3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе мы рассмотрим основные принципы работы приложения. Для этого проведем анализ всех классов, которые входят в состав кода программы, и рассмотрим назначение всех методов, свойств и переменных класса.

3.1 Модуль библиотеки нейронных сетей

Центральная часть программного комплекса представляет собой набор компонентов, с помощью которых можно создавать различные типы нейронных сетей и проводить их обучение. Пользователю приложения предоставляется гибкое API для задания конфигурации нейронной сети и ее обучения. В состав библиотеки нейронных сетей входят:

- реализации различных типов слоев нейронной сети
- компоненты для представления входных и выходных данных сети
- модуль обучения сети
- компоненты клиентского интерфейса для взаимодействия с библиотекой

Рассмотрим каждый из описанных выше модулей

Структура нейронной сети представляет собой несколько слоев нейронов, соединенных последовательно. Каждый слой состоит из определенного количества нейронов. Но на этапе проектирования было принято решение не реализовывать отдельные нейроны как простейшие элементы сети. Это значит, что базовым компонентом для построения структуры сети является слой нейронов. Такой подход позволяет ускорить разработку, а также повысить быстродействие приложения и уменьшить используемую память, так как сокращается количество создаваемых в ходе работы объектов.

3.1.1 Во время работы сети, данные, которые попадают на вход поочередно проходят через каждый слой сети и в итоге преобразуются в выходной набор данных который является ответом сети на входные параметры. При прохождении данных через сеть размерность данных меняется. Например, на вход сети может подаваться цветное RGB изображение (размерность массива данных 25x25x3), а на выходе мы получаем вектор предсказаний размерностью 10. Поэтому модель, которая используется для хранения данных должна иметь возможность менять размерность данных по ходу работы. Было принято решение использовать для хранения данных внутри модели простой одномерный массив, но предоставить клиенту интерфейс, с помощью которого с объектом можно вести работу как с многомерным массивом.

Базовой моделью для представления массива данных в нейронной сети является класс `DataSet`. Данный класс может представлять одномерный,

двумерный либо трехмерный массив и предоставляет набор операций над данными которые необходимы при работе с нейронными сетями. Ниже представлены поля и методы класса `DataSet`:

```
public class DataSet {

private final List<Double> data = new ArrayList<>();
private Dimension dimension;

public List<Double> getData() {}

public DoubleStream getStreamData() {}
public Double[] getWrapperArrayData() {}

public int getSize() {}

public Dimension getDimension() {}

public DataSet update(final List<Double> data, final
Dimension dimension) {}

public DataSet update(final DataSet dataSet) {}

public DataSet update(final DoubleUnaryOperator operator) {}

public DataSet rotate() {}

public DataSet merge(final DataSet other, final
DoubleBinaryOperator operator) {

public double[][] get2DArrayData() {}

public double[][][] get3DArrayData() {}

public double[] getArrayData() {}

public double get(int idx) {}

public double get(int widthIdx, int heightIdx, int
depthIdx){}

public void set(int widthIdx, int heightIdx, int depthIdx,
double value) {}
```

Класс содержит два поля:

- `data` – массив данных типа double;
- `dimension` – объект класса, который хранит информацию о размерности данных (width, height, depth).

Методы класса позволяют проводить различные операции с данными. Перечислим некоторые из них:

- `List<Double> getData()` – возвращает одномерный массив данных, которые хранятся в модели;
- `void rotate()` – производит поворот массива данных на 180 градусов (rot180). Данная операция используется в сверточных нейронных сетях во время обратного прохода по сети. Во время выполнения операции поворота, элемент с индексами (*x, y*) меняется местами с элементом с индексами (*xSize – x - 1*, *ySize – x - 1*), где *xSize, ySize* – размеры массива. Ниже представлен алгоритм работы метода `rotate()`:

```
public DataSet rotate() {
    final int width = dimension.getWidth();
    final int height = dimension.getHeight();
    final int depth = dimension.getDepth();
    final DataSet result = new DataSet(this);

    for (int k = 0; k < depth; k++) {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                result.set(j, i, k, this.get(width - 1 - j,
              height - 1 - i, k));
            }
        }
    }
    return result;
}
```

- `double get (int widthIdx, int heightIdx, int depthIdx)` – позволяет получить единичный элемент массива по заданным координатам. Во время вызова метода происходит проверка на корректность введенных координат чтобы исключить возможность обращения к несуществующим данным. В случае неверных данных происходит создание `IllegalArgumentException`;
- `DataSet update (final DataSet dataSet)` – обновляет данные в текущем экземпляре класса используя информацию из другого экземпляра, переданного в качестве аргумента. Во время вызова метода не происходит проверок размерностей данных, фактически существующие данные просто заменяются новыми и их размер также может измениться.
- `DataSet update (final DoubleUnaryOperator operator)` – данный метод также используется для обновления данных, но в отличие от предыдущего метода для обновления используется не другой экземпляр класса `DataSet`, а лямбда-выражение, которое не принимает параметров и должно возвращает результат типа `Double`. Данная функция вызывается для каждого элемента массива данных.
- `DataSet merge (final DataSet other, final DoubleBinaryOperator operator)` – в качестве аргумента принимает другой экземпляр класс DataSet и применяет функцию `operator` попарно к

каждым двум элементам массивов данных. Для того чтобы метод завершился успешно необходимо чтобы размерности данных объекта, для которого вызывается метод и объекта, переданного в качестве аргумента совпадали. Поэтому перед выполнением обновления данных происходит сравнение полей dimension обоих классов. Если выясняется, что размерности не совпадают, то создается `IllegalArgumentException`;

- `double[] getArrayData()` - данный метод возвращает информацию, хранящуюся в экземпляре класса в виде одномерного массива. Так как целью разработки является создание гибкой библиотеки, которая может быть использована в существующие приложения, а класс DataSet является классом с помощью которого происходит непосредственное взаимодействие кода клиента с функционалом библиотеки необходимо реализовать вспомогательные методы, которые позволят клиентам получать данные в необходимом формате;
- `double[][] get2DArrayData()` – Возвращает информацию, хранящуюся в экземпляре класса в виде двумерного массива. Выполнение метода возможно лишь в том случае, если размерность height данного объекта равна единице. Некоторые библиотеки, используемые для математических вычислений, принимают двумерные массивы в качестве аргументов. Поэтому данный метод может быть полезен при использовании библиотеки с уже существующим приложением либо при написании собственных реализаций классов, выполняющих математические вычисления;
- `double[][][] get3DArrayData()` – Возвращает информацию, хранящуюся в экземпляре класса в виде трехмерного массива.
- `DoubleStream getStreamData()` – возвращает информацию, хранящуюся в экземпляре класса в виде объекта `java.util.stream.Stream`.

Так как класс DataSet разрабатывается для использования в клиентском коде, важной задачей является создание набора конструкторов, с помощью которых клиент сможет создать экземпляр класса `DataSet` используя имеющейся у него формат данных.

Рассмотрим конструкторы класса `DataSet`:

- `DataSet (final Collection<Double> data, final Dimension dimension)` – конструктор принимает в качестве параметра коллекцию данных, а также объект Dimension, который представляет размерность массива данных. Во время работы конструктора производится проверка того, что размерность данных соответствует количеству элементов в коллекции, иначе создается `IllegalArgumentException`;
- `DataSet (final double[] data, final Dimension dimension)` – в этом варианте конструктора данные передаются в виде массива чисел типа double. Во время работы конструктора производится проверка того, что размерность данных соответствует количеству элементов в массиве, иначе создается `IllegalArgumentException`;

- `DataSet (final double[][] data, final Dimension dimension)`- В отличие от предыдущего конструктора, элементы для создания массива данных передаются в виде двумерного массива. Во время работы конструктора производится проверка того, что размерность данных соответствует количеству элементов в массиве, иначе создается `IllegalArgumentException`;
- `DataSet(final Dimension dimension, final DoubleSupplier supplier)` – в данном конструкторе вместо информации для создания массива передается лямбда выражение, которое будет возвращать элементы массива данных. Количество элементов массива определяется аргументом `dimension`.

