Гаусса с μ = -5.9 и D = 35, и с μ = -20 и D = 600 соответственно, г) и д) изображение до и после фильтра спекл-шума

Фильтр спекл-шума для объектов с низким разрешением добавляет мультипликативный шум, который можно охарактеризовать как светлые пятна, крапинки (спеклы), разделенные темными участками изображения. После его применения изображение получает эффект зернистости. Естественной причиной такого эффекта служит интерференция волн (световых, звуковых и т.д.), отраженных от объекта, при его запечатлении. В процедуре `add_noise` этот шум реализуется через сложение значений пикселей с их произведением со случайным числом, которое берется из нормального распределения относительно нуля со среднеквадратическим отклонением равным единице.

После создания массивов «зашумленных» данных `traindata` и `testdata`, можно оценить результат работы фильтров, если вывести на экран изображения из этих массивов, и из исходных массивов `xtrain` и `xtest` с такими же позициями.

Для этого можно использовать процедуру `subplots(n,m,p)` библиотеки `Matplotlib`, которая создает области прорисовки изображений (фигуры), расположенные по `n` строкам и `m` столбцам. Эта процедура возвращает кортеж, через первый элемент которого осуществляется доступ к свойствам фигуры, таким как размер, положение. Через второй элемент, который является набором однородных многомерных массивов (по одному на каждый элемент фигуры), осуществляется запись и вывод на экран изображений, а также подписей к ним, названий и т.п.

Для создания фигуры в 2-ве строки и 2-ва столбца, с последующим выводом в 1-й столбец изображений из массивов исходных и «зашумленных» данных обучающих выборок (рис. 4.а и б) используется следующий код:

```
n, m = 2,2 # количество строк и столбцов массива:
[[0..m],…n-раз…[0..m]];элементы массива 2x2:[0,0],[0,1],[1,0],
[1,1]
f, axes = plt.subplots(n, m) # f – фигура (n x m), axes –
массив (n x m), каждый элемент которого тоже массив
axes[0,0].imshow(xtrain[1100], cmap = "gray" ) # методом imshow
в массив axes[0,0] верхнего элемента 1-го столбца фигуры
записывается и выводится на экран 1100-е изображение из
xtrain, цвет серый (параметр cmap)
axes[1,0].imshow(traindata[1100], cmap = "gray") # в нижний
элемент 1-го столбца выводится это же изображение, но уже
обработанное фильтром, т.к. номер позиции меньше 30.000, то
это фильтр шума Гаусса
```

После оценки результатов работы фильтров, при необходимости в них вносятся изменения. Качество обучающих данных оказывает решающее влияние на эффективность работы искусственных нейронных сетей. Забегая вперед отметим, как будет справляться с задачей по очистке изображений от шума рассматриваемая сверточная нейронная сеть, прошедшая в двух случаях обучение на данных, пропущенных через один и тот же фильтр спекл-шума, но через разные фильтры шума Гаусса. Возникают значительные отличия. В первом случае взяты данные для обучения, прошедшие через фильтр шума Гаусса с μ = -5.9 и D = 35 (рис. 4.б), а во втором с μ = -20 и D = 600 (рис. 4.в). Сравнив результаты работы сети для этих случаев можно увидеть значительные отличия в распознавании сторонних изображений (не из библиотеки `Mnist`) с большим количеством шума, во 2-м случае сеть явно будет справляться лучше. При очистке изображений из библиотеки тестовой выборки, для образцов имеющих шум Гаусса программа сработает примерно одинаково в обоих случаях. Для образцов прошедших через фильтр спекл-шума, программа немного лучше справляется в 1-м случае, когда фильтр Гаусса более мягкий, и нейросети проще выстроить корреляции для очистки от спекл-шума.

После добавления к данным шума, создаются общие объекты хранения для «чистых», «зашумленных» изображений цифр, и их фактического значения. Для обучающей выборки – это массив `trainset`, а для тестовой – `testset`. После чего эти числовые массивы, преобразуются с помощью процедуры `Compose` библиотеки `Torchvision.transforms` в объекты класса `torch.Tensor`. Объекта этого класса представляют собой практически те же многомерные матрицы, но такая смена типа данных требуется для их дальнейшей обработки, средствами библиотеки `Torch`.

```
    tsfms = transforms.Compose([
        transforms.ToTensor(),     # меняем тип данных на ToTensor
        transforms.Normalize(0.1307, 0.3081) # нормализация данных, может отсутствовать, для данных естественной природыравна: 0.5, 0.5.
    ])
    trainset = noisedDataset(traindata,xtrain,ytrain,tsfms)
    testset = noisedDataset (testdata,xtest,ytest,tsfms)
```

С помощью процедуры пользователя noisedDataset в объекты (trainset и testse) записываются три первых ее аргумента, преобразованных методом, хранящемся в 4-м аргументе, т.е. в переменной tsfms, в элементы объекта класса torch.Tensor.

