# BigDL: Distributed Deep Learning on Apache Spark*

BigDL is a distributed deep learning library for Apache Spark*. Using BigDL, you can write deep learning applications as Scala or Python* programs and take advantage of the power of scalable Spark clusters. This article introduces BigDL, shows you how to build the library on a variety of platforms, and provides examples of BigDL in action.

## What Is Deep Learning?

*Deep learning* is a branch of machine learning that uses algorithms to model high-level abstractions in data. These methods are based on artificial neural network topologies and can scale with larger data sets.

Figure 1 explores the fundamental difference between traditional and deep learning approaches. Where traditional approaches tend to focus on feature extraction as a single phase for training a machine learning model, deep learning introduces *depth* by creating a pipeline of feature extractors in the model. These multiple phases in a deep learning pipeline increase the overall prediction accuracy of the resulting model through hierarchical feature extraction.

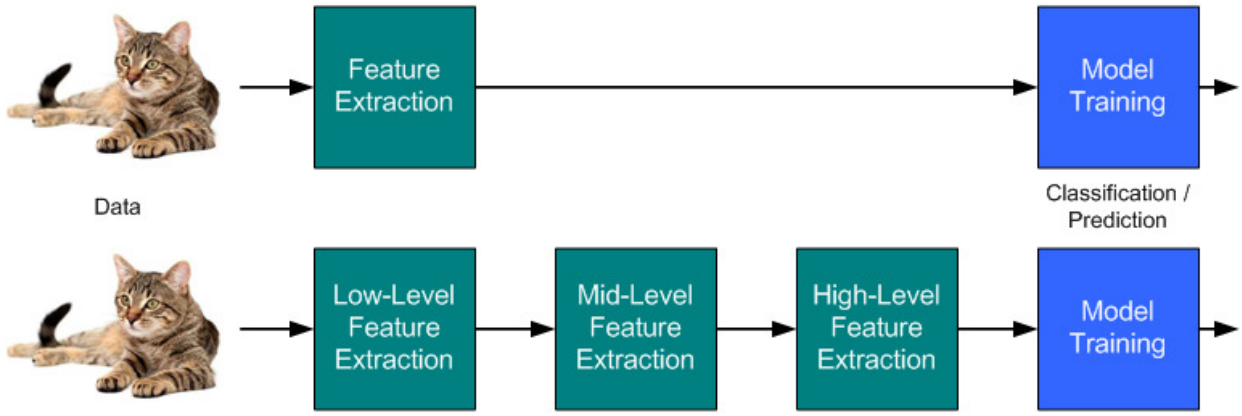



**Figure 1.** Deep learning as a pipeline of feature extractors

## What Is BigDL?

BigDL is a distributed deep learning library for Spark that can run directly on top of existing Spark or Apache Hadoop* clusters. You can write deep learning applications as Scala or Python programs.

- **Rich deep learning support.** Modeled after Torch BigDL provides comprehensive support for deep learning, including numeric computing (via Tensor and high-level neural networks; in addition, you can load pretrained Caffe* or Torch models

into the Spark framework, and then use the BigDL library to run inference applications on their data.

- **Efficient scale out.** BigDL can efficiently scale out to perform data analytics at "big data scale" by using Spark as well as efficient implementations of synchronous stochastic gradient descent (SGD) and all-reduce communications in Spark.
- **Extremely high performance.** To achieve high performance, BigDL uses Intel® Math Kernel Library (Intel® MKL) and multithreaded programming in each Spark task. Consequently, it is orders of magnitude faster than out-of-the-box open source Caffe, Torch, or TensorFlow on a single-node Intel® Xeon® processor (i.e., comparable with mainstream graphics processing units).

## What Is Apache Spark*?

Spark is a lightning-fast distributed data processing framework developed by the University of California, Berkeley, AMPLab. Spark can run in stand-alone mode, or it can run in cluster mode on YARN on top of Hadoop or in Apache Mesos* cluster manager (Figure 2). Spark can process data from a variety of sources, including the HDFS, Apache Cassandra*, or Apache Hive*. Its high performance comes from ability to do in-memory processing via memory-persistent RDDs or DataFrames instead of saving data to hard disks like traditional Hadoop MapReduce architecture.

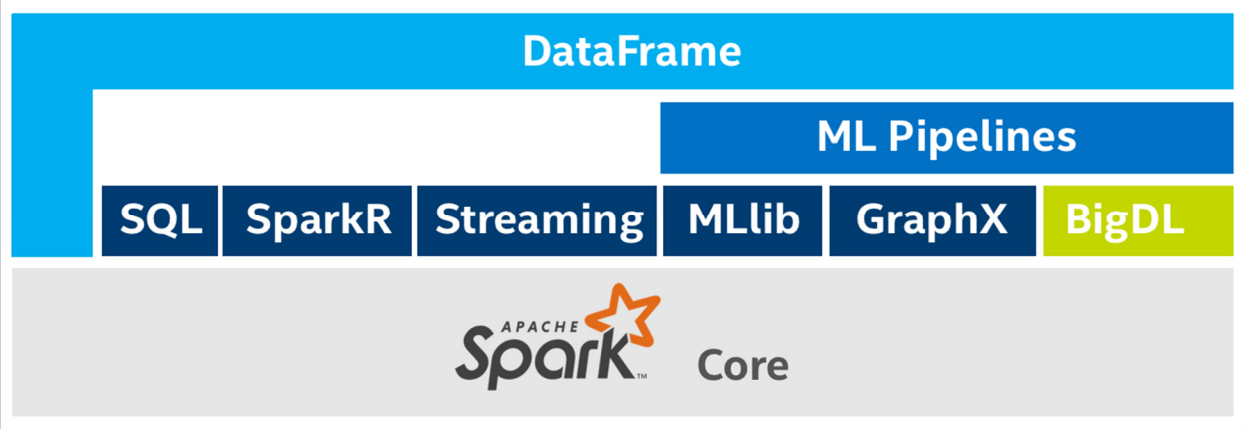



**Figure 2.** BigDL in the Apache Spark* stack

## Why Use BigDL?

You may want to use BigDL to write your deep learning programs if:

- You want to analyze a large amount of data on the same big data Spark cluster on which the data reside (in, say, HDFS, Apache HBase*, or Hive);

- You want to add deep learning functionality (either training or prediction) to your big data (Spark) programs or workflow; or
- You want to use existing Hadoop/Spark clusters to run your deep learning applications, which you can then easily share with other workloads (e.g., extract-transform-load, data warehouse, feature engineering, classical machine learning, graph analytics). An undesirable alternative to using BigDL is to introduce yet another distributed framework alongside Spark just to implement deep learning algorithms.