Для представления размерности массива данных в библиотеке присутствует класс Dimension. Как было сказано выше, класс DataSet представляет собой трехмерный массив и его размерность может меняться в процессе работы приложения. В классе Dimension присутствует 3 целочисленных поля – `height`, `width` и `depth`, которые представляют высоту ширину и глубину массива данных соответственно (см. рисунок 3.1). Методы класса `Dimension` позволяют получить доступ к этим полям, а также получить дополнительную информацию о размерности данных.

- `int getHeight()` – возвращает высоту массива данных;
- `int getWidth()` – возвращает ширину массива данных;
- `int getDepth()` – возвращает глубину массива данных;
- `int getSize()` – возвращает общую размерность массива данных (`height · width · depth`);
- `boolean isSingleDimensional()` – позволяет узнать является ли массив данных одноразмерным, т.е. все его размерности кроме одной равны единице.

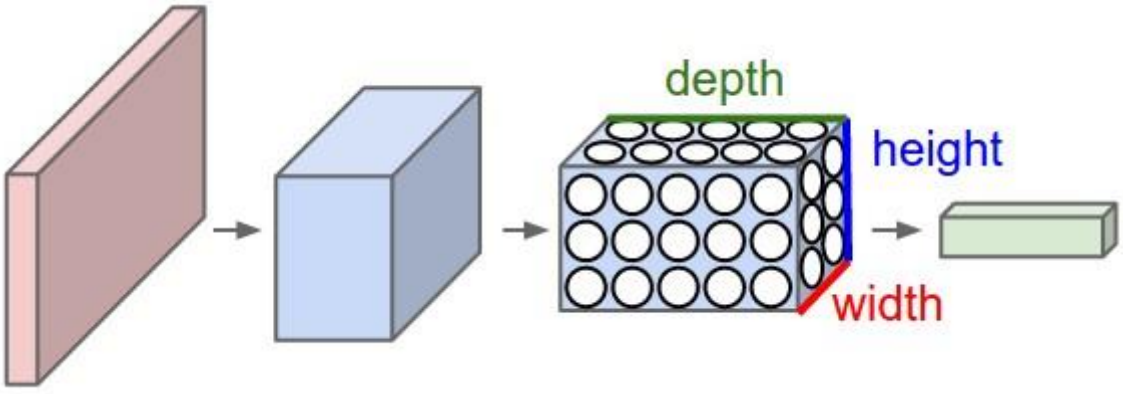

Рисунок 3.1 – Размерности массива данных во время работы нейронной сети.

Класс имеет несколько конструкторов:

- `Dimension (int width);`
- `Dimension (int width, int height);`

- `Dimension (int width, int height, int depth);`

В том случае, если значения полей не предоставляются конструктору (в первом и во втором случаях), для них устанавливается единичное значение. Также во всех конструкторах производится проверка всех переданных аргументов и в случае если какой-либо из них оказался неположительным числом создается стандартное непроверяемое исключение языка Java `IllegalArgumentException`.

Также в классе присутствует статическое константное поле `EMPTY` которое представляет размерность со всеми параметрами, равными единице. Это поле используется

Главная цель данного класса – предоставить информацию о размерности данных в классе `DataSet`, поэтому в большинстве случаев экземпляры данного класса передаются как аргументы конструктора класса `DataSet` и сохраняются в поле `dimension`.

Класс `LayerDimensions` представляет собой контейнер для хранения двух экземпляров класса Dimension. Каждый слой нейронной сети имеет входные и выходные размерности, которые отображают то, как изменяются данные при прохождении через этот слой. В классе присутствует два поля типа `Dimension`: `inputDimension` и `outputDimension`, а также методы для доступа к полям. Данные из экземпляров этих классов используются нейронной сетью для определения размерностей входов и выходов слоев.

3.1.2 Как было сказано выше, ключевым элементом в нейронной сети является слой нейронов. В классах слоев реализована основная логика, связанная с работой нейронной сети. После того как клиент предоставил данные на вход сети, они последовательно проходят обработку всеми слоями, входящими в состав сети, и итоговый результат возвращается клиенту. Данные попадают на вход слоя и после обработки переходят на следующий слой либо на выход сети (если этот слой оказался последним) В общем случае слой обладает такими характеристиками, как:

- размерность входных и выходных данных;
- набор весов (параметров) слоя, которые изменяются во время обучения сети и с помощью которых происходит преобразование данных (в сверточных слоях весами являются ядра свертки, а в полносвязных – веса отдельных нейронов);
- функция активации, которая применяется к данным на выходе из слоя.

Базовая структура слоя описана в классе `NetLayer`. Все существующие типы слоев так или иначе являются наследниками этого класса. В классах наследниках должны быть реализованы методы `forward()` и `backward()`, которые определяют как именно будут изменяться данные при прямом и обратном проходе через сеть. Ниже представлены поля и методы абстрактного класса `NetLayer`:

```
public abstract class NetLayer {

    final LayerDimensions layerDimensions;
    final DataSet deltas;
    final DataSet weights;
    final DataSet prevOutputs;
    final DataSet selfOutputs;
    final ActivationFunction activationFunction;

    abstract void forward(final DataSet dataSet);

    abstract void backward(final DataSet deltas, final
DataSet childrenWeights);

    abstract void lastLayerBackward(final DataSet deltas,
final DataSet y, final DataSet outputs);
}
```

Рассмотрим поля и методы этого класса:

- `LayerDimensions layerDimensions` – входные и выходные размерности для слоя. Определяют то, в каком формате данные должны подаваться на вход и выход слоя;
- `DataSet deltas` – массив ошибок в нейронах слоя. Используется во время работы алгоритма градиентного спуска;
- `DataSet weights` – параметры слоя, которые используются для преобразования данных;
- `DataSet prevOutputs` – сохраненные значения выходов предыдущего слоя. Используются во время работы алгоритма градиентного спуска;
- `DataSet selfOutputs` – Собственные выходы слоя (до применения фунции активации). Используются во время работы алгоритма градиентного спуска;
- `ActivationFunction activationFunction` – функция активации слоя. Во время работы применяется к каждому выходу слоя.
- `MathOperations mathOperations` – интерфейс, который предоставляет реализацию методов для выполнения вычислительных операций (например, свертки)

Методы класса:

- `abstract void forward(final DataSet dataSet)` – данный метод объявлен абстрактным и должен быть переопределен в классе-наследнике. Метод forward вызывается по очереди для каждого слоя сети от первого к последнему. Именно в этом методе происходит преобразование данных, специфичное для данного типа слоя. В качестве аргумента методу передается ссылка на объект `DataSet`, который является массивом данных обрабатываемым в данный момент нейронной сетью. Каждый из различных

типов слоев нейронной сети обладает собственной логикой преобразования данных;

- `abstract void backward(final DataSet deltas, final DataSet childrenWeights)` - метод обратного прохода вызывается во время обучения сети поочередно для каждого слоя сети, начиная от последнего. Во время работы этого метода происходит вычисление ошибок нейронов для данного слоя, а также изменение параметров слоя. В качестве аргументов в метод передается массив ошибок предыдущего слоя, а также параметры предыдущего слоя.

- `abstract void lastLayerBackward(final DataSet y, final DataSet deltas, final DataSet outputs)` – данный метод вызывается при обратном проходе в том случае, если текущий слой является последним слоем сети.

На данный момент в системе присутствует реализация следующих типов слоев нейронной сети:

– полносвязный слой (`FullyConnectedNetLayer`);

– сверточный слой (`ConvolutionNetLayer`);

– softmax-слой (`SoftmaxNetLayer`);

– слой пулинга (`PoolingNetLayer`).

