ТЕХНИЧЕСКИЕ НАУКИ
И ТЕХНОЛОГИИ

АҚПАРАТТЫҚ-КОММУНИКАЦИЯЛЫҚ ТЕХНОЛОГИЯЛАР
ИНФОРМАЦИОННО-КОММУНИКАЦИОННЫЕ ТЕХНОЛОГИИ
INFORMATION AND COMMUNICATION TECHNOLOGIES

DOI 10.51885/1561-4212_2023_1_134
МРНТИ 28.23.15

А.Е. Кусатаев[1], О.Е. Бакланова[2], Ш.Т. Тезекпаева[3]
Восточно-Казахстанский технический университет им. Д. Серикбаева,
г. Усть-Каменогорск, Казахстан
[1]*E-mail: kusatay@mail.ru*
[2]*E-mail: oebaklanova@mail.ru*
[3]*E-mail: shynar8787@mail.ru**

ПРОЕКТИРОВАНИЕ НЕЙРОННОЙ СЕТИ НА ПРИМЕРЕ РАСПОЗНАВАНИЯ РУКОПИСНЫХ ЦИФР

ҚОЛ ЖАЗБА САНДАРДЫ ТАНУ МЫСАЛЫНДА НЕЙРОНДЫҚ ЖЕЛІНІ ЖОБАЛАУ

NEURAL NETWORK DESIGN ON THE EXAMPLE OF HANDWRITTEN DIGIT RECOGNITION

Аннотация. *Данная статья посвящена реализации алгоритма распознавания рукописных цифр с использованием нейронных сетей. Определение полноценной нейронной сети в Keras на реальном примере распознавания рукописных цифр. Улучшение полноценной нейронной сети в Keras посредством добавления сверхточных слоев. В качестве тестового образца рассматривается открытая база данных рукописных цифровых изображений MNIST. Полученная модель может успешно использоваться для решения задач классификации изображений и распознавания образов.*

Ключевые слова: *нейронная сеть, свёрточная нейронная сеть, свёртка, распознавания рукописных цифр, метод градиентного спуска, база данных MNIST, Python.*

Аңдатпа. *Бұл мақала нейрондық желілерді қолдана отырып, қолмен жазылған сандарды тану алгоритмін жүзеге асыруға арналған. Keras кітапханасы көмегімен толық нейрондық желіні анықта. Қатпарлы қабаттарды қосу арқылы Keras-та толық нейрондық желіні жақсарту. Сынақ үлгісі ретінде MNIST қолжазба сандарының ашық кескін базасы қарастырылады. Кескіндерді жіктеу және үлгіні тану мәселелерін шешу үшін алынған модельді сәтті қолданыла алады.*

Түйін сөздер: *нейрондық желі, қатпарлы нейрондық желі, қатпар, қолмен жазылған сандарды тану, градиенттi түсіру әдісі, MNIST мәліметтер базасы, Python.*

Abstract. *This article is devoted to the implementation of an algorithm for recognizing handwritten digits using neural networks. Definition of a full-fledged neural network in Keras on a real example of handwritten digit recognition. Improving a full-fledged neural network in Keras by adding convolutional layers. An open database of MNIST handwritten digit images is considered as a test sample. The resulting model can be successfully used to solve problems of image classification and image recognition.*

Keywords: *neural network, convolutional neural network, convolution, handwritten digit recognition, gradient descent method, MNIST database, Python.*

Введение. Нейронные сети были разработаны с целью имитации нервной системы человека для решения задач машинного обучения на основе вычислительных элементов, работа

которых напоминала бы действие человеческих нейронов [1, 2, 3, 4].

Важнейшим свойством нейронных сетей является их способность учиться на данных окружающей среды и повышать свою производительность в результате обучения. Улучшения производительности происходят с течением времени в соответствии с определенными правилами. Обучение нейронной сети происходит посредством интерактивного процесса настройки синаптических весов и порогов. В идеале нейронная сеть получает знания об окружающей среде на каждой итерации процесса обучения.

Обучение – это процесс, в котором свободные параметры нейронной сети корректируются путем имитации среды, в которую встроена сеть. Тип обучения определяется способом подстройки этих параметров [5, 6, 7, 8]. Данное определение процесса обучения предполагает следующую последовательность событий [9, 10, 11]:

1 – нейронная сеть получает стимулы из внешней среды;

2 – в результате меняются свободные параметры нейронной сети;

3 – после изменения внутренней структуры нейронная сеть по-разному реагирует на возбуждения.

Приведенный список четких правил решения задачи обучения называется алгоритмом обучения. Нетрудно предположить, что универсального алгоритма обучения, подходящего для всех архитектур нейронных сетей, не существует. Существует только один набор инструментов, предлагаемых многими алгоритмами обучения, каждый из которых имеет свои преимущества.

Алгоритмы обучения отличаются друг от друга тем, что регулируют синаптические веса нейронов. Еще одна отличительная черта – то, как обученная нейронная сеть взаимодействует с внешним миром. В этой статье обсуждается парадигма обучения, связанная с модельной средой, в которой работает эта нейронная сеть [12, 13, 14, 15].