## Install the Toolchain

**Note:** This article provides instructions for a fresh installation, assuming that you have no Linux* tools installed. If you are not a novice Linux user, feel free to skip rudimentary steps like Git installation).

### *Install the Toolchain on Oracle* VM VirtualBox*

To install the toolchain on Oracle* VM VirtualBox, perform the following steps:

1. Enable virtualization in your computer's BIOS settings (Figure 3).

   The BIOS settings are typically under the **SECURITY** menu. You will need to reboot your machine and go into the BIOS settings menu.

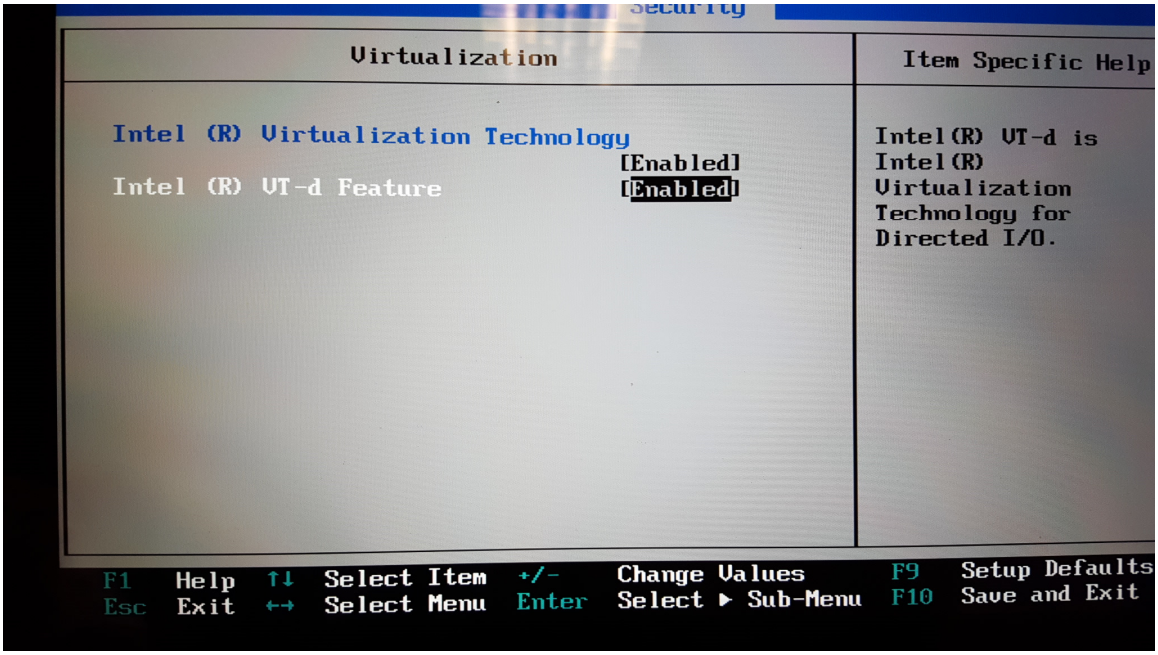



   **Figure 3.** Enable virtualization in your computer's BIOS.

2. Determine whether you have 64-bit or 32-bit machine.

Most modern computers are 64 bit, but just to be sure, look in the Control Panel System and Security applet (Figure 4).

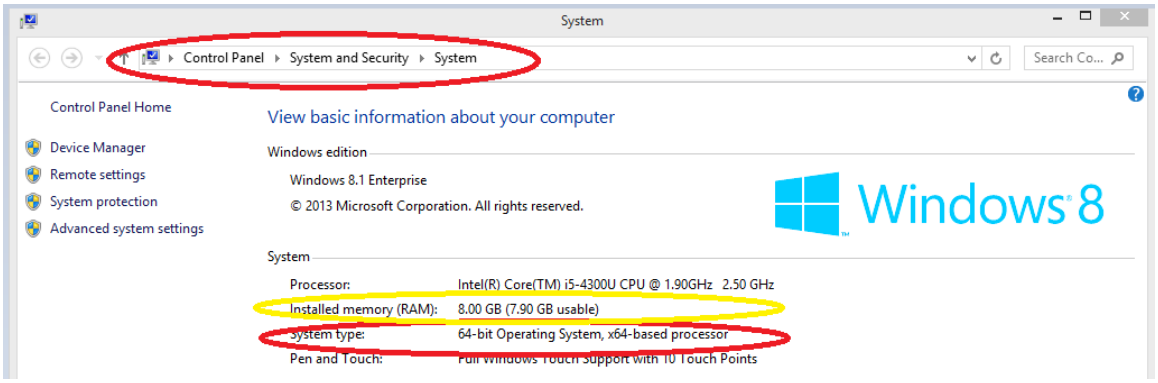
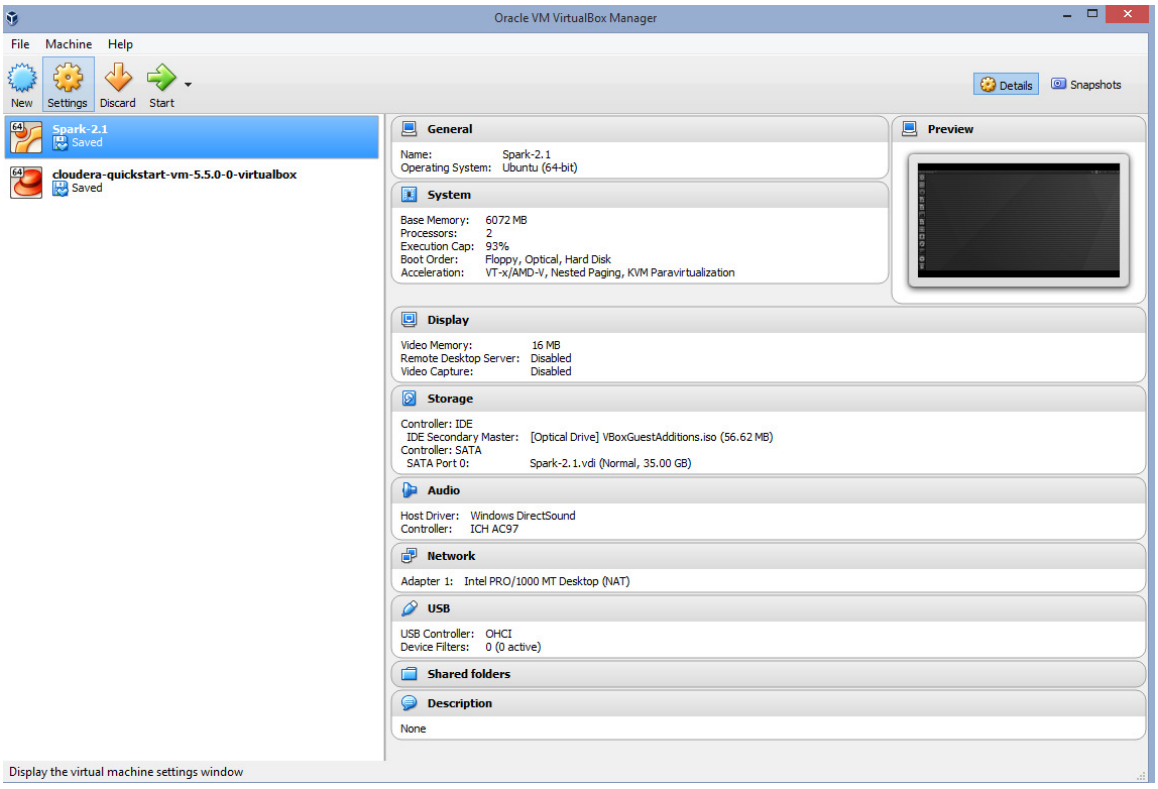



