Министерство науки и высшего образования
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

Факультет «Фундаментальные науки»
Кафедра «Высшая математика»

ОТЧЕТ

ПО ОЗНАКОМИТЕЛЬНОЙ ПРАКТИКЕ

ЗА 4 СЕМЕСТР 2019—2020 ГОДА

Научный руководитель:
ст. преп. кафедры ФН1 _____ Гордеева Н. М.
подпись, инициалы

студент группы ФН1–41Б _____ Сафонов А. А.
подпись, инициалы

Москва
2020

Введение в капсульные нейронные сети

Содержание

Аннотация

В этой статье рассматривается ранее разработанный класс нейронных сетей в машинном обучении – капсульные нейронные сети(Capsule Neural Networks). Изучается принцип работы и обучения данной нейросети, а также преимущества и недостатки по сравнению с традиционными свёрточными нейронными сетями. Заменяя стандартные скалярные активации на векторные, капсульные нейросети готовы стать следующей ступенью развития в приложениях компьютерного зрения. Но для того, чтобы понять, действительно ли данные нейросети работают по-другому, нежели классические, стоит взглянуть на особенность капсул. Также рассматриваются конкретная область действия нейронной сети: распознавание рукописных цифр на изображениях базы данных MNIST.

1 Введение

Человеческий мозг – сложный углеродный компьютер, способный выполнять миллиарды операций в секунду. Механизм работы мозга обычно моделируется с помощью взаимодействия нейронов и нейронных сетей. Мышление, а также анализ информации являются результатом обработки входных сигналов между нейронами из различных слоёв нейронной сети. Также нейронные сети могут и меняться, обновляться, изменяя при этом веса

сигналов, передаваемых между нейронами. Такая модель человеческого мозга и использовалась для воспроизведения его возможностей в компьютерной симуляции – искуственной нейронной сети(ИНС).

Искусственные нейронные сети – математические модели, созданные по аналогии с нейронными сетями нашего мозга. Существует множество различных архитектур нейросетей, каждая из которых способна выполнять определенный класс задач. В данной статье мы поговорим о нейронных сетях, способных выполнять различную обработку и классификацию изображений, а также взглянем на метод их обучения.

2 Сверточные нейронные сети

В 1988 г. была предложена особая архитектура искусственных нейронных сетей(рис. 1) Яном Лекуном – сверточные нейронные сети,Convolutional Neural Networks(CNNs). На рисунке представлен один из вариантов архитектуры CNNs.

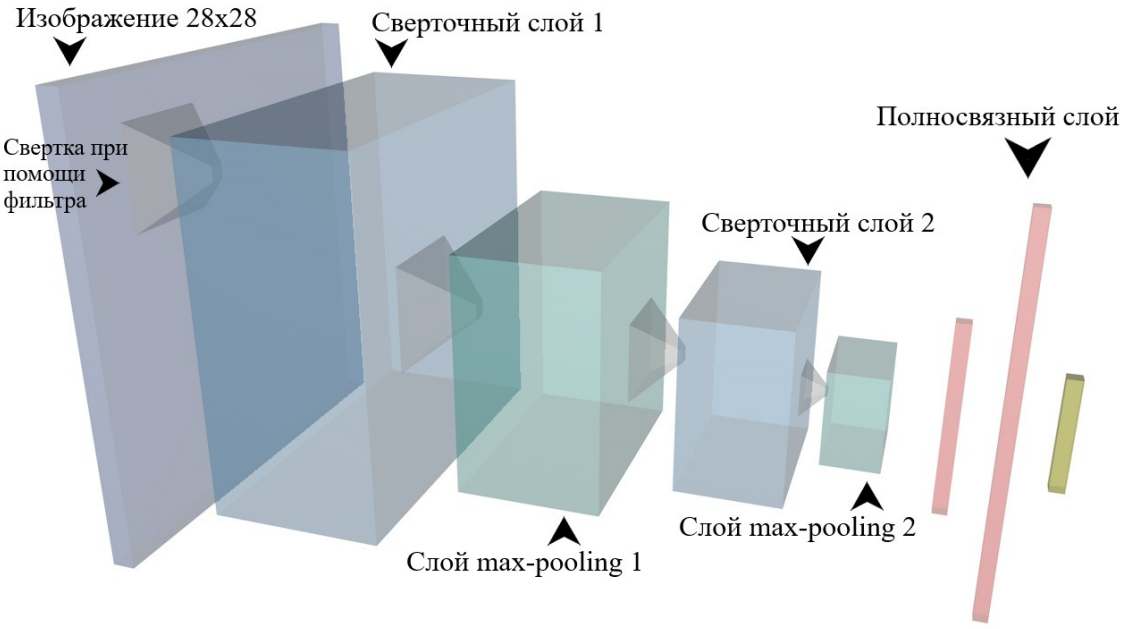

Рис. 1. Архитектура сверточной нейронной сети для обработки базы данных MNIST

На вход подается изображение. В начале алгоритма каждый его пиксель нормируется, чтобы его значение было в промежутке [0;1]. Затем, при помощи фильтра(ядра) происходит классическая свертка, в результате которой мы получаем карту признаков. После чего к полученной карте признаков применяется функция активации ReLU, чтобы избавиться от отрицательных значений. Завершением первого этапа служит слой max-pooling(подвыборки). Если вы забыли или не имеете представления о сверточном слое, то не волнуйтесь, мы его подробнее рассмотрим позже. Заметим, что таких этапов два, но они ничем друг от друга не отличаются. По завершении второго этапа получаем определенное количество более точных карт признаков. Завершающим слоем в сверточной нейронной сети сложит полносвязный слой или же классическая искусственная нейронная сеть, на выходе которой мы получаем наибольшую вероятность наличия какого-то класса объекта на исходном изображении. Данный класс нейросетей достаточно долго был инструментом, который выбирали в качестве решения проблем в компьютерном зрении. Пространственная локализация объектов с помощью CNN значительно выгоднее, чем с

другой архитектурой, применяемой к изображениям или видео. Однако и этот класс имеет свои недостатки. Фильтр или же по-другому ядро(kernel) в сверточном слое должен обучаться нахождению всех присутствующих объектов на входе. Таким образом трансформации, такие как вращение или затемнение объекта, могут нанести вред, в случае если тренировочная база данных не столь объемна.

Однако доктор Джофри Хинтон, известный всем как ученый в области машинного обучения, а также разработчик алгоритма обратного распространения ошибки backpropagation, последние годы искал варианты более усовершенствованной архитектуры нейронных сетей в компьютерном зрении, нежели архитектура CNNs. В основном, ему не нравился слой max-pooling и инвариантность детектирования только лишь к положению на изображении. Иными словами сверточная нейронная сеть испытывает большие трудности в распознавании одного и тото же объекта на изображении, но с другими параметрами(угол поворота, размер, освещённость и т. д.). В связи с этим относительно недавно появился новый класс нейронных сетей – CapsNet(Capsule neural Networks), представленный в статье [1], в котором применяется термин «капсула». По описанию авторов капсула – группа нейронов, представляющая собой наличие особенностей в добавок к параметрам, относящимся к созданию объекта определенного класса. Вопреки скалярным операциям фильтров в традиционных сверточных сетях, капсулы(они же многомерные векторы) могут предоставлять более точную информацию об объекте. Таким образом они должны быть способны закодировать не только наличие особенностей объекта, но и проведенные над ними преобразования на изображении.

3 Принцип работы и архитектура

Так что же конкретно представляют из себя капсульные нейронные сети, и каким образом они смогли достичь значительно лучших результатов в обработке базы данных MNIST, нежели CNNs? Разберемся во всем по порядку.

Итак, каждый объект имеет различные параметры. Глядя на изображение ты стараешься найти объекты, из которых оно состоит, и определить их соответствующие характеристики. Капсульная нейросеть состоит из множества капсул, действие которых и заключается в предположении присутствия и наличии соответствующих параметров определенного объекта в заданном местоположении. Отсюда можно сделать вывод, что основная задача CapsNet – обратное детектирование или **Inverse graphics**.

Рассмотрим отличный пример с домом и лодкой, представленный Aurélien Géron в презентации «Capsule Networks(CapsNets) – Tutorial». Допустим наша нейросеть состоит из 32 капсул(рис. 2).

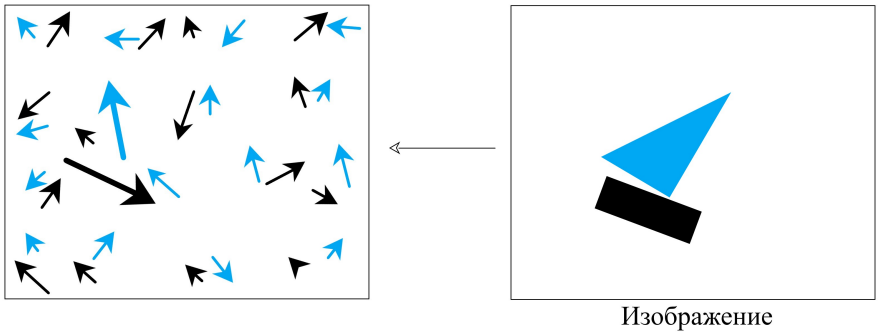

Рис. 2. Обратное детектирование

Стрелками на рисунке являются векторы. Иными словами капсула содержит вектор. Стрелки черного цвета нацелены на поиск прямоугольника, а синие – треугольника. Однако объединяя эти объекты, мы получаем лодку.

Длина каждого вектора соответствует вероятности нахождения того объекта, который ищет капсула. Можно заметить, что длина большинства векторов достаточно короткая. Это означает, что капсулы ничего не нашли. Однако пара векторов имеет достаточно большую длину. Значит капсулы уверены, что нашли искомые объекты.

Для определения направления вектора необходимо закодировать индивидуальные параметры объекта, такие как угол поворота в предыдущем примере. Но также этими параметрами могут быть и толщина, растяжение или сжатие, цвет и т. д. Поэтому в нейронных сетях используются векторы с большей размерностью.

Ниже представлена архитектура CapsNets, упомянутая в статье Хинтона [1](рис. 3). Видно, что в общей сложности она состоит только из 2-х сверточных и одного полносвязного слоев. На входе нейросеть принимает изображение 28×28×1, а на выходе воссоздает его с небольшими изменениями. В данном исследовании мы будем использовать ч/б изображение, чтобы проще понять принцип действия нейросети, однако в общем случае большинство изображений обычно содержат каналы RGB и, возможно, дополнительный канал альфа, отвечающий за прозрачность.

