

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

Тестирование программного обеспечения – процесс исследования программного обеспечения с целью получения информации о качестве продукта. Тестирование является одним из важных этапов разработки, поскольку при написании программного кода невозможно предусмотреть все случаи, и ошибки в работе программы неизбежны. Кроме того, возможно появление ошибок в случае внесения дополнительного функционала в программу даже в уже отлаженном рабочем коде.

Тестирование программы направлено на доказательства её соответствия заявленным требованиям. Таким образом, основной целью тестирования является выделение несоответствия функционирования программы заявленным требованиям и устранение их.

Тестирование программы проводилось в два этапа:

- поэтапное тестирование отдельно каждого модуля в процессе написания программного кода;
- полное тестирование программы после окончания процесса написания программного кода.

Оба этапа являются достаточно важными, ни один из них невозможно исключить. Например, без модульного тестирования, при анализе работы программы в целом, будет происходить достаточное количество сбоев, выявить и локализовать которые может оказаться достаточно сложным заданием, в то время как при анализе работы одного модуля неисправность оказывается достаточно очевидной. И в обратном случае, работоспособность каждого компонента в отдельности не гарантирует корректное поведение всей программы в целом.

Кроме этого одной из важнейших задач тестирования является проверка программы на соответствие заявленным требованиям. После того как был написан и протестирован весь код программного продукта, необходимо удостовериться что выполнены все требования, поставленные заказчиком перед началом разработки.

Тестирование программы проводилось на следующих компьютерах:

- Intel Core i5 7600 4 ядра 3,5 ГГц, оперативная память 16ГБ, видеокарта Intel HD Graphics до 1696 МБ. Операционная система Windows 10;
- Intel Core i3 2 ядра по 1,86 ГГц, оперативная память 3 ГБ, видеокарта GeForce 9800GT 512 МБ. Операционная система Windows 7 Ultimate x63 Service Pack 1;
- Intel Core i5 M2410 2 ядра по 2,3 ГГц (Intel Turbo Boost до 2,9 ГГц), оперативная память 4 ГБ, видеокарта гибридная, Intel HD Graphics 3000 и GeForce M540 1 ГБ. Операционная система Xubuntu Linux;
- Intel Atom 2 ядра по 1,86 ГГц, оперативная память 2ГБ, видеокарта Intel GMA x3100 до 512 МБ. Операционная система Windows 8.1 x64;
- Intel Celeron M420 1,6 ГГц, оперативная память 512 МБ, видеокарта Radeon 64 МБ. Операционная система Windows 7

5.1 Модульное тестирование

Модульное тестирование (unit testing) – это один из наиболее простых и понятных для разработчика способов тестирования. Фактически это тестирование определенных методов какого-то класса программы в изоляции от остальной программы.

Не всякий класс легко покрыть модульными тестами. При проектировании нужно учитывать возможность тестируемости и зависимости класса делать явными.

Часто при разработке применяется методология разработки через тестирование (Test Driven Development). Она подразумевает, что тесты, для методов класса пишутся до того, как будет создана реализация для этих методов. Таким образом разработка приложения представляет собой последовательность коротких циклов, которые состоят из написания модульных тестов и кода, который покрывает данные тесты.

При разработке данного проекта использовалась автоматическая система сборки и управления зависимостями Gradle Build Tool. Этот инструмент был разработан для расширяемых многопроектных сборок, и поддерживает инкрементальные сборки, определяя, какие компоненты дерева сборки не изменились и какие задачи, зависящие от этих частей, не требуют перезапуска. На сегодняшний день Gradle является одной из самых гибких и современных систем сборки для разработки проектов на языке Java. Конфигурация проекта задается при помощи файлов на языке Groovy – скриптовом языке, который работает при помощи JVM. Для написания модульных тестов используется библиотека JUnit – наиболее популярная библиотека для создания модульных тестов в среде JVM. Она предоставляет широкий функционал для тестирования классов и модулей приложения, в том числе:

- большое количество методов проверки (assertion) для определения результатов теста;
- аннотации, позволяющие тестировать создание исключений при работе методов;
- аннотации, позволяющие задавать лимит времени, требуемый для прохождения теста;
- интеграцию со всеми современными средами разработки (IntelliJ IDEA, Eclipse, NetBeans);
- поддержка непрерывного тестирования (Continuous testing).