**Figure 4.** Determine whether your computer is 32 bits or 64 bits.

**Note:** You should have more than 4 GB of RAM. Your virtual machine (VM) will run with 2 GB, but most of the BigDL examples (except for LeNet) won't work well (or at all). This article considers 8 GB of RAM the minimum.

3. Install VirtualBox with GuestAdditions or another VM client from the VirtualBox website.

The VM configurations in Figure 5 are known to work with BigDL.



**Important:**

- Allocate 35–50 GB of hard disk space to your VM to be used for Ubuntu*, Spark, BigDL, and all DeepLearning models
- When allocating hard disk space, select "dynamic" allocation, which allows you to expand the partition later (even though it is nontrivial). If you choose "static," and then run out of disk space, your only option will be to wipe out the entire installation and start over.

4. Install Ubuntu Desktop from the Ubuntu download page.
5. When the download is complete, point your VM to the downloaded file so that you can boot from it. (For instructions, go to the Ubuntu FAQ):

```
sudo apt-get update
sudo apt-get upgrade
```

From this point forward, all installations will take place within Ubuntu or your version of Linux; instructions should be identical for the VM or native Linux except when explicitly said otherwise.

If you have trouble accessing the Internet, you may need to set up proxies. (Virtual private networks in general and Cisco AnyConnect in particular are notorious for modifying the VM's proxy settings.) The proxy settings in Figure 6 are known to work for VirtualBox version 5.1.12 and 64-bit Ubuntu.

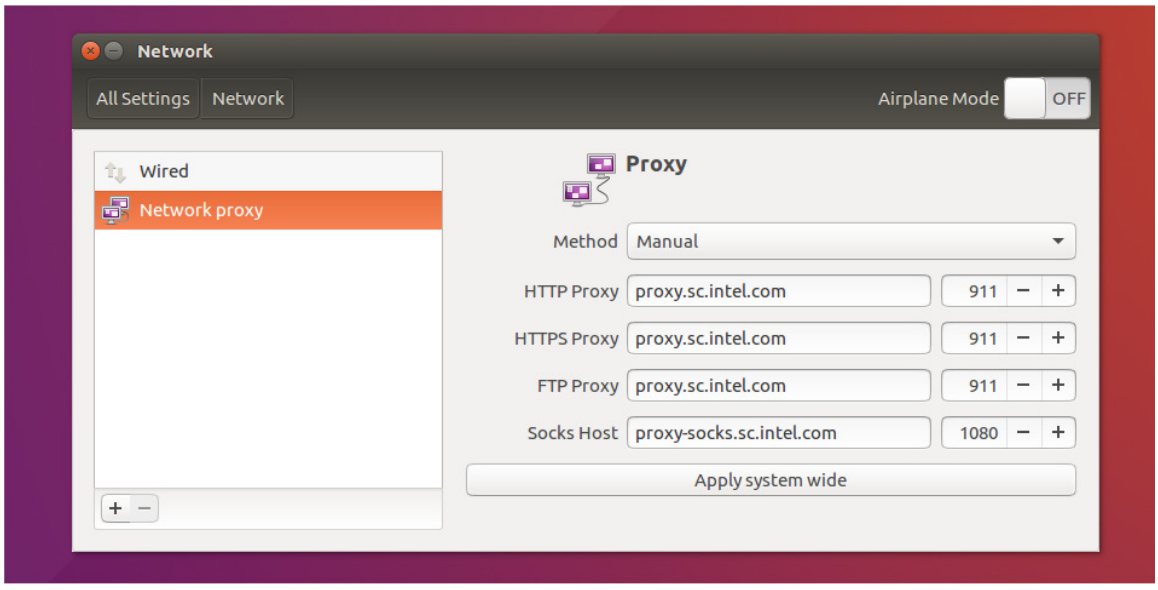



**Figure 6.** Proxy settings for Oracle* VM VirtualBox version 5.1.12 and 64-bit Ubuntu* systems

6. Install Java* with the following commands:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

7. Verify the version of Java:

```
java -version
```

8. Install Scala version 2.11 from the Scala download page.

   When downloading Scala, use the Debian* file format, which downloads into the Downloads folder by default. Open Scala in the file browser, and then click it:

```
sudo apt-get update
```