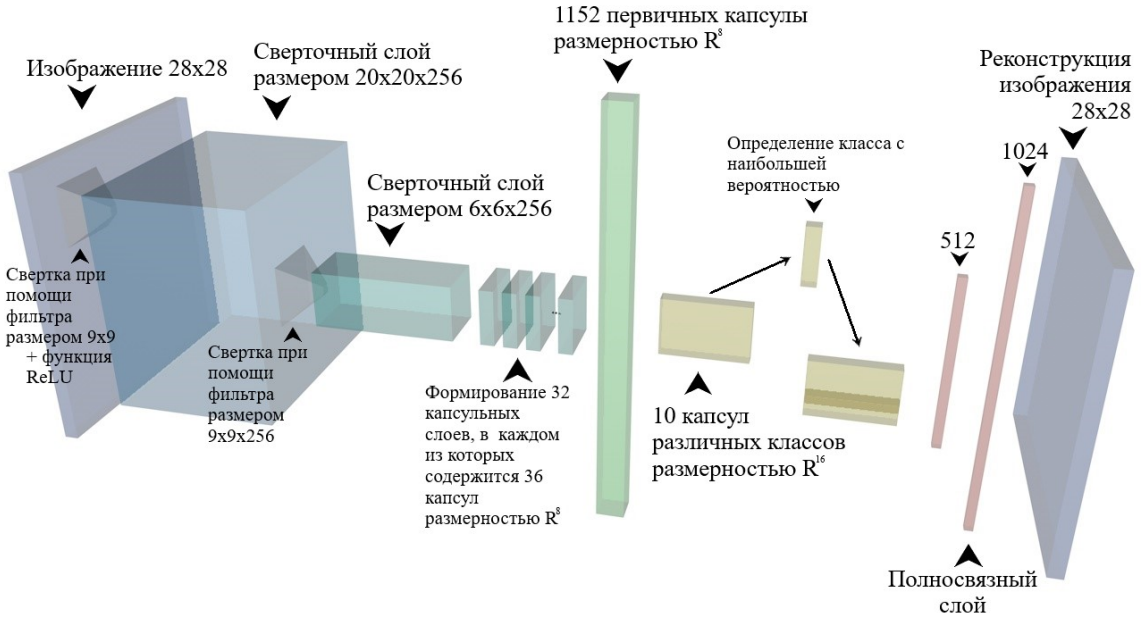

Рис. 3. Архитектура сверточной нейронной сети

3.1 Первый сверточный слой

Данный сверточный слой преобразует интенсивность пикселей входного изображения в локальные особенности «сканируемого» объекта, при помощи фильтра. Фильтр или же ядро(kernel) – окно(обычная матрица весов), скользящее по всей области изображения и находящее определенные признаки объектов. К примеру, если сеть обучали на множестве кошек, то один из фильтров мог бы выдавать «наибольший сигнал» в области ушей, лап или хвоста,в то время как другое ядро могло бы определять другие признаки. Такой этап называют классической сверткой(рис. 4). В описании работы слоя в статьях [1] и [3] использовалось ядро размером 9×9. Такой размер фильтра выбран чтобы модифицировать детектирование требуемых признаков. Всего используется 256 фильтров с

произвольным набором весов. Изначально веса всех ядер устанавливаются произвольно в пределах $[-0.5, 0.5]$, а уже в процессе обучения нейросети изменяются. Опять же такое количество требуется для оптимизации детектирования. В результате чего получаем 256 различных карт признаков размера 20×20 по формуле:

$$(S_{image} - S_{kernel}) + 1 = S_{map} \tag{1}$$

- S_{image} – размер изображения
- S_{kernel} – размер фильтра
- S_{map} – размер карты признака

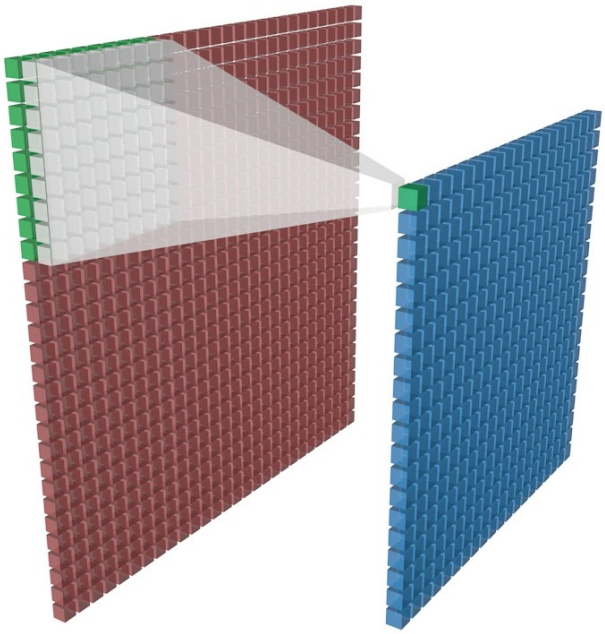

Рис. 4. Сверточный слой

После чего к каждой карте нашего слоя применяем линейную функцию ReLU, чтобы избавиться от отрицательных элементов(рис. 5). Функция действительна на всем $\mathbb{R}$, и чаще всего используется в сверточных слоях для «отсечения» ненужных деталей при детектировании.

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leqslant 0 \end{cases} \tag{2}$$

3.2 Слой первичных капсул

Далее нас ждет еще один сверточный слой. Он не сильно отличается от предыдущего, за исключением того, что свертка будет проходить по всему предыдущему слою размером 20×20×256, составленному из карт признаков, полученных в первом сверточном слое, при помощи фильтра 9×9×256. В предыдущем слое мы искали различные признаки объекта на изображении. Здесь же основная задача – поиск более сложных форм объектов, состоящих из признаков, найденных в предыдущих свертках. Но теперь уже шаг равен 2.

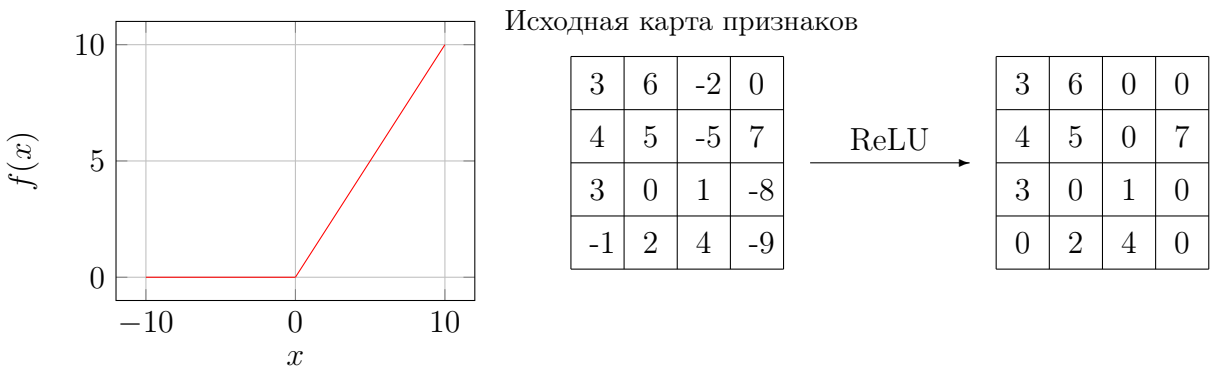

Рис. 5. Линейная функция активации ReLU

Это означает, что вместо перемещения на 1 пиксель, ядро каждый раз будет перемещаться на 2 пикселя(рис. 8). Больший шаг был выбран, чтобы обрабатывать входные данные быстрее. В результате чего снова получаем 256 различных карт признаков, только уже размера 6×6 по формуле:

$$\frac{S_{map_1} - S_{kernel} + 1}{2} = S_{map_2} \tag{3}$$

- S_{map_1} – размер карты признака
- S_{kernel} – размер фильтра
- S_{map_2} – размер новой карты признака

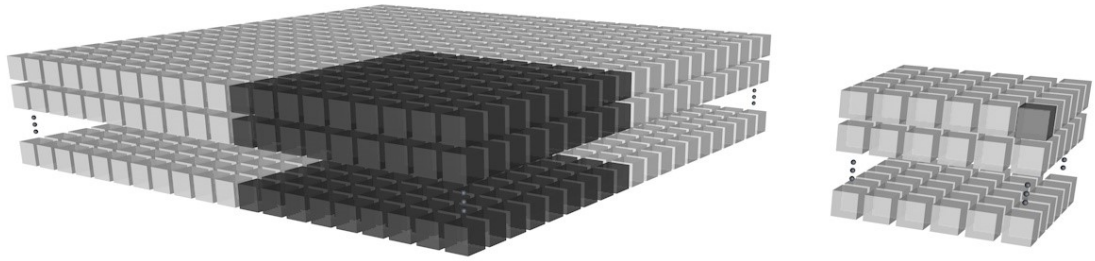

Рис. 6. Свертка по более сложным формам объекта

Теперь мы можем разделить полученный трехмерный массив данных на набор векторов для каждого местоположения. Разрежем этот стек на 32 колоды по 8 карт размера 6×6 в каждой. Будем называть каждую колоду капсульным слоем. Получается, что каждый такой слой имеет по 36 капсул, каждая из которых является вектором $v_j \in \mathbb{R}^8$ (рис. 7).

Теперь капсулы стали для нас новыми пикселями. Используя один такой пиксель, мы можем быть вполне уверены, нашли ли мы нужную грань в этой области или нет. Как мы уже помним, длина вектора дает нам вероятность присутствия искомого объекта на изображении.

В капсуле мы можем хранить 8 значений для данной локации объекта на изображении. Это дает возможность хранить больше информации, чем просто знание о наличии или отсутствии формы в этом месте. Конечно мы могли разделить стек и на большее число колод, но тогда и размерность векторов каждого капсульного слоя уменьшилась

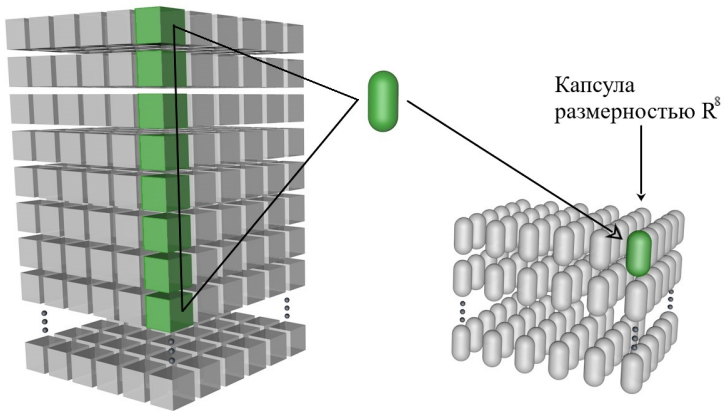

Рис. 7. Инициализация капсул

бы соответственно. Это означает, что капсула смогла бы хранить меньше информации о местоположении, объекте и его характеристиках. А как мы знаем, объект может обладать такими характеристиками как расположение, размер, ориентация, деформация, оттенок, текстура и т. д. Поэтому размерности капсулы может не хватить на вмещение всех требуемых параметров.

Так как теперь мы работаем с капсулами, а капсула – вектор в $\mathbb{R}^8$, то необходимо убедиться в том, что длина каждого вектора не превышает 1, так как он представляет вероятность события. Для этого необходимо нормировать вектор, используя новую нелинейную функцию активации, представленную в статье [1].

$$\mathbf{u}_j = \frac{||\mathbf{u}_j||^2}{1 + ||\mathbf{u}_j||^2} \frac{\mathbf{u}_j}{||\mathbf{u}_j||} \tag{4}$$

Функция также имеет название «**squash function**». Она преобразовывает вектор таким образом, что меняется только его длина, а ориентация в пространстве остается неизменной. Таким образом мы получаем нормированный вектор(рис. 6).

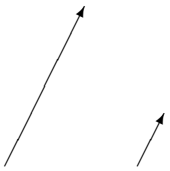

Рис. 8. Применение нелинейной функции $\mathbf{u} := \text{squash}(\mathbf{u})$

3.3 Динамическое взаимодействие между капсулами

В следующем этапе определяется, какую информацию отправить в следующий слой, а именно в слой цифровых капсул. Так как капсульные сети основаны на работе с векторами, то используется новый метод – направление по соглашению (routing by agreement). Это модифицированный метод работы с нейронными связями, по сравнению с другими классическими методами такими как **max–pooling**, при использовании которого можно потерять все связи, кроме самых значимых. Но обо всем по порядку.