– слой дропаута (`DropoutNetLayer`)

Все вышеперечисленные классы являются наследниками класса `NetLayer` и отличаются лишь реализацией методов `forward()` и `backward()`. Рассмотрим подробнее принцип работы каждого типа слоя.

3.1.3 Сверточный слой используется для построения сверточных нейронных сетей - архитектуры искусственных нейронных сетей, нацеленная на эффективное распознавание изображений, входящей в состав технологий глубинного обучения. Главным отличием сверточного слоя от полносвязного является то, что не все нейроны предыдущего слоя связаны с нейронами сверточного слоя. Параметры слоя представляют собой ядро свертки, с помощью которого (см. рисунок 3.2) из входных данных мы получаем карту признаков, которую определяет данный слой.

$$(I * K)_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} K_{i,j} \times I_{x+i-1,y+i-1} \qquad (3.1)$$

где h, w – высота и ширина ядра свертки соответственно.

Имея двумерное изображение I и небольшую матрицу размерности K (так называемое ядро свертки), построенную таким образом, что графически кодирует какой-либо признак, мы вычисляем свернутое изображение I * K, накладывая ядро на изображение всеми возможными способами и записывая сумму произведений элементов исходного изображения и ядра (3.1).

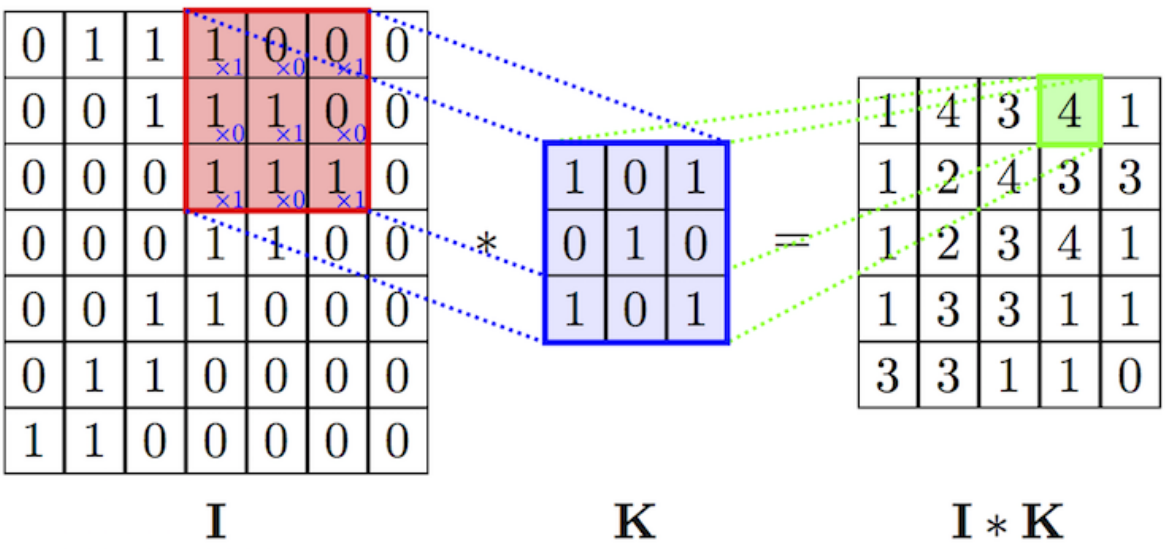

Рисунок 3.2 – Операция двумерной свертки

Кроме размеров ядра свертки, при выполнении операции могут быть использованы такие параметры как stride и padding, где stride – шаг, на который перемещается ядро свертки для вычисления следующего произведения. По умолчанию этот параметр равен 1. Выбор большего значения может ускорить выполнение операции свертки, но эффективность сверточного слоя уменьшится из-за того, что часть пикселей будет пропущена.

Параметр padding определяет начальное смещение ядра свертки относительно исходного изображения. При этом элементы с отрицательными индексами обычно заменяются нулями (см. рисунок 3.3).

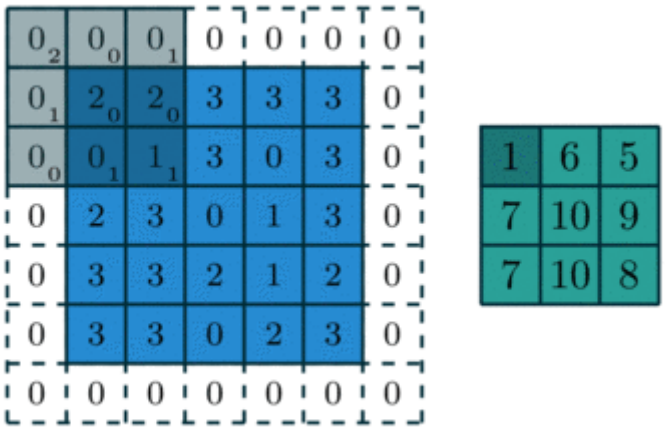

Рисунок 3.3 – Пример двумерной свертки с параметром padding равном единице.

При использовании обычных полносвязных слоев размерность параметров слоя может принимать большие значения. Например, в популярном обучающем наборе CIFAR-10 каждое цветное изображение

обладает размерностью 32 x 32 x 3, и если мы будем считать каждый канал каждого пикселя независимым входным параметром для многослойного персептрона, каждый нейрон в первом скрытом слое добавляет к модели 3072 новых параметров. И с ростом размера изображений ситуация быстро выходит из-под контроля, причем происходит это намного раньше, чем изображения достигают того размера, с которыми обычно работают пользователи реальных приложений.

Поэтому одно из эффективных решений — понижать разрешение изображений до той степени, когда становится применим многослойный персептрон. Тем не менее, когда мы просто понижаем разрешение, мы рискуем потерять большое количество информации, и было бы хорошо, если бы можно было осуществлять полезную первичную обработку информации еще до применения понижения качества, не вызывая при этом взрывного роста количества параметров модели.

Работа сверточных слоев обычно интерпретируется как переход от конкретных особенностей входного сигнала (изображения) к более абстрактным деталям, и далее к ещё более абстрактным деталям вплоть до выделения понятий высокого уровня. При этом в процессе обучения сверточный слой самонастраивается и вырабатывает сам необходимую иерархию абстрактных признаков (карт признаков), фильтруя маловажные детали и выделяя существенное (см. рисунок 3.4).

Оказывается, существует весьма эффективный способ решения этой задачи, который обращает в нашу пользу саму структуру изображения: предполагается, что пиксели, находящиеся близко друг к другу, теснее “взаимодействуют” при формировании интересующего нас признака, чем пиксели, расположенные в противоположных углах. Кроме того, если в процессе классификации изображения небольшая черта считается очень важной, не будет иметь значения, на каком участке изображения эта черта обнаружена.

Ниже представлена реализация метода `forward()` для сверточного слоя:

```
@Override
void forward(DataSet dataSet) {
    this.prevOutputs.update(dataSet);

Validate.isTrue(dataSet.getDimension()
.equals(getLayerDimensions().getInputDimension()),
            "DataSet must match input dimension");

dataSet.update(mathOperations.convolve(weights,dataSet));
    this.selfOutputs.update(dataSet);
dataSet.update(this.activationFunction.getForwardOperator()}
```

Стоит обратить внимание на то, что в начале метода происходит сохранение массива данных в поле `prevOutputs`. Фактически, аргумент,

который передается в метод `forward()` является выходом предыдущего слоя нейронов или же входным сигналом всей сети (если данный слой является первым слоем сети). После выполнения операции свертки происходит сохранение результата в поле `selfOutputs` слоя. Эти данные необходимы для работы алгоритма обратного распространения ошибки с помощью которого происходит обучение сети.

