

## 6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

### 6.1 Требования к аппаратному и программному обеспечению

Для работы приложения персональный компьютер пользователя должен обладать следующими характеристиками и набором библиотек:

- операционная система на базе ядра Linux либо Windows;
- пакет Java Runtime Environment (JRE) не ниже версии 8;
- 2 Гб оперативной памяти;
- жёсткий диск со скоростью 5200 об\мин и выше или твердотельный накопитель;
- процессор с частотой 2.4 ГГц и выше.

### 6.2 Руководство по установке системы

В том случае, если пользователю требуется самостоятельно собрать приложение из исходного кода, сначала необходимо склонировать репозиторий проекта на локальную машину из системы контроля версий git. Для этого на машине необходимо наличие клиента git. Клонирование производится при помощи команды `git clone`, в качестве аргумента к которой передается путь к репозиторию проекта на хостинге GitHub (см. рисунок 6.1).

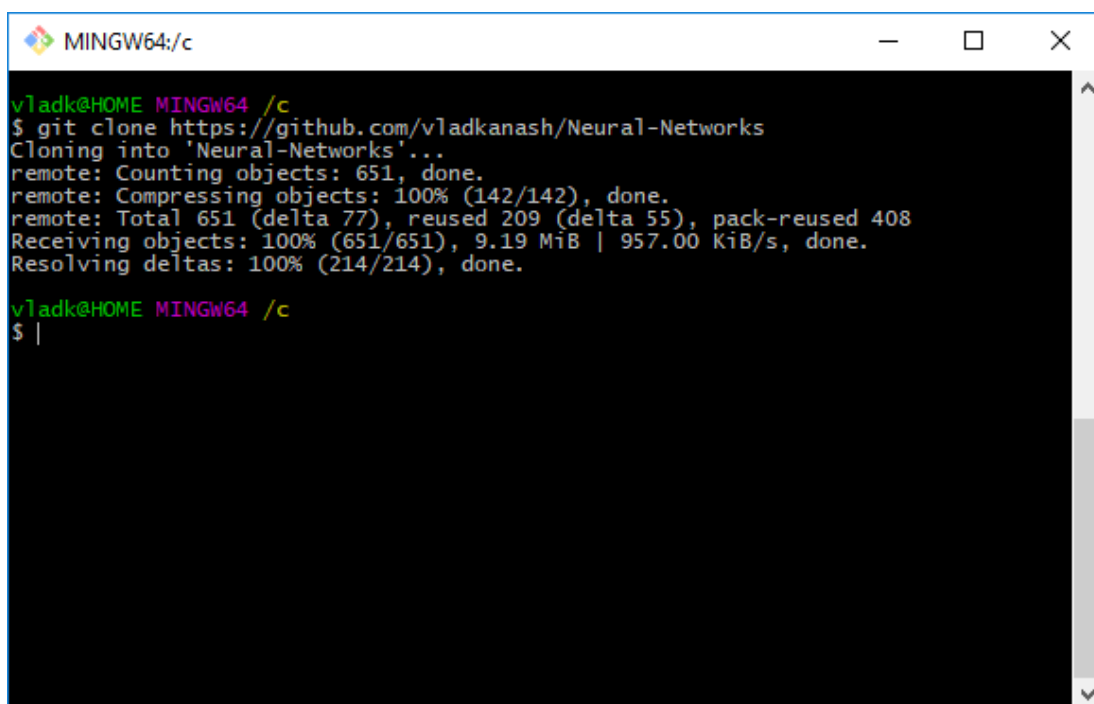
A screenshot of a terminal window titled "MINGW64:/c". The terminal shows a user named "vladk@HOME" in the "MINGW64" shell. The user enters the command `$ git clone https://github.com/vladkanash/Neural-Networks`. The terminal output shows the cloning process: "Cloning into 'Neural-Networks'...", "remote: Counting objects: 651, done.", "remote: Compressing objects: 100% (142/142), done.", "remote: Total 651 (delta 77), reused 209 (delta 55), pack-reused 408", "Receiving objects: 100% (651/651), 9.19 MiB | 957.00 KiB/s, done.", and "Resolving deltas: 100% (214/214), done.". The prompt returns to `$ |`.