3.3.1 Инвариантность детектирования

Ключевая особенность капсульных нейронных сетей – хранение детальной информации о местоположении объекта и о его представлении. Вспомним вышеупомянутый пример Aurélien Géron о представлении лодки в виде капсульных векторов(рис. 2). Если ее слегка повернуть на изображении, то векторы объектов, составляющих лодку, тоже слегка повернутся в этом же направлении(серым цветом представлено исходное положение объектов и векторов)(рис. 9). Это и есть инвариантность детектирования.

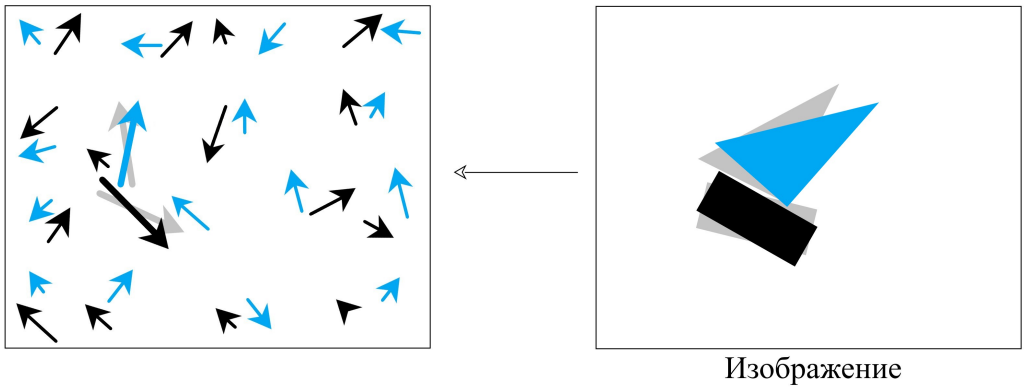

Рис. 9. Изменение положения векторов с изменением положения лодки

В традиционных сверточных нейронных сетях на такой случай обычно есть несколько слоев подвыборки или, как их чаще называют, слои **max–pooling**. Он уменьшает размер предыдущего слоя путем выбора пикселя с максимальным значением в конкретной области и отправлением его в следующий слой. К примеру по карте признаков из левого верхнего угла движется фильтр 2×2 с шагом 2(рис. 10).

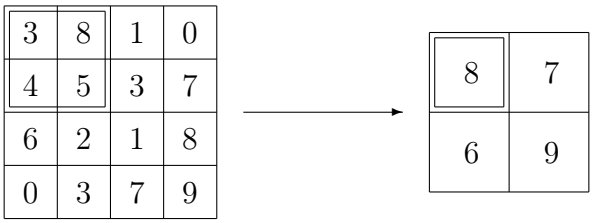

Рис. 10. Слой max–pooling(подвыборки)

К несчастью такие слои имеют тенденцию терять информацию об объекте, такую как о его местоположении и позиции. Это не является столь серьезной проблемой, если нужно классифицировать все изображение, однако будет затруднительно производить точную сегментацию или детектирование объекта(для него требуется информация о местоположении и позиции). Тот факт, что капсулы могут представлять различные положения того же объекта, дает нам понять, что они являются многофункциональными в таких приложениях.

3.3.2 Иерархия частей

Теперь давайте посмотрим, как **CapsNet** справляются с объектами, которые состоят из его иерархических частей. Для этого рассмотрим очередной пример Aurélien Géron. Пусть

у нас есть объект лодка, и мы хотим перейти к ее составляющим, имеющим соответствующие параметры. Такими составляющими являются прямоугольник и треугольник. Также мы можем изобразить дом, используя те же части, что и у лодки, только они будут расположенны по-другому(рис. 11).

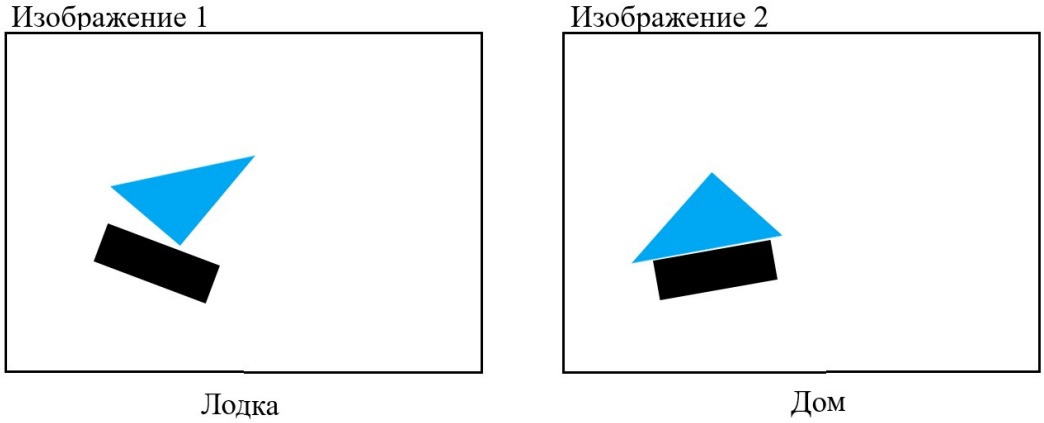

Рис. 11. Изображения лодки и дома с одинаковыми составляющими

Итак, цель будет заключаться в следующем: нужно определить, что объект на Изображении 1, состоящий из прямоугольника и треугольника, имеющих соответствующие параметры(цвет, форма, угол поворота), представляет из себя лодку, но не дом.

С первыми этапами мы уже ознакомились: проходим через пару сверточных слоев, распределяем полученный стек на векторы, а затем нормируем их. Это дает нам выходные данные из слоя **PrimaryCaps**. Следующий этап содержит в себе большую вычислительную сложность, нежели остальные слои. Каждая капсула предыдущего слоя «старается предположить» выходное значение каждой капсулы следующего слоя. На первый взгляд это кажется сложным, поэтому давайте перейдем к примеру.

Выберем капсулу, отвечающую за поиск прямоугольника, будем называть ее капсула-1(рис. 12).

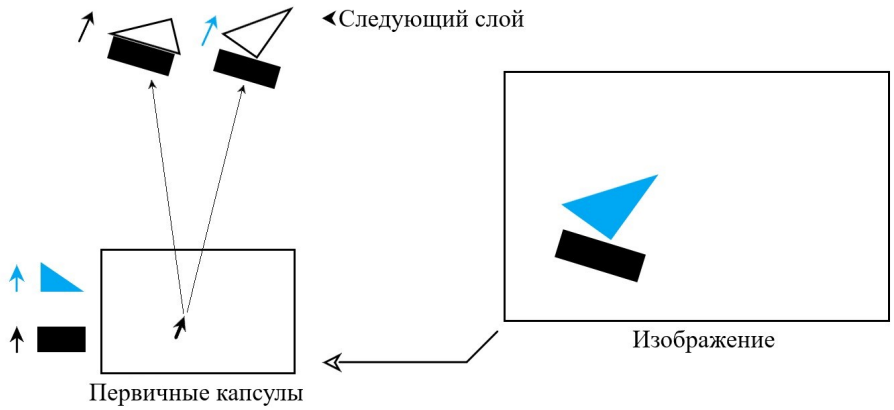

Рис. 12. Предположение значений следующего слоя при помощи капсулы-1

Предположим, что в следующем слое только 2 капсулы: капсула, отвечающая за дом и за лодку соответственно. Когда капсула-1 обнаруживает прямоугольник, она предполагает, что капсула, отвечающая за поиск дома, обнаружила бы дом, а капсула, отвечающая

за поиск лодки, обнаружила бы лодку. Такие варианты могут получиться в соответствии с положением прямоугольника.

Чтобы определить вероятность исхода обоих вариантов, нужно вычислить скалярное произведение вида:

$$\hat{\mathbf{u}}_{i|j} = \mathbf{W}_{ij}\mathbf{u}_i \tag{5}$$

- $\mathbf{W}_{ij}$ – матрица трансформации(в наших исследованиях мы будем использовать матрицу 8×16 для увеличения размерности вектора или же увеличения места для новых параметров)
- $\mathbf{u}_i$ – вектор капсулы-1

В процессе обучения нейросети будут постепенно обучаться и веса матриц $\mathbf{W}_{ij}$ для каждой пары капсус предыдущего и следующего слоев. Иными словами матрица будет обучаться нахождению всех связей между составляющими объекта, таких как угол между парусом и корпуслом лодки, например.

Теперь давайте выясним, что может предположить капсула, отвечающая за поиск треугольника. Назовем ее капсула-2(рис. 13).

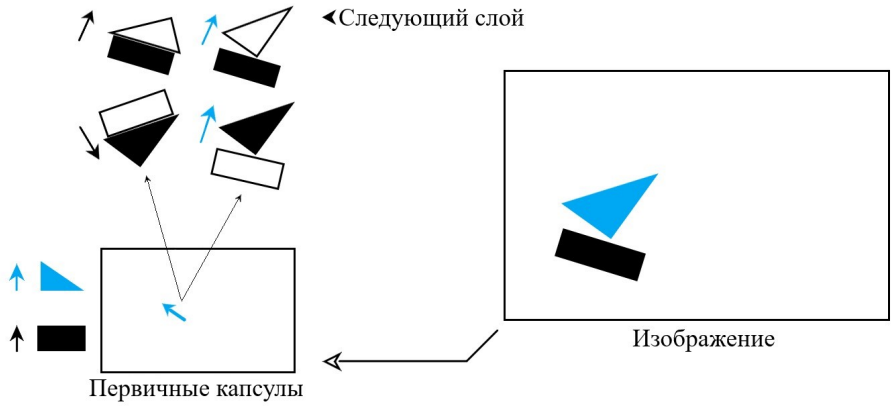

Рис. 13. Предположение значений следующего слоя при помощи капсулы-2

На этот раз, когда капсула-2 обнаруживает треугольник, возникает снова 2 случая: капсула, отвечающая за поиск лодки обнаружила бы лодку, а капсула, отвечающая за поиск дома, обнаружила бы дом, но в перевернутом виде. Такие варианты могут получиться в соответствии с положением треугольника.

3.3.3 Направление по соглашению

После нахождения всех вариантов предсказаний можно заметить, что в одном месте предсказания капсулы-1 и капсулы-2 достаточно сильно совпадают(рис. 14).

Отсюда можно сделать вывод: обе капсулы предыдущего слоя «уверены» в том, что выходным значением следующего слоя будет лодка, а не дом. Поэтому есть все основания предполагать, что такие части объекта как прямоугольник и треугольник являются составляющими именно лодки.

Зная это, можно сделатль вывод, что выходные значения капсулы-1 и капсулы-2 действительно относятся только к капсуле, отвечающей за поиск лодки, и нет необходимости

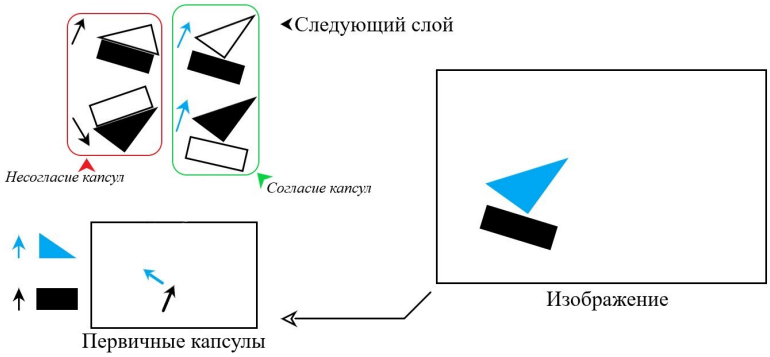

Рис. 14. Совпадение предсказаний капсулы-1 и капсулы-2

отправлять эти данные в любую другую капсулу, это может составить нам больше работы. Данные должны быть направлены только по «обоюдному согласию капсул». Это и называется направление по соглашению(routing by agreement).