Метод `backward()` сверточного слоя производит вычисление ошибок нейронов данного слоя, также с помощью операции двумерной свертки:

```
@Override
void backward(DataSet deltas, DataSet childrenWeights) {
    final DataSet result =
 mathOperations.convolve(
 childrenWeights.rotate(), deltas, 0);

    final DataSet activationGrad = new DataSet(selfOutputs)

.update(this.activationFunction.getBackwardOperator());

    deltas.update(
 result.merge(activationGrad, (a, b) -> a * b));

     this.deltas.update(deltas);

    getWeights().merge(mathOperations.convolve(
        this.deltas, this.prevOutputs.rotate(), 0),
        (a, b) -> a + b);
}
```

Во время выполнения обратного прохода для вычисления ошибок нейронов сверточного слоя производится операция полной (full) двумерной свертки параметров следующего слоя с ошибками следующего слоя. При этом перед выполнением свертки массив параметров следующего слоя разворачивается на 180°. После этого результат умножается на собственный выход слоя, сохраненный во время выполнения прямого прохода по сети, к которому была применена производная функции активации. Получившийся результат является массивом ошибок нейронов данного слоя (3.2).

$$\delta^l = \delta^{l+1} * ROT180(\omega^{l+1})\sigma'(z^l) \quad (3.2)$$

где δ^l – ошибки нейронов текущего слоя;

δ^{l+1} – ошибки нейронов следующего слоя;

$ROT180(\omega^{l+1})$ – массив параметров следующего слоя, развернутый на 180°;

σ' – производная функции активации слоя;

z^l – выходы текущего слоя.

Для того, чтобы получить значение производной для весов текущего уровня, необходимо еще раз провести операцию двумерной свертки, на этот

раз – между ошибками нейронов слоя и выходом предыдущего слоя, также развернутым на 180° (3.3).

$$\frac{\partial C}{\partial \omega^l} = \delta^l * \sigma(ROT180(z^{l-1})) \quad (3.3)$$

где C – функция стоимости;

δ^l – ошибки нейронов текущего слоя;

$ROT180(z^{l-1})$ – массив параметров предыдущего слоя, развернутый на 180°.

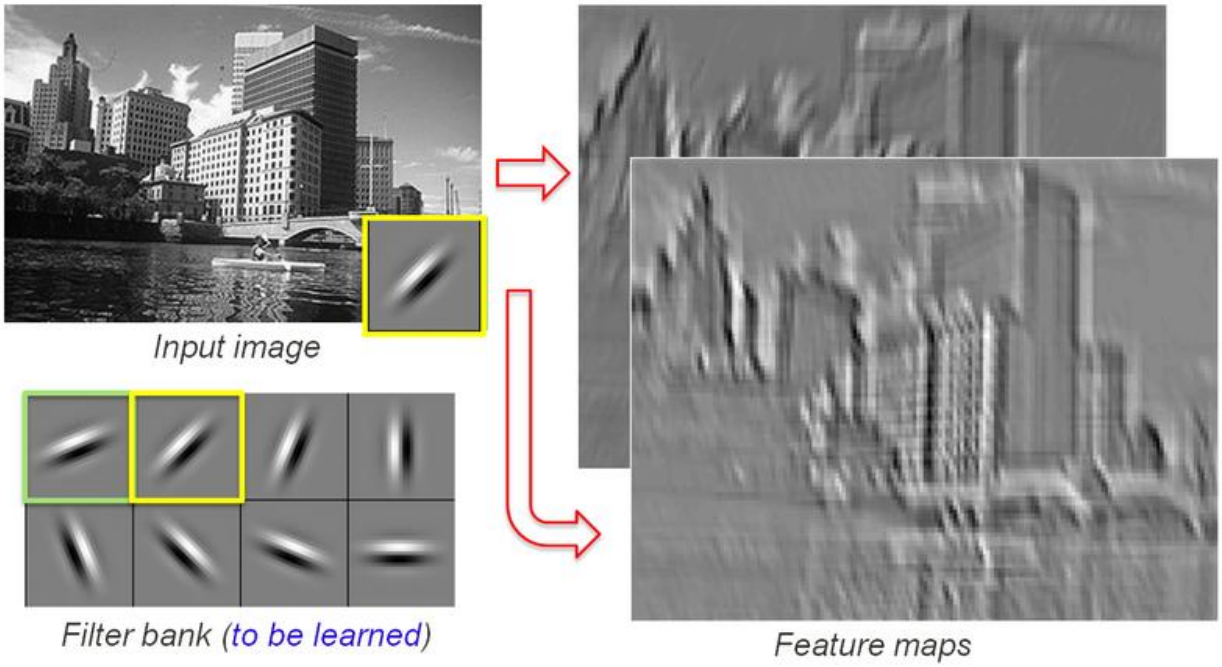

Рисунок 3.4 – Примеры фильтров и полученных с помощью них карт свойств

3.1.4 Полносвязный слой обычно применяется в сверточных нейронных сетях на последних этапах, когда размерность данных была снижена до приемлемого уровня с помощью сверточных слоев и слоев субдискретизации. В этом слое выход каждого нейрона предыдущего слоя поступает на вход всех нейронов текущего слоя. Размерность массива параметров слоя равна $m \times n$, где m – количество нейронов предыдущего слоя, n – количество нейронов текущего слоя. Имея реализацию для слоев только данного типа возможно построение многослойных нейронных сетей, например, многослойных персептронов, пригодных для решения большого количества прикладных задач (см. рисунок 3.4).

При прохождении через полносвязный слой происходит матричное умножение данных, попадающих на вход слоя на матрицу параметров (весов). После этого к данным применяется функция активации и результат передается на вход следующего слоя. Математически это преобразование описывается формулой (3.4):

$$a^l = \sigma(\omega^l \cdot a^{l+1} + b^l) \quad (3.4)$$

где a^l – выход текущего слоя;
a^{l+1} – выход предыдущего слоя;
w^l – матрица параметров слоя;
σ – функция активации слоя;
b^l – смещение слоя.

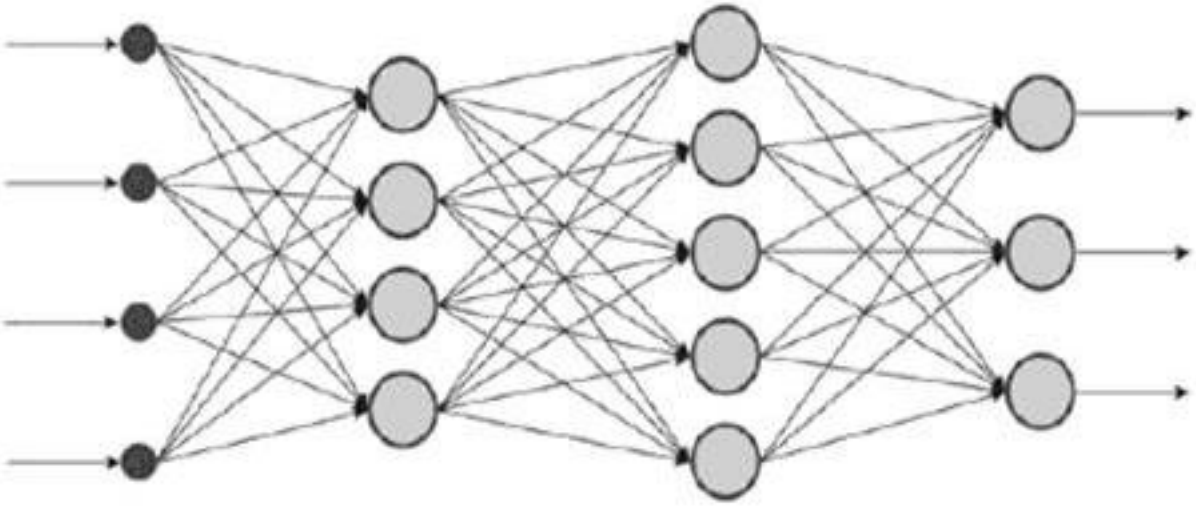

Рисунок 3.5 – Многослойный персептрон

Стоит отметить, что единственное различие между сверточными и полносвязными слоями состоит в том, что нейроны в сверточном слое связаны только с локальной областью на входе и что многие нейроны в сверточных слоях используют одни и те же параметр. Однако нейроны в обоих слоях все еще вычисляют матричные произведения, поэтому их функциональная форма идентична. Таким образом, можно показать, что полносвязный слой (см. рисунок 3.5) является частным случаем сверточного слоя.