Материалы и методы исследования. Рассмотрим автоматическое дифференцирование и обучение нейронных моделей на примере задачи распознавания рукописных цифр. Мы будем обучать на наборе данных MNIST [17, 18, 19, 20]. MNIST состоит из рукописных цифр (рис. 1).

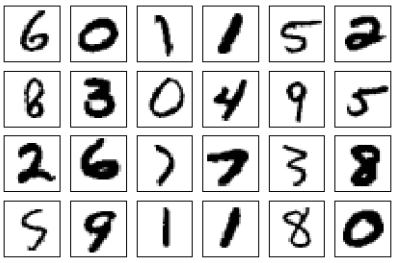

Рисунок 1. Набор данных MNIST: примеры рукописных цифр

Всего в MNIST содержится 70 000 размеченных черно-белых изображений размером 28x28 пикселей. «Размеченных» здесь означает, что каждому изображению в данных уже поставлен в соответствие правильный ответ – цифра, которую хотел изобразить человек на этой картинке.

Для импорта MNIST достаточно написать буквально две строчки кода:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True).
```

Полученные данные уже разбиты на тренировочную (*mnist.train*), тестовую (*mnist.test*) и валидационную (*mnist.validate*) выборки, содержащие 55 000, 10 000 и 5000 примеров соответственно. Каждая из этих выборок состоит из изображений цифр (*mnist.train.images*) и их меток (*mnist.train.labels*); рис. 1 показывает несколько образцов изображений с различными метками.

В этом примере мы не будем использовать двумерную структуру изображений, а представим их в виде обычных векторов размерности 784. В Модуле 5 мы еще вернемся к этому примеру и покажем, как правильно пользоваться двумерным расположением пикселей на картинке и насколько это получится эффективнее, а сейчас для наших нужд достаточно простой одномерной задачи.

Таким образом, тренировочная выборка *mnist.train.images* может быть представлена в виде матрицы размерности 55 000 x 784. Отметим, что и в TensorFlow, и в theano часто требуются не только матрицы, но и многомерные представления данных. Например, MNIST логичнее представить в виде трехмерной «матрицы» размерности 55 000x28x28. Такие «многомерные матрицы» в TensorFlow называются тензорами, однако, как уже было указано, тензоры не настоящие, а просто многомерные массивы.

Рассмотрим в качестве модели для обучения softmax-регрессию. Это обобщение логистической регрессии на случай нескольких классов. Чтобы получить «вероятности» классов, которые необходимо оценить, применяем так называемую softmax-функцию к вектору получившихся ненормализованных оценок:

$$softmax(x)_j = \frac{\exp(x_j)}{\sum_i \exp(x_i)}$$

Идея softmax-функции состоит в том, чтобы несколько заострить, преувеличить разницу между полученными значениями: softmax будет выдавать значения, очень близкие к нулю, для всех x_j, существенно меньших максимального.

В качестве функции потерь используем стандартную для логистической регрессии перекрестную энтропию (кросс-энтропию, cross-entropy):

$$H_t(y) = -\sum_i t_i \log y_i ,$$

где y – предсказанное нами значение, t – исходная разметка (правильный ответ).

Для удобства метки изображений можно представить в виде так называемых one-hot vectors – векторов, в которых единица стоит в позиции, соответствующей исходной метке, а в остальных позициях стоят нули. Например, вектор [0; 0; 0; 1; 0; 0; 0; 0; 0; 0] будет соответствовать ответу 3. Тогда *mnist.train.labels* – это тензор (матрица) размера 55 000x10.

Обычно в Python для сложных вычислений используются вспомогательные библиотеки численных вычислений, например NumPy. Такие библиотеки выполняют дорогие операции (например, умножение матриц) не силами самого интерпретатора языка Python, а с помощью оптимизированного кода, написанного на других языках (обычно C или Fortran). К сожалению, даже переключение между операциями в Python и вне его может оказаться слишком дорогим, особенно когда речь идет о вычислениях на GPU. Поэтому вместо того, чтобы выполнять каждую отдельную операцию независимо от Python, TensorFlow предлагает возможность описать в Python граф вычислений, а затем запускать всю модель вне Python.

Чтобы использовать TensorFlow, нужно его сначала импортировать:

```
import tensorflow as tf
```

В TensorFlow требуемые операции выражаются с помощью символьных переменных, поэтому создадим переменную для тренировочных данных:

$$x = tf.placeholder(tf.float32, [None, 784]).$$

В данном случае x – это не какой-то заранее заданный тензор, а так называемая заглушка (placeholder), которую мы заполним, когда попросим TensorFlow произвести вычисления. Чтобы использовать произвольное число 784-мерных векторов для обучения, в качестве одной из размерностей указываем None. Для TensorFlow это значит, что данная размерность может иметь произвольную длину.

Кроме заглушки для тренировочных данных, также потребуются переменные, которые мы будем изменять при обучении нашей модели:

$$W = tf.Variable(tf.zeros([784, 10])),$$
$$b = tf.Variable(tf.zeros([10])).$$

Здесь W имеет размерность 784_10, так как мы собираемся умножать вектор размерности 784 на W и получать предсказание для 10 возможных меток, а вектор b размерности 10 – это свободный член, bias, который добавляется к выходу.

После импортирования нужных модулей и объявления всех переменных нашу модель на TensorFlow можно записать в одну строчку:

$$y = tf.nn.softmax(tf.matmul(x, W) + b).$$

Перемножаем матрицы x и W с помощью tf.matmul(x,W), затем добавляем к результату b и для получения вероятностей классов применяем tf.nn.softmax.

Для того чтобы обучить модель, нужно зафиксировать некий способ оценки качества предсказаний (именно эту оценку мы и будем в конечном счете оптимизировать). Опишем в терминах TensorFlow функцию потерь. Для исходной разметки понадобится заглушка:

$$y\_ = tf.placeholder(tf.float32, [None, 10]).$$

Теперь функцию потерь тоже можно записать в одну строчку:

$$cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ * tf.log(y), reduction\_indices=[1])).$$

Посмотрим, что будет происходить от внутренних операций к внешним:

– вычисляем tf.log(y), логарифм каждого элемента y;

– умножаем каждый элемент y_ на соответствующий ему *tf.log(y)* (операция умножения на векторах, матрицах и тензорах здесь понимается покомпонентно);

– суммируем результат с помощью tf.reduce_sum по второму измерению (первое измерение – это примеры из тестового или валидационного множества, второе – возможные классы), то есть суммируем по размерности каждого вектора y; для этого мы указали в качестве параметра reduction_indices=[1];

– последняя операция *tf.reduce_mean* подсчитывает среднее значение по всем примерам в выборке.

Теперь можно попросить TensorFlow оптимизировать эту функцию. Так как граф вычислений полностью описан в терминах, понятных TensorFlow, все вершины в этом графе содержат известные классические функции, градиенты которых уже реализованы в TensorFlow, библиотека может воспользоваться алгоритмом обратного распространения ошибки для того, чтобы подсчитать градиенты, узнать, как веса влияют на функцию по-

терь, которую мы хотим минимизировать, и затем применить тот или иной алгоритм оптимизации для уточнения этих весов.

Для обучения модели будем использовать метод градиентного спуска со скоростью обучения 0;5:

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy).

Перед началом обучения инициализируем переменные:

init = tf.initialize_global_variables().

Теперь можно запускать обучение. Для этого создаем TensorFlow-сессию

sess = tf.Session()
sess.run(init)

и запускаем

for i in range(1000):batch_xs, batch_ys = mnist.train.next_batch(100)
sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys}).

На каждом проходе цикла выбираем случайные 100 примеров из обучающей выборки и передаем их в функцию train_step для обучения. Такой подход – это типичный стохастический градиентный спуск, применяемый в данном случае потому, что обучающая выборка слишком велика для того, чтобы проходить по ней целиком на каждом шаге обучения (собственно, так будет всегда на любых данных хоть сколько-нибудь реального размера). Вместо этого каждый раз выбираем небольшой новый случайный набор обучающих данных и используем его.

Осталось понять, насколько хорошо распознаются рукописные цифры. Модель на данный момент выдает предсказания в виде softmax-результатов – суммирующихся в единицу чисел, отражающих уверенность модели в том или ином ответе. Для того чтобы понять, какая метка предсказана для очередного изображения, можно просто взять максимальное значение из этих результатов.

В TensorFlow это выражается как tf.argmax – функция, выдающая позицию максимального элемента в тензоре по заданной оси. Для того чтобы понять, верно ли предсказана метка, достаточно просто сравнить между собой

tf.argmax(y, 1) и tf.argmax(y_, 1):correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1)).

Это дает на выходе список булевых значений, показывающих, верно или неверно предсказан результат; итоговой точностью может быть, например, среднее значение соответствующего вектора:

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)).

Остается только вычислить его и вывести (на всякий случай: цифры, которые получатся у вас, наверняка будут немного отличаться от наших, они зависят от многих случайных факторов, например от случайной инициализации, но порядок величины должен быть именно такой):

print("Точность: %s" % sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
>> Точность: 0.9154.

На каждом слое полносвязной сети повторяется одна и та же операция: на вход подается вектор, он умножается на матрицу весов и к результату добавляется вектор свободных членов; только тогда к результату применяется некоторая нелинейная функция активации.

Во всех сетях, которые мы построили до сих пор, этот подход последовательно использовался, независимо от структуры или происхождения данных. Будь то изображения, текст

или музыка, мы многократно применяем аффинные преобразования на каждом уровне нашей сети, сначала преобразовывая данные в векторную форму.

Большинство типов данных имеют заранее известную внутреннюю структуру. Базовым примером такой структуры является изображение, которое обычно представляется в виде массива числовых векторов: если изображение черно-белое, то это массив только интенсивности, а если цветное, то массив векторов из трех чисел, представляющих интенсивность трех основных цветов (красный, зеленый и черный в стандартном RGB, синий, зеленый и красный в трех типах колбочек в человеческом глазу и т.д.). Если обобщить такую подструктуру до максимальной общности, то описание будет следующим:

1) исходные данные представляют собой многомерный массив («тензор»);