Таким образом, в объекте trainset хранится:

- `trainset[0]` (1-й элемент) – массив `traindata` (изображения с шумом);
- `trainset[1]` (2-й элемент) – массив `xtrain` (исходные изображения);
- `trainset[2]` (3-й элемент) – массив `ytrain`, (значения цифр на изображениях).

Также в процедуре `noisedDataset` переопределяется метод `len`, так чтобы он определял количество элементов в датасете, и метод `getitem`, через который будет осуществляться итерация датасета.

Далее с помощью утилиты `DataLoader`, для увеличения скорости обработки массивов, загружаются подготовленные датасеты в виде потока пакетированных данных. Один пакет данных обрабатывается за один шаг итераций. Объем данных на итерационный шаг задается переменной `batch_size`:

```
    trainloader = DataLoader(trainset, batch_size = 32,
     shuffle = True)
    testloader = DataLoader(testset, batch_size = 1,
     shuffle = True)
```

Атрибут `shuffle` отвечает за случайный порядок доступа к пакетам (перемешивает их).

Далее создается класс `denoising_model(nn.Module)`, описывающий модель нейросети. Здесь `nn.Module` – это базовый класс для всех модулей нейронных сетей. Описанный пользователем класс наследует его, и при необходимости добавляет и переопределяет методы и свойства.

```
class denoising_model(nn.Module):
    def __init__(self):
# наследуется родительский метод инициализации с добавлением
2-х новых полей: энкодера и декодера
        super(denoising_model,self).__init__() # позволяет
дополнить родительский init
        # КОДЕР:
        self.layer1 = nn.Sequential( # последовательная
упаковка нескольких сетевых уровней
                                    # свертка:
            nn.Conv2d(1, 32, kernel_size = 7, stride = 3),
              # функция активации линейного выпрямителя:
            nn.ReLU(),
    # Пакетная нормализация с распределением вероятностных
                 характеристик: μ = 0 и D = 1:
            nn.BatchNorm2d(num_features =32),
             # отсев 20% случайных нейронов слоя:
            nn.Dropout(0.2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size = 3, stride = 1),
            nn.BatchNorm2d(num_features = 64),
            nn.ReLU(),
            nn.Dropout(0.2)
        )
         # ДЕКОДЕР:
        self.layer02 = nn.Sequential(
            nn.ConvTranspose2d(64, 32, kernel_size = 3,
stride = 1),
            nn.BatchNorm2d(num_features = 32),
            nn.ReLU(),
            nn.Dropout(0.2)
        )
        self.layer01 = nn.Sequential(
            nn.ConvTranspose2d(32, 1, kernel_size = 7, stride
= 3),
            nn.Sigmoid()
# layer1-2: датасет пропускается через кодер
# layer01-02: датасет пропускается через декодер
    def forward(self, x):
```

```
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer02(out)
        out = self.layer01(out)
        return out
```

В модели мскусственной нейронной сети используются библиотеки с GPU-ускорением, для работы которых хорошо подходит графический процессор с поддержкой технологии CUDA. Программой проводится проверка на наличие такого аппаратного оборудования.

```
if torch.cuda.is_available() == True:
    device = "cuda:0"
else:
    device = "cpu"
```

При отрицательном ответе для проведения вычислений нейросети выбирается центральный процессор (CPU). Также можно использовать облачные провайдеры, такие как Linode, Google Colaboratory, Kaggle и т.д. В любом случае в переменной `device` должен храниться тип доступного устройства для проведения вычислений. Далее через переменную `model` запускается модель искусственной нейронной сети (`denoising_model`) на типе устройства записанном в `device`.

model = denoising_model().to(device)

Для обучения ИНС требуется метрика контроля качества. В данном случае в ее роли выступает среднеквадратичная функция потерь – MSELoss. Оптимизация искусственной нейронной сети осуществляется алгоритмом Adam (adaptive moment estimation), который сочетает в себе идеи накопления движения и слабого обновления весов для типичных признаков. При таком алгоритме скорость обучения поддерживается для каждого веса сети (параметра) и отдельно адаптируется по мере развития обучения.

```
criterion = nn.MSELoss()  # Метрика контроля качества - функцию потерь, выбирается MSELoss - среднеквадратичная функция
optimizer = optim. Adam(model.parameters(), lr = 0.003)
# Оптимизатор со стандартными настройками, ставится только скорость обучения
```

Остальные параметры ИНС имею следующий вид:

```
epochs = 120 # количество эпох
l   =   len(trainloader)   #   количество   пакетированных массивов: trainset/batch_size = 60000/32 = 1875
losslist = list() # список потерь
running_loss = 0 # текущие потери
KOst = 0.0063 # критерий остановки процесса обучения
```

