4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

4.1 Алгоритм работы метода `trainSingle()`

В классе `SGDTrainer` определены методы, с помощью которых можно проводить обучение нейронной сети. Одним из методов данного интерфейса является метод `trainSingle()`, используемый для обучения сети методом стохастического градиентного спуска (Stohastic Gradient Descent). Данный метод относится к оптимизационным алгоритмам и нередко используется для настройки параметров модели машинного обучения. При стохастическом (оперативном) градиентном спуске значение градиента аппроксимируются градиентом функции стоимости, вычисленном только на одном элементе обучения, в отличие от стандартного градиентного спуска, где происходит накопление градиента за счет множества обучающих наборов. Затем параметры изменяются пропорционально приближенному градиенту. Таким образом параметры модели изменяются после каждого объекта обучения. Достоинством метода стохастического градиентного спуска является тот факт, что обучение можно проводить в режиме реального времени, по мере того как поступают обучающие наборы. Для больших массивов данных стохастический градиентный спуск может дать значительное преимущество в скорости по сравнению со стандартным градиентным спуском.

Для вычисления градиентов в многослойных нейронных сетях применяется метод обратного распространения ошибки – итеративный алгоритм, основная идея которого состоит в распространении сигналов ошибки от выходов сети к её входам, в направлении, обратном прямому распространению сигналов в обычном режиме работы.

Рассмотрим код метода `trainSingle()`

```
public void trainSingle(DataSet input, DataSet reference)
{
        final DataSet outputs = network.forward(data);
        network.backward(ref, outputs);
        network.updateWeights();
}
```

В качестве аргументов в метод передаются два объекта `DataSet`: набор входных данных, для которых производится обучение (`DataSet input`) и эталонная реакция сети на эти входные данные (`DataSet reference`). Как видно, метод состоит из трех основных операций. Сначала осуществляется прямой проход по сети:

```
final DataSet outputs = network.forward(data);
```

После этого полученный результат вместе с эталонным выходом передаются в качестве аргумента в метод обратного прохода по сети:

```
network.backward(ref, outputs);
```

Во время выполнения обратного прохода в слоях сети было сохранено значение градиента. Последней операцией стохастического градиентного спуска является обновление параметров сети:

```
network.updateWeights();
```

Все три операции реализованы в виде методов класса `Network`, ссылка на экземпляр которого была передана в класс `SGDTrainer` во время инициализации. Рассмотрим подробнее каждую из них.

4.1.1 Первым этапом стохастического градиентного спуска является прямой проход по сети с использованием входных данных, для которых необходимо произвести обучение. Метод `Network.forward(DataSet input)` принимает в качестве аргумента массив входных данных и поочередно пропускает их через каждый слой нейронной сети. По мере прохода через слои нейронной сети размерность данных меняется и в итоге мы получаем выходной массив данных, который возвращается клиенту. Рассмотрим алгоритм работы метода `forward()`:

```
public DataSet forward(final DataSet data) {
    if (layers.isEmpty()) {
        throw new IllegalStateException("Cannot call
              forward() on empty network");
    }

    final DataSet dataSet = new DataSet(data);
    this.layers.forEach(l -> l.forward(dataSet));
    return dataSet;
}
```

Перед началом работы метода происходит проверка на наличие слоев в сети. Очевидно, что сеть, для которой не задана конфигурация слоев, не подлежит тренировке:

```
if (layers.isEmpty()) {
        throw new IllegalStateException("Cannot call
              forward() on empty network");
    }
```

После этого происходит создание копии аргумента `data` с помощью конструктора копирования класса `DataSet`. Данный шаг необходим в связи с тем, что массив данных, который передается в метод `forward()` меняет свою размерность и содержимое. Изменение данных, хранящихся в качестве

обучающего набора является нежелательным, поэтому для прямого прохода по сети предоставляется копия данных:

```
final DataSet dataSet = new DataSet(data);
```

После этого происходит последовательный вызов метода `forward()` класса `NetLayer` для каждого из слоев нейронной сети и результат работы возвращается клиенту:

```
this.layers.forEach(l -> l.forward(dataSet));
```

Главной целью работы метода `forward()` для алгоритма градиентного спуска является получение выходов для каждого слоя сети. Эти данные сохраняются в поле `selfOutputs` и используются во время обратного прохода.