9. Install Spark from the Spark download page.

   The instructions that follow are for Spark-1.6. If you install a different version, replace `1.6.x` with your flavor of choice:

   a. Replace Spark with the version you downloaded:

```
$ cd Downloads
$ tar -xzf spark-1.6.1-bin-hadoop2.6.tgz
```

   b. Move the files to their proper location:

```
$ sudo mkdir /usr/local/spark
$ sudo mv spark-1.6.1-bin-hadoop2.6 /usr/local/spark
```

   c. Test the installation:

```
$ cd /usr/local/spark
$ cd spark-2.1.0-bin-hadoop2.6
$ ./bin/spark-shell
```

   d. Install Git:

```
$ sudo apt-get install git
```

## *Download BigDL*

BigDL is available from GitHub* using the Git source control tool. Clone the BigDL Git repository as shown:

```
$ git clone https://github.com/intel-analytics/BigDL.git
```

When you're done, you'll see a new subdirectory called `BigDL`, which contains the BigDL repository. You must also install Apache Maven, as shown below, to compile the Scala code:

```
$ sudo apt-get install maven
```

## Build BigDL

This section shows you how to download and build BigDL on your Linux distribution. The prerequisites for building BigDL are:

- Java 8 (for best performance);
- Scala 2.10 or 2.11 (you need Scala 2.11 if you plan to use Spark 2.0 or later);
- Maven 3; and
- Git.

### *Apache Maven Environment*

You need Maven 3 to build BigDL. You can download it from the Maven website.

After installing Maven 3, set the environment variable `MAVEN_OPTS` as follows:

```
$ export MAVEN_OPTS="-Xmx2g -XX:ReservedCodeCacheSize=512m"
```

When compiling with Java 7, you must add the option `-XX:MaxPermSize=1G`.

### *Building*

It is highly recommended that you build BigDL by using the `make-dist.sh` script.

When the download is complete, build BigDL with the following commands:

```
$ bash make-dist.sh
```

After that, you can find a `dist` folder that contains all the files you need to run a BigDL program. The files in `dist` include:

- **dist/bin/bigdl.sh.** Use this script to set up proper environment variables and launch the BigDL program.
- **dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies.jar.** This Java archive (JAR) package contains all dependencies except Spark.
- **dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies-and-spark.jar.** This JAR package contains all dependencies, including Spark.

### *Alternative Approaches*

If you're building for Spark 2.0, you should amend the `bash` call for Scala 2.11. To build for Spark 2.0, which uses Scala 2.11 by default, pass `-P spark_2.0` to the `make-dist.sh` script:

```
$ bash make-dist.sh -P spark_2.0
```

It is strongly recommended that you use Java 8 when running with Spark 2.0. Otherwise, you may observe poor performance.

By default, `make-dist.sh` uses Scala 2.10 for Spark version 1.5.x or 1.6.x and Scala 2.11 for Spark 2.0. To override the default behaviors, pass `-P scala_2.10` or `-P scala_2.11` to `make-dist.sh`, as appropriate.

## Getting Started

Prior to running a BigDL program, you must do a bit of setup, first. This section identifies the steps for getting started.

### *Set Environment Variables*

To achieve high performance, BigDL uses Intel MKL and multithreaded programming. Therefore, you must first set the environment variables by running the provided script in `PATH_To_BigDL/scripts/bigdl.sh` as follows:

```
$ source PATH_To_BigDL/scripts/bigdl.sh
```

Alternatively, you can use `PATH_To_BigDL/scripts/bigdl.sh` to launch your BigDL program.

### *Working with the Interactive Scala Shell*

You can experiment with BigDL codes by using the interactive Scala shell. To do so, run:

```
$ scala -cp bigdl-0.1.0-SNAPSHOT-jar-with-dependencies-and-spark.jar
```

Then, you can see something like:

```
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_91).
Type in expressions for evaluation. Or try :help.


scala>
```

For instance, to experiment with the Tensor application programming interfaces (APIs) in BigDL, you can then try:

```
scala> import com.intel.analytics.bigdl.tensor.Tensor
import com.intel.analytics.bigdl.tensor.Tensor


scala> Tensor[Double](2,2).fill(1.0)
res9: com.intel.analytics.bigdl.tensor.Tensor[Double] =
1.0     1.0
1.0     1.0
[com.intel.analytics.bigdl.tensor.DenseTensor of size 2x2]
```

For more details about the BigDL APIs, refer to the BigDL Programming Guide.

## *Running a Local Java* Program (Single Mode)*

You can run a BigDL program—for example, the VGG model training —as a local Java program that runs on a single node (machine). To do so, perform the following steps:

1. Download the CIFAR-10 data from the CIFAR-10 Dataset page.

   Remember to choose the binary version.

2. Use the `bigdl.sh` script to launch the example as a local Java program:

```
./dist/bin/bigdl.sh -- \
java -cp dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies-and-spark.jar \
com.intel.analytics.bigdl.models.vgg.Train \
--core core_number \
--node 1 \
--env local \
-f cifar10_folder \
-b batch_size
```

   This command uses the following parameters:

   - `--core`. The number of *physical* cores the machine used in the training
   - `--node`. The number of nodes (machines) used in the training (As the example runs as a local Java program in this case, it is set to `1`.)
   - `--env`. Either `"local"` or `"spark"` (In this case, it is set to `"local"`.)
   - `-f`. The folder in which you put the CIFAR-10 data set
   - `-b`. The mini-batch size (The program expects that the mini-batch size is a multiple of `node_number` × `core_number`. In this example, `node_number` is `1`, and you should set the mini-batch size to `core_number` × 4.)

## *Running a Spark Program*

You can run a BigDL program—for example, the VGG model training — as a standard Spark program (running in either local mode or cluster mode). To do so, perform the following steps:

1. Download the CIFAR-10 data from the CIFAR-10 Dataset page.

   Remember to choose the binary version.

2. Use the `bigdl.sh` script to launch the example as a Spark program:

```
./dist/bin/bigdl.sh -- \
spark-submit --class com.intel.analytics.bigdl.models.vgg.Train \
dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies.jar \
--core core_number_per_node \
--node node_number \
```

```
--env spark \
-f cifar10_folder/ \
-b batch_size
```

This command uses the following parameters:

- `--core`. The number of physical cores used in each executor (or container) of the Spark cluster
- `--node`. The executor (container) number of the Spark cluster (When running in Spark local mode, set the number to `1`.)
- `--env`. Either `"local"` or `"spark"` (In this case, it is set to `"spark"`.)
- `-f`. The folder in which you put the CIFAR-10 data set (Note that in this example, this is just a local file folder on the Spark drive. As the CIFAR-10 data is somewhat small [about 120 MB], you send it directly from the driver to executors in the example.)
- `-b`. The mini-batch size (It is expected that the mini-batch size is a multiple of `node_number` × `core_number_per_node`. In this example, you should set the mini-batch size to `node_number` × `core_number_per_node` × `4`.)