После задания всех параметров происходит перебор 120 эпох через цикл с предусловием. В каждой эпохе запускается цикл последовательного извлечения пакетов из `trainloader` в переменные `dirty,clean` и `label` соответственно с дальнейшим выводом на экран шкалы состояния `(tqdm)` их извлечения для данной эпохи. Извлечение содержимого пакетов осуществляется с помощью метода `getitem`, переопределенного раннее при формировании датасетов, таким образом, чтобы перебор элементов пакета осуществлялся автоматически без добавления в код циклов.

```
for epoch in range(epochs): # цикл эпох
    print("Entering Epoch: ",epoch)
    for i1, (dirty, clean, label) in tqdm (enumerate
(trainloader)):
        dirty = dirty.view(dirty.size(0),1,28,28).type(torch.
FloatTensor)
        clean = clean.view(clean.size(0),1,28,28).type(torch.
FloatTensor)
        dirty, clean = dirty.to(device),clean.to(device)

#----------------Forward Pass----------работает нейросеть
        output = model.forward(dirty) # запускается модель НС
        loss = criterion(output,clean) # считается критерий
nn.MSELoss():  (output-clean)^2/n

#-------------Backward Pass-----------------обратный проход
        optimizer.zero_grad() # обнуляются градиентные буферы
        loss.backward() # вычисляется градиент критерия loss=
criterion()
        optimizer.step()  # обновление с новым градиентом
        running_loss += loss.item() #  потери текущей эпохи
        epochloss += loss.item() #  потери всего
#----------------Log--------------------------------
    sr_loss = running_loss/l #
    losslist.append(sr_loss) # пополнение списка потерь
    running_loss=0 # обнуление потери для новой эпохи
    print("======> epoch: {}/{}, Loss:{}".format(epoch,
epochs, sr_loss))
    if sr_loss <= KOst: break # проверка на выполнение
остановки
```

[6-8]

5. ИТОГИ ТЕСТИРОВАНИЯ

На рис. 8 приведены результаты очистки изображений из тестовой части библиотеки Mnist, линейной искусственной нейронной сети [6] обученной на 2-х разных фильтрах. Сеть содержит полносвязанные линейные слои.

Для создания тренировочных данных использовался один и тот же фильтр спекл-шума, а фильтр шума Гаусса в первом случае (группа А) имел параметры: μ = -5.9 и D = 35, а во втором (группа Б): μ = -20 и D = 600. По итогам тестирования стало понятно, что в группе Б несколько ухудшен результат очистки от спекл-шума.

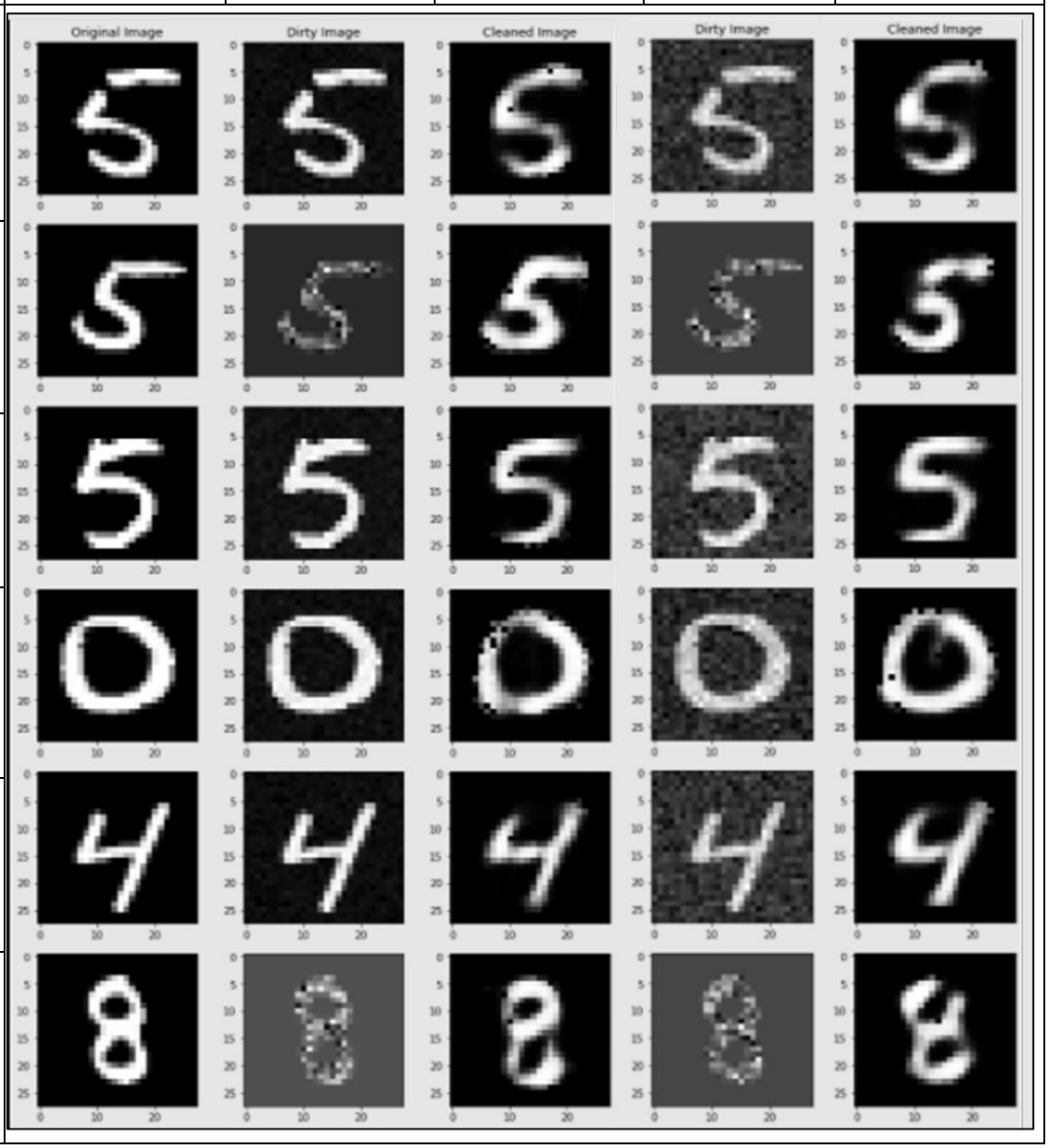
	исходное изо.	μ = -5.9, D = 35	очищеное изо.	μ = -20, D = 600	очищеное изо.
id _897 ш. Гауса					
0 id _8834 спекл-ш.					
id _4226 ш. Гауса					
id _4554 ш. Гауса					
id _910 ш. Гауса					
id _6049 спекл-ш.					