У этого алгоритма есть несколько преимуществ:

1. после того как капсулы предыдущего слоя направили выходные данные в наиболее подходящую капсулу следующего слоя, капсулы из слоя ниже получат более отфильтрованный входной сигнал, а также будут тщательнее определять позицию объекта.

2. рассматривая варинты активации, мы легко можем управлять иерархическими составляющими, и точно знать, какая часть к какому объекту относится(к примеру прямоугольник относится к лодке или треугольник относится к лодке и т. д.).

3. направление по соглашению может без труда распознать наложенные друг на друга объекты на изображении(такой пример мы рассмотрим позже).

А теперь давайте посмотрим, как направление по соглашению реализовано в Capsnet. Ниже представлены различные позиции лодки, предполагаемые в следующем слое(рис. 15). Точками на рисунке изображены векторы предсказаний. К примеру одна из точек может показать, что капсула-1 может «думать» о самой удачной позиции лодки, а другая – что капсула-2 «думает» об этом. Если мы предположим, что есть еще много других низкоуровневых капсул, то можем получить множество векторов с предсказаниями, как одна такая точка. В этом примере присутствует 2 параметра: угол поворота лодки и ее размер. Как упоминалось ранее, параметров у объекта может быть значительно больше, но для простоты представления мы используем не больше 2–х(используем пространство $\mathbb{R}^2$).

Итак, первое, что нужно сделать, это вычислить среднее значение всех предсказаний. Это даст нам вектор, обозначенный оранжевым крестом(рис. 16).

Определив среднее значение, найдем расстояние между каждым вектором предсказаний и вектором среднего значения. Здесь мы будем использовать евклидово расстояние, однако **CapstNet** на самом деле используют скалярное произведение. Сейчас мы хотим

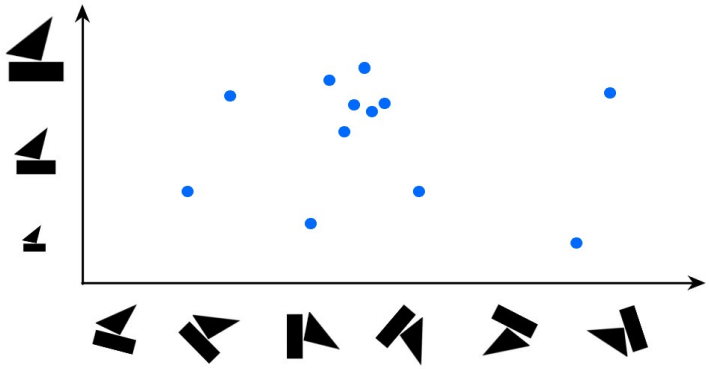

Рис. 15. Графическое представление набора предположений

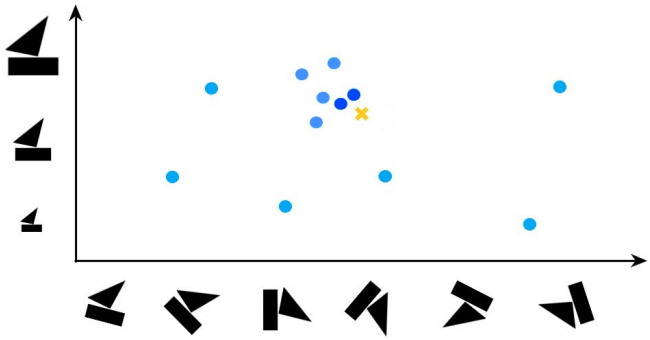

Рис. 16. Нахождение среднего значения

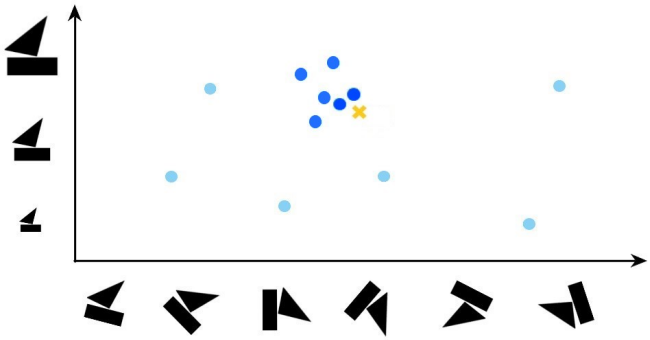

Рис. 17. Определение весов предсказаний

определить, насколько каждый вектор предсказаний «согласен» с вектором среднего значения. Используя эту меру, мы можем обновлять вес каждого вектора предсказаний соответсвтенно. Мы видим, что векторы, находящиеся далеко от среднего значения, имеют достаточно маленькие веса(изображены оттенками голубого цвета), а векторы, находящиеся близко – достаточно большие веса(изображены синим цветом)(рис. 17).

Определим снова среднее значение, но уже учитывая важность каждого вектора(его вес), а затем вычислим расстояние от вектора среднего значения до остальных(рис. 18).

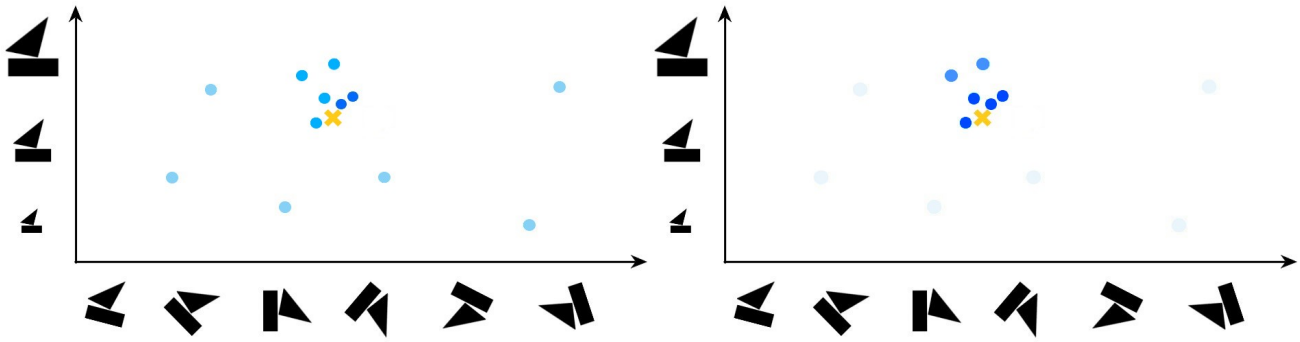

Рис. 18. Определение среднего значения с учетом важности всех векторов предсказаний

Можно заметить, что среднее значение постепенно смешается к центру скопления векторов. После этого мы снова обнавляем веса всех предсказаний с учетом нового положения вектора среднего значения. Мы также можем повторить этот процесс еще пару раз. На практике 3-5 итераций обычно бывает достаточно.

Теперь мы знаем, как **CapstNet** определяет требуемый набор предсказаний. Давайте

теперь взглянем, как работает весь алгоритм в более подробных деталях.

Ниже представлен алгоритм динамического взаимодействия между капсулами, представленный в статьях [1] и [3].

begin
  Для $\forall$ i–тых капсул слоя l и j–тых капсул слоя $(l+1)$: $b_{ij} \leftarrow 0$;
  for r *iterations* **do**
    для $\forall$ i–тых капсул слоя l: $\mathbf{c}_i \leftarrow \text{softmax}(\mathbf{b}_i)$;
    для $\forall$ j–тых капсул слоя $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$;
    для $\forall$ j–тых капсул слоя $(l+1)$: $\mathbf{v}_j \leftarrow \text{squash}(\mathbf{s}_j)$;
    для $\forall$ i–тых капсул слоя l и j–тых капсул слоя $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}\mathbf{v}_j$;
  end
end

Algorithm 1: Алгоритм динамического взаимодействия между капсулами

Итак в начале алгоритма мы устанавливаем каждому предсказанию, представленному на рис. 12 и 13, вес b_{ij}, равный 0(то есть для всех 4–х случаев). Затем к каждой первичной капсуле(в данном случае k=2) мы применяем функцию «**softmax**» вида:

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})} \tag{6}$$

В действительности это дает нам веса для всех предсказаний, равный 0.5 каждый(рис. 19).

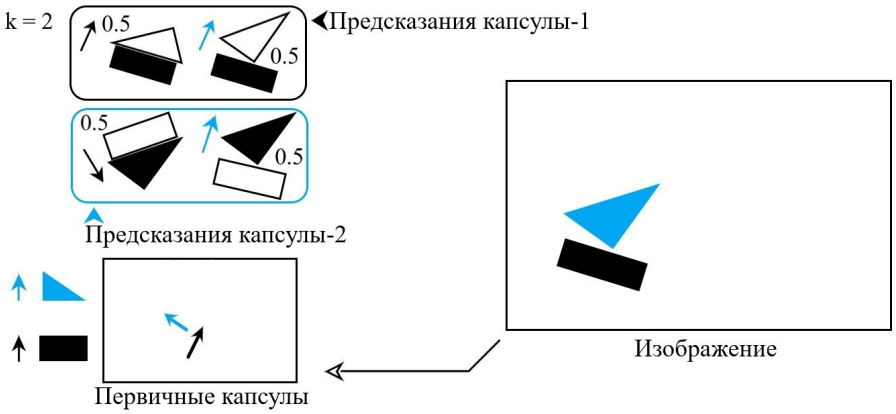

Рис. 19. Определение весов каждого предсказания

После этого мы вычисляем взвешенную сумму предсказаний для каждой капсулы следующего слоя, используя формулу (5):

$$\mathbf{s}_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j|i} \tag{7}$$

В результате этого мы можем получить векторы, длина которых больше 1, поэтому следует применить функцию «**squash**» для их нормализации:

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1+||\mathbf{s}_j||^2}\frac{\mathbf{s}_j}{||\mathbf{s}_j||} \tag{8}$$

В итоге мы получаем выходные данные капсулы-1 и капсулы-2 в виде векторов. Однако это не окончательные значения, так как выполнена лишь одна итерация алгоритма. Сейчас мы можем заметить, какие предсказания оказались более точными.

К примеру капсула-1 сделала успешное предсказание для капсулы, отвечающей за поиск лодки(рис. 20). Она действительно представляет объект достаточно точно.

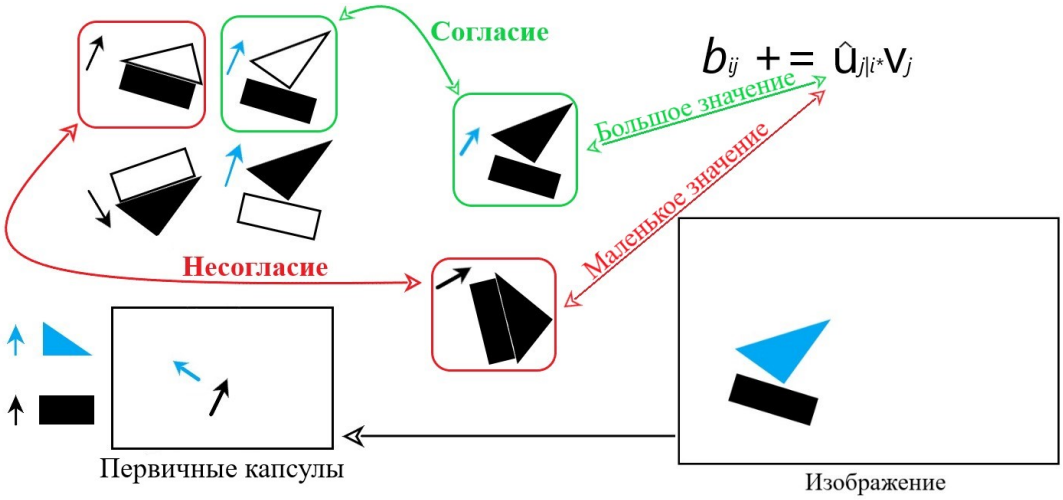

Рис. 20. Обновление весов векторов предсказаний

Такая оценка получилась путем вычисления скалярного произведения вектора предсказания $\hat{\mathbf{u}}_{j|i}$ и полученного вектора $\mathbf{v}_j$. Полученный результат мы просто добавляем и исходным весам b_{ij}. Отсюда следует, что некоторые веса предсказаний увеличились.