4.1.2 Вторым этапом метода градиентного спуска является обратный проход по сети. Во время этой операции происходит вычисление градиента в слоях нейронной сети с помощью сохраненных ранее выходов слоев. Ниже представлен код функции `backward()`, которая осуществляет обратный проход:

```
void backward(final DataSet y, final DataSet outputs) {
  Iterator<NetLayer> iter = layers.descendingIterator();
  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;
  }
}
```

Во время работы метода осуществляется проход слоев в обратном порядке – от последнего слоя к первому, поэтому первым шагом является создание итератора:

```
Iterator<NetLayer> iter = layers.descendingIterator();
```

После этого начинается перебор слоев сети. Сначала происходит вызов метода `lastLayerBackward()` для последнего слоя сети, так как для

вычисления градиента на последнем слое требуется только эталонный выход сети и фактический выход, полученный при помощи прямого прохода:

```
layer.lastLayerBackward(deltas, y, outputs);
```

Кроме этого в метод `lastLayerBackward()` в качестве аргумента передается еще один объект типа `DataSet – deltas`, который представляет собой массив ошибок нейронов слоя. Для вычисления собственных ошибок слою необходимы данные об ошибках слоя, который следует за ним. Поэтому при передаче ссылки на объект `deltas` в метод `backward()` каждого слоя он считывает данные которые хранятся в нем, использует их для вычисления ошибок собственных нейронов и записывает результат обратно в `deltas`. Таким образом, каждый слой получает доступ к ошибкам предыдущего. В начале работы метода, deltas инициализируется единицами, а размерность данных равна размерности выхода сети:

```
DataSet deltas = new DataSet(y.getDimension(), () -> 1);
```

Кроме массива ошибок предыдущего слоя на вход метода `backward()` каждого слоя, кроме последнего подаются параметры слоя, следующего за ним:

```
layer.backward(deltas, lastLayer.getWeights());
```

Поэтому во время работы цикла слой, участвующий в предыдущей итерации, сохраняется в переменной `lastLayer`:

```
lastLayer = layer;
```

В результате работы метода `backward()` в слоях накапливаются градиенты, полученные от обучения единственного обучающего набора.

4.1.3 Последним шагом алгоритма стохастического градиентного спуска является обновление параметров всех слоев. Для этого необходимо лишь сложить полученные на предыдущем шаге градиенты с текущими значениями параметров. Но так как во время стандартного градиентного спуска перед обновлением параметров происходит накопление градиента за счет множества обучающих наборов, эта логика вынесена в отдельный метод:

```
void updateWeights() {
    this.layers.forEach(l -> l.updateWeights());
}
```

После обновления параметров слоя его градиент обнуляется и слой становится готов к новому циклу обучения. Таким образом после описанных

выше операций параметры нейронной сети будут обновлены с учетом единственного обучающего набора. Диаграмма последовательности данного алгоритма представлена на чертеже ГУИР.400201.139 РР.1.

4.2 Алгоритм работы метода `trainFull()`

При стандартном (пакетном) градиентном спуске для корректировки параметров модели используется градиент. Градиент обычно считается как сумма градиентов, вызванных каждым элементом обучения. Вектор параметров изменяется в направлении обратном градиенту с заданным шагом. Поэтому стандартному градиентному спуску требуется один проход по обучающим данным до того, как он сможет менять параметры.

Главным отличием данного алгоритма от описанного ранее стохастического градиентного спуска является то, что обучение производится с использованием множества обучающих наборов, а не одного. Рассмотрим код метода `trainFull()`:

```
public void trainFull(Map<DataSet, DataSet> trainingData) {
  trainingData.forEach((x, y) -> {
    final DataSet outputs = network.forward(x);
    network.backward(y, outputs);
  });

  network.updateWeights();
}
```

В качестве аргумента методу передается ассоциативный массив, каждым элементом которого является обучающий набор и соответствующий ему эталонный выход сети. Первые два этапа градиентного спуска (прямой проход по сети и обратный проход по сети) выполняются в цикле для всех обучающих наборов:

```
trainingData.forEach((x, y) -> {
  final DataSet outputs = network.forward(x);
  network.backward(y, outputs);
});
```

При этом в слоях нейронной сети происходит накопление градиентов. После этого происходит вызов метода `updateWeights()` и параметры всех слоев сети обновляются с учетом всех обучающих наборов.

Обычно данный метод обучения применяется в том случае, когда заранее имеется большое количество обучающих наборов и производительность системы позволяет провести обучение с использованием всех имеющихся данных. На чертеже ГУИР.400201.139 ПД представлена блок-схема работы программы, использующей для обучения сети метод `trainFull()`.

4.3 Алгоритм работы метода `trainBatch()`

Кроме двух описанных выше методов обучения нейронной сети существует еще один, который можно описать как нечто среднее между стохастическим и пакетным градиентным спуском. Данный метод носит название «mini-batch». В этом случае имеется множество обучающих наборов, но градиент рассчитывается лишь для определенного подмножества из них, размер которого задается заранее.