Рис. 8. Результат тестирования линейной искусственной нейронной сети на данных Mnist, обучение с: μ = -5.9, D = 35 (группа А), μ = -20, D = 600 (группа Б)

На рис. 9 представлены результаты тестирования этой же линейной искусственной нейронной сети на сторонних изображениях (не из библиотеки Mnist).

Для загрузки в программу изображений цифр использовался следующий код:

```
from PIL import Image
rw, cl = 28, 28
n = 6
name = []
AryIzo = np.zeros((n, rw, cl))
for k in range(n):
    name .append ("Izo" + str(k) + ".jpg")
    izo = Image.open(name[k])
    pix = izo.load()
    for i in range(rw):
        for j in range(cl):
            AryIzo[k,i,j] = (pix[i,j][0]+ pix[i,j][1]
+ pix[i,j][2])/3
```

Вывод на экран изображений из массива `AryIzo`, и их очищенных версий осуществляется аналогично подобной процедуре для массива `testset`, с внесением корректировок:

– замена строки a на b:

a) `dirty = testset[test_imgs[idx]][0]`

b) `dirty = tsfms(np.transpose(AryIzo[idx]))`

– удаление вывода изображений группы `clear`.

Очевидно, что качество очистки изображений, не входящих в библиотеку `Mnist` с помощью данной искусственной нейронной сети без сверточного слоя невысоко.

Далее на рис. 9 представлены результаты тестирования на этих же входных данных только сверточной ИНС, архитектура которой представлена в табл. 1.