2) между измерениями этого массива имеется одна или несколько осей, порядок вдоль которых играет важную роль; например, это может быть расположение пикселей на изображении, временная шкала музыкального произведения, порядок слов или символов в тексте;

3) другие оси представляют собой «каналы», описывающие свойства каждого элемента в соответствии с предыдущим набором осей; например, три компонента для изображения, два компонента (левый и правый) для стереозвука и т.д.

При обучении полносвязных нейронных сетей это дополнительное знание структуры задачи никак не используется. Так в сети распознавания рукописных чисел изображение 28 x 28 пикселей было просто преобразовано в вектор длиной 784 и передано в качестве входных данных. Получается, что наши аффинные преобразования не учитывают структуру изображения, топологию данных.

Основная идея сверточной сети заключается в том, что обработка части изображения должна быть очень частой, вне зависимости от точного расположения этой части. Конечно, важную роль играет взаимное расположение объектов, но сначала их еще нужно распознать, причем это распознавание локально и независимо от фактического положения области с объектом в пределах более крупной картины.

Таким образом, сверточная сеть делает это предположение просто очевидным: давайте покроем вход маленькими окнами, допустим 5 x 5 пикселей, и извлечем признаки в каждом таком окне с помощью небольшой нейронной сети. Более того, и здесь важный момент – мы выбираем одни и те же признаки в каждом окне, а это значит, что будет только одна маленькая нейросеть, у которой будет всего 5 x 5 = 25 входов, и из каждого изображения она может извлечь очень многие различные входы.

Затем результаты этой нейронной сети можно представить заново, заменив окна 5x5 их центральными пикселями, используя второй сверточный слой, еще одну небольшую нейронную сеть и так далее. Вскоре мы увидим, что каждый сверточный слой имеет очень мало свободных параметров, особенно по сравнению с полносвязными аналогами.

Свертка – это особый тип линейного преобразования входных данных. Если xl является картой объектов в слое l, то результатом двумерного преобразования с ядром размера $2d + 1$ и весовой матрицей размера W $(2d + 1)$ x $(2d + 1)$ в следующем слое будет

$$y_{i,j}^{l} = \sum_{-d \le a,b \le d} W_{a,b} x_{i+a,j+b}^{l}$$

где $y_{i,j}^{l}$ – результат свертки на уровне l, а $x_{i,j}^{l}$ – ее вход, то есть выход всего предыдущего слоя. Иначе говоря, чтобы получить компоненту (i, j) следующего уровня, мы применяем линейное преобразование к квадратному окну предыдущего уровня, то есть скалярно

умножаем пиксели из окна на вектор свертки.

На рис. 2 проиллюстрировано применение свертки с матрицей весов W размера 3x3 к матрице X размера 5x5. Отметим, что умножение подматрицы исходной матрицы X, соответствующей окну, и матрицы весов W – это не умножение матриц, а просто скалярное произведение соответствующих векторов. Всего окно умещается в матрице X девять раз, и получается

$$\begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 4 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 2 & 3 & 1 & 0 \\ 0 & 4 & 3 & 2 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 9 & 5 & 4 \\ 8 & 8 & 10 \\ 8 & 15 & 12 \end{pmatrix}$$

Здесь мы обозначили свертку карты признаков X с помощью матрицы весов W через $X * W$, как принято обозначать свертку в математике.

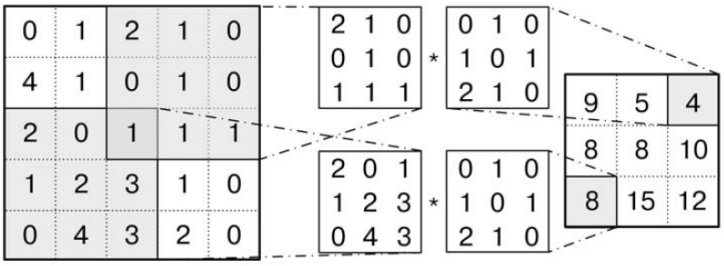

Рисунок 2. Пример подсчета результата свертки: два примера подматрицы и общий результат

Если размер свертки будет выражаться четным числом, то в одном из случаев в пределах суммирования неравенство станет строгим; здесь больше нет «естественного» выбора для центра окна, и его выбор из четырех вариантов может зависеть от реализации в конкретной библиотеке.

Это преобразование обладает как раз теми свойствами, о которых говорилось выше:

– свертка сохраняет структуру входных данных (порядок в одномерном случае, взаиморасположение пикселей в двумерном случае и т.д.), поскольку применяется к каждой части входных данных в отдельности;

– операция свертки обладает свойством разреженности, так как значение каждого нейрона следующего слоя зависит только от небольшой части входных нейронов (например, в полносвязной нейронной сети каждый нейрон зависит от всех нейронов предыдущего слоя);

– свертка повторно использует одни и те же веса несколько раз, поскольку они повторно применяются к разным частям ввода.

Почти всегда после свертки в нейронной сети следует нелинейность, которую можно записать так:

$$z_{i,j}^{l} = h(y_{i,j}^{l}) .$$

В качестве функции h часто используют ReLU, особенно в очень глубоких сетях, но и классические σ и $tanh$ тоже встречаются. Подробно останавливаться на типах нелинейностей, использующихся в сверточных сетях, мы сейчас не будем; наша первоочередная задача – сформировать интуиции по поводу сверточных сетей.

Вернемся к распознаванию рукописных цифр из датасета MNIST и попробуем существенно улучшить наше решение. Начнем снова с загрузки данных и импорта библиотек:

import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True).

Этот код скачает датасет MNIST (если он еще не был импортирован), а затем преобразует правильные ответы из него в one-hot представление: правильными ответами станут векторы размерности десять, в которых одна единица на месте нужной цифры.

Зададим заглушки для тренировочных данных:

x = tf.placeholder(tf.float32, [None, 784]),

y = tf.placeholder(tf.float32, [None, 10]).

В полносвязной нейронной сети мы считали вход просто вектором длины 784. Это сильно осложняло задачу нейронной сети: значения разных пикселей получались совершенно независимыми друг от друга, и полностью терялась информацию о том, какие из них расположены ближе друг к другу и, соответственно, должны больше влиять друг на друга. На этот раз мы будем применять сверточные сети, для которых пространственная структура изображений важна и в которых она постоянно используется. Поэтому переформатируем входной вектор в виде двумерного массива из 28x28 пикселей:

x_image = tf.reshape(x, [-1,28,28,1]) .

Формально массив теперь четырехмерный: первая размерность - 1 соответствует заранее неизвестному размеру мини-батча, а в четвертой размерности в каждом пикселе стоит только одно число. Для цветной картинки, например, в каждом пикселе могли бы стоять три числа, соответствующие интенсивностям красного, зеленого и синего цветов (RGB).