## *Notes on Mac OS installation*

In general, installation on Mac OS is simpler than within a windows VM since Mac OS is linux-based. However, command syntax is obviously different. Here are some notes:

1. Install Java

```
Got to java.com and install mac OS version of the JDK. Follow installation
prompts. You need to install JDK (Java development kit) for MacOS.
Alternatively, you can open terminal prompt and type
```

$java –version.

This will also prompt you to install JDK (click on “more info” rather than ‘OK’ button).

2. Verify the version of Java:

```
 java –version
```

Expected output:

```
DN0a181774:scripts svermoli$ java --version
Unrecognized option: --version
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
DN0a181774:scripts svermoli$ java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
DN0a181774:scripts svermoli$
```

Install Scala per instructions here http://biercoff.com/how-to-install-scala-on-mac-os/

Install Spark the same way as described above in installation steps for a VM

Install Homebrew: https://brew.sh

Install Git:

```
        $ brew install git

Do a git pull as directed above.

Maven may have been installed already as part of one of the previous steps. To check:
        $ mvn –v.
Expected output:
```

```
Sergeys-MacBook-Pro:BigDL svermoli$ mvn -v
Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426; 2017-04-03T12:39:06-07:00)
Maven home: /usr/local/Cellar/maven/3.5.0/libexec
Java version: 1.8.0_131, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.12.3", arch: "x86_64", family: "mac"
Sergeys-MacBook-Pro:BigDL svermoli$
```

If maven is not installed, install it via the following command

$brew install maven.

## *Debugging Common Issues*

Currently, BigDL uses synchronous mini-batch SGD in model training, and it will launch a single Spark task (running in multithreaded mode) on each executor (or container) when processing a mini-batch:

- Set `Engine.nodeNumber` to the number of executors in the Spark cluster.

- Ensure that each Spark executor has the same number of cores (`Engine.coreNumber`).
- The mini-batch size must be a multiple of `nodeNumber` × `coreNumber`.

**Note:** You may observe poor performance when running BigDL for Spark 2.0 with Java 7, so use Java 8 when you build and run BigDL for Spark 2.0.

In Spark 2.0, use the default Java serializer instead of Kryo because of Kryo Issues 341: "Stackoverflow error due to parentScope of Generics being pushed as the same object" (see the Kryo Issues page for more information). The issue has been fixed in Kryo 4.0, but Spark 2.0 uses Kryo 3.0.3. Spark versions 1.5 and 1.6 do not have this problem.

On CentOS* versions 6 and 7, increase the max user processes to a larger value, such as, 514585. Otherwise, you may see errors like "unable to create new native thread."

Currently, BigDL loads all the training and validation data into memory during training. You may encounter errors if BigDL runs out of memory.

## BigDL Examples

**Note:** if you're using a VM, your performance won't be as good as it would be in a native operating system installation, mostly due to the overhead of the VM and limited resources available to it. This is expected and is in no way reflective of genuine Spark performance.

### *Training LeNet on MNIST*

The "hello world" example for deep learning is training LeNet (a convolutional neural network—on the MNIST database).

Figure 7 shows the clone LeNet example.

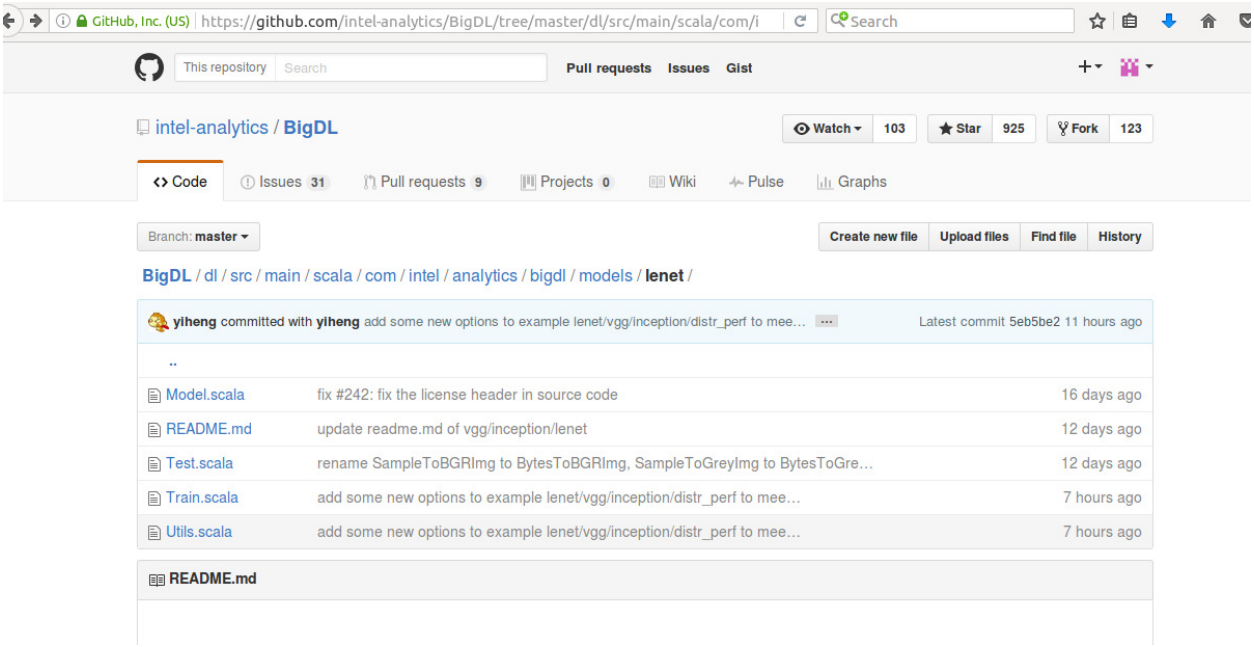



**Figure 7.** The clone LeNet example

A BigDL program starts with an import of `com.intel.analytics.bigdl._`. Then, it initializes the engine, including the number of executor nodes, the number of physical cores on each executor, and whether it runs on Spark or as a local Java program:

```
  val sc = Engine.init(param.nodeNumber, param.coreNumber, param.env ==
"spark").map(conf => {
    conf.setAppName("Train Lenet on MNIST")
      .set("spark.akka.frameSize", 64.toString)
      .set("spark.task.maxFailures", "1")
    new SparkContext(conf)
  })
```

If the program runs on Spark, `Engine.init()` returns a `SparkConf` with proper configurations populated, which you can then use to create the `SparkContext`. Otherwise, the program runs as a local Java program, and `Engine.init()` returns `None`.