Для любого сверточного слоя существует полносвязный слой, который работает идентично. Весовая матрица этого слоя будет большой матрицей, по большей части нулевой, за исключением некоторых элементов, где веса во многих из блоков равны (из-за совместного использования параметров).

Например, полносвязный слой с количеством нейронов 4096, на вход которого подается сигнал размерностью 7 × 7 × 512, может быть эквивалентно выражен как сверточный слой с фильтрами размером 7 × 7 и количеством каналов равным количеству нейронов в полносвязном слое. Другими словами, мы устанавливаем размер фильтра равным размеру данных на входе, и, следовательно, в нашем случае вывод будет просто 1 × 1 × 4096, что дает идентичный результат в качестве исходного полносвязного слоя.

3.1.5 При задаче классификации, когда необходимо получить на выходе нейронной сети вероятности принадлежности входного образа одному из не пересекающихся классов. Очевидно, что суммарный выход сети по всем

нейронам выходного слоя должен равняться единице (так же, как и для выходных образов обучающей выборки).

```
@Override
void forward(final DataSet dataSet) {
    final double sum = dataSet.getStreamData().sum();
    dataSet.update(e -> e / sum);
}
```

Обычно, Softmax-слой является последним слоем нейронной сети, количество нейронов в слое равно количеству классов, к которым необходимо отнести изображение. Математически, выход каждого нейрона слоя можно описать формулой (3.5):

$$y_i = \frac{e_i}{\sum_{j=1}^{n} e_j} \tag{3.5}$$

где y_i – выход i-го нейрона;

e_i – вход i-го нейрона.

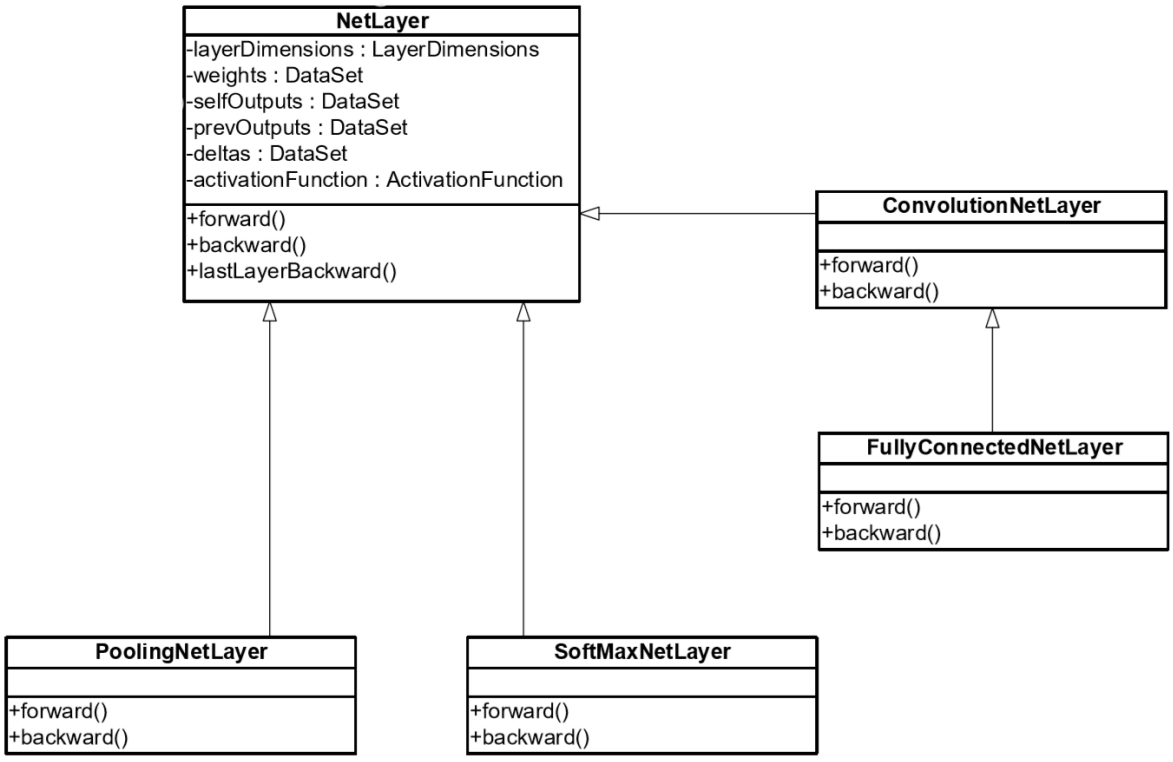

Рисунок 3.6 – Диаграмма классов слоев нейронной сети

3.1.6 В сверточных нейронных сетях между слоями свертки присутствуют так называемые слои субдискретизации, целью которых является уплотнение карты признаков, полученной от предыдущего слоя свертки. Эти слои не обладают параметрами, которые изменяются в процессе

обучения. Операция субдискретизации выполняет уменьшение размерности сформированных карт признаков. В архитектуре сверточной сети считается, что информация о факте наличия искомого признака важнее точного знания его координат, поэтому из нескольких соседних нейронов карты признаков выбирается максимальный и принимается за один нейрон уплотнённой карты признаков меньшей размерности. За счёт данной операции, помимо ускорения дальнейших вычислений, сеть становится более инвариантной к масштабу входного изображения.

Слой субдискретизации работает независимо на каждом канале входного сигнала и изменяет его размерность. Наиболее распространенной формой применения является слой субдискретизации с фильтрами размером 2 на 2, при этом количество каналов остается неизменным. Кроме функции максимума (см. рисунок 3.7), пул-блоки могут также реализовывать другие функции, такие как среднее арифметическое или L2-нормирование. Функция среднего арифметического часто использовалась ранее, но в последнее время она все чаще заменяется функцией максимума, которая, как было установлено, лучше работает на практике.

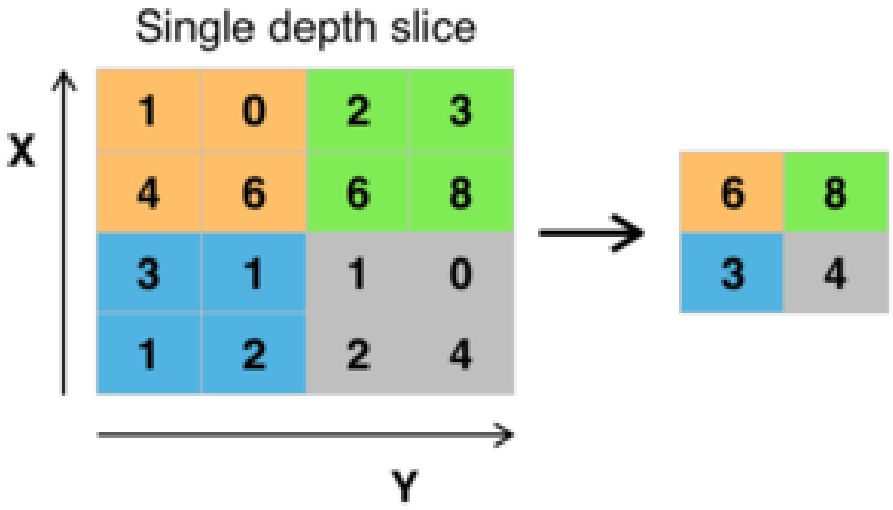

Рисунок 3.7 – Операция субдискретизации с использованием функции максимума

Класс `PoolingNetLayer` представляет собой базовый шаблон для различных слоев субдискретизации. В качестве аргумента конструктора в класс передается объект типа Dimension, который определяет размерность пул-блока. Очевидно, что входная размерность слоя должна быть кратна размерности пул-блока. Если данное условие не выполняется, в момент инициализации происходит создание `IllegalArgumentException`. также в конструктор класса передается функция субдискретизации, задачей которой является получение результатов пулинга из отдельных пул-блоков. Таким образом, клиент может создать собственный слой пулинга путем реализации функции субдискретизации. Изначально клиенту доступны три реализации

слоев пулинга, доступные в виде статических членов класса `PoolingNetLayer`:

- с функцией максимума (`MaxPoolingNetLayer`);
- с функцией среднего арифметического (`AveragePoolingNetLayer`);
- с функцией L2-нормирования (`L2PoolingNetLayer`);

Стоит помнить о том, что во время обучения сети происходит обратный проход – то есть движение по сети от последнего слоя к первому. Поэтому слои субдискретизации должны обладать возможностью восстановить состояние сигнала до прохождения слоя. Например, при использовании max-пулинга сохраняется индекс элемента, который оказался максимальным в каждой части входного сигнала.

