Implement a Minimal Viable Data Product Using Deeplearning4J

This article demonstrates data flows and interactions between components used to implement automatic image labeling in a data-product. Our example use-case is automatic classification of images using neural networks.

After reading this article you will be able:
- to use DL4J in a Spark session / CDSW
- persist labeled training data in Kudu (in order to slice and dice the training set)
- execute a learned model in a Spark-shell session
- execute a learned model in a Spark-streaming job

This means: we go from learning to production, not perfectly robust, but end-2-end!

Content

Scope

In order to solve the image classification problem, we have to act in multiple roles. Data engineering is needed in order to prepare data and a robust data pipeline. The input for our machine learning framework needs to be normalized and learning parameters have to chosen carefully – eventually automatically. Finally, after the data scientist job is done, the model has to be executed and maintained in a production environment - this is a great time for Dev-Ops people.

We have seen that, in many cases only one single person plays all the three roles. But in a real world scenario in enterprise environments this isn't a standard. People normally act in different contexts and multiple collaboration models for co-exist. In this article you will learn, how to implement a minimal viable data product using Apache Spark and Deeplearning4J on a CDH cluster.

Figure 1 illustrates our desired data flow which allows us to learn the unknown inherent structure of a labeled image dataset.

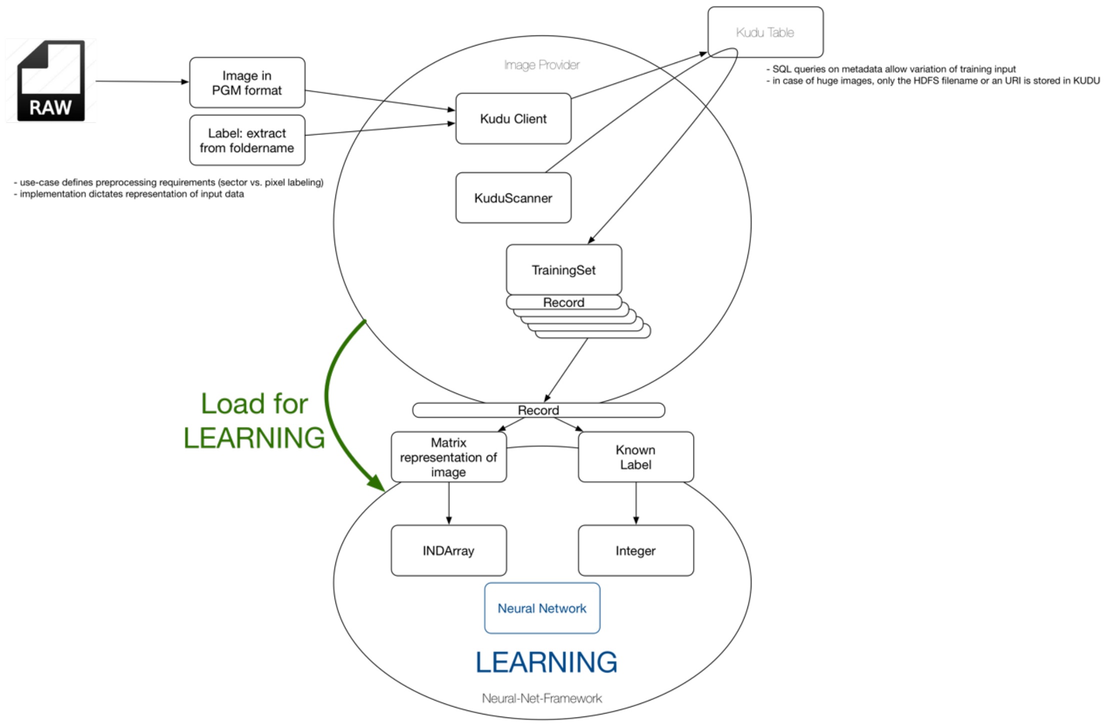
RAW
Image in PGM format
Label: extract from foldername
- use-case defines preprocessing requirements (sector vs. pixel labeling)
- implementation dictates representation of input data
Image Provider
Kudu Table
- SQL queries on metadata allow variation of training input
- in case of huge images, only the HDFS filename or an URI is stored in KUDU
Kudu Client
KuduScanner
TrainingSet
Record
Load for LEARNING
Record
Matrix representation of image
Known Label
INDArray
Integer
Neural Network
LEARNING
Neural-Net-Framework

Figure 1: Overview of data flows and data type conversion during automatic image classification with Deeplearning4J.