After the initialization, begin creating the LeNet model by calling `LeNet5()`, which creates the LeNet-5 convolutional network model as follows:

```
  val model = Sequential()
  model.add(Reshape(Array(1, 28, 28)))
    .add(SpatialConvolution(1, 6, 5, 5))
    .add(Tanh())
    .add(SpatialMaxPooling(2, 2, 2, 2))
    .add(Tanh())
    .add(SpatialConvolution(6, 12, 5, 5))
```

```
    .add(SpatialMaxPooling(2, 2, 2, 2))
    .add(Reshape(Array(12 * 4 * 4)))
    .add(Linear(12 * 4 * 4, 100))
    .add(Tanh())
    .add(Linear(100, classNum))
    .add(LogSoftMax())
```

Next, load the data by creating the `DataSet.scala` commands (either a distributed or local one, depending on whether it runs on Spark—and then applying a series of `Transformer.scala` commands (for example, `SampleToGreyImg`, `GreyImgNormalizer`, and `GreyImgToBatch`):

```
  val trainSet = (if (sc.isDefined) {
      DataSet.array(load(trainData, trainLabel), sc.get, param.nodeNumber)
    } else {
      DataSet.array(load(trainData, trainLabel))
    })
     -> SampleToGreyImg(28, 28)
     -> GreyImgNormalizer(trainMean, trainStd)
     -> GreyImgToBatch(param.batchSize)
```

After that, you create the Optimizer — either a distributed or local one, depending on whether it runs on Spark) by specifying the data set, the model, and the criterion, which, given input and target, computes gradient per given loss function:

```
  val optimizer = Optimizer(
    model = model,
    dataset = trainSet,
    criterion = ClassNLLCriterion[Float]())
```

Finally, after optionally specifying the validation data and methods for the Optimizer, you train the model by calling `Optimizer.optimize()`:

```
  optimizer
    .setValidation(
      trigger = Trigger.everyEpoch,
      dataset = validationSet,
      vMethods = Array(new Top1Accuracy))
    .setState(state)
    .setEndWhen(Trigger.maxEpoch(param.maxEpoch))
    .optimize()
```

The following command executes the LeNet example from the command line:

```
$ dist/bin/bigdl.sh -- \
  java -cp dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies-and-spark.jar \
  com.intel.analytics.bigdl.models.lenet.Train \
  -f ~/data/mnist/ \
  --core 1 \
  --node 1 \
  --env local \
  --checkpoint ~/model/model_lenet \
  -b 50
```

The following is an example command of LeNet Spark Local mode on a VM:

**Note:** Before you run Spark local mode, export `SPARK_HOME=your_spark_install_dir` and `PATH=$PATH:$SPARK_HOME/bin`:

```
$ ./dist/bin/bigdl.sh \
  -- spark-submit \
  --master local[1] \
  --driver-class-path dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies.jar \
  --class com.intel.analytics.bigdl.models.lenet.Train \
    dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies.jar \
  -f ~/data/mnist/ \
  --core 1 \
  --node 1 \
  --env spark \
  --checkpoint ~/model/model_lenet_spark
```

## *Image Classification*

**Note:** For an example of running an Image inference on pre-trained model, see the BigDL `README.md`.

Download the images into the directory `ILSVRC2012_img_val.tar`. This is the image `.tar` file for the pre-trained model. Put it in `~/data/imagenet`, for example, and untar it. Next, download the model `resnet-18.t7` and put it in `~/model/resnet-18.t7`, for example.

Run the following command from a command line (this example is for a VM, so memory sizes are set to `1g`):

```
$ ./dist/bin/bigdl.sh --spark-submit --master local[1] \
```

```
  --driver-memory 1g \
  --executor-memory 1g \
  --driver-class-path dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies.jar \
  --class com.intel.analytics.bigdl.example.imageclassification.ImagePredictor \
  dist/lib/bigdl-0.1.0-SNAPSHOT-jar-with-dependencies-and-spark.jar \
  --modelPath ~/model/resnet-18.t7/resnet-18.t7 \
  --folder ~/data/imagenet/predict_100 \
  --modelType torch -c 1 -n 1 \
  --batchSize 4 \
  --isHdfs false
```

The output of the program is a table with two columns: the `*.JPEG` file name and a numeric label. To check the accuracy of prediction, refer to the `imagenet1000_clsid.txt` label file. (Note that the file labels are 0-based, while the output of the example is 1-based.)

The first few lines of the image file are as follows:

```
{0: 'tench, Tinca tinca',
 1: 'goldfish, Carassius auratus',
 2: 'great white shark, white shark, man-eater, man-eating shark, Carcharodon
carcharias',
 3: 'tiger shark, Galeocerdo cuvieri',
 4: 'hammerhead, hammerhead shark',
 ...
```

The following are the first few lines of the example output:

```
[ILSVRC2012_val_00025162.JPEG,360]
[ILSVRC2012_val_00025102.JPEG,958]
[ILSVRC2012_val_00025113.JPEG,853]
[ILSVRC2012_val_00025153.JPEG,867]
[ILSVRC2012_val_00025132.JPEG,229]
[ILSVRC2012_val_00025133.JPEG,5]
```

The image `ILSVRC2012_val_00025133.JPEG` (Figure 8) has been correctly identified as label `5`, which corresponds to the entry "4: 'hammerhead, hammerhead shark'" in the `imagenet1000_clsid.txt` file.