При написании модульных тестов требуется протестировать работу определенного класса приложения в изоляции от всех остальных классов и модулей. Но у класса обычно имеются зависимости, без которых невозможна корректная работа методов. Для решения данной проблемы используются так называемые мок-фреймворки, предоставляющие функциональность для создания фиктивных зависимостей для тестируемых классов. Для необходимых методов фиктивных классов (тех, которые использует

тестируемый объект) определяется требуемое поведение, они подставляются в качестве зависимостей в тестируемый класс и таким образом мы можем быть уверены, что тестируется только логика работы выбранного нами класса. Наиболее популярным мок-фреймворком в связке с библиотекой JUnit является Mockito. Ниже представлен отрывок из конфигурационного файла Gradle где подключаются модули JUnit и Mockito:

```
testCompile group: 'junit', name: 'junit', version: '4.11'
testCompile group: 'org.mockito', name: 'mockito-all',
version: '2.0.2-beta'
```

В библиотеке JUnit модульные тесты представлены в виде методов тестовых классов, причем обычно структура тестовых классов повторяет структуру тестируемых классов приложения. Рассмотрим один из таких классов: `ConvolutionNetLayerTest`, который тестирует работу полносвязного слоя нейронной сети:

```
@RunWith(MockitoJUnitRunner.class)
public class ConvolutionNetLayerTest {

    @Spy
    private ConvolutionNetLayer layer = new
        ConvolutionNetLayer(
            new Dimension(10, 10, 3),
            new Dimension(3, 3, 3),
            1,
            ActivationFunction.SIGMOID);

    @Test
    public void forwardTest() throws Exception {
        final DataSet dataSet = new DataSet(new
            Dimension(10, 10, 3), () -> 1);
        layer.forward(dataSet);

        final Dimension expected = new Dimension(8, 8, 1);
        Assert.assertEquals(expected,
            dataSet.getDimension());
    }
}
```

Класс `ConvolutionNetLayerTest` помечен аннотацией `@RunWith(MockitoJUnitRunner.class)`. Это сигнализирует о том, что в тесте используется функциональность фреймворка Mockito. Сами тестовые методы должны быть помечены аннотацией `@Test`. Тестовый метод `forwardTest()` производит тестирование метода `forward()` класса `ConvolutionNetLayer`. Для этого происходит вызов данного метода с тестовым аргументом. Так как в данном случае происходит тестирование

только корректности изменения размерностей массива данных после прохождения через слой, то массив данных заполняется единицами.

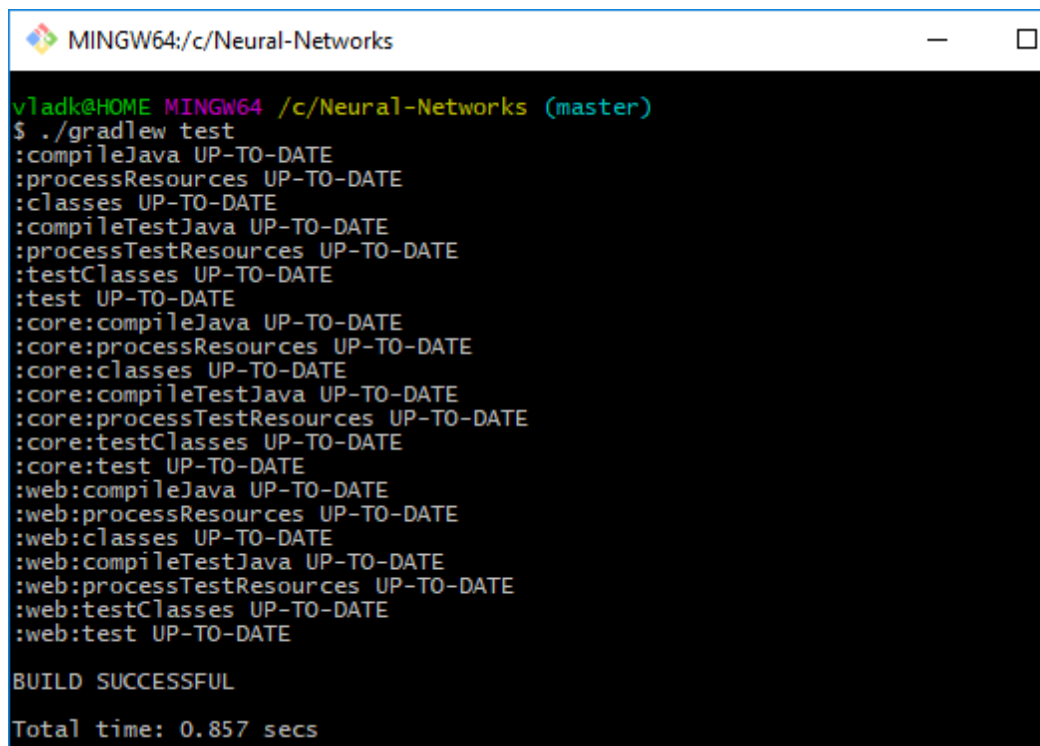
```
final DataSet dataSet = new DataSet(new
    Dimension(10, 10, 3), () -> 1);
layer.forward(dataSet);
```