3.1.7 Переобучение — это излишне точное соответствие нейронной сети конкретному набору обучающих примеров, при котором сеть теряет способность к обобщению. Другими словами, построенная нами модель смогла выучить обучающее множество (вместе с шумом, который в нем присутствует), но она не смогла распознать скрытые процессы, которые породили данное множество.

У глубоких сверточных нейронных сетей масса разнообразных параметров, особенно это касается полносвязных слоев. Переобучение может проявить себя в следующей форме: если у нас имеется недостаточно обучающих примеров, маленькая группа нейронов может стать ответственной за большинство вычислений, а остальные нейроны станут избыточны; или наоборот, некоторые нейроны могут нанести ущерб производительности, при этом другие нейроны из их слоя не будут заниматься ничем, кроме исправления их ошибок.

Чтобы помочь сети не утратить способности к обобщению в данных обстоятельствах, вводятся приемы регуляризации: вместо сокращения количества параметров, накладываются ограничения на параметры модели во время обучения, не позволяя нейронам изучать шум обучающих данных. Одним из приемов, позволяющих достичь этого является добавление в сеть dropout-слоев.

В частности, dropout с параметром p за одну итерацию обучения проходит по всем нейронам определенного слоя и с вероятностью p полностью исключает их из сети на время итерации. Это заставит сеть обрабатывать ошибки и не полагаться на существование определенного нейрона (или группы нейронов), а полагаться на “единое мнение” (consensus) нейронов внутри одного слоя. Это довольно простой метод, который эффективно борется с проблемой переобучения сам, без необходимости вводить другие регуляризаторы. (см. рисунок 3.8).

Понятно, что исключение большой части результатов работы предыдущих слоев скорее всего понизит эффективность работы сети. В то же время применение использование dropout положительно влияет на скорость

обучения сети. Задачей специалиста, который работает над решением конкретной задачи является выбор правильного параметра для dropout-слоя для того чтобы решить проблему переобучения и при этом не понизить точность работы модели.

Класс `DropoutNetLayer` представляет собой слой нейронной сети, эффект от которого виден только во время обучения сети. В качестве параметра конструктора в класс передается единственный параметр — целое положительное число, который представляет собой количество нейронов, которые должны быть исключены из данного слоя во время обучения. Конкретные входы, результаты работы которых будут отброшены, выбираются случайным образом при каждом цикле тренировки. При этом необходимо сохранить индексы элементов, которые остаются в массиве данных, так как при обратном проходе необходимо знать какие параметры предыдущих слоев принимали участие в создании результата работы сети.

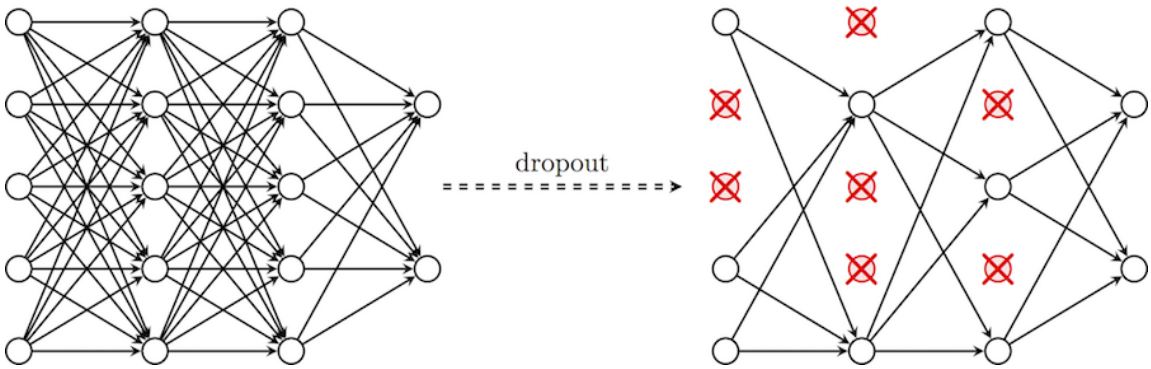

Рисунок 3.8 – Пример работы dropout слоя

С помощью описанных выше слоев возможно создание множества конфигураций нейронных сетей предназначенных для решения задач распознавания изображений и других многопараметрических объектов, аппроксимации функций,

3.1.8 В сверточных и полносвязных слоях перед тем как попасть на выходы нейронов, к данным применяется функция активации (передаточная функция) - преобразование, которое применяется к сигналу каждого нейрона перед тем как он будет передан на выход слоя.

Первоначально (с 1950-х) модели персептронов были полностью линейными, то есть в качестве функции активации служило только тождество. Но вскоре стало понятно, что основные задачи чаще имеют нелинейную природу, что привело к появлению других функций активации. Сигмоидальные функции (обязанные своему названию характерному S-образному графику) хорошо моделируют начальную “неопределенность” нейрона относительно бинарного решения, когда аргумент функции близок к нулю, в сочетании с быстрым насыщением при смещении аргумента в каком-либо направлении. (см. рисунок 3.9)

В последние годы в глубоком обучении получили широкое распространение полулинейные функции и их вариации (Rectified Linear Unit - ReLU) — они появились в качестве простого способа сделать модель нелинейной (“если значение отрицательно, обнулим его”), но в конце концов оказались успешнее, чем исторически более популярные сигмоидальные функции, к тому же они больше соответствуют тому, как биологический нейрон передает электрический импульс.