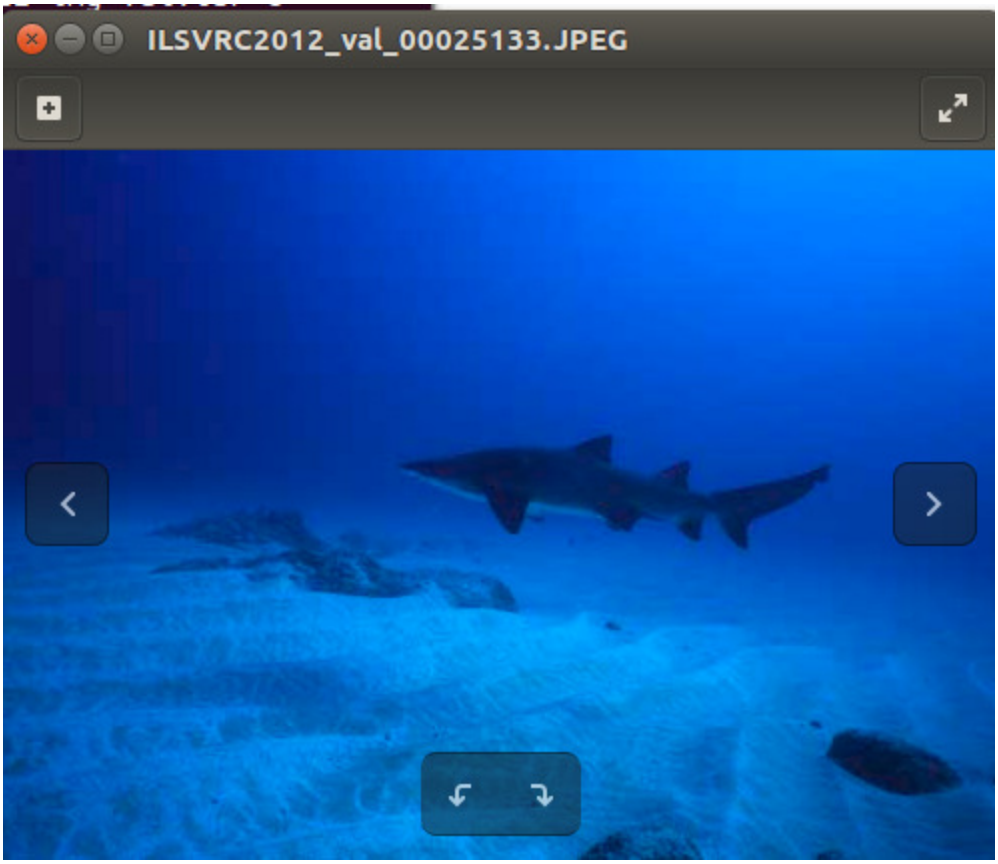



**Figure 8.** The image file `ILSVRC2012_val_00025133.JPEG`

To run the classification example from within the IntelliJ IDE, you must set arguments slightly differently when running in VM, as Figure 9 shows.

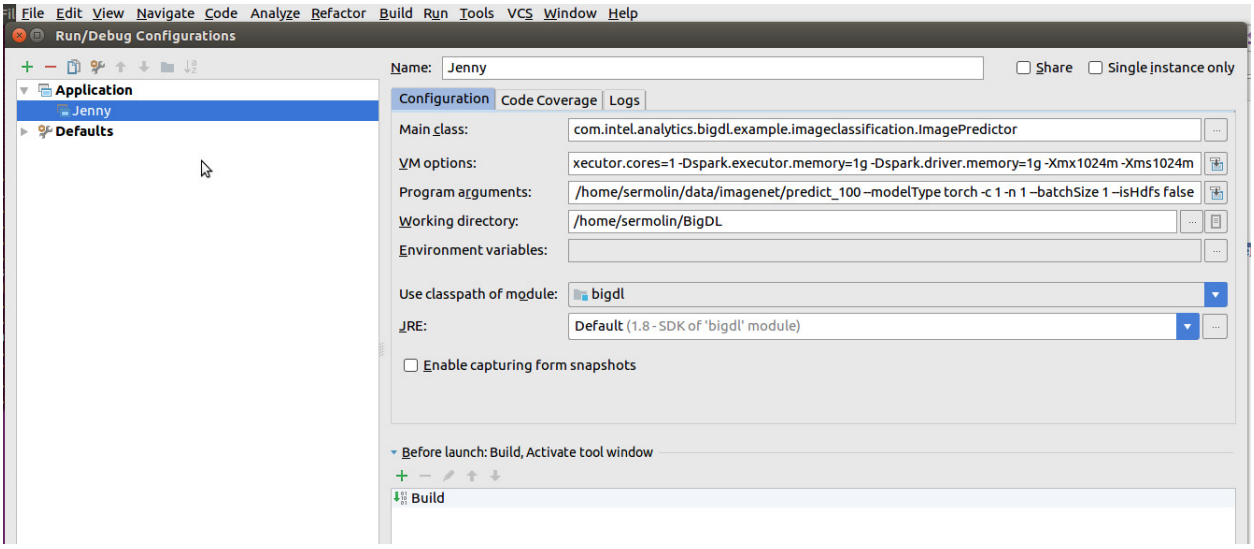



**Figure 9.** The classification example in the IntelliJ integrated development environment

Because the IntelliJ IDE in Figure 9 does not display the entire lines, here are the VM options:

```
-Dspark.master=local[1]
-Dspark.executor.cores=1
-Dspark.total.executor.cores=1
-Dspark.executor.memory=1g
-Dspark.driver.memory=1g
-Xmx1024m -Xms1024m
```

And here are the program arguments:

```
ImageClassifier Program arguments:
  --modelPath ~/model/resnet-18.t7/resnet-18.t7
  --folder ~/data/imagenet/predict_100
  --modelType torch
  -c 1
  -n 1
  --batchSize 1
  --isHdfs false
```

## Breadth and Depth of Neural Networks Support

BigDL currently supports the following-preconfigured neural network models:

- Autoencoder
- Inception
- LeNet
- Resnet
- Rnn
- VGG

For more information, see the BigDL Models page.

In addition, BigDL supports more than 100 standard neural network "building blocks," allowing you to configure your own topology (Figure 10).

