Deep Learning on a Cloudera Enterprise Data Hub: From Prototyping to Data Products

Authors: Rob Siwicki, and Mirko Kämpf

In this article we discuss application of deep learning techniques and how to integrate deep learning technology into the Cloudera Enterprise Data Hub. We start with an introduction to deep learning artificial intelligence (AI) features. Apache Kudu and Apache Spark will help us to manage and conduct a variety of data analysis experiments. The goal is to provide a viable data product for a data driven business. The activity stands as an example for the digital transformation towards data driven business decisions and can be generalized to other algorithms and other data sets. Our demo is using an artificial neural network to classify complex unstructured data (in this case, images). Furthermore, we expose metadata for presentation in BI dashboards using Cloudera Impala and Apache Solr. In our case we use Apache Spark to classify input image data that has been loaded into Apache Kudu; then load a Solr index with helpful metadata to supply the dashboard. Apache Spark is used for processing: to learn how to apply an image classification model. In our case the raw data has been staged in Apache Kudu. The loading of results and metadata into a Solr index supports an increase in the variety of integration patterns and is therefore a worthwhile activity.

Motivation

Availability of scalable compute power together with scalable storage capacity, as provided by Big-Data platforms such as Cloudera EDH, has led to a renaissance of neuronal network based methods. Especially recently the concept of deeplearning is attracting data scientists from multiple disciplines. Multiple implementations of deeplearning libraries coexist.

Deeplearning allows extraction of structures from datasets without the applicability of a learned model on a different platform is a critical aspect and is crucial for success in a data driven economy.

Our motivation for using Kudu is to facilitate the future ingest of data in real-time. Image post-processing and optimization techniques need random access to particular images (or frames in case of video processing) and the retraining of models and re-labeling of input data require random read/write access to particular records. Traditionally, this is not possible with core Hadoop with its write once semantics. Since we manage multiple classes of metadata and derived data besides the raw data in order to achieve traceability we apply multiple storage technologies (polyglot persistence).

An example we use is image classification based on neural networks. Application of one or multiple models is essential. We use an iterative classification process. From multiple experiments with multiple parameters we generate a variety of small model files. But in HDFS we shouldn't store too many small data artifacts. In our example we ignore this for simplicity. Here we simply store models locally on a workstation and in HDFS. Optionally, we can use HBase to store small models, and SOLR or Kudu to keep track of model properties. This allows quick analysis on model information in the context of learning optimization and model tuning. Furthermore it is a step towards metadata integration using a multi-layer approach.

The Bigger Picture: AI for Data Driven Business

An iterative process is a widely used pattern. Multiple experiments and incremental improvements build the baseline for a data driven business. The diagram in figure 1 illustrates three aspects of a data driven business as a ternary phase diagram and using the three dimensions: science (in general), engineering, and software development. Activities of each category (on each dimension) contributes to the final business goals. Different business cases require different activities and this leads to different permutations of emphasis of these activities.

