# Implement a Minimal Viable Data Product Using Deeplearning4J

This article demonstrates data flows and interaction between core components of a data-product. Our goal is an automatic classification of images using neural networks. After reading the article 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

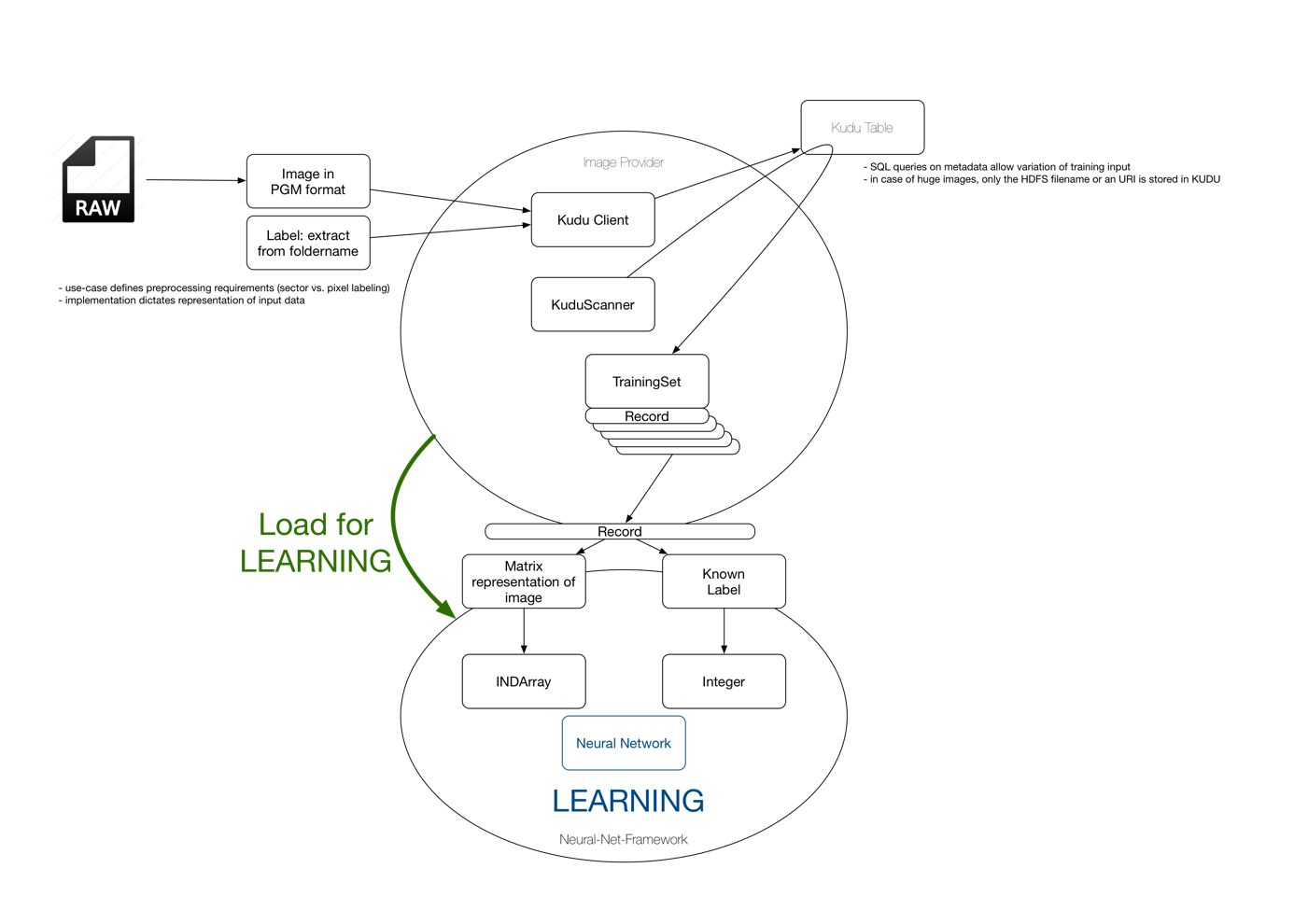
- execute a learned model in a Spark-streaming job

This means: From learning to production …. not perfectly robust, but end-2-end!

## Scope

In order to solve the problem, we have to act in multiple roles. Data engineering is needed in order to prepare a robust data pipeline. The input for our machine learning framework needs to be normalized and learning parameters have to chosen carefully. Finally, after the data scientist job 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 possible. People act in different contexts and multiple collaboration models exist. In this article you will learn, how to implement a minimal viable data product using Apache Spark and Deeplearning4J.



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

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

Parameter-variation and model validation lead to multiple models represented by multiple neural networks. Furthermore, multiple network architectures can be trained. 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 a variety of new input. And finally, continuous learning offers a reference model which can be compared with the current production model. This stability analysis provides information about model update requirements.

Persisting model metadata and metadata of raw images are key factors to build a meaningful model in an cost effective way. We use Apache Kudu 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.

Model deployment has to be managed without any interruption to existing data flows. Since Apache Flume reloads the configuration file automatically, we can simple offer the name or ID of the model which has to be loaded in the Flume configuration. 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:

## Ingest and convert raw images

* Use a simple Java program (encapsulate logic in a helper-class)
* Use the helper class in a Flume interceptor and write via Kudu-Sink

## Train a model from labeled images

* Load image files from a prepared folder structure where folder names represent image labels
* This is called fixed training set approach (as long as no sampling is applied)

## Query for a specific training set

* Provide a RecordScanner for DL4J
* This is called flexible or adjustable training set approach

## Variation of model parameters

* Repeat multiple training runs on the same data with variable learning parameters.

## Evaluation of model quality

* Inspect model quality as a function of learning parametes and image preparation procedure
* This allows us to do an impact analysis on additional preprocessing

## Predict the class of unknown images

* Java program with on particular image
* Flume Interceptor using the helper class and a SOLR sink
* Spark program with an RDD of images
* Spark streaming job Morphline or SolrJ client

Finally we can draw a sketch to illustrate how all the components work together.

You will find the source code in this repository:

<https://github.com/kamir/cdsw-dl4j-mvdp-on-cdh>