Дальше нужно создать сверточный слой. Во-первых, необходимо выбрать размер ядра свертки – для этого примера возьмем ядро размера 5x5; во-вторых, определиться с числом фильтров, которые будем обучать, – пусть на первом слое их будет 32; в-третьих, определить число каналов, то есть, сколько чисел задают каждый пиксель, – в нашем изображении, так как датасет черно-белый, цветовой канал всего один. Теперь можно создавать переменные для весов свертки:

W_conv_1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1)).

Инициализируем веса с помощью обрезанного нормального распределения с заданным стандартным отклонением 0;1 и ожиданием 0. Такое распределение выбрано потому, что в качестве функции активации планируется использовать ReLU. Массив из четырех чисел на входе – это форма тензора, который инициализируем с помощью нормального распределения. Первые два параметра задают размер ядра, третий отвечает за число входных каналов, четвертый определяет число выходных каналов. По сути, двигая наше окно-фильтр по исходному изображению, на выходе получаем вместо одного значения столбик из 32 значений, что можно представить себе, как применение 32 разных фильтров.

Теперь остается только задать свободные члены (biases). Несмотря на сложную структуру тензора весов, для свободных членов достаточно отвести всего 32 переменные: для каждого из 32 фильтров, независимо от того, к какой именно области изображения он приложен, результат свертки сдвигается на одно и то же число

b_conv_1 = tf.Variable(tf.constant(0.1, shape=[32])) .

Важно отметить, как мало здесь переменных: весов на сверточном слое всего 5 x 5 x 1 x 32 = 800 и 32 свободных члена. Когда мы задавали полносвязный слой для работы с теми же MNIST-изображениями, весов получалось 784 x 10 = 7840, а если добавить скрытый слой размером 32, как здесь, то на первом слое весов станет 784 x 32 = 25 088, что гораздо больше. Это яркая иллюстрация того, как сверточные сети используют дополнительную информацию о структуре входов для того, чтобы делать такую «абсолютную регуляризацию», объединяя массу весов. Мы знаем, что изображение имеет двумерную структуру, знаем его геометрию и заранее определяем, что хотели бы обрабатывать каждое окно в изображении одними и теми же фильтрами: нам все равно, в какой части картинки будут расположены штрихи, определяющие цифру 5, нужно просто распознать, что это именно 5, а не 8.

Теперь у нас определены переменные для всех весов сверточного слоя:

conv_1 = tf.nn.conv2d(x_image,W_conv_1, strides=[1, 1, 1, 1], padding="SAME") + b_conv_1.

Функция *tf.nn.conv2d* только применяет сверточные фильтры, она делает линейную часть работы, а функцию активации необходимо задать самостоятельно. В качестве функции активации возьмем ReLU:

h_conv_1 = tf.nn.relu(conv_1).

Чтобы соблюсти стандартную архитектуру сверточной сети, осталось только добавить слой субдискретизации:

h_pool_1 = tf.nn.max_pool(h_conv_1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME").

Функция *tf.nn.max_pool* определяет max-pooling слой, выбирая максимальное значение из каждого окна. Параметр *ksize* здесь как раз и задает размер этого окна, в котором мы выбираем максимальный элемент. Он имеет ту же структуру, что и *strides*. Здесь уже вполне можно представить себе ситуацию, когда первая компонента будет не равна единице и мы захотим выбирать «самые подходящие» из нескольких последовательных изображений. Можно даже задать первую размерность -1, тогда слой субдискретизации будет выбирать «самое подходящее» изображение из всего мини-батча.

Параметры *strides* и *padding* обозначают здесь то же самое, что и для сверточного слоя, только в этот раз мы двигаемся по изображению в обе стороны с шагом 2. Понятно, что после этого слоя размер изображения в обоих направлениях уменьшится вдвое, до 14 x 14.

По той же схеме, что и выше, добавим еще один сверточный слой и слой субдискретизации, используя на этом слое 64 фильтра:

W_conv_2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv_2 = tf.Variable(tf.constant(0.1, shape=[64]))
conv_2 = tf.nn.conv2d(h_pool_1, W_conv_2, strides=[1, 1, 1, 1], padding="SAME") + b_conv_2
h_conv_2 = tf.nn.relu(conv_2)
h_pool_2 = tf.nn.max_pool(h_conv_2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME").

Как правило, в глубоких нейронных сетях за сверточными слоями следуют полносвязные, задача которых состоит в том, чтобы «собрать вместе» все признаки из фильтров и перевести их в самый последний слой, который выдаст ответ. Но для начала необходимо из двумерного слоя сделать плоский. В TensorFlow это делается функцией reshape:

*h_pool_2_flat = tf.reshape(h_pool_2, [-1, 7*7*64]).*

Число 7 х 7 х 64 возникло из-за того, что дважды была применена субдискретизация и в последнем слое использовано 64 фильтра. Теперь остается только добавить полносвязные слои.

Добавляем первый слой из 1024 нейронов:

*W_fc_1 = tf.Variable(tf.truncated_normal([7*7*64, 1024], stddev=0.1))*
b_fc_1 = tf.Variable(tf.constant(0.1, shape=[1024]))
h_fc_1 = tf.nn.relu(tf.matmul(h_pool_2_flat, W_fc_1) + b_fc_1).

Регуляризуем его дропаутом:

keep_probability = tf.placeholder(tf.float32)
h_fc_1_drop = tf.nn.dropout(h_fc_1, keep_probability).

Теперь добавляем второй, самый последний слой с десятью выходами:

W_fc_2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))
b_fc_2 = tf.Variable(tf.constant(0.1, shape=[10]))
logit_conv = tf.matmul(h_fc_1_drop, W_fc_2) + b_fc_2
y_conv = tf.nn.softmax(logit_conv).

Определим ошибку и введем оптимизатор, используя алгоритм оптимизации Adam:

cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logit_conv, y))
train_step = tf.train.AdamOptimizer(0.0001).minimize(cross_entropy).

Оцениваем точность:

correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)).

Остается запустить обучение и дождаться результата:

init = tf.initialize_global_variables()
sess = tf.Session()
sess.run(init)
for i in range(10000):batch_xs, batch_ys = mnist.train.next_batch(64)
sess.run(train_step, feed_dict={x: batch_xs, y: batch_ys, keep_probability: 0.5})
print(sess.run(accuracy, feed_dict={x: mnist.test.images, y: mnist.test.labels, keep_probability: 1.}))
>> 0.9906.

Для сравнения сделаем то же самое в Keras. Суть процесса не меняется, но код получается заметно меньше. Как и в TensorFlow, в Keras набор данных MNIST можно загрузить средствами самой библиотеки:

from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
(X_train, y_train), (X_test, y_test) = mnist.load_data().

Подготовим данные к подаче их на вход сети, для чего потребуются три вспомогательные процедуры.

Во-первых, нужно будет так же превратить каждую картинку в двумерный массив:

```
batch_size, img_rows, img_cols = 64, 28, 28
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1).
```

Во-вторых, входные данные MNIST в Keras, в отличие от TensorFlow, представляют собой целые числа от 0 до 255. Можно было бы обучать сеть и на таких данных, но для единообразия приведем их к типу float32 и нормализуем от 0 до 1:

```
X_train = X_train.astype("float32")
X_test = X_test.astype("float32")
X_train /= 255
X_test /= 255.
```

В-третьих, переведем правильные ответы в one-hot представление. Для этого служит вспомогательная процедура to_categorical из keras.np_utils:

```
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10).
```

Теперь задаваем собственно модель. В Keras это выглядит примерно так же, как в TensorFlow, но некоторые названия и параметры более узнаваемы. Сначала инициализируем модель:

```
model = Sequential().
```

Теперь добавим сверточные слои. Нам понадобится слой Convolution2D, основными аргументами которого являются число фильтров и размер окна, аргумент border_mode имеет ровно тот же смысл, что аргумент padding в TensorFlow, а аргумент input_shape сообщает Keras, какой размерности тензор ожидать на входе:

```
model.add(Convolution2D(32, 5, 5, border_mode="same", input_shape=input_shape))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), border_mode="same"))
model.add(Convolution2D(64, 5, 5, border_mode="same", input_shape=input_shape))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), border_mode="same")).
```

В этом примере сохранена та же архитектура, что была выше. Отметим, что в Keras не нужно отдельно инициализировать переменные, которые будут затем использоваться в качестве весов или свободных членов в слоях: Keras сам понимает, сколько должно быть весов у той или иной сверточной архитектуры, и сам поймет, что по ним нужно модель оптимизировать. Единственное, чем ему нужно для этого помочь, – задать явно размерность input_shape.

Полносвязные слои тоже получаются достаточно просто. За «переформатирование» тензора, которое в TensorFlow делалось функцией tf.reshape, теперь отвечает дополнительный «слой», который называется Flatten и способен сам понять, тензор какой формы подается ему на вход:

```
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation("relu"))
model.add(Dropout(0.5))
model.add(Dense(10))
```

model.add(Activation("softmax"))

И все, можно компилировать и запускать обучение:

model.compile(loss="categorical_crossentropy",
optimizer="adam", metrics=["accuracy"])
model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=10,
verbose=1, validation_data=(X_test, Y_test))
score = model.evaluate(X_test, Y_test, verbose=0)
print("Test score: %f" % score[0])
print("Test accuracy: %f" % score[1]).

На выходе получится примерно такая картина:

Epoch 10/10
60000/60000 - loss: 0.0096 - acc: 0.9973 - val_loss: 0.0321 - val_acc: 0.9915
Test score: 0.03208116913667295
Test accuracy: 0.99150000000000005.

Как видим, нейронная сеть работает примерно так же, достигая в данном случае точности в 99,15 % на валидационном множестве MNIST.

Заключение. В результате проведенного исследования смогли улучшить выбранный показатель качества с 91,54 % до 99,15 %. Сначала мы определили полноценную нейронную сеть средством Keras. Затем улучшили качество, добавив сверточные слои. Дальнейшего улучшения удалось добиться путем включения случайного прореживания сети и выбора подходящего оптимизатора. Было выявлено преимущество подхода с различными размерами ядра свёртки над подходом с несколькими последовательными слоями с минимальным размером ядра свёртки для задачи распознавания символов из базы MNIST. Также было определено превосходство рассмотренного подхода и в скорости обучения нейронной сети – она существенно быстрее выходит на уровень высокой точности.

Список литературы

1. Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение = Deep Learning. – М.: ДМК-Пресс, 2017. – 652 с.
2. Ерёмин Д. М., Гарцеев И. Б. Искусственные нейронные сети в интеллектуальных системах управления. – М.: МИРЭА, 2015. – 75 с.
3. Козин, Н.Е. Поэтапное обучение радиальных нейронных сетей / Н.Е. Козин, В.А. Фурсов // Компьютерная оптика. – 2019. – № 26. – С. 138-141.
4. Björklund T., Fiandrotti A., Annarumma M., Francini G., Magli E. Robust license plate recognition using neural networks trained on synthetic images // Pattern Recognition. – 2019. – 93. – P. 134-146
5. Puranic A., Deepak K.T., Umadevi V. Vehicle Number Plate Recognition System: A Literature Review and Implementation using Template Matching // International Journal of Computer Applications. – 2016. –134(1). – P. 12-16.
6. Bharath B. P., Mahalakshmi S. Automatic license plate detection using deep learning techniques // International Journal of Scientific Research Today. – 2017. – 5(1) . – P. 107-112
7. Rizvi S. T. H., Patti D., Bjorklund T., Cabodi G, Francini G. Deep classifiers-based license plate detection, localization and Recognition on GPU-Powered Mobile Platform // Future Internet. – 2017. – http://www.mdpi.com/1999-5903/9/4/66 (01.02.2018).
8. Tlebaldinova A, Denissova N, Baklanova O, Krak Iu, Györök G. Normalization of Vehicle License Plate Images Based on Analyzing of Its Specific Features for Improving the Quality Recognition //Acta Polytechnica Hungarica. – 2020. – Vol. 17, No. 6. – P. 193-206. DOI: 10.12700/APH.17.6.2020.1.7.
9. Бакланова О.Е., Рыжкова Е.В., Азаматова Ж.К., Азаматов Б.Н. Разработка алгоритмов обработки изображений распознавания образов в задачах автоматического контроля качества //Вестник ВКГТУ. – 2019. – №4. – С. 77-81.

10. Baklanova O.E., Ryzhkova E.V., Iskakova M.M. Automatization of algorithms for visual quality control of coatings and contours on the products for medical purposes // Proceeding of International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing 2018, Madrid, Spain. – P.432-435.
11. Ryzhkova E., Baklanova O., Baklanov A., and Pronina M. Development and research of computer vision algorithms for visual control of geometric parameters of objects (defining the boundaries of the contour of the part) // Совместный выпуск научных журналов «Вестник» ВКГТУ им.Д.Серикбаева и «Вычислительные технологии» Института СО РАН. – 2018. – №3. – Т.1. – Ч.2. – С. 190-200.
12. LeCun, Y. Gradient Based Learning Applied to Document Recognition / Y. LeCun, L. Bottou, P. Haffner. – IEEE Press, 2018. – P.46.
13. Simard, P.Y. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis / P.Y. Simard, D. Steinkraus, J. Platt // International Conference on Document Analysis and Recognition (ICDAR), IEEE Computer Society. – 2018. – P. 958-962.
14. Хайкин, С. Нейронные сети: полный курс / С. Хайкин. – М.: Вильямс, 2016. – 1104 с.