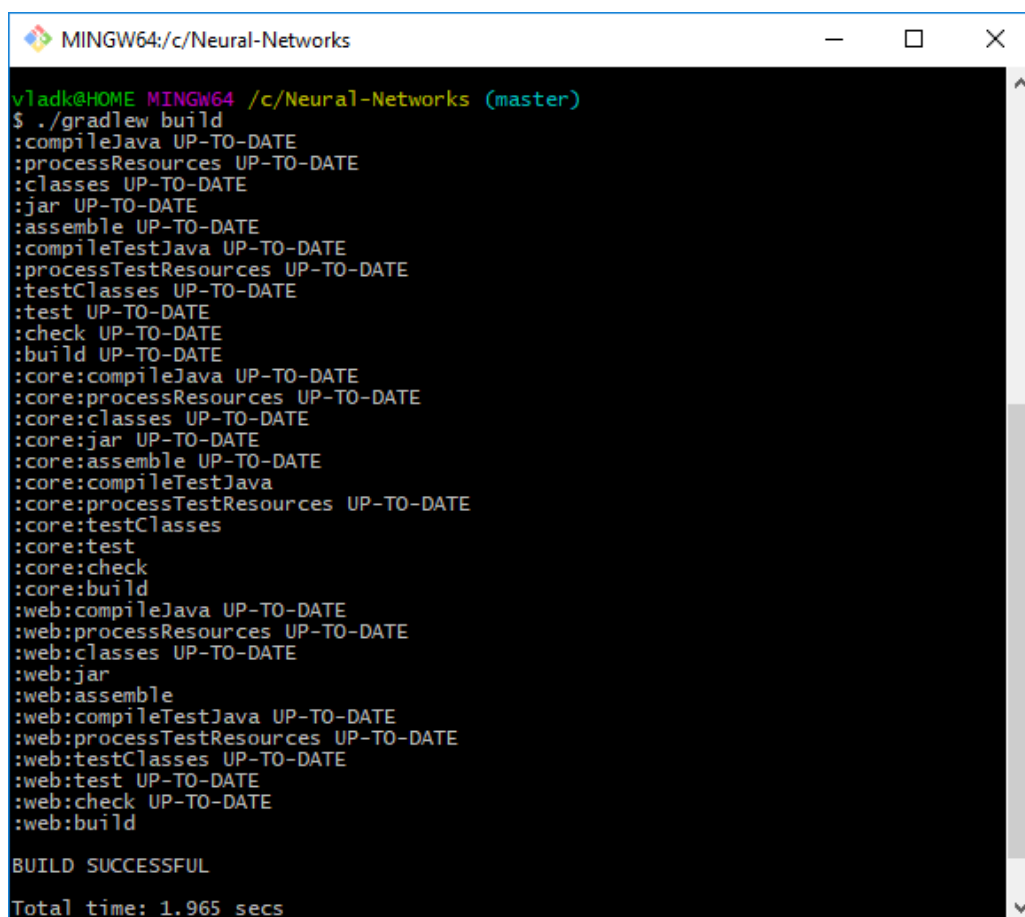
Рисунок 6.1 – Клонирование репозитория с исходным кодом проекта

Сборка программного продукта осуществляется при помощи автоматической системы сборки Gradle Build Tool. Одной из особенностей

данной системы является то, что для сборки проекта не требуется локально установленная система Gradle. В репозитории проекта находится небольшой скрипт (Gradle Wrapper), с помощью которого происходит запуск сборки. Перед началом непосредственной компиляции и сборки проекта скрипт сам загрузит и локально установит Gradle Build Tool. В репозитории данного программного комплекса присутствует две версии скрипта:

- gradlew.bat для систем Windows;
- gradlew.sh для систем Linux.

Сборка запускается при помощи предустановленной задачи build. Во время сборки также производится запуск модульных и интеграционных тестов. Результаты выполнения этапов сборки выводятся в консоль. После успешной сборки (см. рисунок 6.2) в категории build/libs можно найти скомпилированный .jar файл Neural-Networks.SNAPSHOT. После подключения .jar файла в качестве зависимости в проект, программный продукт готов к использованию.



```
MINGW64:/c/Neural-Networks
vladk@HOME MINGW64 /c/Neural-Networks (master)
$ ./gradlew build
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar UP-TO-DATE
:assemble UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build UP-TO-DATE
:core:compileJava UP-TO-DATE
:core:processResources UP-TO-DATE
:core:classes UP-TO-DATE
:core:jar UP-TO-DATE
:core:assemble UP-TO-DATE
:core:compileTestJava UP-TO-DATE
:core:processTestResources UP-TO-DATE
:core:testClasses UP-TO-DATE
:core:test UP-TO-DATE
:core:check UP-TO-DATE
:core:build UP-TO-DATE
:web:compileJava UP-TO-DATE
:web:processResources UP-TO-DATE
:web:classes UP-TO-DATE
:web:jar UP-TO-DATE
:web:assemble UP-TO-DATE
:web:compileTestJava UP-TO-DATE
:web:processTestResources UP-TO-DATE
:web:testClasses UP-TO-DATE
:web:test UP-TO-DATE
:web:check UP-TO-DATE
:web:build UP-TO-DATE
BUILD SUCCESSFUL
Total time: 1.965 secs
```