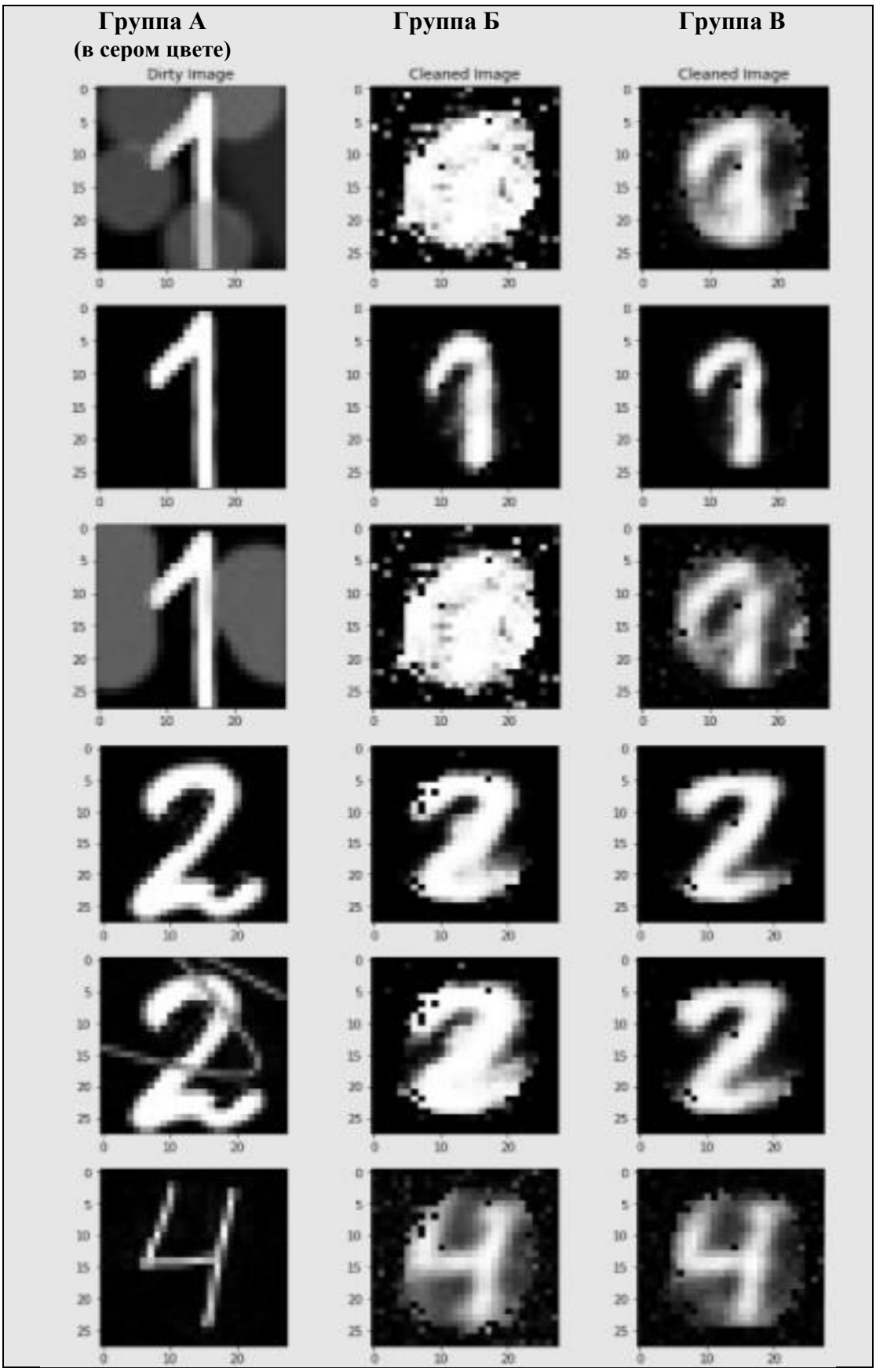


Рис. 9. Результат тестирования линейной искусственной нейронной сети на сторонних данных, обучение с μ = -5.9, D = 35 (группа Б) и μ = -20, D = 600 (группа В)