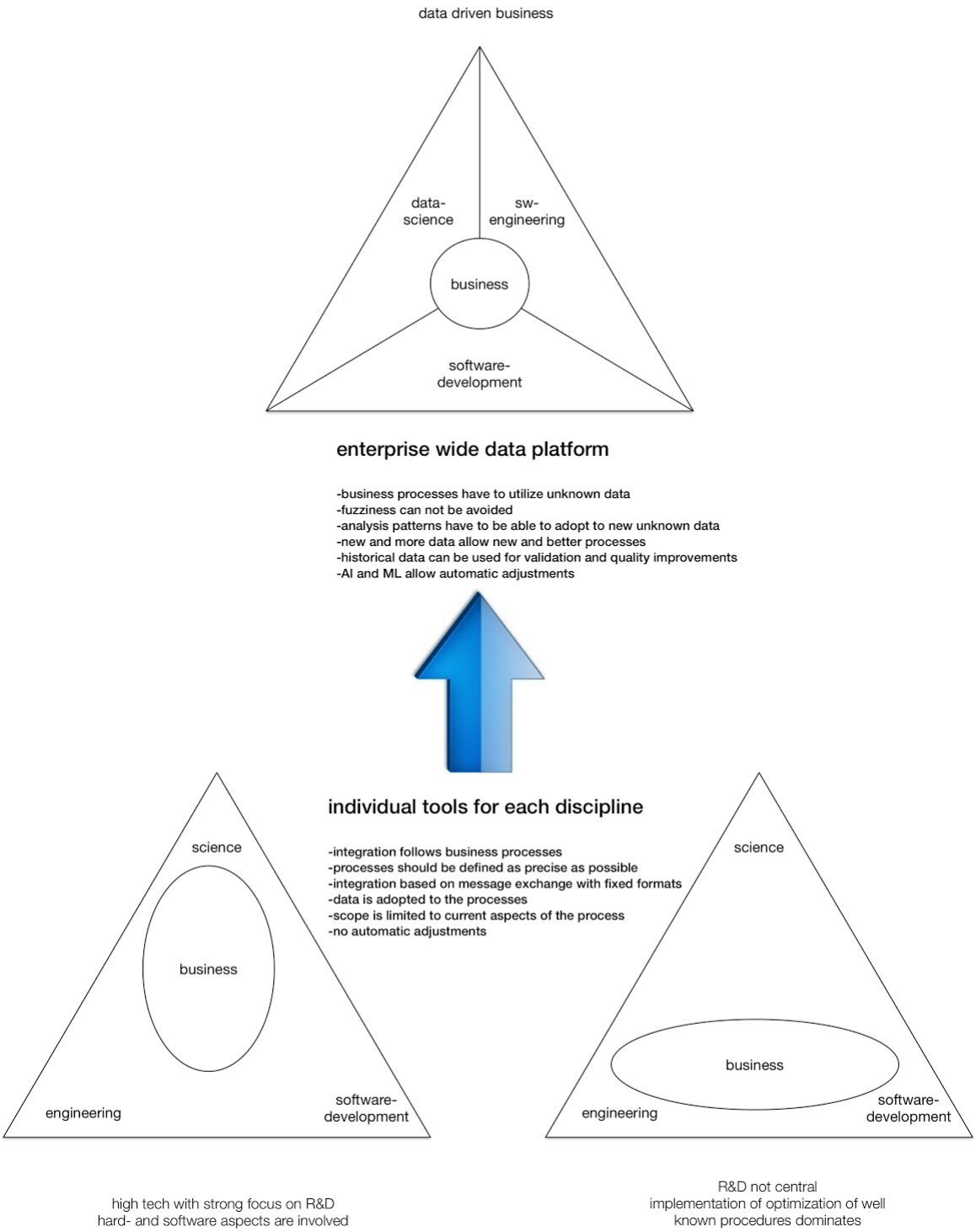


Figure 1: The digital transformation towards data driven business models.

The *digital transformation* moves organisations forward to more data driven patterns. Software development, data engineering, and data science - all together - contribute in a coherent approach to the business, which defines the overall context. Having this high-level figure in mind we can think about an architectural pattern as shown in figure 11. In order to achieve this, all three disciplines have to speak the same language, and this is the visual language of charts and in some cases the symbolic language of math. The journey starts with a transformation of an idea into formulas and algorithms. Software engineering provides implementations (as in our case the Deeplearning4J library, or MLlib which comes with Apache Spark) and the large amount of data needs to be managed by data engineers and curators. There is nice quote: "Data is the next oil." It is certain that IT infrastructure evolves from the background support of the business to the next reactor facility - or perhaps using the oil analogy, refinery.

The Role of CDH & CDSW for Data Driven Business

CDH comes as a scalable data storage and processing platform. The integration of engineering and software development is obvious, since the typical tools such as IDEs and ETL tools exist and even standards exist for both in the market. A different scenario coexists for the data scientist. Also for the data scientist, a variety of tools exist - some are well integrated with Hadoop others not: but do we really need a different tool for the data scientist? Wouldn't it be great if she or he can directly work on the CDH stack? Using the spark-shell or the Jupyter notebook are great options. Another one is the recently released product named Cloudera Data Science Workbench (CDSW).

And for sure, before we can use an artifact within the spark-shell we have to test locally in our IDE (in a real world even using automatically using Jenkins or other CI tools). Thus, the IDE becomes naturally a place where the implementation journey starts; but a bit later comes the critical point, at the time when we release the artifacts to the wild, with the goal of using them in a productive data product. Usually a maven repository is used for this purpose, from where the implemented algorithms can be picked up by the data scientists, who do not touch the internal bolts, but who are instead keen on doing the best refining one can think of.

Data scientists, data engineers, and software developers come together at this point and this article aims on showing how this three roles can work together in an efficient way, using the CDH stack. Deeplearning4J is an example, carefully chosen, to emphasize the ease of use, even for not so obvious code artifacts. Finally this should also motivate you to bring over your own code to the spark-shell or to CDSW to enable it to operate on every growing data sets.

Introduction

In general, there are three types of metadata in our images. (A) color profiles and pixel statistics, object counts etc. (B) technical metadata related to image creation such as geo-coordinates or aperture, and finally, predicted information which depends on the model, usually derived from a reference data set. All those facts can be combined in advanced

analytics tasks. It was also important to our solution that we had the ability to overwrite previous results as we improved the quality of our classification function.

The high level architecture of our first solution is illustrated below (Fig. 1).

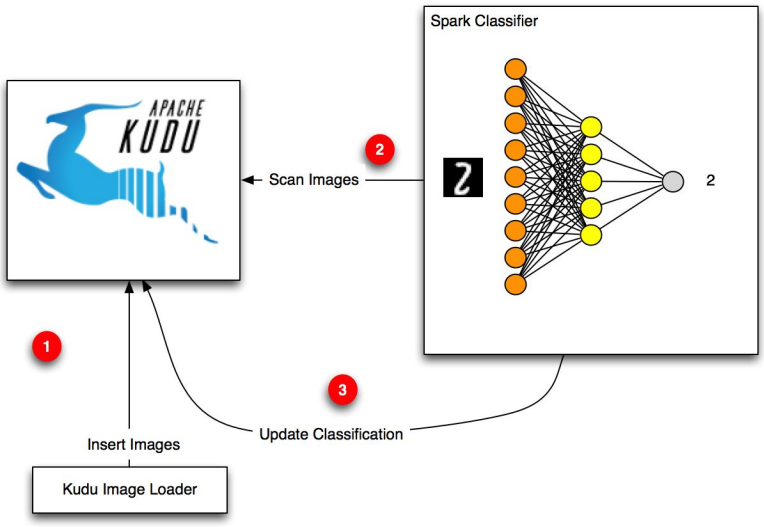


Fig 1. High Level Architecture Diagram

Having used Artificial Intelligence techniques in the past that were mostly hand coded Artificial Neural Networks (ANNs), we wanted to derive a useful and perhaps more pain free way of utilising such capabilities; but, also doing so with a modern data platform like CDH.

Why Use Deep Learning and Neural Networks

A neural network has the capacity to derive a function f that can map an input vector x to an output y. Consider:

$$f(x) = y$$

In our case we want to find the function f that when provided with an image can map to y, which is the image's classification. When the neural network learns, it is finding the function f.

Our first decision was to choose a neural network framework for our implementation. We had previously written our own implementations in C, C# and Scala for solving various problems such as image classification and forecasting stock market movements. We had also experimented with TensorFlow in the past; but, one of our colleagues suggested deeplearning4j. After a quick review of the excellent learning material and quickstart information we decided to use this. The API is simple to understand and the framework is also Java based providing a lower complexity and ease of integration when considering our own skills base.