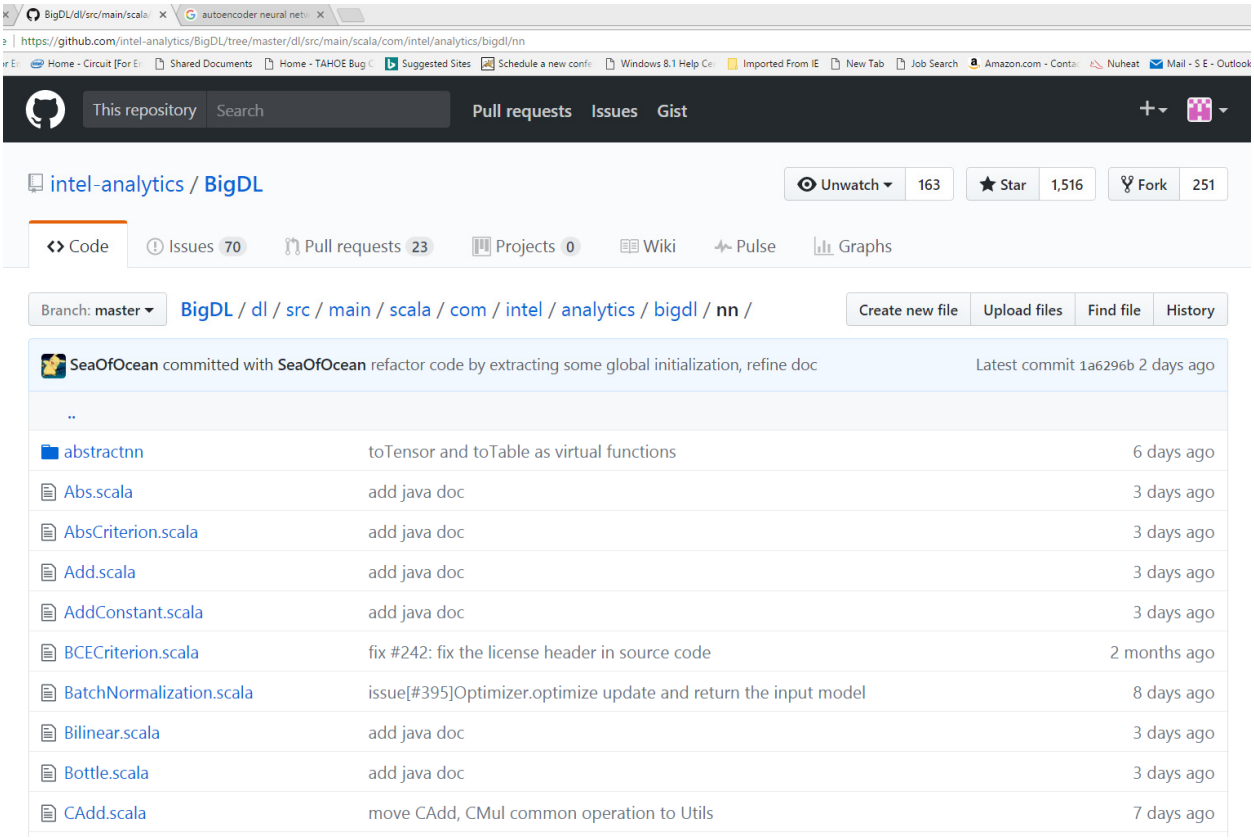



**Figure 10.** BigDL standard neural network building blocks

For more information, see the BigDL Neural Networks page.

In addition, BigDL comes with out-of-the-box examples of end-to-end implementations for LeNet, ImageNet, and TextClassification (Figure 11).

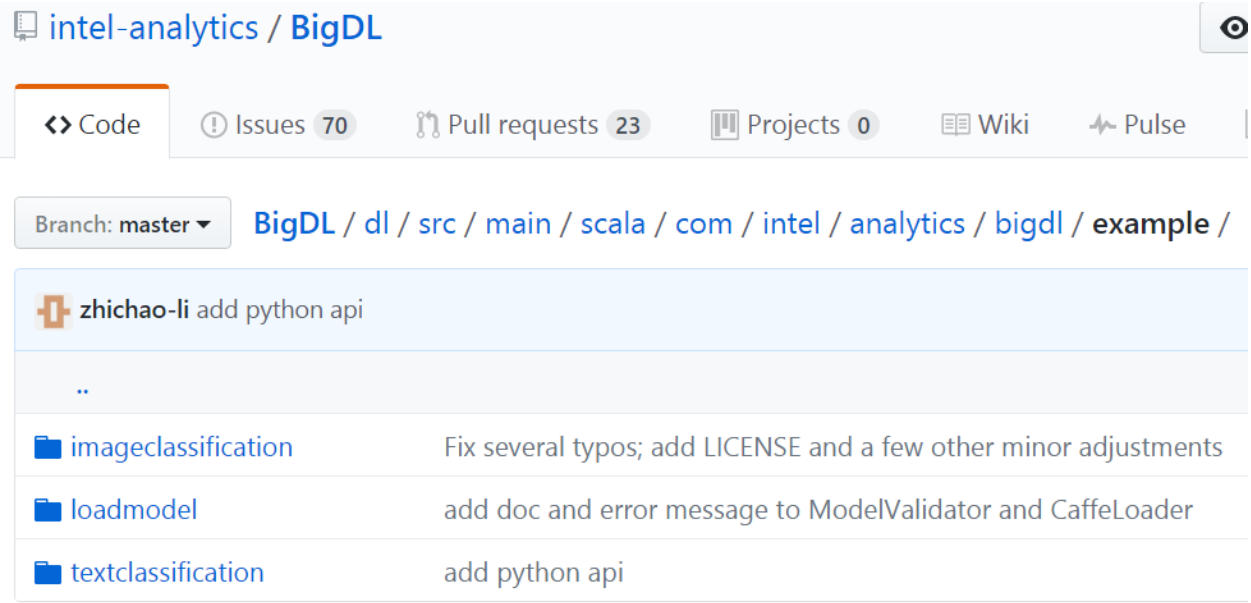



**Figure 11.** BigDL examples

For more information, see the [BigDL Examples page].

## Abbreviations

| | |
|---|---|
| HDFS | Hadoop Distributed File System |
| IDE | Integrated Development Environment |
| MKL | Math Kernel Library |
| RDD | Resilient Distributed Dataset |
| RNN | Recurrent Neural Networks |
| SGD | Stochastic Gradient Descent |
| VM | Virtual Machine |
| YARN | Yet Another Resource Negotiator |