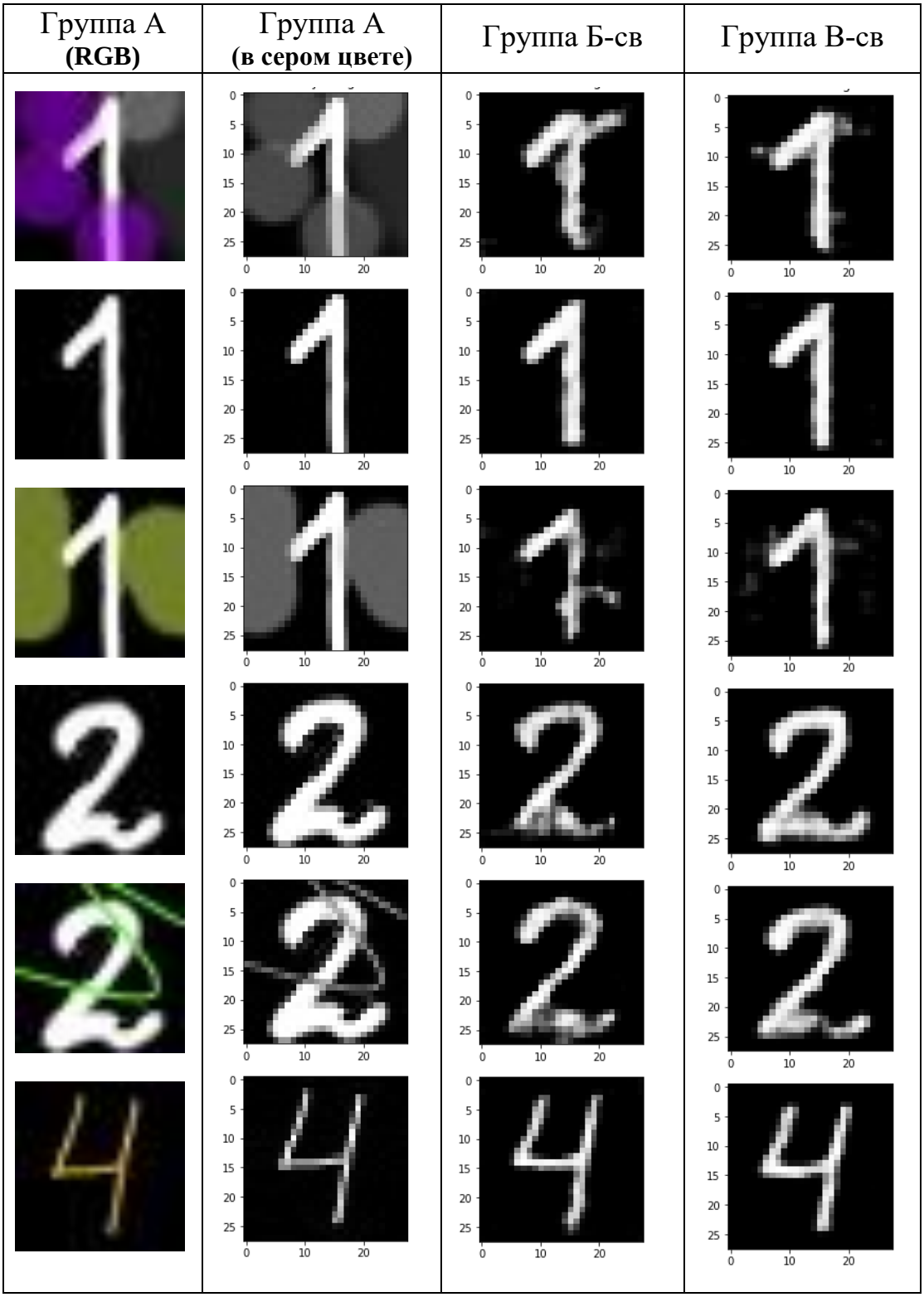


Рис.10. Результат тестирования сверточной нейронной сети на сторонних данных, обучение с μ = -5.9, D = 35 (группа Б) и μ = -20, D = 600 (группа В)

Качество очистки заметно выросло после добавления сверточных операций. Исходя из результатов работы ИНС, представленных на рис. 9 и рис. 10, следкет, что при использовании в обучении более грубого фильтра (группа В), качество очищенных сторонних изображений возрастает.

6. ПРАКТИЧЕСКИЕ ЗАДАНИЯ

1. Запустить программу, приведенную в приложении.
2. Подготовить собственные зашумленные изображения цифр размерами 28×28 для дальнейшего распознавания. Изображения можно сгенерировать программой или фотографированием рукописных букв с конвертированием фала изображения в нужный размер. Выбрать вариант самостоятельно.
3. Описать процедуру подготовки данных для распознавания цифр.
4. Привести полученные изображения.
5. Запустить программу распознавания изображения с помощью предобученной сверточной нейронной сети автокодировкщика.
6. Привести результаты работы автокодировщика.
7. Сделать выводы и составить отчет о выполнении задания.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Николенко С., Кадурин А., Архангельская Е. Н63 Глубокое обучение. — СПб.: Питер, 2018. – 480 с.
2. Хьюбел Д. Глаз, мозг, зрение / Д. Хьюбел; Пер. с англ. О. В. Левашова и Г. А. Шараева.- М.: «Мир», 1990. – 239 с.
3. Быков В. Л. Кора полушарий большого мозга // Частная гистология человека. – СПб.: СОТИС, 2001. – С. 260-271. – 304 с.
4. Фролова Ю. Г. Клиническая нейропсихология [Электронный ресурс]: пособие /. – Минск: БГУ, 2016.
5. Гудфеллоу Я., Бенджио И., Курвилль А. Г93 Глубокое обучение/ пер. с анг. А.А. Слинкина.– 2-е изд., испр. – М.: ДМК Пресс, 2018. – 652 с.
6. Электронный ресурс: https://github.com/pranjaldatta/Denoising-Autoencoder-in-Pytorch – Denoising Autoencoder Pytorch
7. Понтер Ян. Программируем с PyTorch: Создание приложений глубокого обучения / пер. с анг. Попова А. – изд. Питер, 2020 – 256 с.
8. Макмахан Б., Рао Делип. Знакомство с PyTorch / пер. с анг. Попова А. – изд. Питер, 2020 – 256 с.