We wanted to choose a simple implementation to experiment with. So a feed forward neural network using back-propagation as the training method. We also utilised pre-prepared data from the deeplearning4j examples. Handwriting recognition is a standard example in this field, so we choose to use the MNIST data.

Example Dataset : The MNIST Database

The MNIST data examples (https://en.wikipedia.org/wiki/MNIST_database) represent a series of standard test cases for handwriting classification problems. It contains 60,000 training images and 10,000 testing images. Fig. 2 shows examples of hand-written digits.

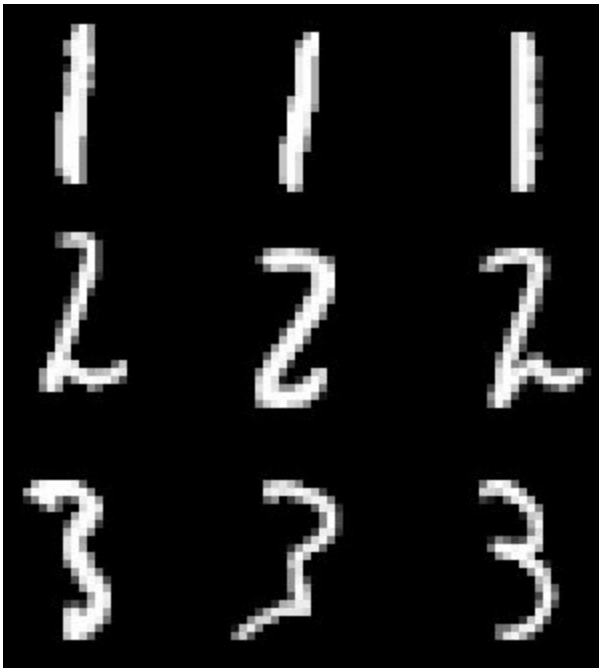


Fig 2. MNIST Images

We then wanted to integrate the classifier into an Apache Spark job that would be able to read image files from a CDH cluster and then output the classification data.

Introduction to Artificial Neural Networks

For those not familiar with the concepts and function of ANNs we will highlight the concepts of a simple feed forward neural network; however, there are several different types of network that can be used with different effect to solve different problems. For example, Convolutional AANs are well applied to the field of image recognition and recurrent ANNs have been applied to time

series analysis problems. In the following section we go deeper into the details of the mechanics of a feed forward neural network.

Neuron Architecture

Nearly every ANN has input and output neurons. Inputs and outputs are generally vectors of floats [1.0,0.6,0.96,0.77] and can be mapped into grids, cubes or other higher dimensional entities.

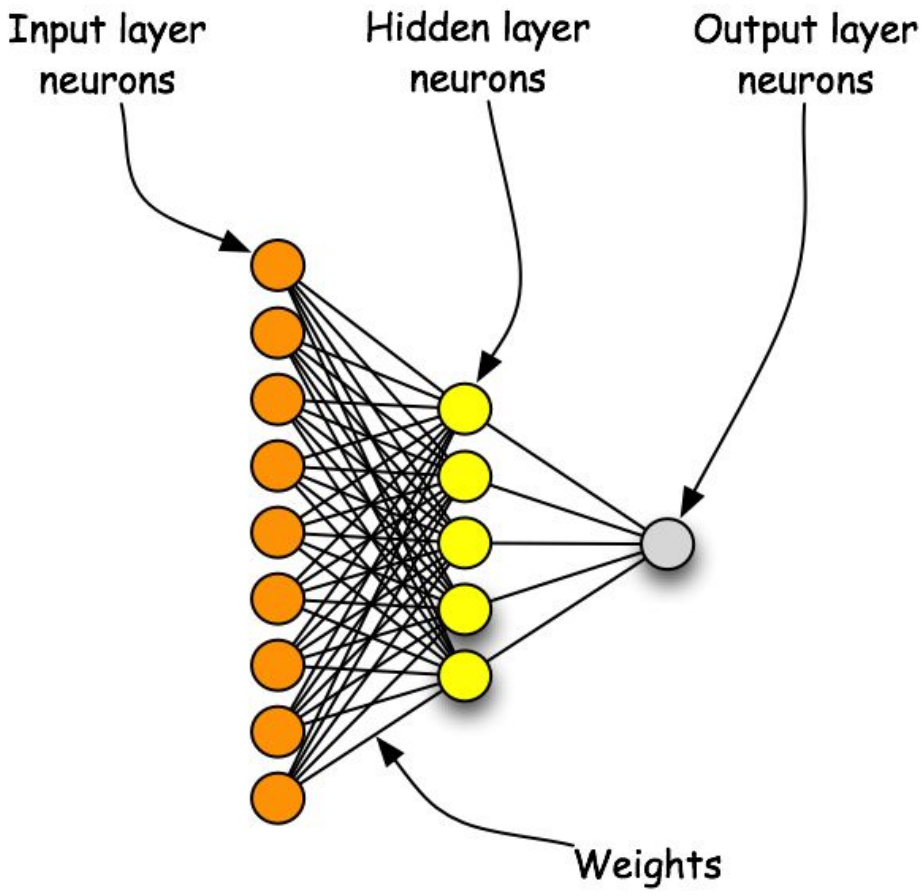


Fig 3. High Level ANN Architecture

In the case of the MNIST data set, the input vector represents the matrix of the bitmap image of each handwriting character. The input image is normalised so that the input vector represents a series of 0s and 1s for either colored or blank pixels and the two dimensional 28 x 28 image is represented as a one dimensional vector (d=28^2).