После этого размерность объекта `dataSet` должна измениться согласно конфигурации данного сверточного слоя. Для проверки этого используется один из статических методов класса `Assert`, предоставляемого библиотекой `JUnit`:

```
Assert.assertEquals(expected, dataSet.getDimension());
```

Если размерность данных после прямого прохода через слой совпала с ожидаемой, тест считается успешно выполненным. Иначе, `JUnit` помечает данный тест как не пройденный.

В системе `Gradle` существует предопределенная задача `test`, предназначенная для запуска модульных и интеграционных тестов. Согласно структуре проекта, `Gradle` тестовые классы располагаются в директории `test/java`. Каждый тестовый класс находится в том же пакете, что и тестируемый класс программного продукта. При запуске задачи `test` будут последовательно запущены все классы, содержащиеся в данной директории. При этом на вывод консоли будут передаваться результаты прохождения тестов. (см. рисунок 5.1)



```
MINGW64:/c/Neural-Networks

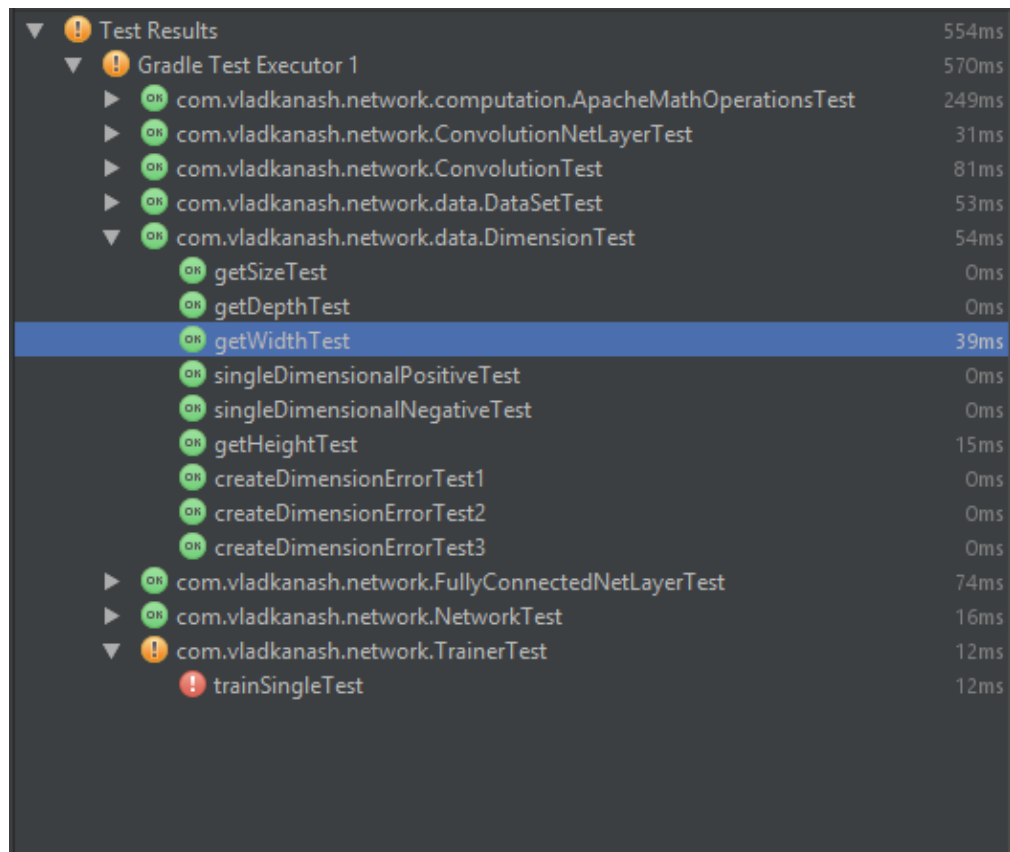
vladk@HOME MINGW64 /c/Neural-Networks (master)
$ ./gradlew test
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:core:compileJava UP-TO-DATE
:core:processResources UP-TO-DATE
:core:classes UP-TO-DATE
:core:compileTestJava UP-TO-DATE
:core:processTestResources UP-TO-DATE
:core:testClasses UP-TO-DATE
:core:test UP-TO-DATE
:web:compileJava UP-TO-DATE
:web:processResources UP-TO-DATE
:web:classes UP-TO-DATE
:web:compileTestJava UP-TO-DATE
:web:processTestResources UP-TO-DATE
:web:testClasses UP-TO-DATE
:web:test UP-TO-DATE

BUILD SUCCESSFUL

Total time: 0.857 secs
```