ОГЛАВЛЕНИЕ

ПРИЛОЖЕНИЕ 1

Полный код программы:

```
import numpy as np
from keras.datasets import mnist
import matplotlib.pyplot as plt
from tqdm import tqdm
from torchvision import transforms
import torch.nn as nn
from torch.utils.data import DataLoader,Dataset
import torch
import torch.optim as optim
from torch.autograd import Variable

def add_noise(img,noise_type="gaussian"):
row,col=28,28
img=img.astype(np.float32)
if noise_type=="gaussian":
mean=-20
var=600
sigma=var**.5
noise=np.random.normal(mean,sigma,img.shape)
noise=noise.reshape(row,col)
img=img+noise
return img

if noise_type=="speckle":
noise=np.random.randn(row,col)
noise=noise.reshape(row,col)
img=img+img*noise
return img

(xtrain,ytrain),(xtest,ytest)=mnist.load_data()
print("No of training datapoints:{}\nNo of Test
datapoints:{}".format(len(xtrain),len(xtest)))

noises=["gaussian","speckle"]
noise_ct=0
noise_id=0
traindata=np.zeros((60000,28,28))

for idx in tqdm(range(len(xtrain))):
if noise_ct<(len(xtrain)/2):
```

```
noise_ct+=1
traindata[idx] = add_noise(xtrain[idx], noise_type = noises[noise_id])
else:
print("\n{} noise addition completed to imag-es".format(noises[noise_id]))
noise_id += 1
noise_ct = 0

print("\n{} noise addition completed to imag-es".format(noises[noise_id]))

noise_ct = 0
noise_id = 0
testdata = np.zeros((10000,28,28))
for idx in tqdm(range(len(xtest))):
if noise_ct < (len(xtest)/2):
noise_ct += 1
testdata[idx] = add_noise(xtest[idx], noise_type = noises[noise_id])
else:
print("\n{} noise addition completed to imag-es".format(noises[noise_id]))
noise_id += 1
noise_ct = 0

print("\n{} noise addition completed to imag-es".format(noises[noise_id]))

f, axes = plt.subplots(2,2)
f.set_figwidth(7)
f.set_figheight(7)

#showing images with gaussian noise
axes[0,0].imshow(xtrain[1100],cmap="gray")
axes[0,0].set_title("Original Image")
axes[1,0].imshow(traindata[1100],cmap='gray')
axes[1,0].set_title("Noised Image")
#showing images with speckle noise
axes[0,1].imshow(xtrain[31000],cmap='gray')
axes[0,1].set_title("Original Image")
axes[1,1].imshow(traindata[31000],cmap="gray")
axes[1,1].set_title("Noised Image")

class noisedDataset(Dataset):
def __init__(self, datasetnoised, datasetclean, labels, transform):
self.noise = datasetnoised # 1)
self.clean = datasetclean # 2)
self.labels = labels # 3)
```

```
self.transform = transform
def __len__(self):
return len(self.noise)
def __getitem__(self,idx):
xNoise = self.noise[idx] # 1)
xClean = self.clean[idx] # 2)
y = self.labels[idx] # 3)
if self.transform != None:
xNoise = self.transform(xNoise) # 1)
xClean = self.transform(xClean) # 2)
return (xNoise,xClean,y)
tsfms = transforms.Compose([
transforms.ToTensor()
])

trainset = noisedDataset(traindata,xtrain,ytrain,tsfms)
testset = noisedDataset(testdata,xtest,ytest,tsfms)

batch_size = 32
trainloader = DataLoader(trainset, batch_size=32, shuffle=True)
testloader = DataLoader(testset, batch_size=1, shuffle=True)

class denoising_model(nn.Module):
def __init__(self):
super(denoising_model,self).__init__()
self.layer1 = nn.Sequential(
nn.Conv2d(1, 32, kernel_size=7, stride=3),
# padding=1),
nn.ReLU(),
nn.BatchNorm2d(num_features=32),
#nn.MaxPool2d(kernel_size=2, stride=2, padding=1),
nn.Dropout(0.2)
)
self.layer2 = nn.Sequential(
nn.Conv2d(32, 64, kernel_size=3, stride=1),
nn.ReLU(),
nn.BatchNorm2d(num_features=64),
nn.Dropout(0.2)
)
self.layer02 = nn.Sequential(
nn.ConvTranspose2d(64, 32, kernel_size = 3, stride=1),
nn.ReLU(),
nn.BatchNorm2d(num_features = 32),
```

```
#nn.Upsample(size = 13, mode='bilinear'),
nn.Dropout(0.2)
)
self.layer01 = nn.Sequential(
nn.ConvTranspose2d(32, 1, kernel_size=7, stride=3),
nn.Sigmoid()
)
#self.drop_out = nn.Dropout()
#self.fc1 = nn.Linear(7 * 7 * 64, 1000)
#self.fc2 = nn.Linear(1000, 10)
def forward(self, x):
out = self.layer1(x)
out = self.layer2(out)
out = self.layer02(out)
out = self.layer01(out)
return out

if torch.cuda.is_available() == True:
device = "cuda:0"
else:
device ="cpu"
model=denoising_model().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr = 0.003)
epochs = 120
l = len(trainloader)
losslist = list()
epochloss = 0
running_loss = 0
KOst = 0.0063

for epoch in range(epochs):
print("Entering Epoch: ",epoch)
for i1, (dirty,clean,label) in tqdm(enumerate(trainloader)):
dirty = dirty.view(dirty.size(0), 1, 28, 28).type(torch.FloatTensor)
clean = clean.view(clean.size(0), 1, 28, 28).type(torch.FloatTensor)
dirty,clean = dirty.to(device),clean.to(device)

#-----------------Forward Pass----------------------
output = model.forward(dirty)
loss = criterion(output,clean)

#-----------------Backward Pass----------------------
```