Each neuron is connected to other neurons by weights. When the ANN operates weights are used to amplify or minimize the strength of output of each neuron. It is in the phase of learning that the neural network adjusts the weights and thereby begins to approximate the desired function (see lower part in Fig.1). In simple ANNs, no links exist between neurons of one layer; but more complex architectures are possible where this is an apparent feature.

Activation Functions

It is important to understand Activation Functions (AFs). AFs defines the output of each neuron. For example, given an input value the AF will map the value to an output value. The step function is one of the simplest examples. Given an input value x between 0 and 1 a step function would typically map values below a threshold to 0 and over a threshold to 1.

i.e. *f(x)={x<0.5=0,x<1.0=1}*

Thereby we use AFs to control the **firing** of neurons. There are several types of AF:

- Linear activation function *f(x)=x*
- Step activation function f*(x)={x<0.5=0,x<1.0=1}*
- Sigmoid activation function *f(x)=1/(1+e^-x))*
- Hyperbolic tangent activation function *f(x)= tanh(x)*
- ReLU – Rectified Linear Units *f(x)=max(0,x)* – does not saturate to 1 or 0
- Softmax activation function *f(x)=e^zi/sum(e^zj)*

Bias

Note that bias is an important concept. The effect of the weights essentially adjust the _slope_ of the activation function. The bias can be used to move the function output left and right. If we consider a sigmoid function such as Fig. 4:

$$f(x, w, b) = 1 / (1 + e^{-(wx+b)})$$

Fig 4. Sigmoidal Activation Function Formula

Where x is the input number, w is the weight and b is the bias, we can see from a rapid example in R, how the bias can affect the output of the function.

```
sigmoid = function(x, bias) {
  1 / (1 + exp(-x + bias))
}

x <- seq(-4, 4, 0.01)
b <- 0.0

plot(x, sigmoid(x, b), type="l", col="red")
b <- 0.5
```

```
lines(x, sigmoid(x, b), col="red")
b <- -0.5
lines(x, sigmoid(x, b), col="red")
b <- 1.0
lines(x, sigmoid(x, b), col="red")
b <- -1.0
lines(x, sigmoid(x, b), col="red")
b <- 1.5
lines(x, sigmoid(x, b), col="red")
b <- -1.5
lines(x, sigmoid(x, b), col="red")
b <- 2.0
lines(x, sigmoid(x, b), col="red")
b <- -2.0
lines(x, sigmoid(x, b), col="red")
```

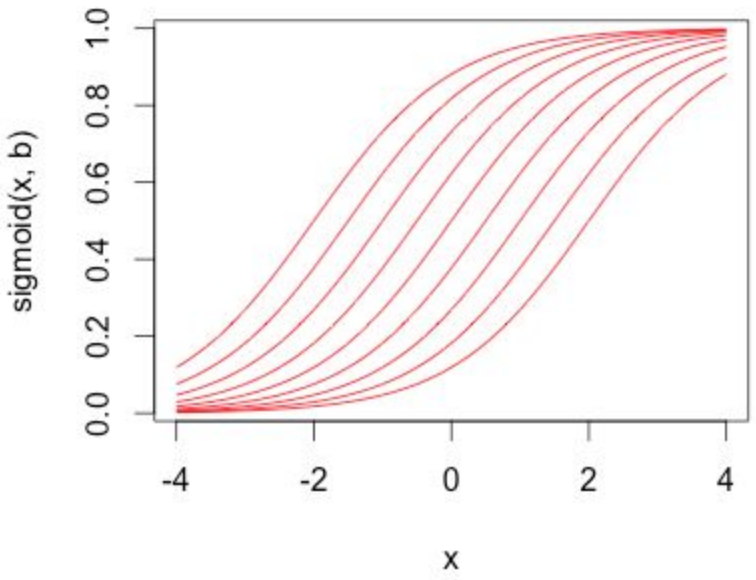


Fig 5. Example of Bias Affecting Sigmoidal Activation Function Placement

It is useful to illustrate a simple example here of something known as a perceptron. This will aid the users understanding of what is happening inside the network when it calculates a prediction.

A perceptron is an early conceptual precursor implementation of a neural network. For our example we will represent a simple perceptron based AND logic function.

In this case there are two input neurons and one output neuron. There is no hidden layer. The inputs represent the cases of a standard binary AND operation ([0,0],[1,0],[0,1],[1,1])

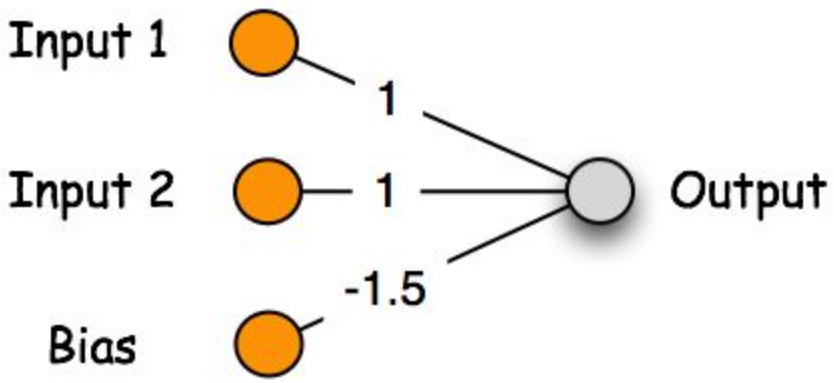


Fig 6. Perceptron Logical AND Gate

Consider what happens if we present this perceptron with each of the logical AND permutations ([0,0],[1,0],[0,1],[1,1]) .

We get the following calculations:

*[0,0] => (0*1) + (0*1) + (-1.5) = -1.5*
*[1,0] => (1*1) + (0*1) + (-1.5) = -0.5*
*[0,1] => (0*1) + (1*1) + (-1.5) = -0.5*
*[1,1] => (1*1) + (1*1) + (-1.5) = +0.5*

Nothing special until we apply the AF. In this case we utilise the Step AF *step(x) = {1, if x >= 0.5 | else 0}*.