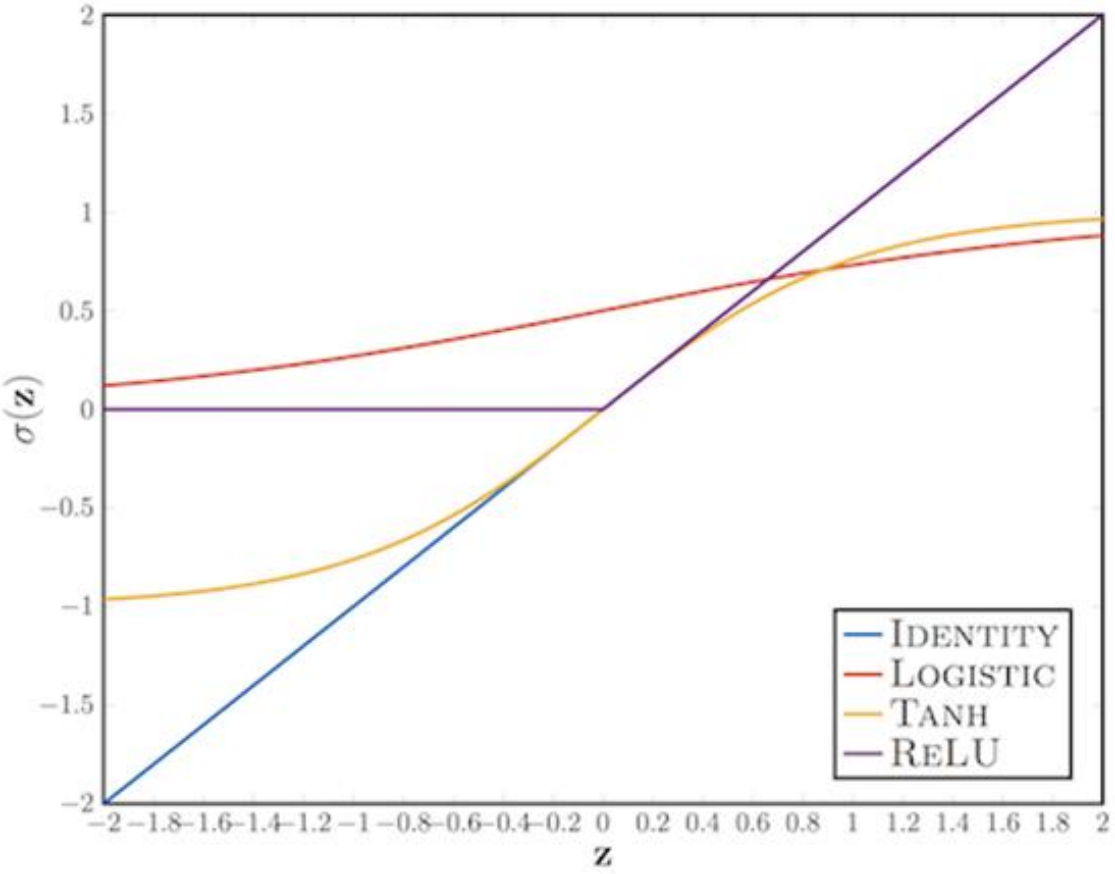

Рисунок 3.9 – Графики различных функций активации

Класс `ActivationFunction` инкапсулирует логику работы функции активации. В классе имеется два поля: `forwardOperator` и `backwardOperator`, которым соответствует сама функция активации, а также ее производная, которая используется в алгоритме обратного распространения ошибки. Класс содержит статические поля, которые представляют собой наиболее часто используемые функции активации. Подробная информация о всех реализованных пороговых функциях описана в таблице 3.1.

Все поля класса задаются как параметры конструктора, поэтому клиент может создать собственную реализацию класса с функцией, подходящей для решения его задачи. Так как функция активации используется в сверточных и полносвязных слоях, экземпляр класса `ActivationFunction` передается в качестве параметра конструктора при создании данных классов.

Во время работы метода `forward()` происходит вызов функции `forwardOperator`, которая применяется к результату матричного умножения входных параметров с весами слоя. Соответственно, во время обратного прохода по сети функция `backwardOperator` применяется к выходам текущего слоя.

Таблица 3.1 – Функции активации

Название функции	Формула	Реализация
Тождество	$f(x) = x$	`ActivationFunction.IDENTITY`
Полулинейная функций (ReLU)	$f(x) = \max(x, 0)$	`ActivationFunction.RELU`
Сигмоида	$f(x) = \frac{1}{1 + \exp(-x)}$	`ActivationFunction.SIGMOID`
Гиперболический тангенс	$f(x) = \tanh(x)$	`ActivationFunction.TANH`

Ниже представлены реализации всех пороговых функций, входящих в состав библиотеки:

```
public static final ActivationFunction SIGMOID = new
ActivationFunction(
                e -> 1/( 1 + Math.pow(Math.E,(-1*e))),
                e -> e * (1 - e));

public static final ActivationFunction RELU = new
ActivationFunction(
                e -> e > 0 ? e : 0,
                e -> e > 0 ? 1 : 0);

public static final ActivationFunction IDENTITY = new
ActivationFunction(
                e -> e, e -> e);

public static final ActivationFunction TANH = new
ActivationFunction(
          e -> Math.tanh(e)
          e -> 1 – Math.tanh(e) * Math.tanh(e));
```

3.1.9 Класс `Network` представляет собой модель нейронной сети. Он является контейнером для слоев сети и имеет методы для запуска прямого и обратного прохода по сети, добавления новых слоев и извлечения статистических данных о работе модели. Во время работы приложения клиент непосредственно взаимодействует с объектами данного класса.

Класс обладает единственным конструктором, который принимает параметр типа `Dimension` – размерность входного слоя данной нейронной сети. Наиболее значимыми полями класса `Network` являются:

- `LinkedList<NetLayer> layers` – контейнер для всех слоев сети. Так как необходимо обеспечить возможность выполнять прямой и обратный проход по всем слоям сети, для реализации данного поля использовался класс `java.util.LinkedList`;
- `Dimension inputDimension.` – размерность входного слоя сети. Определяет требование к данным, которые будут подаваться на вход нейронной сети.

Рассмотрим методы класса `Network`:

- `void addLayer(final Layer layer)` – добавляет новый слой в контейнер слоев данной сети. В качестве параметра передается объект типа Layer, который является частью модуля конструирования. Во время работы данного метода происходит инициализации слоя сети в виде наследника класса `NetLayer`: проверка всех параметров слоя на корректность, вычисление входных и выходных размерностей слоя. При успешной инициализации происходит добавление нового объекта в контейнер слоев;
- `void forward(final Dataset dataset)` – данный осуществляет прямой проход по сети. В качестве аргумента методу передается экземпляр класса DataSet, который представляет собой входной сигнал, для которого требуется получить реакцию сети. Во время работы метода происходит проверка на соответствие размерности переданного параметра размерности входного слоя сети. Если это условие не выполняется, то происходит создание `IllegalArgumentException` и работа метода завершается. Если проверка прошла успешно, то объект DataSet по очереди передается методам `forward()` каждого из слоев, начиная с первого слоя. В результате клиент получает ответ сети на заданный входной сигнал.
- `void forward(final Double[] data)` – данный метод также осуществляет прямой проход по сети, но в качестве аргумента ему передается массив чисел double которые представляют входной сигнал. Размер массива должен совпадать с размером входного слоя сети;
- `void backward(final DataSet reference, final DataSet outputs)` – данный метод используется в алгоритме градиентного спуска, с помощью которого происходит обучение сети. На вход методу подаются два массива данных: `reference` представляет собой эталонный выход сети, по которому следует провести обучение, `outputs` – реальные выходные данные, полученные с помощью прямого прохода по сети. Таким образом перед вызовом данного метода необходимо вызвать метод `forward()` чтобы получить массив данных `outputs`, а также для того чтобы каждый слой сети сохранил информацию о собственных выходах (поле `selfOutputs` класса `NetLayer`). Во время работы метода создается объект

DataSet который представляет собой массив ошибок слоя. Для последнего слоя сети происходит вызов lastLayerBackward(), куда передаются эталонный выход сети, фактический выход сети и массив ошибок, который изначально инициализирован единицами. После этого происходит вызов метода backward() для каждого из оставшихся слоев сети. Таким образом в каждом слое накапливается величина ошибок, с помощью которых происходит обновление весов слоя. Ниже представлен алгоритм работы метода backward():

```
void backward(final DataSet y, final DataSet outputs) {
    final Iterator<NetLayer> iter =
layers.descendingIterator();
    final DataSet deltas = new DataSet(
 y.getDimension(), () -> 1);

    NetLayer lastLayer = null;
    while (iter.hasNext()) {

        final NetLayer layer = iter.next();
        if (lastLayer == null) {
            layer.lastLayerBackward(deltas, y, outputs);
        } else {
            layer.backward(deltas, lastLayer.getWeights());
        }

        lastLayer = layer;
    }
}
```

- boolean isEmpty() - вспомогательный метод, который позволяет узнать содержит ли данная модель сети какие-либо слои кроме входного слоя. Если сеть не содержит слоев, то вызовы методов forward() или backward() будут завершаться с ошибкой.