Parameter-variation and model validation lead to multiple models represented by multiple neural networks. Furthermore, multiple network architectures can be trained and compared with each other.

In a robust data-product it is important to identify an appropriate model type and reasonable learning parameters which provide training in a reasonable amount of time. Model robustness has to be evaluated over time using new – so far un seen – input. And finally, continuous learning offers a reference model which can be compared with the current production model. This kind of stability analysis provides information about or even triggers model update requirements.

Persisting model metadata and metadata of raw images are key factors in the process of build a meaningful model in a cost efficient way. We use Apache Kudu as storage layer for this purpose. Running a learned model in production requires a direct embedding of the ML toolkit into your data pipelines. Using a Java based DL framework allows a direct integration inside Apache Flume and Spark Streaming jobs, as long as the trained model hasn't to be parallelized over multiple nodes.

Model deployment has to be managed without any interruption to existing data flows. Since Apache Flume reloads the configuration file automatically regularly, we can simply offer the name or ID of the model which has to be loaded in the Flume configuration via Flume's standard configuration mechanism. A more enterprise like approach could be based on a Flume source which receives a signal via Zookeeper as soon as the model needs to be changed. Reloading of the model from Kudu or HDFS is done fully transparently then.

Let’s go and implement the data product. This is our TODO-list:

1. Ingest and convert raw images
2. Train a model from labeled images
3. Query for a specific training set
4. Variation of model parameters
5. Evaluation of model quality
6. Predict the class of unknown images

1. Ingest and convert raw images

For our first step we use the simple Java program `Step1.java` and convert a batch of PNG files[1] into PGM format[2]. The image encoding is encapsulated in a helper-class named `ImageConverte.java`. The current approach uses the `convert` command provided by ImageMagic. We generate a shell script and execute that using the `ProcessBuilder` class. Other options for image conversion are discussed in this article on StackOverflow[3].

The Flume based solution uses an `ImageInterceptor`. You can find a Flume configuration stub file in the `cfg` folder of the project. Finally, the ingestion into Hadoop is accomplished by writing the transformed data entity - in our case a PGM version of the initial image into HDFS.

Since such images are pretty small, one should consider an Avro file for image aggregation. HBase and Kudu tables can be considered as convenient approach as long as the image size less then 2MB (for HBase) or 64kB (for Kudu) respectively. The benefit of both technologies is the random access pattern both provide. Solr should only be considered to collect metadata for later data exploration. The raw image should always be references from a Solr document rather than being persisted in the index.

In order to initialize the Kudu image store you have to run the program `Step2.java` once.

2. Train a model from labeled images

Before we start learning multiple models, we prepare a model store using the Java program `Step3.java` which initializes the `KuduModelStore.`

There are two different implementations in our project. First, the `ImageClassifierHDD`, which simply persists the model files in a local folder on the workstation where the code is executes. We use this simply during development to test our network. You can run the Java program `Step4` in order to create a simple model for character recognition.

The input data is provided by a `DataSetIterator` component. Deeplearning4J provides the `MnistDataSetIterator` out of the box. It allows accessing the binary MNIST database files.

[1] PNG: https://en.wikipedia.org/wiki/Portable_Network_Graphics
[2] PGM: https://en.wikipedia.org/wiki/Netpbm_format
[3] ImageMagic: http://stackoverflow.com/questions/5150503/image-magick-java

```
DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize, true, rngSeed);
```

If we chose to use a different data source – such as an HBase table or a Kudu table – we have to implement a custom `DataSetIterator`.

This brings us to the next step – but pretty essential part – of our data product. We need a reliable, but flexible way of training set preparation. Here we simply use SQL queries on image metadata. The idea can be generalized to HBase scans, Solr queries, or even SPARQL queries if an external triple store is used. The essence is: any kind of query or filter gives us a bunch of image IDs or file names which allows us to bring the data into Deeplearning4J via a `DataSetIterator`.