Consider the step function in R.

```
step = function(x) {
  ifelse(x>=0.5,1,0)
}

x <- seq(-4, 4, 0.001)

plot(x, step(x), type="l", col="red")
```

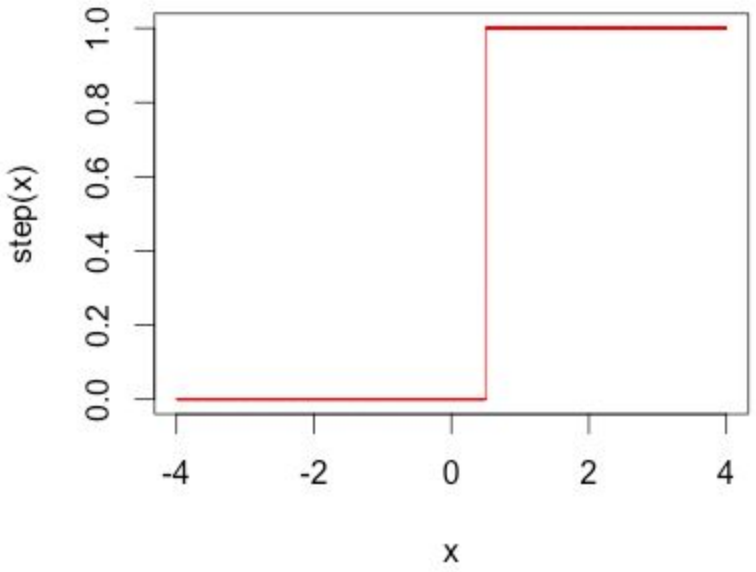


Fig 7. Example of Step Activation Function

Once applied to our perceptron output we get the following result.

[0,0] => step(-1.5) = 0
[1,0] => step(-0.5) = 0
[0,1] => step(-0.5) = 0
[1.1] => step(+0.5) = 1

There we have it, we have constructed a logical AND gate truth table using a simple perceptron. To complete this introductory example we can also illustrate an OR gate.

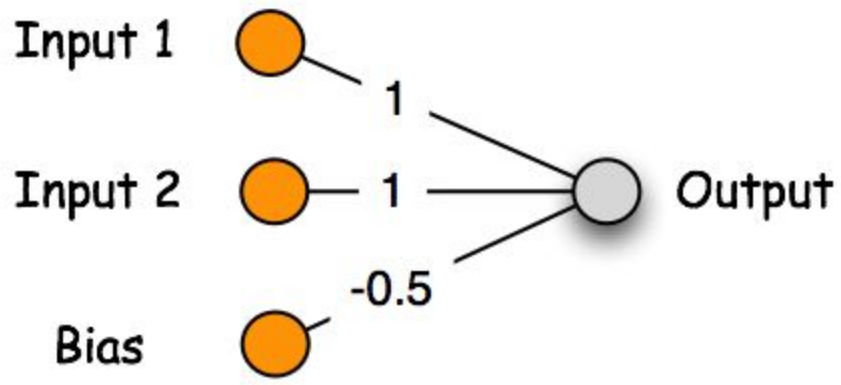


Fig 8. Perceptron Logical OR Gate

We get the following calculations:

*[0,0] => (0*1) + (0*1) + (-0.5) = -0.5*
*[1,0] => (1*1) + (0*1) + (-0.5) = +0.5*
*[0,1] => (0*1) + (1*1) + (-0.5) = +0.5*
*[1,1] => (1*1) + (1*1) + (-0.5) = +1.5*

Again, nothing special until we apply the AF. In this case we also utilise the Step AF *step(x) = {1, if x >= 0.5 | else 0}*.

[0,0] => step(-1.5) = 0
[1,0] => step(-0.5) = 1
[0,1] => step(-0.5) = 1
[1.1] => step(+0.5) = 1

Consider that in all cases we have multiplied the inputs by the weights for each neuron and then added the bias. After this we have applied the activation function.

If you scale this simple perceptron model through the addition of layers of hidden neurons and alternative AFs you can start to approximate more complex functions within the ANN. Also consider that by the manipulation of weights and biases the ANN can learn different more complex functions. In general, the selection of the right network type and also the activation function depend on the type of data and the function one wants to approximate using the ANN. A lot of discussions and research are going on in this field. So far we almost exclusively used sigmoidal, step and or softmax as activation function.

This is what we do when we train the AAN. We manipulate the weights and bias to minimise the error in prediction in our network and even more complex AAN architectures can map more complex functions to the extent that we can create classifiers for images or recognise stock trading patterns.

Note that input layer neurons generally have no activation function and are just weighted and summed and hidden layer neurons are connected to input neurons, other hidden neurons or output neurons. Strictly speaking a deep learning neural network consists of multiple hidden layers of neurons. We will expand on this in a later post.

The Feed Forward Neural Network

In our more complex feed forward neural network architecture we interpose the input layer and output layer with a hidden layer of neurons. We also scale out the network with many more inputs to match the number of pixels (28 X 28) in our images and the number of output neurons matches the number of classes - each representing the potential classification of the handwritten digits from 0 to 9.

Instead of hardcoding the weights as our perceptron example, we employ back-propagation to adjust the weights automatically, hence this is the algorithm by which the neural network learns.

The utilisation of our neural network is dominated by two phases. The first is a training phase where the neural network is exposed to the data set. After initialisation of the neural network's weights the training algorithm repeats iterations of forward calculation, known as feed forward and then back propagation to minimise the error within the network by adjusting the weights. When we demonstrated the calculation of logic gates using perceptrons we essentially applied a simple feed forward process; though, with precalculated weights and bias.