Когда достигается согласие между капсулами предыдущего и следующего слоев, скалярное произведение $\hat{\mathbf{u}}_{j|i}\mathbf{v}_j$ достаточно большое, а значит точные предсказания имеют большой вес.

С другой стороны капсула-1 выдала неточное предсказание для капсулы, отвечающей за поиск дома, поэтому скалярное произведение в таком случае будет достаточно маленьким, и веса b_{ij} векторов предсказаний сильно не возрастут(рис. 20).

На второй итерации мы обновляем веса b_{ij} путем тех же вычислений при помощи формулы(5)(рис. 21).

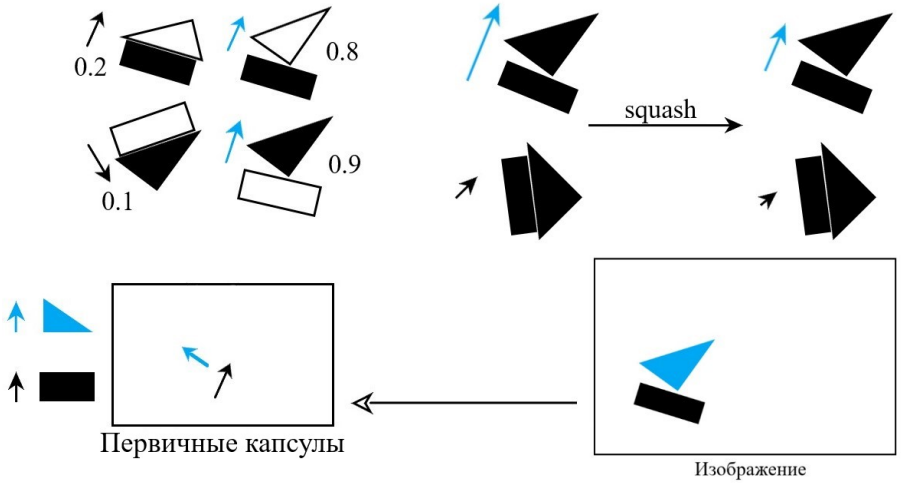

Рис. 21. Обновление весов векторов предсказаний

Видно, что вес предсказания о наличии лодки reccap теперь равен 0.8, в то время как вес предсказания о наличии дома этой же капсулы равен 0.2. Аналогично и с результатами капсулы-2. Поэтому наибольшие значения будут переданы капсуле, отвечающей за лодку, а не дом.

После прохождения второй итерации алгоритма замечаем, что вектор детектирования лодки достаточно длинный, поэтому его следует нормировать, а вектор дома – совсем маленький.

Итак, только за пару итераций мы полностью отказались от вероятности наличия дома на изображении, и без сомнений выбрали лодку.

3.3.4 Наложение объектов

Как уже говорилось ранее, алгоритм «**Routing by agreement**» отлично справляется с распознаванием объектов, нагроможденных друг на друга, как на картинке ниже(рис. 22).

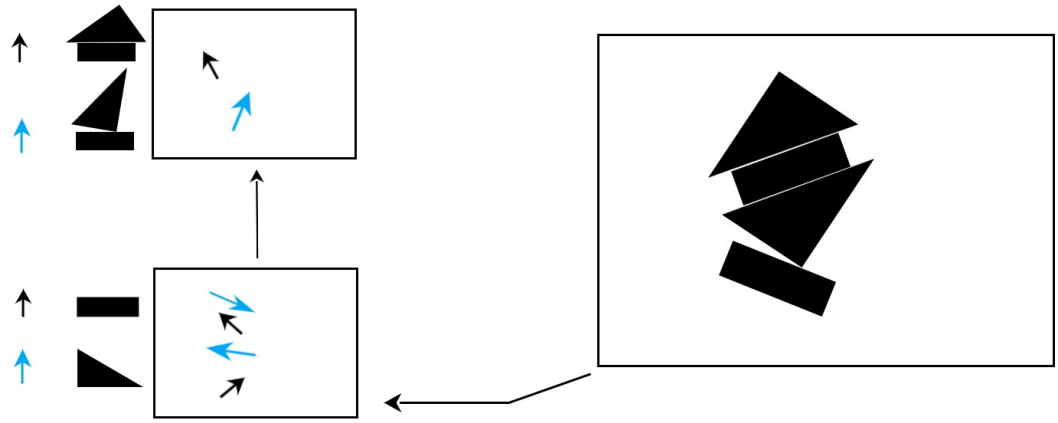

Рис. 22. Распознавание нагроможденных на изображении

Один из вариантов детектирования предусматривает наличие перевернутого дома как первую интерпретацию изображения(рис. 23).

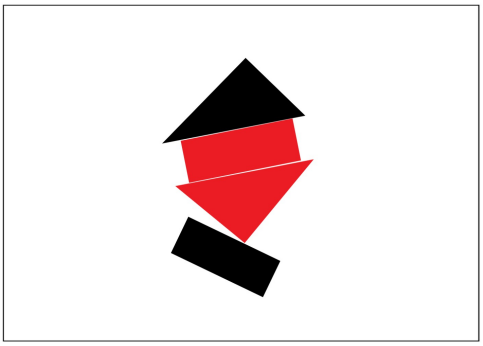

Рис. 23. Нахождение перевернутого дома

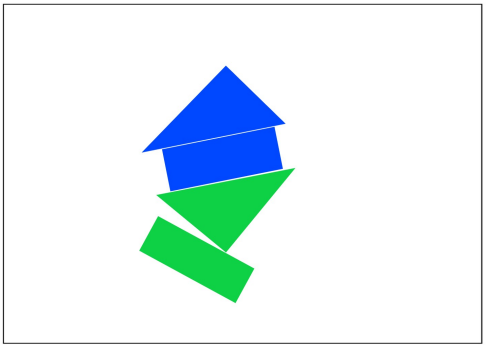

Рис. 24. Нахождение дома и лодки

Однако, если учитывать этот случай, тогда не будет объяснения присутствия нижнего прямоугольника и верхнего треугольника.

Иной вариант детектирования намного успешнее. В результате мы видим дом сверху и лодку снизу(рис. 24). При помощи направления по соглашению капсульная нейросеть

будет стремиться выбрать именно этот вариант по завершении алгоритма. Таким образом, наша **CapsNet** без труда справилась с неоднозначностью.

3.4 Цифровые капсулы

Итак, после определения наилучших предсказаний мы переходим к завершающему капсульному слою. В нем и происходит классификация поданного на вход изображения. Определяется она путем нахождения наибольшей длины вектора. Это означает, что длина вектора отвечает за вероятность присутствия конкретного класса, в котором конкретная капсула уверена. Если же рассматривать тот же пример с лодкой и домом, то, сравнивая длины полученных на предыдущем этапе векторов предсказаний, нейросеть без сомнений отнесет объект к классу «лодки», а не к классу «дом»(рис. 25).

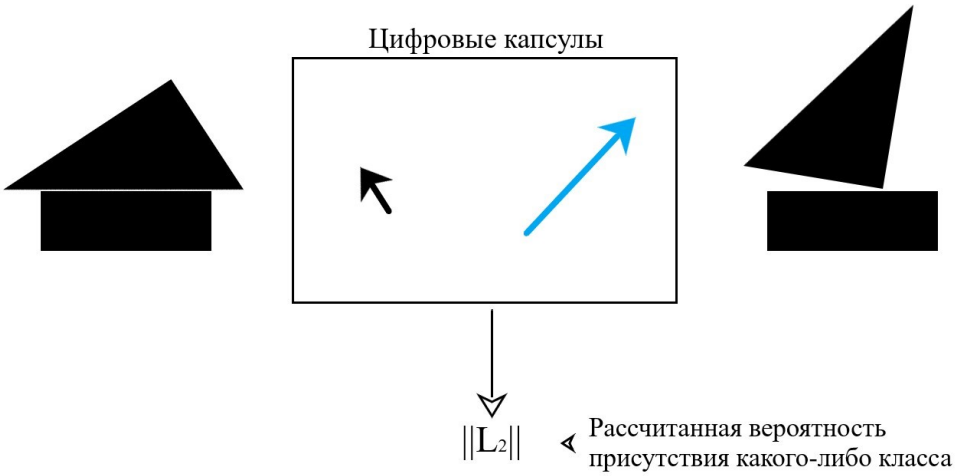

Рис. 25. Классификация детектируемого объекта

Вдобавок к этому можно натренировать классификацию нейронной сети, минимизируя потери перекрестной энтропии. Иными словами это поспособствует оптимизации оценки вероятностей исходных событий.

Однако в статье Хинтона [1] используется такая функция как «**margin loss**» – логистическая функция ошибки, при помощи которой возможно распознавать множество различных классов на изображении. Такая функция имеет вид:

$$L_k = T_k \max(0, m^+ - ||\mathbf{v}_k||)^2 + \lambda(1 - T_k)\max(0, ||\mathbf{v}_k|| - m^-)^2 \tag{9}$$

- $T_k = 1$, если класс k присутствует
- $m^- = 0.1$
- $m^+ = 0.9$
- $\lambda = 0.5$

Если не углубляться в подробности, то можно объяснить ее действие следующим образом: если объект класса k присутствует на изображении, то $||\mathbf{v}_k||^2$ должна быть не меньше 0.9, а если же объект не обнаружен, то $||\mathbf{v}_k||^2$ не должна превышать 0.1 . Общая потеря данных – сумма потерь каждого класса.

3.5 Реконструкция изображения

Одна из дополнительных задач, которую авторы статьи [1] и [3] решили рассмотреть, это воспроизведение исходных изображений после их классификации. Для этого добавляется еще один слой, представляющий собой классическую искусственную нейронную сеть(ИНС), — полносвязный слой или декодер(рис. 26).

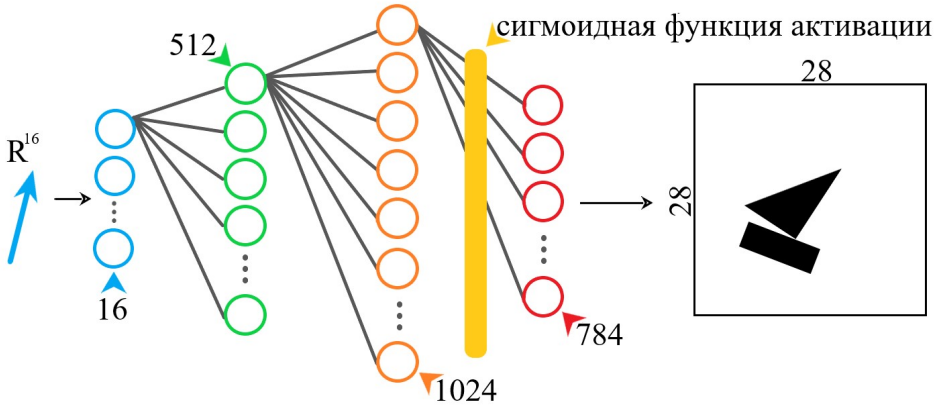

Рис. 26. Реконструкция при помощи классической ИНС

Сигмоидная функция активации – функция вида:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{10}$$

Функция является нелинейной, и действует на всем $\mathbb{R}$, а ее множество значений лежит в диапазоне $(0; 1)$(рис. 27).