To train a neural network we define its architecture first by instantiating a `MultiLayerConfiguration`:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(rngSeed)
         //include a random seed for reproducibility
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
         // use stochastic gradient descent as an optimization algorithm
        .iterations(1)
        .activation(Activation.RELU)
        .weightInit(WeightInit.XAVIER)
        .learningRate(rate) //specify the learning rate
        .updater(Updater.NESTEROVS).momentum(0.98)
         //specify the rate of change of the learning rate.
        .regularization(true).l2(rate * 0.005)
         // regularize learning model
        .list()
        .layer(0, new DenseLayer.Builder()
         //create the first input layer.
                .nIn(numRows * numColumns)
                .nOut(500)
                .build())
        .layer(1, new DenseLayer.Builder()
         //create the second input layer
                .nIn(500)
                .nOut(100)
                .build())
        .layer(2, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
         //create hidden layer
                .activation(Activation.SOFTMAX)
                .nIn(100)
                .nOut(outputNum)
                .build())
        .pretrain(false).backprop(true)
         //use backpropagation to adjust weights
        .build();
```

The code was taken from DL4J examples. We build three layers and activate backpropagation. For more details and an introduction to artificial neural networks you should read the long version of our article in the `docs` folder.

Learning is done after model initialization:

```
    MultiLayerNetwork model = new MultiLayerNetwork(conf);
    model.init();

    //print the score with every 1 iteration
    model.setListeners(new ScoreIterationListener(1));
    model.fit(mnistTrain);
```

Using a ModelWrapper class we are able to persis the model in a local folder or in HDFS.

```
    ModelWrapper.saveModelOnLocalDisc( model, modelID, time);
```

Model evaluation follows in step 5.

3. Query for a specific training set

In this step we need a DataSetIterator for Kudu. The boundary conditions are simple: (1) we use a single machine for training the model. This means, the parallel query execution on Kudu leads to a single stream of input data to the CDSW edge node.
The same approach can also be used to define partitions for parallel learning in multiple containers – one simple specifies a query per partition or training batch.

The following link shows an example of a `DataSetIterator`, which has to be used as a starting point for our own implementation:
https://github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/recurrent/word2vecsentiment/SentimentExampleIterator.java

Note: This isn't part of the first phase of the project. We will add such an implementation in the future.

4. Variation of model parameters

Now it is time to repeat multiple training runs on the same data with variable learning parameters.

The program we prepared for parameter variation and producing multiple models is named `ParameterVariationExperiment1.`

The logic inside the three exemplary loops is subject to parallelization. As long as a single model can easily be trained on one CPU we can use a Spark job to manage all possible parameter sets and start model training in parallel for each individual parameter set in one executor. The downside of this approach is that the training data needs to be loaded into each particular executor completely. But since we deal with a problem which needs multiple iterations on a huge data set, the only reasonable alternative would be a model chain which is trained in one executor. For small models this can be considered – but if the storage capacity required by one particular model is in the range of the executor's heap size this is not possible any more.

5. Evaluation of model quality

Model evaluation means systematic comparison of prediction results with known truth – also called *ground truth*. DL4J offers a convenient method to do such a comparison after training a model. Again, we need the `DataSetIterator`.

```
DataSetIterator mnistTest = new MnistDataSetIterator(batchSize, false, rngSeed);
…
Evaluation evil = new Evaluation(outputNum); //create an evaluation object with 10 possible
classes

while (mnistTest.hasNext()) {
    DataSet next = mnistTest.next();
    INDArray featureMatrix = next.getFeatureMatrix(); //get the network for prediction
    INDArray output = loadedModel.output( featureMatrix ); //get the networks prediction
    evil.eval(next.getLabels(), output); //check the prediction against the true class
}
System.out.println(evil.stats());
```

This gives us the following output:

```
…
Examples labeled as 8 classified by model as 9: 3 times
Examples labeled as 9 classified by model as 0: 2 times
Examples labeled as 9 classified by model as 1: 2 times
Examples labeled as 9 classified by model as 3: 6 times
Examples labeled as 9 classified by model as 4: 9 times
Examples labeled as 9 classified by model as 5: 2 times
Examples labeled as 9 classified by model as 7: 1 times
Examples labeled as 9 classified by model as 8: 1 times
Examples labeled as 9 classified by model as 9: 986 times