The second phase is the actual classification and prediction itself; where the neural network is presented with new images to classify and is essentially ready for utilisation in an application.

The feedforward phase is simple and can be represented as the multiplication of inputs by weights and application of bias.

For each iteration the training algorithm gets the data, applies the neural network and compares the obtained result with the expected one.

$$f(x,w) = activationFunction(\sum(w \cdot x) + b)$$

Fig 9. Feedforward Calculation

On each iteration of network learning, the network is provided the inputs and invokes the feedforward mechanism, the output is tested to ascertain the error in the context of the learning data and weights are adjusted to minimise the error utilising a learning rate hyperparameter. Essentially SGD is used to calculate the extent to optimise how the weights must change (see https://en.wikipedia.org/wiki/Stochastic_gradient_descent).

It is important to initialise the weights correctly to get a good classifier. In our case we use the Xavier initialisation algorithm. Essentially this algorithm sets all the weights to normally distributed random numbers (for further details see [1]).

Implementation using Deeplearning4J on Spark

Deeplearning4j allows you to create the architecture of such a network programmatically, e.g., using the following code snippet, taken directly from the deeplearning4j examples:

```
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(LossFunction.NEGATIVELOGLIKELIHOOD) //create hidden layer
        .activation(Activation.SOFTMAX)
        .nIn(100)
        .nOut(outputNum)
        .build())
  .pretrain(false).backprop(true) //use backpropagation to adjust weights
  .build();
```

Our strategy was to train the network offline using a Java application, then to store the generated weights. If we could store the weights in a file we could then use the file to re-populate multiple instances of the network and thereby scale it out across our CDH cluster. From previous experience we knew that once the neural network had learned to solve the

classification problem, the actual classification of examples themselves was relatively fast - so this would be a good strategy.

Since this article is not about a the philosophical question: Java vs. Scala vs. Python - but rather about integration of an existing library into the enterprise wide data lake (including the Hadoop stack). Currently, in Java 7 and Java 8, there is interactive shell available. Thus we prefer to use the spark-shell. From here, we can use the Scala programming language and integration of Java libraries is as easy as demonstrated in our examples. Finally, the goal of this blog is not to teach „Scala“, but rather to emphasize the generic usability of the CDSW which uses the spark-shell under the hood.

Unfortunately, the integration of all libraries doesn't yet work out of the box. Thus, our article shows, how to achieve this integration and proper artifact deployment using a Maven based project and Hadoop distributed storage.

Persistence of Trained Models

The deeplearning4j framework provides a mechanism to save the model using the `ModelSerializer` class. It produces a zip file with a JSON descriptor of the model including all relevant parameters and a binary representation of the model data: the learned weights. The model descriptor doesn’t contain information about the training data set. This information has to be tracked separately.

In order to generate some test weights we added the serializer code to the `MLPMnistSingleLayerExample` from the deeplearning4j examples code, as follows:

```
File locationToSave = new File("bpann.zip");
boolean saveUpdater = true;
ModelSerializer.writeModel(model, locationToSave, saveUpdater);
```

We could then reinitialise the trained BPANN in a Spark-shell session using the Java-wrapper code and begin classification.

Managing Random Access to Small Data Entities

To read the MNIST data within Spark the sensible strategy to adopt was to utilise the `MnistDataSetIterator`. The `MnistDataSetIterator` contains the code necessary to read an example of the data.

The `MnistDataSetIterator` converts the standard MNIST images to ppm format. In order to load the images into Apache Kudu we had to transform it to ppm first. The transformation is

done because it is a simple representation of an image pixel by pixel that is easy to map to input nodes of a neural network, represented by the `INDArray` in our case.

> ***Warning:*** *the objects which have to be represented as input vectors - or more specifically as* `INDArray` *of shape* `[3,5]` *- must not be bigger than 64kb. Kudu limits the cell size to 64 kb and the number of columns to 300 to achieve great performance.*

Loading the data into Kudu with the Java API is straightforward. We use a table named 'MNIST' on Kudu-master 'quickstart.cloudera'. The following code snippet illustrates the table initialisation.

```
/*      Our Kudu image store uses the following settings:       */
public static String tableName = "MNIST";
public static String kuduMaster = "quickstart.cloudera";
public static ArrayList<ColumnSchema> columnList = null;
public static Schema schema = null;

public static void init() {

kuduClient = new KuduClient.KuduClientBuilder(kuduMaster).build();

columnList = new ArrayList<>();
columnList.add(new ColumnSchema.ColumnSchemaBuilder("ID", Type.STRING).key(true).build());
columnList.add(new ColumnSchema.ColumnSchemaBuilder("IMAGE_AS_PNG", Type.STRING).key(false).build());
columnList.add(new ColumnSchema.ColumnSchemaBuilder("IMAGE_AS_PGM", Type.STRING).key(false).build());
columnList.add(new ColumnSchema.ColumnSchemaBuilder("PREDICTION", Type.INT32).key(false).build());
columnList.add(new ColumnSchema.ColumnSchemaBuilder("KNOWNLABEL", Type.INT32).key(false).build());

  schema = new Schema(columnList);

try {

  CreateTableOptions options = new CreateTableOptions();

  ArrayList<String> hashPartition = new ArrayList<String>() {{add("ID");}};

  options.setNumReplicas(1);
  options.addHashPartitions(new ArrayList<String>(hashPartition),2);

  System.out.println("CREATE table for MNIST data in Kudu");

  kuduClient.createTable("MNIST", schema, options);

}
catch (KuduException e) {
  e.printStackTrace();
}

}
```

Images (in PNG format) are extracted from the raw MNIST-DB. We convert them to PGM format (a special case of PPM).