Рисунок 5.1 – Запуск тестов при помощи консоли

В то же время многие современные средства разработки имеют встроенные инструменты для запуска и управления процессом тестирования. На рисунке 5.1 представлено окно с результатами тестирования в среде разработки IntelliJ IDEA. Как видно, результаты работы тестовых классов отображаются в виде дерева и для каждого класса и метода выводится время работы.



Test Item	Execution Time
Test Results	554ms
Gradle Test Executor 1	570ms
com.vladkanash.network.computation.ApacheMathOperationsTest	249ms
com.vladkanash.network.ConvolutionNetLayerTest	31ms
com.vladkanash.network.ConvolutionTest	81ms
com.vladkanash.network.data.DataSetTest	53ms
com.vladkanash.network.data.DimensionTest	54ms
getSizeTest	0ms
getDepthTest	0ms
getWidthTest	39ms
singleDimensionalPositiveTest	0ms
singleDimensionalNegativeTest	0ms
getHeightTest	15ms
createDimensionErrorTest1	0ms
createDimensionErrorTest2	0ms
createDimensionErrorTest3	0ms
com.vladkanash.network.FullyConnectedNetLayerTest	74ms
com.vladkanash.network.NetworkTest	16ms
com.vladkanash.network.TrainerTest	12ms
trainSingleTest	12ms

Рисунок 5.2 – Окно результатов тестирования в среде IntelliJ IDEA

5.2 Интеграционное тестирование

Главным отличием интеграционного тестирования от модульного является то, что при модульном тестировании происходит проверка взаимодействия нескольких компонентов программного продукта в связке друг с другом, в то время как при модульном тестировании проверяется лишь работа отдельного компонента в изоляции от всей остальной системы. Поэтому при интеграционном тестировании реже используется создание фиктивных классов, так как компоненты системы тестируются с реальными зависимостями. Интеграционное тестирование обычно проводится после того как была проведена разработка всех классов и методов программного продукта (либо отдельного модуля) и для данных классов было проведено модульное тестирование.

Рассмотрим тестовый метод `trainSingleTest()`, который тестирует способность простой сверточной сети обучаться при помощи метода стохастического градиентного спуска. Ниже представлен код данного тестового метода:

```
@Test
public void trainSingleTest() throws Exception {
    final Network network = new Network(
        new Dimension(3, 3));

    network.addLayer(Layer.conv(new Dimension(2, 2),
        1).withSigmoidActivation());

    network.addLayer(Layer.conv(new Dimension(2,2),
        2).withSigmoidActivation());

    final Double[] initialResult = network.forward(new
        Double[] {-4.56, 9.03, -3.0, -2.3,
        5.0, -0.1, -4.9, -2.27, 0.0});

    final Trainer trainer = new SGDTrainer(network);
    for (int i = 0; i < 50; i++) {
        trainer.trainSingle(new DataSet(
            new Double[]{-4.56, 9.03, -3.0, -2.3, 5.0,
            -0.1, -4.9, -2.27, 0.0},
            new Dimension(3, 3)),
            new DataSet(new Double[]{0.3, 0.36},
            new Dimension(1, 1, 2)));
    }

    final Double[] newResult = network.forward(
        new Double[] {-4.56, 9.03, -3.0, -2.3, 5.0,
        -0.1, -4.9, -2.27, 0.0});

    Assert.assertEquals(newResult[0], 0.3, 0.05);
    Assert.assertEquals(newResult[1], 0.36, 0.05);
}
```