==========================Scores========================================
 Accuracy:       0,9821
 Precision:      0,982
 Recall:         0,982
 F1 Score:       0,982
========================================================================
```

Multiple model quality metrics exist as shown in the output above. But you have to be careful, not all metrics make sense in all cases. E.g., in a multi-class classification setting, only accuracy makes sense. One can only define precision and recall in a 1-vs-all sense.

Metric	Value	Explanation
Accuracy:	0,9821	The degree to which a given quantity is correct and free from error. The accuracy of a number x is given by the number of significant decimal (or other) digits to the right of the decimal point in x. Accuracy is also a description of systematic errors, a measure of statistical bias; *http://mathworld.wolfram.com/*
Precision:	0,982	Precision (also called positive predictive value) is the fraction of retrieved instances that are relevant. , while recall The number of digits used to perform a given computation. The precision of x is the total number of significant decimal (or other) digits. *https://en.wikipedia.org/wiki/Precision_and_recall*
Recall:	0,982	Recall (also known as sensitivity) is the fraction of relevant instances that are retrieved. *https://en.wikipedia.org/wiki/Precision_and_recall*
F1 Score:	0,982	The F_1 score (also F-score or F-measure) is a measure of a procedure's (e.g., a test's or a prediction's) accuracy. It considers

		both the precision p and the recall r to compute the score: p is the number of correct positive results divided by the number of all positive results, and r is the number of correct positive results divided by the number of positive results that should have been returned. *https://en.wikipedia.org/wiki/F1_score*

Since we have multiple models – one per parameter set – we want to inspect model quality as a function of learning parameters, model parameters and also dependent on the image preparation procedure which defines the properties of the training set.
Keeping all this information (which is training related metadata) allows us to do an impact analysis and cost analysis or further training sessions on the CDH cluster.

```
public static void updateModelEvaluationResult( String id, Object model )
TODO: Add all relevant artifacts...
```

How do you present the result of all your tuning effort? Let's say, initially, the accuracy was 99.2 % and after some optimization you achieved 99.47 %. The absolute change in accuracy is 0.27 percent points. The error dropped from 0.8% to 0.53% which is 34% error reduction. This is an impressive number! But what can you learn from it? Not much. We suggest to avoid setting the wrong goals: instead of simply shrinking the error by a certain factor, you should specify the error level which can be accepted first. Please remember, there is not such a thing like an ideal model with 100% accuracy.

6. Predict the class of unknown images

Finally, we want to apply the trained model to new unlabeled data. We instantiate the model and load its state from disc.

```
MultiLayerNetwork loadedModel = ModelWrapper.getMultiLayerNetworkFromLocalDisc_latest( ID );
```

Now we have to load the PGM image and convert it into an INDArray.

```
    // Process one image ... in two formats ...
    String fnPGM = "./MNIST_images_pgm/3_01.pgm";
    File f1 = new File(fnPGM);
    ImageLoader loader = new ImageLoader();

    // we have to convert an arbitrary PNG image to INDArray ... which is our feature vector!
    INDArray fm1 = loader.asMatrix(f1);

    // try to predict one label loaded from outside ...