The following code snippet shows how we put the raw image data into Kudu:
Note: It is not possible to store objects larger than 64k in a Kudu table!

```
byte bytesPGM[] = new byte[0];

String bytesPGM_as_BASE64 = "";

try {
   bytesPGM = extractBytesFromPGM( f.getAbsolutePath() );
   bytesPGM_as_BASE64 = Base64.encodeBase64String( bytesPGM );
}
catch (Exception e) {
   e.printStackTrace();
}

Insert insert = table.newInsert();

// take the known label from filename
String labelKnown = f.getName().substring(0,1);

PartialRow row = insert.getRow();
row.addString(0, f.getAbsolutePath() );
row.addString(1, bytesPNG_as_BASE64);
row.addString(2, bytesPGM_as_BASE64);
row.addInt(3, -1);
row.addInt(4, Integer.parseInt(labelKnown) );

try {
   session.apply(insert);
   session.flush();
}
catch (KuduException e) {
   e.printStackTrace();
}
```

The BASE64 encoded image is loaded from Kudu using the following code:

```
...
List<String> projectColumns = new ArrayList<>(1);
projectColumns.add("IMAGE_AS_PGM");
projectColumns.add("ID");

KuduSession session = KuduImageStore.kuduClient.newSession();
KuduTable table = KuduImageStore.getImageTable();

KuduScanner scanner = KuduImageStore.kuduClient.newScannerBuilder(table)
      .setProjectedColumnNames(projectColumns)
      .build();
while ( scanner.hasMoreRows() ) {
   RowResultIterator results = scanner.nextRows();
   while (results.hasNext()) {
      RowResult result = results.next();
      String id = result.getString("ID");
      String bytesAsNase64 = result.getString("IMAGE_AS_PGM");
      byte[] bytes = Base64.decodeBase64( bytesAsNase64.getBytes() );
…
   }
}
…
```

Crucially, the PGM image files need to be converted into the input matrix. This can be achieved with the following code, extracted from deeplearning4j examples:

```
public static INDArray convertBinaryToFeatureMatrix(int numExamples, byte[] img) throws IOException {

  float[][] featureData = new float[numExamples][0];
  float[] featureVec = new float[img.length];
  featureData[0] = featureVec;

  for (int j = 0; j < img.length; j++) {
     float v = ((int) img[j]); //byte is loaded as signed -> convert to unsigned
     if (v > 30.0f) {
        featureVec[j] = 1.0f;
     }
     else {
        featureVec[j] = 0.0f;
     }
  }

  featureData = Arrays.copyOfRange(featureData,0,1);

  INDArray features = Nd4j.create(featureData);

  return features;

}
```

Essentially the image is binarized with colored pixels represented as the float value 1.0 and uncolored pixels represented as the float value 0.0.

The matrix is then supplied to the newly instantiated AAN as follows:

```
INDArray featureMatrix = convertBinaryToFeatureMatrix(1, bytes);
INDArray output = restoredModel.output(featureMatrix);
```

This pretty short video on Youtube illustrates the conversion of a raw image to an INDArray.

Classification of Unknown Images

The output INDArray represents the AAN classification of the image retrieved from Kudu. The output is then processed to extract the classification that has the best match.

```
winnerTakesAllDecision(output);
```

The output is then supplied back to Kudu as the classification of the image using the method named `updateClassificationResult` in the `KuduImageStore` class.

```
public static void updateClassificationResult( String id, int label ) {
  try {
     KuduSession session = KuduImageStore.kuduClient.newSession();
     KuduTable table = KuduImageStore.getImageTable();

     Update update = table.newUpdate();
     PartialRow row = update.getRow();
     row.addString("ID", id);
     row.addInt("PREDICTION", label);
     session.apply(update);
     session.flush();
  }
  catch( Exception ex ) { … }
}
```

So far so good. We are able to use the deeplearning4j library in our program IDE. Please execute the class `demo.ImageClassifierHDD` to train the model. The class `demo.ImageClassifierKudu` is used to ingest the MNIST data to a new Kudu table and then it performs the classification. Results are written directly to Kudu. Training of a classification model and using it for prediction in our Java code works just fine.

Bringing all Pieces to one Table: Integration Pattern on CDH

But is this the right way to do data science? Usually, software development tasks lead to programs. In this particular case, we need the data in the right shape, the right model, and the right infrastructure to apply the model at scale. All this is a combination of multiple techniques, usually applied by different roles or even different teams. Nowadays, people use the name “data product” a lot. Such a data product includes the logic, fancy math, appropriate data pre- and post-processing. And all this has to happen in a scalable environment for one purpose: to solve a business problem.