Рассмотрим код метода `trainBatch()`, который реализует данный алгоритм:

```
public void trainBatch(final Map<DataSet, DataSet> data,
        final int batchSize) {
Validate.isTrue(batchSize > 0, "Batch size not positive");
int batchCount = 0;
for (Map.Entry<DataSet, DataSet> trainExample :
        data.entrySet()) {
   batchCount++;
   final DataSet outputs =
        network.forward(trainExample.getKey());

   network.backward(trainExample.getValue(), outputs);

   if (batchCount >= batchSize) {
       network.updateWeights();
       batchCount = 0;
   }
}
network.updateWeights();

}
```

Как видно, в отличие от метода `trainFull()`, метод `trainBatch()` требует передачи второго аргумента – размера пакета `batchSize`. Этот аргумент представляет собой целое положительно число, поэтому сначала происходит его валидация:

```
Validate.isTrue(batchSize > 0, "Batch size not positive");
```

После этого запускается цикл по массиву обучающих наборов, аналогично методу trainFull():

```
for (Map.Entry<DataSet, DataSet> trainExample :
        data.entrySet()) {
```

Перед началом цикла происходит инициализация переменной `batchCount`, которая показывает сколько обучающих наборов из текущего пакета было уже обработано. В цикле происходит выполнение прямого и

обратного обхода по сети, а кроме этого – вызов метода `updateWeights()` в том случае, если было обработано количество обучающих наборов, равное `batchSize`:

```
if (i >= batchSize) {
        network.updateWeights();
        batchCount = 0;
}
```

После завершения работы цикла происходит последний вызов метода `updateWeights()` для последних обучающих наборов, количество которых было меньше чем `batchSize`. Даже в том случае, если `updateWeights()` был вызван в цикле применительно ко всем обучающим наборам и перед вторым вызовом метода в сеть не был передан новый обучающий набор, величина градиента в слоях будет равна 0, что никак не повлияет на текущие параметры сети:

```
network.updateWeights();
```

Описанные выше методы позволяют обучать нейронные сети с помощью различных вариаций алгоритма градиентного спуска. Скорость обучения будет зависеть от таких параметров, как размеры слоев сети, их количество и используемый метод вычисления двумерной свертки.

4.4 Алгоритм работы метода `getChannel()`

Метод `getChannel(final int channel)` является публичным методом класса `DataSet` и предназначен для извлечения отдельных каналов из трехмерного массива данных. Данный метод используется при работе сверточных сетей для того, чтобы производить операцию свертки над отдельными каналами изображения.

Рассмотрим код данного метода:

```
public DataSet getChannel(final int channel) {
    Validate.isTrue(channel >= 0 && channel <
        this.getDimension().getDepth(),
       "this should be a valid channel index");

    final int channelSize = getDimension().getHeight() *
        getDimension().getWidth();

    final double[] channelData = new double[channelSize];

    final int channelCount = this.getDimension().getDepth();
     for (int i = channel, j = 0; i < this.getSize();
          i+=channelCount) {
                channelData[j++] = this.data.get(i);}
```

```
        return new DataSet(channelData, new
           Dimension(getDimension().getWidth(),
           getDimension().getHeight()));
    }
```

В качестве единственного аргумента метод получает целое число `channel` – индекс канала, который необходимо извлечь из массива данных. Поэтому в первую очередь производится валидация аргумента. Индекс канала не должен быть больше чем общее количество каналов в массиве данных (поле `depth` класса `DataSet`):

```
Validate.isTrue(channel >= 0 && channel <
       this.getDimension().getDepth(),
      "this should be a valid channel index");
```

После этого происходит вычисление количества элементов в одном канале и сохранение данной величины в переменной `channelSize`:

```
final int channelSize = getDimension().getHeight() *
          getDimension().getWidth();
```

Теперь, когда известна величина массива данных, который будет возвращен в качестве результата, инициализируется массив, который будет хранить данные канала:

```
final double[] channelData = new double[channelSize];
```

Теперь необходимо выбрать из текущего массива данных только те элементы, которые относятся к каналу с заданным индексом. Для этого начнем цикл по элементам массива данных. В классе `DataSet` элементы, имеющие одинаковые координаты, но принадлежащие разным каналам расположены друг за другом. Поэтому чтобы выбрать элементы одного канала, необходимо совершить проход по массиву с шагом, равным количеству каналов, начиная с элемента, равного индексу искомого канала.

```
for (int i = channel, j = 0; i < this.getSize();
          i+=channelCount) {
```

При каждом шаге цикла будет происходить копирование элемента в новый массив `channelData`. Таким образом после завершения цикла в данном массиве будут собраны все элементы, принадлежащие каналу с заданным индексом.