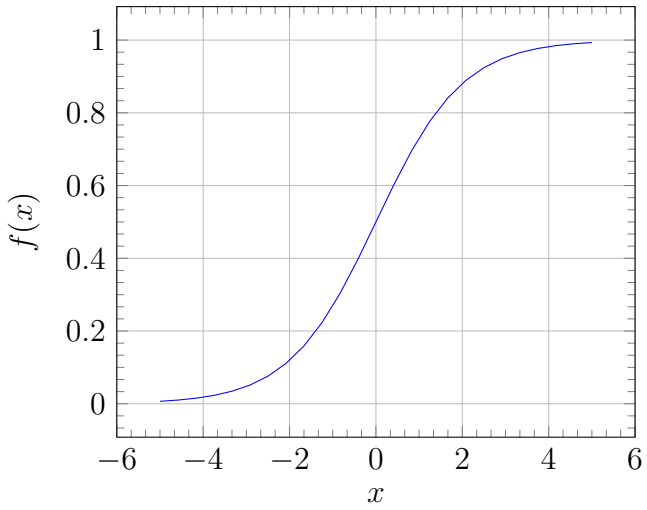

Рис. 27. Нелинейная функция активации сигмоида

Данная ИНН учится воссоздавать входное изображение, минимизируя разницу в квадрате между входным и восстановленным изображениями.

Итак, теперь общая потеря данных представляет собой:

$$\text{Loss} = \text{margin loss} + \alpha\ \text{reconstruction loss}$$

- $\alpha = 0.0005$ – коэффициент масштабирования(подобран для того, чтобы значение потери данных при реконструкции не привышало значение **margin loss** во время тренировки нейросети)

Преимущество добавления потери данных при реконструкции изображения заключается в тренировке нейросети сохранять всю информацию, требуемую для воссоздания картинки. Также коэффициент α играет некую роль регулировщика: он уменьшает риск переобучения нейросети.

4 Обработка базы данных MNIST

Как мы уже знаем, основной задачей капсульных нейронных сетей является классификация различных изображений. Самым распространенным способом обучения и проверки работоспособности **CapsNet** является классификация базы данных MNIST(Modified National Institute of Standards and Technology). Эта база данных является стандартом, предложенным национальным институтом стандартов и технологий в США с целью калибрации и сопоставления методов распознавания изображений при помощи машинного обучения в первую очередь на основе нейронных сетей.

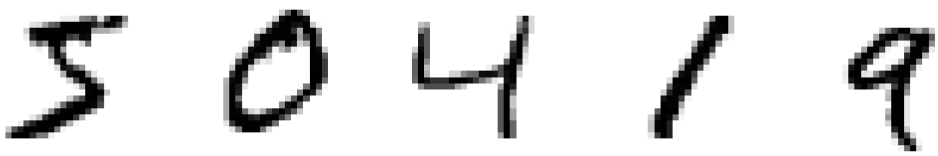

Рис. 28. Элементы базы данных MNIST

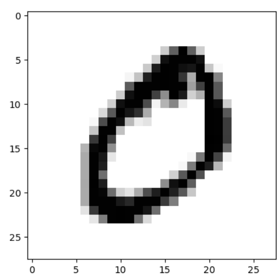

Рис. 29. Рукописная цифра 0

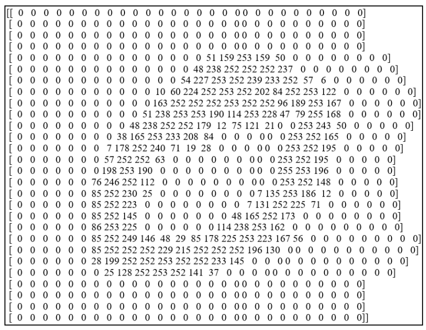

Рис. 30. Представление цифры 0 в виде массива значений пикселей

База представляет собой набор из 60.000 тренировочных экземпляров рукописных цифр и 10.000 тестовых экземпляров(рис. 28). Данный набор составлен на основе записей, собранных у 500 человек. Экземпляр является ч/б изображением размером 28×28 пикселей(рис. 29). Нужно понимать, что наша нейронная сеть работает не с самим изображением, а с его представлением в виде двумерного массива числовых значений цветов пикселей этого изображения(рис. 30).

В нашей работе основной задачей является обычная классификация и реконструкция изображений рукописных цифр. Также будет измеряться точность распознавания и

потеря данных при обработки информации, а также качество реконструкции исходного экземпляра.

5 Реализация в Python при помощи Tensorflow

На сегодняшний день самым простым и доступным для большинства языком является Python. Популярность этого языка для написания нейросети обусловлена тем, что для данной задачи в нем доступно огромное множество вспомогательных библиотек, которые могут в разы уменьшить количество строк кода и работы в целом. Так развитие дошло до того, что на языке Python нейросети можно писать в несколько строк кода. Познакомимся же с одной из таких библиотек, а также с наиболее комфортной средой разработки.

Tensorflow — открытая программная библиотека для машинного обучения, разработанная компанией Google для построения и тренировки нейронной сети с целью нахождения и классификации образов, стараясь приблизиться к качеству человеческого восприятия. Применяется как для исследований, так и для разработки собственных проектов Google. Основной API(application programming interface) для работы с библиотекой реализован для Python, а также существуют реализации для C#, C++, Haskell, Java, Go и Swift.

PyCharm — интегрированная среда разработки для языка Python. Предоставляет средства для анализа кода, графический отладчик, инструмент для запуска юнит–тестов и поддерживает веб–разработку на Django. PyCharm разработана компанией JetBrains на основе IntelliJ IDEA. Это кросс-платформенная среда разработки, совместимая с Windows, MacOS, Linux.

Ниже представлен программный код на языке Python с использованием библиотек Tensorflow 2.0 в интегрированной среде PyCharm, реализующий капсульную нейронную сеть, задача которой — детектирование, классификая и реконструкция базы данных MNIST.