3.2 Модуль вычислений

Во время работы нейронной сети необходимо производить множество операций с числовыми массивами. Самой затратной в плане машинного времени операцией является двумерная свертка, которая применяется как в сверточных, так и в полносвязных слоях, при прямом проходе данных через сеть и при обучении. На этапе проектирования было принято решение, о переносе всей логики, которая отвечает за математические вычисления в отдельный модуль. Таким образом, клиент может самостоятельно выбрать метод вычислений, используемый в системе, а при необходимости – создать собственную реализацию.

3.2.1 Интерфейс `MathOperations` описывает математические операции, используемые при работе нейронных сетей. Он содержит методы, задачей которых является осуществление математических операций над массивами данных, представленными в виде объектов `DataSet`. Рассмотрим методы данного интерфейса:

- `DataSet convolve(final List<DataSet> kernels, final DataSet input, final int padding, final int stride)` – данный метод используется в сверточных слоях, осуществляет двумерную свертку данных, переданных в качестве аргумента `input`. Аргумент `kernels` представляет собой массив ядер свертки. Также в метод передаются параметры `padding` и `stride` – начальное смещение ядра свертки и шаг свертки соответственно. Код, который вызывает метод `convolve()` гарантирует, что входные данные и ядра свертки будут иметь корректные размерности. После завершения работы метода размерность результата также проходит проверку и в случае ошибки создается `IllegalStateException`. Клиенту также предоставлена возможность создания исключения `MathOperationException`, если во время вычислений произошла ошибка. В этом случае произойдет корректное завершение работы сети, а информация об ошибке будет сохранена;
- `DataSet dotProduct(final DataSet A, final DataSet B)` – данный метод используется в полносвязных слоях (`FullyConnectedNetLayer`) как при прямом, так и при обратном проходе. Фактически, метод выполняет умножение двух матриц, переданных в качестве аргументов. Гарантируется, что входные данные имеют корректные размерности. После завершения работы метода размерность результата также проходит проверку и в случае ошибки создается `IllegalStateException`;
- `DataSet outerProduct(final DataSet vectorA, final DataSet vectorB)` – данный метод используется в полносвязных слоях в алгоритме обратного распространения ошибки. При вызове этого метода осуществляется вычисление внешнего продукта двух векторов, переданных в качестве аргументов `vectorA` и `vectorB` (3.6)

$$u \otimes v = uv^T = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 & u_1v_3 & u_1v_4 \\ u_2v_1 & u_2v_2 & u_2v_3 & u_2v_4 \\ u_3v_1 & u_3v_2 & u_3v_3 & u_3v_4 \\ u_4v_1 & u_4v_2 & u_4v_3 & u_4v_4 \end{bmatrix} \quad (3.6)$$

Гарантируется, что данные которые переданы в метод в качестве аргументов являются векторами (другими словами, все размерности `width`, `height`, `depth` кроме одной равны единице). На выход метода должен быть передан объект `DataSet`, который представляет собой матрицу (то есть размерность `depth` этого объекта должна быть равна единице).

3.2.2 Как было сказано выше, интерфейс `MathOperations` описывает математические операции над массивами данных, которые производятся во время работы слоев нейронных сетей. Самой простой реализацией этого интерфейса является класс `ApacheMathOperations`, который реализует перечисленные выше методы с помощью библиотеки Apache Common Math. Данная библиотека является наиболее популярной библиотекой математических операций для языка Java. Она предоставляет возможность производить базовые математические операции над матрицами и векторами, однако в ней отсутвует реализация двумерной свертки, поэтому в классе `ApacheMathOperations` метод convolve реализован без помощи сторонних библиотек:

```
@Override
public DataSet convolve(final List<DataSet> kernels, final
DataSet input, final int padding) {
    final List<Double> result = new ArrayList<>(outputDepth
* outputHeight * outputWidth);

    int ay = -padding;
    for (int y = 0; y < outputHeight; y++, ay++) {
      int ax = -padding;
      for (int x = 0; x < outputWidth; x++, ax++) {
       for (DataSet kernel : kernels) {
          double res = 0.0;
          for (int fx = 0; fx < kernelWidth; fx++) {
            int ox = ax + fx;
            for (int fy = 0; fy < kernelHeight; fy++) {
              int oy = ay + fy;
              if (oy >= 0 && oy < inputHeight && ox >= 0 &&
                    ox < inputWidth) {

                for (int z = 0; z < inputDepth; z++) {
                        res += kernel.get(fx, fy, z) *
                          input.get(ox, oy, z);
                  }
                }
              }
            }
            result.add(res);
          }
        }
      }
    return new DataSet(result, new Dimension(outputWidth,
outputHeight, outputDepth));
}
```

Таким образом, класс `ApacheMathOperations` представляет собой простейшую реализацию интерфейса `MathOperations`. Вычисления производятся с использованием одного потока исполнения, поэтому при

использовании больших объемов данных эффективность работы системы, использующей данную реализацию может быть низкой.

3.2.3 При выполнении операций свертки, умножения матриц, вычисления внешнего продукта можно повысить скорость работы путем распараллеливания задачи между несколькими потоками исполнения. Соответствующей реализацией интерфейса `MathOperations` является класс `ConcurrentMathOperations`. Использование данного класса может повысить работоспособность приложения при обработке больших объемов данных, но следует помнить, что многопоточные вычисления могут работать медленнее чем однопоточные в том случае, если затраты на создание и поддержку работы множества потоков превышают затраты на вычисление данных без использования многопоточности.

3.2.4 Класс `MathOperationsException` наследуется от класса `java.lang.Exception` и является проверяемым исключением, хранящим информацию об ошибке, произошедшей во время выполнения операций вычисления. Класс не имеет дополнительных методов и полей, кроме тех, которые определены супер-классом `java.lang.Exception`. Клиенты могут использовать данный тип исключения в собственных реализациях интерфейса `MathOperations` для того чтобы работа системы была завершена корректно.

3.3 Модуль клиентского интерфейса

Для создания конфигурации модели сети клиенту предоставляется набор классов, с помощью которых можно описать архитектуру требуемого решения и задать все необходимые параметры для построения модели. Размеры ядер свертки и их количество в сверточных слоях, количество нейронов в полносвязных слоях, функции активации, используемые слоями – эти параметры должны быть выбраны клиентом. Рассмотрим классы и входящие в их состав поля и методы, разработанные для этих целей.

Класс Layer содержит в себе набор шаблонов, которые позволяют клиенту задать параметры, необходимые для создания модели сети. Рассмотрим поля и методы данного класса:

- `int neuronCount` – целое положительное число, которое обозначает количество нейронов в полносвязном слое;
- `Dimension filterSize` – объект типа `Dimension`, обозначающий размерность ядер свертки в сверточном слое;
- `ActivationFunction activationFunction` – функция активации данного слоя;

- LayerType type – объект перечисления LayerType, обозначающий тип слоя, представленного данным экземпляром класса Layer. Ниже представлены члены перечисления LayerType:

```
public enum LayerType {
    CONVOLUTION,
    FULLY_CONNECTED,
    SOFTMAX,
    POOLING,
    DROPOUT
}
```

Методы класса Layer позволяют получить доступ к полям класса, а также создать шаблоны различных типов слоев:

- Layer fullyConn(int neuronCount)- метод принимает в качестве аргумента количество нейронов и возвращает шаблон для создания полносвязного слоя;

- Layer conv(Dimension kernelSize, int kernelCount) – метод принимает в качестве аргумента размерность ядра свертки, а также количество ядер и возвращает шаблон для создания сверточного слоя;

- Layer pool(Dimension poolSize, Function<List<Double>, Double> poolingFunction) – метод принимает размер блока пулинга и функцию пулинга, возвращает шаблон для создания слоя пулинга;

- Layer softmax() – возвращает шаблон для создания softmax-слоя;

- Layer dropout(Dimension dropoutSize) – метод принимает в качестве аргумента размерность выхода dropout слоя и возвращает шаблон для создания данного типа слоя.

Диаграмма классов всей системы представлена на чертеже ГУИР.400201.139 РР.2.