15. LeCun, Y. Scaling learning algorithms towards AI / Y. LeCun, Y. Bengio – MIT Press, 2017.
16. Гаршин, А.А., Солдатова, О.П. Автоматизированная система распознавания рукописных цифр на основе свёрточной нейронной сети // Свидетельство об официальной регистрации программ для ЭВМ № 2010610988.
17. LeCun, Y. The MNIST database of handwritten digits. – http://yann.lecun.com/exdb/mnist. (дата обращения 12.09.2022).
18. LeCun, Y. Efficient BackProp in Neural Networks: Tricks of the trade / Y. LeCun, L. Bottou, G. Orr, K. Muller. – Springer, 2008. – 44 p.
19. Bishop, C.M. Neural Networks for Pattern Recognition. – Oxford University Press, 2019. – 498 p.
20. Gaussian blur. – http://en.wikipedia.org/wiki/Gaussian_blur. (дата обращения 12.09.2022)

References

1. Goodfellow Y., Bengio I., Courville A. Deep Learning = Deep Learning. – M.: DMK-Press, 2017. – 652 p.
2. Eremin D. M., Gartseev I. B. Artificial neural networks in intelligent control systems. – M.: MIREA, 2015. – 75 p.
3. Kozin, N.E. Step-by-step training of radial neural networks / N.E. Kozin, V.A. Fursov // Computer Optics. –2019. – No. 26. – P. 138-141.
4. Björklund T., Fiandrotti A., Annarumma M., Francini G., Magli E. Robust license plate recognition using neural networks trained on synthetic images // Pattern Recognition, 2019, 93, 134-146
5. Puranic A., Deepak K. T., Umadevi V. Vehicle Number Plate Recognition System: A Literature Review and Implementation using Template Matching // International Journal of Computer Applications. – 2016. – 134(1). – P. 12-16.
6. Bharath B. P., Mahalakshmi S. Automatic license plate detection using deep learning techniques // International Journal of Scientific Research Today. – 2017. – 5(1) . – P. 107-112.
7. Rizvi S. T. H., Patti D., Bjorklund T., Cabodi G, Francini G. Deep classifiers-based license plate detection, localization and Recognition on GPU-Powered Mobile Platform // Future Internet. – 2017. – http://www.mdpi. com/1999-5903/9/4/66 (02/01/2018).
8. Tlebaldinova A, Denissova N, Baklanova O, Krak Iu, Györök G. Normalization of Vehicle License Plate Images Based on Analyzing of Its Specific Features for Improving the Quality Recognition //Acta Polytechnica Hungarica. – Vol. 17, no. 6. – 2020. – P. 193-206. – DOI: 10.12700/APH.17.6.2020.1.7
9. Baklanova O.E., Ryzhkova E.V., Azamatova Zh.K., Azamatov B.N. Development of image processing algorithms for pattern recognition in tasks of automatic quality control // Bulletin of the EKSTU. – No. 4. – 2019. – P. 77-81.
10. Baklanova O.E., Ryzhkova E.V., Iskakova M.M. Automatization of algorithms for visual quality control of coatings and contours on the products for medical purposes // Proceeding of International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing 2018, Madrid, Spain. – P.432-435.
11. Ryzhkova E., Baklanova O., Baklanov A., and Pronina M. Development and research of computer vision algorithms for visual control of geometric parameters of objects (defining the boundaries of the contour of the part) // Joint issue of scientific journals "Bulletin" of the EKSTU named after D. Serikbaev and "Computational Technologies" of the Institute of the Siberian Branch of the Russian

Academy of Sciences. – No. 3, v. 1, Part 2. – 2018. – P. 190-200.
12. LeCun, Y. Gradient Based Learning Applied to Document Recognition / Y. LeCun, L. Bottou, P. Haffner. – IEEE Press, 2018. – P.46.
13. Simard, P.Y. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis / P.Y. Simard, D. Steinkraus, J. Platt // International Conference on Document Analysis and Recognition (ICDAR), IEEE Computer Society. – 2018. – P. 958-962.
14. Khaikin, S. Neural networks: full course / S. Khaikin. – M.: Williams, 2016. – 1104 p.
15. LeCun, Y. Scaling learning algorithms towards AI / Y. LeCun, Y. Bengio – MIT Press, 2017.
16. Garshin, A.A., Soldatova, O.P. Automated system for recognition of handwritten digits based on a convolutional neural network // Certificate of official registration of computer programs No. 2010610988.
17. LeCun, Y. The MNIST database of handwritten digits, http://yann.lecun.com/exdb/mnist. (accessed 12.09.2022).
18. LeCun, Y. Efficient BackProp in Neural Networks: Tricks of the trade / Y. LeCun, L. Bottou, G. Orr, K. Muller - Springer, 2008. – 44 p.
19. Bishop, C.M. Neural Networks for Pattern Recognition. – Oxford University Press, 2019. – 498 p.
20. Gaussian blur. – http://en.wikipedia.org/wiki/Gaussian_blur. (accessed 12.09.2022).