5.1 Программный код

```
# Capsule Neural Networks: introduction and its implementation
# in classification data base MNIST

# Импортирование требуемых библиотек
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib
import pandas as pd

# Воспроизведение изучаемых изображений
tf.compat.v1.reset_default_graph()
np.random.seed(42)
tf.compat.v1.set_random_seed(42)

# Загрузка базы данных MNIST
data = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = data.load_data()

# Визуализация произвольной части исследуемой базы данных
n_samples = 5
plt.figure(figsize=(n_samples * 3, 3))
for index in range(n_samples):
plt.subplot(1, n_samples, index + 1)
sample_image = train_images[index].reshape(28, 28)
plt.imshow(sample_image, cmap=''binary'')
plt.axis(''off'')
plt.show()

# Визуализация меток, соответствующих своему изображению
print(train_labels[:n_samples])

# Представление набора данных(набора изображений) в виде тензора
# Вместо None в дальнейшем будет стоять число изображений
X = tf.compat.v1.placeholder(shape=[None, 28, 28, 1], dtype=tf.float32,
name=''X'')

# Primary Capsules
# Инициализация параметров для сверточных слоев
caps1_n_maps = 32 # количество частей, на которые делится полученный стек
caps1_n_caps = caps1_n_maps * 6 * 6 # 1152 первичные капсулы
caps1_n_dims = 8 # размерность первичных капсул

# Первый сверточный слой
# Количество фильтров - 256
# Размер фильтра - 9×9
# Шаг фильтра - 1
```

```
conv1 = tf.keras.layers.Conv2D(
filters=256, kernel_size=9, strides=(1, 1), padding='valid',
dilation_rate=(1, 1), activation=tf.nn.relu, use_bias=True,
kernel_initializer='glorot_uniform', bias_initializer='zeros',
kernel_regularizer=tf.keras.regularizers.l2(0.05),name='conv1')
out_conv1 = conv1(X)

# Второй сверточный слой
# Количество фильтров - 256
# Размер фильтра - 9×9×256
# Шаг фильтра - 2
conv2 = tf.keras.layers.Conv2D(
filters=256, kernel_size=9, strides=(2, 2), padding='valid',
dilation_rate=(1, 1), activation=tf.nn.relu, use_bias=True,
kernel_initializer='glorot_uniform', bias_initializer='zeros',
kernel_regularizer=tf.keras.regularizers.l2(0.05),name='conv2')
out_conv2 = conv2(out_conv1)

# Разделение полученного стека данных на 1152 вектора размерностью 8
caps1_raw = tf.reshape(out_conv2, [-1, caps1_n_caps, caps1_n_dims],
name=''caps1_raw'')

# Теперь каждый полученный вектор необходимо нормировать, так как его длина
# может превышать 1
# Для этого воспользуемся новой нелинейной функцией squash(s)
def squash(s, axis=-1, epsilon=1e-7, name=None):
with tf.name_scope(name, default_name=''squash''):
squared_norm = tf.math.reduce_sum(tf.square(s), axis=axis, keepdims=True)
safe_norm = tf.math.sqrt(squared_norm + epsilon)
squash_factor = squared_norm / (1. + squared_norm)
unit_vector = s / safe_norm
return squash_factor * unit_vector
caps1_output = squash(caps1_raw, name=''caps1_output'')

# Digit Capsules
# Инициализация параметров для слоя с цифровыми капсулами
caps2_n_maps = 10 # Количество капсул соответствует количеству классов
caps2_n_dims = 16 # Размерность цифровых капсул
init_sigma = 0.1 # Область отклонения случайных значений матриц W

# Инициализация обучаемых матриц преобразования для каждой первичной капсулы
# предсказаний
W_init = tf.random.normal(
shape=(1, caps1_n_caps, caps2_n_caps, caps2_n_dims, caps1_n_dims),
stddev=init_sigma, dtype=tf.float32, name=''W_init'')
W = tf.Variable(W_init, name=''W'')

# Количество наборов матриц W соответствует количеству обрабатываемых
# изображений
```

```
batch_size = tf.shape(X)[0]
# Создание копий наборов матриц W, количество которых соответствует
# количеству изображений
W_tiled = tf.tile(W, [batch_size, 1, 1, 1, 1], name=''W_tiled'')
# ВНИМАНИЕ! Индексирование размерностей идет справа налево!(индексирование
# как обычно начинается с 0
# создание размерности под индексом -1(размерность будет равна 1 и будет
# стоять на 1 месте справа в скобках)
caps1_output_expanded = tf.compat.v1.expand_dims(caps1_output, -1,
name=''caps1_output_expanded'')
# Создание размерности под индексом 2(размерность будет равна 1 и будет
# стоять на 2 месте справа в скобках)
caps1_output_tile = tf.compat.v1.expand_dims(caps1_output_expanded, 2,
name=''caps1_output_tile'')
# Создание копий набора первичных капсул, количество которых соответствует
# количеству классов
caps1_output_tiled = tf.compat.v1.expand_dims(caps1_output_tile,
[1,1,caps2_n_caps,1,1],name=''caps1_output_tiled'')
# Капсулы, полученные в результате скалярного перемножения матриц W и
# первичных векторов
caps2_predicted = tf.linalg.matmul(W_tiled, caps1_output_tiled,
name=''caps2_predicted'')

# Routing by Agreement
# Round 1
# Инициализируем веса предсказаний, присваивая им начальные значения равные 0
raw_weights = tf.zeros([batch_size, caps1_n_caps, caps2_n_caps, 1, 1],
dtype=np.float32, name=''raw_weights'')
# Применение функции softmax
routing_weights = tf.nn.softmax(raw_weights, axis=2, name=''routing_weights'')
# Перемножение весов и векторов предсказаний
weighted_predictions = tf.math.multiply(routing_weights, caps2_predicted,
name=''weighted_predictions'')
# Вычисляется сумма произведений векторов и весов
weighted_sum = tf.math.reduce_sum(weighted_predictions, axis=1, keepdims=True,
name=''weighted_sum'')
# Нормирование полученного вектора
caps2_output_round_1 = squash(weighted_sum, axis=-2,
name=''caps2_output_round_1'')

# Создание большего тензора
caps2_output_round_1_tiled = tf.tile(caps2_output_round_1, [1, caps1_n_caps, 1,
1, 1], name=''caps2_output_round_1_tiled'')
# Скалярное перемножение капсул текущего и предыдущего уровней
agreement = tf.matmul(caps2_predicted, caps2_output_round_1_tiled,
transpose_a=True,name=''agreement'')
# Обновление весов для следующего раунда
raw_weights_round_2 = tf.add(raw_weights, agreement, name=''raw_weights_round_2'')
```

```
# Round 2
routing_weights_round_2 = tf.nn.softmax(raw_weights_round_2, axis=2,
name=''routing_weights_round_2'')
weighted_predictions_round_2 = tf.math.multiply(routing_weights_round_2,
caps2_predicted,name=''weighted_predictions_round_2'')
weighted_sum_round_2 = tf.math.reduce_sum(weighted_predictions, axis=1,
keepdims=True,name=''weighted_sum_round_2'')
caps2_output_round_2 = squash(weighted_sum_round_2, axis=-2,
name=''caps2_output_round_2'')
caps2_output = caps2_output_round_2

# Очередная функция squash для нормирования векторов без изменения направления
def safe_norm(s, axis=-1, epsilon=1e-7,keep_dims=None,name=None):
with tf.name_scope(name, default_name=''safe_norm''):
squared_norm = tf.math.reduce_sum(tf.math.square(s), axis=axis, keepdims=True)
return tf.math.sqrt(squared_norm + epsilon)

y_proba = tf.math.argmax(y_proba,axis=-2,name=''y_proba'') # Нормализация всех
# векторов
y_proba_argmax = safe_norm(caps2_output,axis=2,name=''y_proba_argmax'') # Находим
# индекс вектора с наибольшим значением

# Избавляемся от последних 2-х измерений(отсчет идет справа налево)
y_pred = tf.compat.v1.squeeze(y_proba_argmax,axis=[1,2],name=''y_pred'')
y = tf.compat.v1.placeholder(shape=[None],dtype=tf.int64,name=''y'') # Тензор,
# содержащий в себе классы цифр 0..9

# Значения, требуемые для margin los
m_plus = 0.9 # m+
m_minus = 0.1 # m-
lambda_ = 0.5 # lambda

# Значение T_k для каждого экземпляра и класса
T = tf.one_hot(y, depth=caps2_n_caps,name=''T'')

# Вычисляем норму векторов предсказаний
caps2_output_norm = safe_norm(caps2_output, axis=-2, keep_dims=True,
name=''caps2_output_norm'')
# Вычисляем квадрат max(0,m+ - ||v_k||)
present_error_raw = tf.math.square(tf.math.maximum(0.,m_plus-caps2_output_norm),
name=''present_error_raw'')
# Перегруппируем результат
present_error = tf.reshape(present_error_raw,shape=(-1,10),
name=''present_error'')
# Вычисляем квадрат max(0,||v_k|| - m-)
absent_error_raw = tf.math.square(tf.math.maximum(0.,caps2_output_norm-m_minus),
name=''absent_error_raw'')
# Перегруппируем результат
absent_error = tf.reshape(present_error_raw,shape=(-1,10),
```

```
name=''absent_error'')
# Считаем потерю данных для каждой цифры
L = tf.math.add(T*present_error,lambda_(1.0-T)*absent_error,name='L'')
# Вычисляем среднее значение от суммы всех потерь данных для каждой цифры
margin_loss = tf.math.reduce_mean(tf.math.reduce_sum(L, axis=1),
name='margin_loss'')

# Реконструкция исходного изображения
# Тензор, в котором содержатся скрытые векторы, не требуемые на выходе
mask_with_labels = tf.compat.v1.placeholder_with_default(False, shape=(),
name='mask_with_labels'')

# Определяем объект для реконструкции, используя метки y - True, y_pred - False
reconstruction_targets = tf.compat.v1.cond(mask_with_labels,lambda: y,
lambda: y_pred, name='reconstruction_targets'')

# Маска для реконструкции(равна 1 для требуемого объекта, 0 - для иного)
reconstruction_mask = tf.one_hot(reconstruction_targets, depth=caps2_n_caps,
name='reconstruction_mask'')

# Изменение размерности для возможности перемножения
reconstruction_mask_reshaped = tf.reshape(reconstruction_mask,
[-1, 1, caps2_n_caps, 1, 1], name='reconstruction_mask_reshaped'')

# Перемножение масок с выходными капсулами
caps2_output_masked = tf.math.multiply(caps2_output,
reconstruction_mask_reshaped, name='caps2_output_masked'')

# Снова изменяем размерность тензора
decoder_input = tf.reshape(caps2_output_masked,[-1,caps2_n_caps*caps2_n_dims],
name='decoder_input'')

# Decoder
n_hidden1 = 512 # Количество нейронов в первом скрытом слое
n_hidden2 = 1024 # Количество нейронов во втором скрытом слое
n_output = 28*28 # Количество нейронов в выходном слое(изображение 28x28)

# Построение декодера
# Первый скрытый слой
hidden1 = tf.keras.layers.Dense(n_hidden1,activation=tf.nn.relu,
kernel_initializer='glorot_uniform',bias_initializer='zeros')(decoder_input)

# Второй скрытый слой
hidden2 = tf.keras.layers.Dense(n_hidden2,activation=tf.nn.relu,
kernel_initializer='glorot_uniform',bias_initializer='zeros')(hidden1)

# Выходной слой
decoder_output = tf.keras.layers.Dense(n_output,activation=tf.nn.sigmoid,
kernel_initializer='glorot_uniform',bias_initializer='zeros')(hidden2)
```

```
# Вычисление потерь данных при реконструкции
# Изменение размерности тензора X
X_flat = tf.reshape(X,[-1, n_output], name=‘‘X_flat’’)
# Вычисление квадрата разности между входным и воссоздаваемым изображениями
squared_difference = tf.math.square(X_flat-decoder_output,
name=‘‘squared_difference’’)
# Вычисление среднего значения результата
reconstruction_loss = tf.math.reduce_mean(squared_difference,
name=‘‘reconstruction_loss’’)

# Вычисление общей потери данных как сумма 2-х значений
alpha = 0.0005 # Коэффициент масштабирования
loss = tf.math.add(margin_loss,alpha*reconstruction_loss,c=‘‘loss’’)

# Вычисляем количество экземпляров, которые были правильно классифицированы
correct = tf.math.equal(y,y_pred,name=‘‘correct’’) # Определяем активную метку
# Вычисляем среднее значение всех требуемых меток
accuracy = tf.math.reduce_mean(tf.cast(correct, tf.float32),
name=‘‘accuracy’’)

# Алгоритмы для тренировки нейросети
optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate=0.001,beta1=0.09,
beta2=0.999, epsilon=1e-8, use_locking=False,name=’optimizer’)
training_op = optimizer.minimize(loss, name=‘‘training_op’’)

init = tf.compat.v1.global_variables_initializer()
saver = tf.compat.v1.train.Saver()

# Стандартная тренировка нейросети
# Загрузка базы данных рукописных цифр
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(‘‘/tmp/data/’’)

n_epochs = 10 # Количество эпох
batch_size = n_samples # Количество изображений
restore_checkpoint = True
n_iterations_per_epoch = 5000 // batch_size # Кол-во итераций за эпоху
n_iterations_validation = 1000 // batch_size # Кол-во подтверждающих итераций
best_loss_val = np.infty # Бесконечность
checkpoint_path=‘‘./CNN’’ # Проверка наличия MNIST в одной папке с НС

# Проверка наличия требуемых экземпляров по адресу checkpoint_path
with tf.compat.v1.Session() as sess:
if restore_checkpoint and tf.compat.v1.train.checkpoint_exists(checkpoint_path):
saver.restore(sess,checkpoint_path)
else:
init.run()
```

```
# Тренировка нейронной сети, используя экземпляры MNIST с метками
for epoch in range(n_epochs):
for iteration in range(1, n_iterations_per_epoch+1):
X_batch, y_batch = mnist.train.next_batch(batch_size)
_, loss_train = sess.run([training_op,loss],
feed_dict={X: X_batch.reshape([-1,28,28,1]), y: y_batch, mask_with_labels:True})
print(‘‘\rIteration: {}/{} ({:.1f}%) Loss: {:.5f} ’’.format(iteration,
n_iterations_per_epoch,iteration*100/n_iterations_per_epoch,loss_train),end=‘‘ ’’)

loss_vals = [] # Потеря данных
acc_vals = [] # Точность детектирования
# Закрепление тренировки на данных без меток
for iteration in range(1, n_iterations_validation+1):
X_batch, y_batch = mnist.validation.next_batch(batch_size)
loss_val, acc_val = sess.run([loss,accuracy],
feed_dict={X: X_batch.reshape([-1,28,28,1]), y: y_batch})
loss_vals.append(loss_val)
acc_vals.append(acc_val)
print(‘‘\rEvaluating the model: {}/{} ({:.1f}%) ’’.format(iteration,
n_iterations_validation,iteration*100/n_iterations_validation),end=‘‘ ’’*10)
loss_val = np.mean(loss_vals)
acc_val = np.mean(acc_vals)
print(‘‘\rEpoch: {} Val accuracy: {:.4f}% Loss: {:.6f}{} ’’.format(epoch+1,
acc_val*100,loss_val)‘‘(improved) ’’ if loss_val < best_loss_val else ‘‘’’))

# Обновление данных
if loss_val < best_loss_val:
save_path = saver.save(sess, checkpoint_path)
best_loss_val = loss_val

n_iterations_test = 1000 // batch_size # Количество тестируемых значений

# Увеличение точности распознавания
with tf.compat.v1.Session() as sess:
saver.restore(sess,checkpoint_path)
loss_tests = []
acc_tests = []
for iteration in range(1, n_iterations_test+1):
X_batch, y_batch = mnist.test.next_batch(batch_size)
loss_test, acc_test = sess.run([loss,accuracy],
feed_dict={X: X_batch.reshape([-1,28,28,1]), y: y_batch})
loss_tests.append(loss_test)
acc_tests.append(acc_test)
print(‘‘\rEvaluating the model: {}/{} ({:.1f}%) ’’.format(iteration,
n_iterations_validation,iteration*100/n_iterations_validation),end=‘‘ ’’*10)
loss_test = np.mean(loss_tests)
acc_test = np.mean(acc_tests)
print(‘‘\rFinal test accuracy: {:.4f}% Loss: {:.6f} ’’.format(acc_test*100,
loss_test))
```

```
# Проверка распознавания и реконструкции различных экземпляров
sample_images = mnist.test.images[:n_samples].reshape([-1,28,28,1])

with tf.compat.v1.Session() as sess:
saver.restore(sess,checkpoint_path)
caps2_output_value,decoder_output_value,y_pred_value = sess.run([caps2_output,
decoder_output,y_pred],feed_dict=X: sample_images,y: np.array([],dtype=np.int64))

sample_images = sample_images.reshape(-1,28,28)
reconstructions = decoder_output_value.reshape([-1,28,28])

# Входные изображения с метками
plt.figure(figsize=(n_samples * 2, 3))
for index in range(n_samples):
plt.subplot(1, n_samples, index + 1)
plt.imshow(sample_images[index], cmap="binary")
plt.title("Label:" + str(mnist.test.labels[index]))
plt.axis("off")
plt.show()

# Обработанные программой выходные изображения с предсказаниями
plt.figure(figsize=(n_samples * 2, 3))
for index in range(n_samples):
plt.subplot(1, n_samples, index + 1)
plt.title("Predicted:" + str(y_pred_value[index]))
plt.imshow(reconstructions[index], cmap="binary")
plt.axis("off")
plt.show()
```