Рисунок 6.2 – Пример вывода информации о ходе сборки проекта

### 6.3 Руководство по использованию программного средства

Рассмотрим типичный алгоритм работы с программным комплексом. Для использования системы, клиент должен добавить скомпилированный

файл библиотеки `Neural-Networks.jar` в качестве зависимости в разрабатываемый проект. После этого ему становятся доступны классы и методы программного комплекса.

Конфигурирование любой нейронной сети начинается с создания экземпляра класса `Network`, который представляет собой модель сети. В качестве аргумента конструктора данному классу передается объект класса `Dimension`, который представляет собой размерность входных данных для новой сети:

```
Network network = new Network(new Dimension (28, 28, 3));
```

В свою очередь, для создания экземпляра класса `Dimension` в общем случае необходимо передать в конструктор 3 целочисленных параметра, которые отображают три размерности – ширину (`height`), высоту (`width`) и количество каналов (`depth`). Стоит отметить, что существуют и другие конструкторы класса `Dimension`, в которых часть описанных выше параметров опускается. При этом значение не указанных явно полей устанавливается в 1.

После того как был создан объект нейронной сети необходимо описать слои, которые будут входить в данную сеть. Это можно осуществить при помощи вызова метода `addLayer()` класса `Network`. Стоит отметить, что попытка обучения либо получения реакции на входной сигнал сети, для которой не задано ни одного слоя, приведет к созданию `IllegalStateException`.

В качестве аргумента методу `addLayer()` должен передаваться экземпляр класса `Layer`, который представляет собой конфигурацию нейронной сети. Класс `Layer` располагает статическими фабричными методами, с помощью которых можно получить конфигурации для различных типов слоев нейронной сети. Клиенту доступны такие методы, как:

- `Layer.fullyConn()` – позволяет задать конфигурацию полносвязного слоя;
- `Layer.conv()` – позволяет задать конфигурацию сверточного слоя;
- `Layer.softmax()` – позволяет задать конфигурацию Softmax-слоя;
- `Layer.pool()` – позволяет задать конфигурацию слоя пулинга (субдискретизации);
- `Layer.dropout()` – позволяет задать конфигурацию слоя дропаута

В каждый из перечисленных выше методов требуется передать параметры, соответствующие типу данного слоя. Например, для создания сверточного слоя необходимо указать размерность и количество ядер свертки с помощью экземпляра `Dimension` и целого числа. Для полносвязного слоя

требуется указать лишь число нейронов, для слоя субдискретизации – размер блока пулинга и функцию пулинга. Для создания конфигурации Softmax-слоев и слоев дропаута не требуется передача каких-либо параметров:

```
network.addLayer(Layer.conv(
    new Dimension(5, 5, 3), 4).withSigmoidActivation());
network.addLayer(Layer.softmax(
    new Dimension(5, 5));
network.addLayer(Layer.conv(
    new Dimension(5, 5, 3), 8).withReLUActivation());
network.addLayer(Layer.fullyConn(20)
    .withSigmoidActivation());
network.addLayer(Layer.fullyConn(10)
    .withSigmoidActivation());
```

Кроме этого для сверточных и полносвязных слоев можно также указать пороговую функцию. Для этого в классе Layer существуют следующие методы:

- `withSigmoidActivation()` – устанавливает сигмоиду в качестве пороговой функции слоя
- `withReLUActivation()` – устанавливает частично линейную функцию (Rectified Linear Unit) в качестве пороговой функции слоя;
- `withTanhActivation()` – устанавливает гиперболический тангенс в качестве пороговой функции слоя;
- `withIdentityActivation()` – устанавливает тождество в качестве пороговой функции слоя;
- `appendActivationFunction()` – позволяет указать собственную пороговую функцию, которая будет добавлена к существующей функции.