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
running_loss += loss.item()
epochloss += loss.item()

#-----------------Log-------------------------------
sr_loss = running_loss/l
losslist.append(sr_loss)
running_loss=0
print("======> epoch: {}/{}, Loss:{}".format(epoch,epochs, sr_loss))
if sr_loss <= KOst: break

plt.plot(range(len(losslist)), losslist)

f, axes = plt.subplots(6,3,figsize = (20, 20))
axes[0, 0].set_title("Original Image")
axes[0, 1].set_title("Dirty Image")
axes[0, 2].set_title("Cleaned Image")
test_imgs = np.random.randint(0, 10000, size=6)
print(test_imgs)

for idx in range((6)):
dirty = testset[test_imgs[idx]][0]
clean = testset[test_imgs[idx]][1]
label = testset[test_imgs[idx]][2]
dirty = dirty.view(dirty.size(0), 1, 28, 28).type(torch.FloatTensor)
dirty = dirty.to(device)
output = model(dirty)
output = output.view(1, 28, 28)
output = output.permute(1, 2, 0).squeeze(2)
output=output.detach().cpu().numpy

dirty=dirty.view(1,28,28)
dirty=dirty.permute(1,2,0).squeeze(2)
dirty=dirty.detach().cpu().numpy()
clean=clean.permute(1,2,0).squeeze(2)
clean=clean.detach().cpu().numpy()
axes[idx,0].imshow(clean,cmap="gray")
axes[idx,1].imshow(dirty,cmap="gray")
axes[idx,2].imshow(output,cmap="gray")

from PIL import Image
```

```
rw, cl = 28, 28
n = 6
name = []
AryIzo = np.zeros((n, rw, cl))
for k in range(n):
name .append ("Izo" + str(k) + ".jpg")
izo = Image.open(name[k])
pix = izo.load()
for i in range(rw):
for j in range(cl):
AryIzo[k,i,j] = (pix[i,j][0]+ pix[i,j][1] + pix[i,j][2])/3

f, axes = plt.subplots(6, 3, figsize = (20, 20))
axes[0, 0].set_title("Original Image")
axes[0, 1].set_title("Dirty Image")
axes[0, 2].set_title("Cleaned Image")
test_imgs = np.random.randint(0, 10000, size = 6)

for idx in range((6)):
dirty = tsfms(np.transpose(AryIzo[idx]))
clean = testset[test_imgs[idx]][1]
label = testset[test_imgs[idx]][2]
dirty = dirty.view(dirty.size(0), 1, 28, 28).type(torch.FloatTensor)
dirty = dirty.to(device)
output = model(dirty)
output = output.view(1, 28, 28)
output = output.permute(1, 2, 0).squeeze(2)
output = output.detach().cpu().numpy()
dirty = dirty.view(1, 28, 28)
dirty = dirty.permute(1, 2, 0).squeeze(2)
dirty = dirty.detach().cpu().numpy()
clean = clean.permute(1, 2, 0).squeeze(2)
clean = clean.detach().cpu().numpy()
axes[idx, 0].imshow(clean,cmap = "gray")
axes[idx, 1].imshow(dirty,cmap = "gray")
axes[idx, 2].imshow(output,cmap = "gray")

import os
PATH = "PATHS101"
#"username/directory/lstmmodelgpu.pth"
torch.save(model.state_dict(),PATH)
```

СВЕРТОЧНЫЕ СЕТИ НА PYTHON

Методические указания
к выполнению практических работ
для студентов 2-го курса магистратуры, обучающихся по образовательным программам
направления 09.04.03 «Прикладная информатика» всех форм обучения

Составители: Головинский Павел Абрамович,
Тарасова Анна Сергеевна,
Шаталова Ангелина Олеговна,
Еникеев Эльдар Ильясович

Подписано в печать 00.00.2022 Формат 60x84 1/16. Уч.-изд. л. 0,0.
Усл. печ. л. 0.0. Бумага для множительных аппаратов. Тираж 50 экз. Заказ №___.

ФГБОУ ВО «Воронежский государственный технический университет»
394026 г. Воронеж, Московский проспект, 14

Участок оперативной полиграфии издательство ВГТУ
394026 г. Воронеж, Московский проспект, 14