6 Результаты работы программы

В процессе обучения нейросети мы получаем результаты изменения точности детектирования и потери данных за каждую эпоху.

```
Epoch: 1   Val accuracy: 98.6000%   Loss: 0.016452 (improved)
Epoch: 2   Val accuracy: 99.0000%   Loss: 0.013760 (improved)
Epoch: 3   Val accuracy: 99.2000%   Loss: 0.010516 (improved)
Epoch: 4   Val accuracy: 99.6000%   Loss: 0.009154 (improved)
Epoch: 5   Val accuracy: 98.0000%   Loss: 0.018975
Epoch: 6   Val accuracy: 98.6000%   Loss: 0.012574
Epoch: 7   Val accuracy: 98.3000%   Loss: 0.019033
Epoch: 8   Val accuracy: 98.8000%   Loss: 0.012060
Epoch: 9   Val accuracy: 98.6000%   Loss: 0.014504
Epoch: 10  Val accuracy: 99.1000%   Loss: 0.012454
```

Итак, по завершении тренировки мы получаем конечную точность детектирования `99.1%`, что достаточно неплохо для начала! Однако это еще не все. Впереди нам предстоит оценить результат пройденной тренировки, на подобранных тестовых экзэмплярах, но уже без меток. В конце тестирования мы получаем финальные значения точности детектирования и потери данных.

```
Final test accuracy: 99.2000% Loss: 0.012688
```

Тем самым мы даже улучшили результаты обработки базы данных, что здорово!

И последним нашим тестированием будет предсказание и реконструкция рукописных цифр. Для начала мы выбираем и фиксируем несколько(в нашем случае `n_samples = 5`) изображений. Затем снова начинаем сессию, восстанавливаем обученную модель, оцениваем `caps2_output` для получения выходных векторов нашей каспсульной нейросети, `decoder_output` для получения реконструкции и `y_pred` для предсказаний искомых классов данных. Результаты обработки представлены на рис. 31.

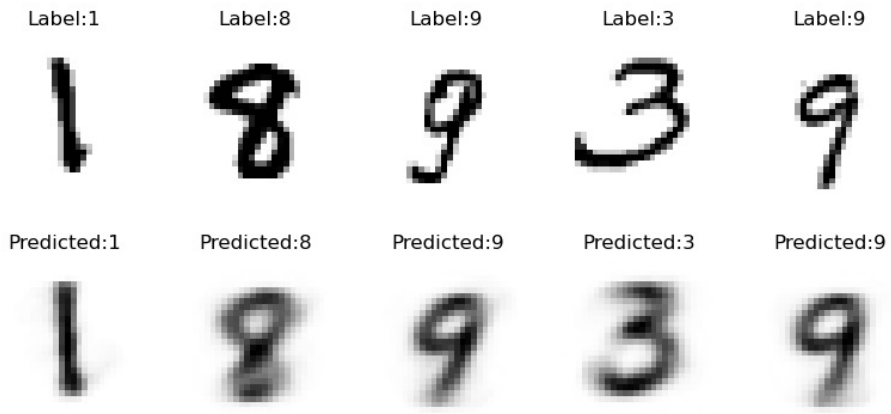

Рис. 31. Детектирования, реконструкция и классификация рукописных цифр

Свехру представлены входные данные в виде ч/б изображений. Несмотря на то, что у каждого изображения указана соответствующая метка(Label – отвечает за классификацию объекта к такому то классу), в тестовых экземплярах при обработке она отсутствует.

Ниже представлены реконструкции исходных изображений(каждое из них соответствует исходному изображению, находящемуся над реконструкцией соответственно), а вместо меток представлены предсказания(Predicted) класса объекта. Можно заметить, что все предсказания сделаны верно и реконструкция выглядит достаточно хорошо.

7 Заключение и дальнейшие исследования

Как показали результаты обработки изображений, капсульные нейронные сети очень хорошо справляются с поставленной задачей и превосходят по точности остальные классы нейросетей. Однако далеко еще не во всех промышленных областях они доказали свое превосходство. Сейчас капсульные нейросети используются только в научных исследованиях, поэтому лидером пока остаются сверточные нейронные сети.

В дальнейших исследованиях будет рассматриваться применение капсульных нейронных сетей к гиперспектральным изображениям для классификации местности. Для решения данной задачи архитектура классической **CapsNet** будет изменена, так как будет осуществляться обработка растровых изображений.

Также я хотел бы выразить свою благодарность компаниям Google за предоставление открытой библиотеки Tensorflow и JetBrains за предоставление бесплатной версии PyCharm Community.

Список литературы

[1] S. Sabour, N. Frosst, and G. E. Hinton, *Dynamic routing between capsules*, in Advances in neural information processing systems, 2017, pp. 3856–3866.

[2] G. E. Hinton, S. Sabour and N. Frosst, *Matrix capsules with em routing*, 2018, pp. 1–15.

[3] A. Punjabi, J. Schmid, and A. K. Katsaggelos, *Examining the Benefits of Capsule Neural Networks*, 2020, pp. 1–13.

[4] P. J. Grother, *NIST Special Database 19 Handprinted Forms and Characters Database*, 1995, pp. 1–28.

[5] K. Sood, *Capsule Networks: An Alternative Approach to Image Classification Using Convolutional Neural Networks*, 2020, pp. 1–11.

[6] T. Zeng, H. K.-H. So, and E. Y. Lam, *RedCap: residual encoder-decoder capsule network for holographic image reconstruction*, 2020, pp. 4876–4887.

[7] J.-T. Hsu, C.-H. Kuo, (Member, Ieee), and D.-W. Chen, *Image Super–Resolution Using Capsule Neural Networks*, 2020, pp. 9751–9759.

[8] О. Жерон, *Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow: концепции, инструменты и техники для создания интеллектуальных систем.*:Пер. с англ. — СпБ.:ООО"Альфа-книга 2018. — 688 с.: ил. — Парал. тит. англ.

[9] Капсульные сети от Хинтона, 2017. URL: `https://habr.com/ru/company/recognitor/blog/343726/`. (Дата обращения: 13.02.2020)

[10] Введение в архитектуры нейронных сетей, 2017. URL: `https://habr.com/ru/company/oleg-bunin/blog/340184/`. (Дата обращения: 15.02.2020)

[11] Введение в машинное обучение с tensorflow, 2017. URL: https://habr.com/ru/post/326650/. (Дата обращения: 15.03.2020)

[12] Применение нейросетей в распознавании изображений, 2009. URL: https://habr.com/ru/post/74326/. (Дата обращения: 14.02.2020)

[13] Сверточная нейронная сеть, часть 1: структура, топология, функции активации и обучающее множество, 2018. URL: https://habr.com/ru/post/348000/. (Дата обращения: 20.02.2020)

[14] Сверточная нейронная сеть, часть 2: обучение алгоритмом обратного распространения ошибки, 2018. URL: https://habr.com/ru/post/348028/. (Дата обращения: 21.02.2020)

[15] Глубокое обучение для новичков: распознаем рукописные цифры, 2016. URL: https://habr.com/ru/company/wunderfund/blog/314242/. (Дата обращения: 15.02.2020)

[16] О реализации библиотеки для глубокого обучения на Python, 2020. URL: https://habr.com/ru/company/ruvds/blog/486686/. (Дата обращения: 10.03.2020)

[17] Нейронные сети для начинающих. Часть 1, 2016. URL: https://habr.com/ru/post/312450/. (Дата обращения: 07.02.2020)

[18] Нейронные сети для начинающих. Часть 2, 2017. URL: https://habr.com/ru/post/313216/. (Дата обращения: 08.02.2020)

[19] Математика для искусственных нейронных сетей для новичков, часть 1 — линейная регрессия, 2016. URL: https://habr.com/ru/post/307004/. (Дата обращения: 09.02.2020)

[20] Математика для искусственных нейронных сетей для новичков, часть 2 — градиентный спуск, 2016. URL: https://habr.com/ru/post/307312/. (Дата обращения: 10.02.2020)

[21] Математика для искусственных нейронных сетей для новичков, часть 3 — градиентный спуск продолжение, 2016. URL: https://habr.com/ru/post/308604/. (Дата обращения: 11.02.2020)

[22] Нейросети и глубокое обучение, глава 1: использование нейросетей для распознавания рукописных цифр, 2019. URL: https://habr.com/ru/post/456738/. (Дата обращения: 18.02.2020)

[23] Что такое свёрточная нейронная сеть, 2016. URL: https://habr.com/ru/post/309508/. (Дата обращения: 19.02.2020)

[24] Алгоритм обучения многослойной нейронной сети методом обратного распространения ошибки (Backpropagation), 2013. URL: https://habr.com/ru/post/198268/. (Дата обращения: 20.02.2020)

[25] Капсульные нейронные сети, 2018. URL: https://habr.com/ru/post/417223/. (Дата обращения: 21.02.2020)

[26] Капсульная нейронная сеть или CapsNet: введение [Электронный ресурс] URL: https://neurohive.io/ru/osnovy-data-science/kapsulnaja-nejronnaja-set-capsnet/. (Дата обращения: 22.02.2020)

[27] TensorFlow MNIST example [Электронный ресурс] URL: https://www.singularis-lab.com/docs/materials/03_02_TensorFlow_Example.pdf. (Дата обращения: 15.02.2020)

[28] Алгоритмы компьютерного зрения на основе сверточных нейронных сетей [Электронный ресурс] URL: https://nnov.hse.ru/data/2018/09/29/1156640581/lecture2_CNN.pdf. (Дата обращения: 15.02.2020)

[29] Tensorflow [Электронный ресурс], URL: https://www.tensorflow.org/. (Дата обращения: 01.04.2020).

[30] How to implement CapsNets using TensorFlow [Электронный ресурс], URL: https://www.youtube.com/watch?v=2Kawrd5szHE. (Дата обращения: 05.03.2020).

[31] Capsule Networks (CapsNets) — Tutorial [Электронный ресурс], URL: https://www.youtube.com/watch?v=pPN8d0E3900&t=1160s. (Дата обращения: 01.03.2020).

[32] TensorFlow 2.0 Changes [Электронный ресурс], URL: https://www.youtube.com/watch?v=WTNH0tcscqo&t=3s. (Дата обращения: 03.04.2020).

[33] TensorFlow 2.0 Complete Course — Python Neural Networks for Beginners Tutorial [Электронный ресурс], URL: https://www.youtube.com/watch?v=tPYj3fFJGjk&t=8046s. (Дата обращения: 05.04.2020).