Так как результат работы метода требуется возвратить в виде объекта класса `DataSet`, то происходит инициализация объекта данного класса с помощью конструктора, принимающего в качестве параметра массив `channelData`:

```
return new DataSet(channelData, new
        Dimension(getDimension().getWidth(),
        getDimension().getHeight()));
```

Размерности `height` и `width` в данном случае равны размерностям исходного объекта `DataSet`, а количество каналов устанавливается равным единице.

4.5 Алгоритм работы метода `expand()`

Метод `expand(final int newDepth)` также является публичным методом класса `DataSet`. Так же как и метод `getChannel()`, метод `expand()` используется для получения нового массива данных на основе существующего. Но если метод `getChannel()` возвращает единственный канал из массива данных, то метод `expand()` напротив, на основе единственного канала создает трехмерный массив данных.

Ниже представлен код метода `expand()` :

```
public DataSet expand(final int newDepth) {
    Validate.isTrue(this.getDimension().getDepth() == 1);
    Validate.isTrue(newDepth > 1);

    final List<Double> newData = this.getStreamData()
            .flatMap(e -> DoubleStream.generate(() ->
                    e).limit(newDepth))
            .boxed()
            .collect(Collectors.toList());
    this.dimension = new
Dimension(this.getDimension().getWidth(),
this.getDimension().getHeight(), newDepth);

    return this.update(newData, this.dimension);
}
```

Единственный аргумент, передаваемый методу – `newDepth` – целое положительное число, представляющее собой количество каналов в новом массиве данных. В первую очередь производится валидация аргумента, а также проверка на то, что текущий объект DataSet имеет один единственный канал. (это равнозначно единице в поле depth):

```
Validate.isTrue(this.getDimension().getDepth() == 1);
Validate.isTrue(newDepth > 1);
```

После этого запускается цикл по всем элементам массива данных при помощи вызова метода `getStreamData()`, который возвращает объект потока данных `Stream`. Объекты `java.lang.stream.Stream` являются частью стандартной библиотеки языка java, с они позволяют совершать

проход по всем элементам какой-либо коллекции данных, вносить в них изменения и получить на выходе новую коллекцию данных.

```
final List<Double> newData = this.getStreamData()
```

Целью работы метода является добавление новых каналов в текущий массив путем копирования существующего канала. Для того чтобы добавить n каналов в объект `DataSet`, необходимо каждый элемент существующего массива заменить на последовательность из n одинаковых элементов. Данное преобразование осуществляется при помощи вызова метода `flatMap()`:

```
.flatMap(e -> DoubleStream.generate(() ->
                          e).limit(newDepth))
```

В качестве аргумента в данный метод передается функция, которая на основе каждого элемента потока данных e новый поток данных путем вызова статического метода generate класса `DoubleStream`. Затем, при помощи вызова метода `limit()` размерность этого потока данных сокращается до величины `newDepth`. В конце работы метода `flatMap()` все потоки данных, количество которых равно количеству элементов исходного потока, объединяются в один.

После этого необходимо собрать данные в переменную `newData`. После вызова метода `boxed()` тип элементов в потоке данных меняется на объектный тип Double. Это необходимо С помощью вызова метода `collect()` данные, содержащиеся в потоке преобразуются в список:

```
.collect(Collectors.toList());
```

В качестве аргумента методу `collect()` требуется передать экземпляр класса `Collector`, полученный с помощью вызова метода `toList()` вспомогательного класса `Collectors`. С помощью этого экземпляра поток данных будет преобразован в объект `java.util.List` Теперь в переменной `newData` хранятся элементы нового, многоканального массива. Перед обновлением данных необходимо изменить размерность текущего объекта `DataSet`:

```
this.dimension = new
     Dimension(this.getDimension().getWidth(),
     this.getDimension().getHeight(), newDepth);
```

Для создания нового объекта `Dimension` используется конструктор с тремя аргументами. Новая размерность массива данных имеет такие же параметры `width` и `height`, но параметр `depth` устанавливается равным аргументу метода `newDepth`.

Последним шагом является обновление данных с помощью массива, хранящегося в переменной `newData`. Для этого используется метод update, который принимает в качестве аргументов экземпляр класса `java.util.ArrayList`, содержащий данные для этого массива и экземпляр класса `Dimension`, определяющий размерность нового массива данных. Также после этого метод возвращает ссылку на текущий объект `DataSet`. Это делает возможным использование рассматриваемого метода в цепочке вызовов:

```
return this.update(newData, this.dimension);
```

Экземпляры класса `DataSet` используются в большинстве модулей системы: для хранения параметров и выходов слоев сети, для представления обучающих наборов при обучении сети и входных данных при ее функционировании. Так как с помощью экземпляров этих классов происходит импорт данных в нейронную сеть со стороны клиента, они являются частью открытого API. Рассмотренные выше методы класса `DataSet`, позволяют безопасно осуществлять разнообразные преобразования массивов данных, необходимые клиенту.