В начале работы данного тестового метода происходит инициализация модели сети. Создаваемая конфигурация имеет небольшие размерности, так как наличие большого количество слоев, имеющих большие размерности, сильно увеличит время работы тестового метода:

```
final Network network = new Network(
    new Dimension(3, 3));
network.addLayer(Layer.conv(new Dimension(2, 2),
    1).withSigmoidActivation());
network.addLayer(Layer.conv(new Dimension(2,2),
    2).withSigmoidActivation());
```

После этого на вход сети подается массив данных для того чтобы получить первоначальную реакцию сети (полученную без какого-либо обучения). Этот результат будут использованы для проверки того, что сеть действительно способна обучаться и ее реакция на такой же массив входных данных будет отличаться. Результат прямого прохода по сети сохраняется в переменную `initialResult`:

```
final Double[] initialResult = network.forward(new
    Double[] {-4.56, 9.03, -3.0, -2.3,
        5.0, -0.1, -4.9, -2.27, 0.0});
```

После этого происходит создание объекта класса `Trainer`, с помощью которого будет производиться обучение сети. В качестве параметра конструктору класса передается ссылка на объект `Network`, представляющий модель сети. Затем запускается цикл, в ходе которого происходит обучение сети с помощью единственного массива входных данных. Таким образом сеть приобретает способность реагировать на данный сигнал с заданной реакцией:

```
for (int i = 0; i < 50; i++) {
    trainer.trainSingle(new DataSet(
        new Double[]{-4.56, 9.03, -3.0, -2.3, 5.0,
            -0.1, -4.9, -2.27, 0.0},
        new Dimension(3, 3)),
        new DataSet(new Double[]{0.3, 0.36},
            new Dimension(1, 1, 2)));
}
```

Обучение происходит при помощи вызова метода `trainSingle()`, в качестве параметров которому передаются 2 объекта типа `DataSet` – входные данные для которых следует произвести обучение и эталонный выход сети для этих входных данных.

После завершения процесса обучения на вход сети снова подаются тот же массив данных, который был использован в процессе обучения. Новый результат прямого прохода сохраняется в переменную `newResult`

```
final Double[] newResult = network.forward(
    new Double[] {-4.56, 9.03, -3.0, -2.3, 5.0,
        -0.1, -4.9, -2.27, 0.0});
```

Последним шагом является проверка того, что новый результат работы сети совпадает с эталонным выходом, который был использован при обучении:

```
Assert.assertEquals(newResult[0], 0.3, 0.05);
Assert.assertEquals(newResult[1], 0.36, 0.05);
```

Выход сети представляет собой массив, состоящий из двух чисел, поэтому проверка происходит с помощью двух вызовов метода `assertEquals()`. Так как при данном количестве циклов обучения не гарантируется что выход сети будет полностью совпадать с эталонным выходом, используется метод `assertEquals`, который проверяет сравнение двух числовых значений с заданной погрешностью. Таким образом, данный тестовый метод будет успешно пройден в том случае, если процесс обучения тестовой сверточной нейронной сети прошел успешно и сеть приобрела способность реагировать на входные данные в соответствии с обучающим набором.

Во время работы описанного выше интеграционного теста проводилась проверка взаимодействия между практически всех компонентов системы: классов `DataSet` и `Dimension`, предназначенных для хранения и передачи данных, классов `NetLayer` и `ConvolutionNetLayer`, содержащих логику работы отдельных слоев сверточной нейронной сети, класса `ApacheMathOperations`, предоставляющего методы для математических вычислений, классов `Network` и `Trainer`, которые ответственны за логику работы всей модели сети и процесс обучения с применением алгоритма градиентного спуска.

Наличие интеграционных тестов позволяет быть уверенным в корректности работы всей системы при разных вариантах использования, а также удостовериться в том, что различные модули системы способны взаимодействовать без ошибок.