Если пороговая функция не указана явно, то по умолчанию используется функция сигмоиды.

После каждого вызова метода `addLayer()` происходит обработка конфигурации данного слоя. При этом, если введенные пользователем данные не являются корректными (размерность ядра в сверточном слое больше размерности входных данных и т.п.), то создается `IllegalArgumentException`.

Таким образом с помощью последовательных вызовов метода `addLayer()` клиент добавляет к данной нейронной сети новые слои. Стоит отметить что слои добавляются к сети в том порядке, в котором происходит вызов `addLayer()`. При вызове данного метода из нескольких потоков для единственного класса `Network`, поведение программы не определено.

После того как требуемая конфигурация сети была задана, можно переходить к следующему этапу – обучению сети. Для инициализации процесса обучения с помощью метода градиентного спуска необходимо создать экземпляр класса `SGDTrainer`. В качестве аргумента конструктора в класс передается созданный ранее объект `Network`.

Класс `SGDTrainer` содержит методы, позволяющие провести обучение нейронной сети с помощью метода градиентного спуска.

```
final Trainer trainer = new SGDTrainer(network);
```

Предполагается, что клиент обладает каким-либо обучающим набором, предназначенным для обучения данной сети. Перед началом обучения необходимо создать объект класса `DataSet` на основе имеющегося обучающего набора. Класс `DataSet` обладает рядом конструкторов для того чтобы позволить клиентам использовать для обучения имеющиеся у них данные в любом формате:

- `DataSet(final Double[] data, final Dimension dimension)` - создает объект `DataSet` на основе массива из оберточных классов `Double`;

- `public DataSet(final double[] data, final Dimension dimension)` - создает объект `DataSet` на основе массива чисел `double`;

- `public DataSet(final double[][] data, final Dimension dimension)` - создает объект `DataSet` на основе двумерного массива `double`;

- `public DataSet(final Collection<Double> data, final Dimension dimension)` - создает объект `DataSet` на основе коллекции объектов оберточного класса `Double`;

- `public DataSet(final DataSet dataSet)` - создает объект `DataSet` на основе существующего экземпляра.

- `public DataSet(final Dimension dimension, final DoubleSupplier supplier)` - создает объект `DataSet` на основе данных полученных от лямбда-выражения `supplier`.

После того, как данные для обучения были приведены к необходимому формату, можно начинать цикл обучения сети. Для этого необходимо вызвать один из методов обучения класса `SGDTrainer` – `trainSingle()`, `trainBatch()` либо `trainFull()`. Эти методы отличаются в зависимости от того, каким количеством тестовых наборов обладает клиент и как он хочет проводить обучение. Ниже представлен пример обучения сети с использованием метода `trainSingle()`:

```
for (int i = 0; i < trainingExamples.length; i++) {  
    trainer.trainSingle(trainingExamples[i],  
        trainingReferences[i])  
}
```

В данном случае в массиве `trainingExamples` хранятся экземпляры класса `DataSet`, содержащие тестовые входные сигналы для сети, а в массиве `trainingReferences` содержатся эталонные реакции сети для данных

тестовых сигналов, также представленные в виде экземпляров класса `DataSet`. Необходимо иметь в виду что размерность тестовых сигналов должна совпадать с входной размерностью сети (заданной при создании объекта `Network`), а размерность эталонной реакции сети – с выходной размерностью последнего слоя сети. Несоответствие этих параметров приведет к созданию `IllegalArgumentException`.

После того как был успешно пройден этап обучения сети можно начать непосредственное применение нейронной сети. Для этого необходимо преобразовать имеющиеся входные данные в экземпляр класса `DataSet`. После этого этот объект передается в качестве аргумента метода `forward()` класса `Network`:

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

Данный метод пропускает массив данных через все слои сети и возвращает реакцию сети на входные данные в виде объекта `DataSet`.

Несмотря на то, что этап обучения уже был пройден, при наличии новых обучающих данных можно проводить дальнейшее обучение сети при этом не переставая использовать ее в работе.

Рассмотренный выше алгоритм работы с программным комплексом требует лишь базовых знаний в области машинного обучения и при этом позволяет создавать сложные системы глубинного обучения, способные решать широкий спектр задач.