    if( fm1 != null ) {

        INDArray output1 = loadedModel.output(fm1); //get the networks prediction

        System.out.println("dimensions : " + output1.length() + "\n");
        for (int i = 0; i < output1.length(); i++) {
            System.out.print(output1.getInt(i) + " : ");
        }
    }
```

This logic can be executed in any Java program or even in a Spark-shell session as demonstrated in the spark shell script `step6.scala` in the project's `bin` folder.

Another Flume Interceptor can be implemented using this logic. Enrichment of raw data is a very common requirement in any industry. Even if the labeling is not perfect, a good pre-labeling provides some additional background. As a side effect we can easily track the dataset quality. As long as a homogeneous dataset flows in, we could expect also a stable distribution classification results. A deviation from this can be seen as an indicator for either new features or bad data.

A Spark-Streaming job could also be implemented this way.

Summary

The benefit of a single node for learning the neural network is the ability to separate the learning path and the prediction path. We illustrate this idea in figure 2:

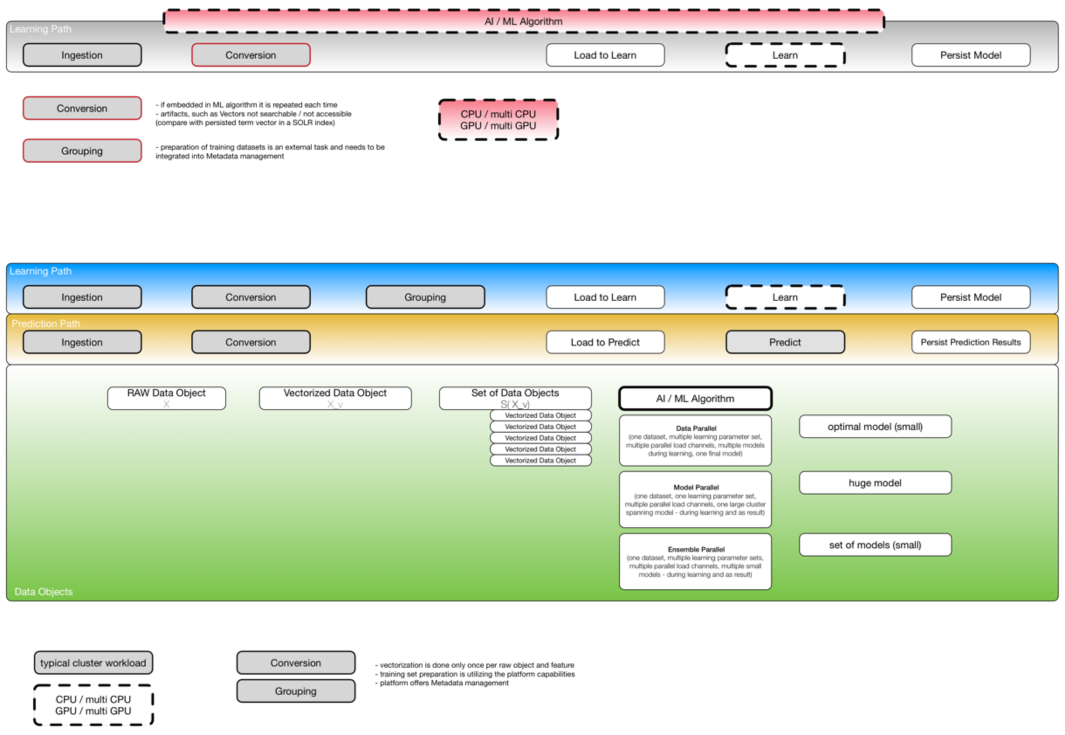
AI / ML Algorithm
Learning Path
Ingestion
Conversion
Load to Learn
Learn
Persist Model
Conversion
- if embedded in ML algorithm it is repeated each time
- artifacts, such as Vectors not searchable / not accessible (compare with persisted term vector in a SOLR index)
Grouping
- preparation of training datasets is an external task and needs to be integrated into Metadata management
CPU / multi CPU
GPU / multi GPU
Learning Path
Ingestion
Conversion
Grouping
Load to Learn
Learn
Persist Model
Prediction Path
Ingestion
Conversion
Load to Predict
Predict
Persist Prediction Results
RAW Data Object
X
Vectorized Data Object
X_v
Set of Data Objects
S(X_v)
Vectorized Data Object
Vectorized Data Object
Vectorized Data Object
Vectorized Data Object
Vectorized Data Object
AI / ML Algorithm
Data Parallel
(one dataset, multiple learning parameter set, multiple parallel load channels, multiple models during learning, one final model)
Model Parallel
(one dataset, one learning parameter set, multiple parallel load channels, one large cluster spanning model - during learning and as result)
Ensemble Parallel
(one dataset, multiple learning parameter sets, multiple parallel load channels, multiple small models - during learning and as result)
optimal model (small)
huge model
set of models (small)
Data Objects
typical cluster workload
CPU / multi CPU
GPU / multi GPU
Conversion
Grouping
- vectorization is done only once per raw object and feature
- training set preparation is utilizing the platform capabilities
- platform offers Metadata management

Figure 2: High level overview of processing stages learning and prediction in an auto-labeling use-case.

The blue box represents the learning path. Often this is done on a subset of the full dataset. This subset is already labeled and defines the *ground truth*. The yellow box works on unknown images. Thus, the prediction step can be done in a different place. The green box in this figure illustrates which data objects have to be provided at each stage. Different ML frameworks use different data types, so we have to be careful. A strong dependency between our processing pipeline and the ML implementation should be avoided – if flexibility in terms of frameworks and algorithms is one of the requirements.

Finally, we can draw a sketch to illustrate how the data flows through all the components and how they work together in figure 3.

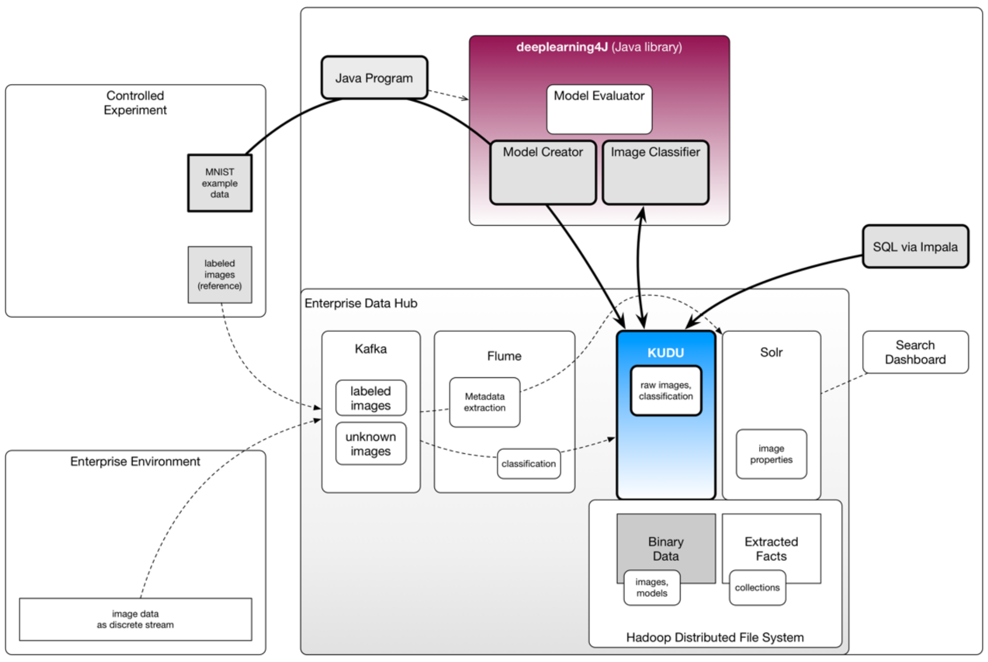

Figure 3: High level architecture and data flows in our minimal viable data product using Deeplearning4J and CDH.

Hands-On Session

You will find the source code in this repository:
https://github.com/kamir/cdsw-dl4j-mvdp-on-cdh

In order to follow the examples, please prepare a 1 node CDH quickstart VM which includes Apache Kudu as provided here.

Checkout our Github repository into the folder `/home/cloudera` inside the VM.

```
$ git clone https://github.com/kamir/cdsw-dl4j-mvdp-on-cdh
```

You can build the artifacts using:

```
$ mvn clean build
```

Now, if no error occurred, you can run the example by using the demo script where $NR indicates the step you want to run:

```
$ bin/run_demo.sh $NR
```

Conclusion & Outlook

This article illustrates a recurring pattern or simply: the anatomy of a scalable data product.

Multiple optimizations and variations exist. In many cases it is simply a different UI or a different source data set. Sometimes you need a very specific neural network and the appropriate training algorithm may not exist yet. We used Deeplearning4J - but the components shown here can be seen as a generalized architectural pattern – no matter which ML framework has been chosen.

Our next goal is to evaluate model parallelity and data parallel training, using Deeplearning4J on CDH.