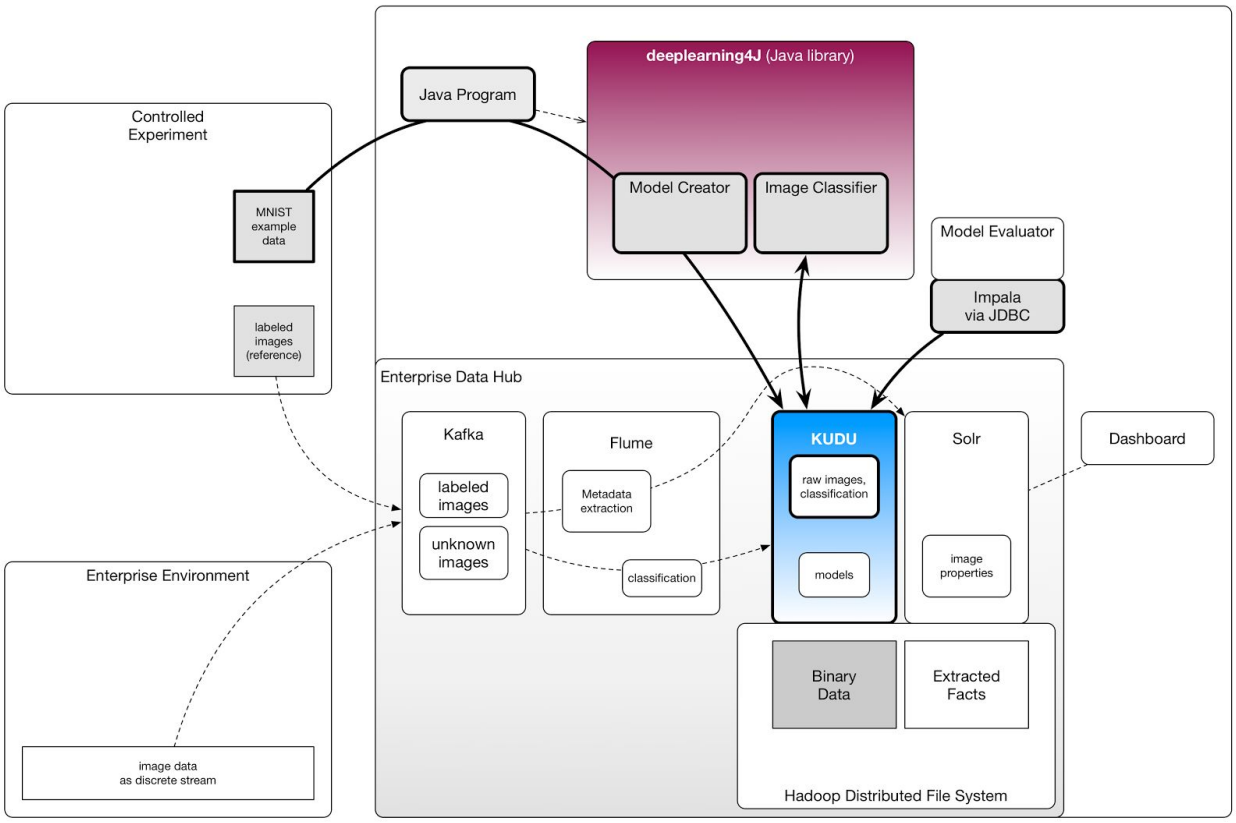


Figure 11: The minimal viable product for automatic image classification. Bold borders illustrate components which have been implemented in the first iteration.

In order to achieve this, we extend our initial demo program. We created a wrapper module around deeplearning4J with all required Java code for the particular BPANN example. Our goal is clear: we want to train and evaluate multiple models using a spark-shell or the Jupyter notebook - not our Java IDE.

This allows us to apply the best identified model to a huge dataset in a reliable repeatable way already during ingestion time. Since models may have to be updated, our procedure must track the state of individual training and evaluation runs together with all related properties.

Loading individual images in a spark-shell session can be done easily. Just run a spark-shell with the following package: `org.apache.kudu:kudu-spark_2.10:1.1.0`. Next, we import the required packages, define a dataframe and query our data, stored in Kudu.

```
scala> import org.apache.kudu.spark.kudu._
scala> val df = sqlContext.read.options(Map("kudu.master" -> "quickstart.cloudera:7051","kudu.table" -> "MNIST")).kudu
scala> df.select("ID", "KNOWNLABEL", "PREDICTION").take(1)
res3: Array[org.apache.spark.sql.Row] =
Array([/GITHUB.cloudera.internal/cdsw-deeplearning4j-demo-02/MNIST_images_pgm/0_01.pgm,0,0])
```

Besides our spark-shell, also the Impala shell enable queries on our image-table. First, you login into the Kudu-quickstart VM using the demo user account (password: demo).

```
$ ssh demo@
$ impala-shell
```

Now you can create a table to query existing data using Impala:

```
CREATE EXTERNAL TABLE mnist
STORED AS KUDU
TBLPROPERTIES (
 'kudu.table_name' = 'MNIST'
);

select count(PREDICTION),PREDICTION as all_prediction
from mnist group by prediction order by prediction;

select count(PREDICTION),PREDICTION as correct_prediction
from mnist where prediction=knownlabel
group by prediction order by prediction;
```

In our first iteration we have implemented the following 3 phase procedure:

Phase 1 : Preparation of raw data:

- load data from IDX files and save PNG
- convert PNG to PGM
- read PGM from disc and convert to BASE64 string and put into KUDU.

Phase 2 : Train an appropriate model:

- take BASE64 string and known labels from Kudu-table
- generate a model (or multiple models for comparison)
- store models in a reusable format

Phase 3 : Application of the model for prediction of unknown labels

- load a chosen model from model repository
- take new image and convert to PGM and to *`INDArray`*
- apply the ANN model to the vector representation of the image to get the label prediction

With this components in place we have our minimal viable data product. Sure, the model needs to be improved, and also the toolset needs some more features. But this will be done next time.

We want to be able to learn classification models from arbitrary images, stored in a CDH cluster. Model evaluation with multiple parameter sets have to done and compared. The model should than be applied to new images already during data ingestion.

The deeplearning4j library offers a data pipeline concept using a `DataSetIterator`.
"It iterates through input datasets, fetching one or more (batchSize) new examples with each iteration, and loading those examples into a `DataSet(INDArray)` object that neural nets can work with." In our special case each loaded image has to be converted to a line vector represented by the `INDArray` object.

In order to scale our approach to larger datasets and to multiple parameter sets we plan to utilize Apache Kudu and Apache Spark on a CDH cluster. How we implement the Kudu-based `DataSetIterator` will be shown in the next part. How trained models are applied automatically in different ways during data ingestion, e.g., using Apache Flume and Apache Kafka will be shown in the third part.

Evaluation of Model Quality & Model Stability using CDH

How to train an appropriate model and how to apply it in a data product are two crucial questions - especially if we deal with data in a data driven business context. Possible paths to answer both questions are manifold and vary much for both cases.

First of all, one must have a good understanding of the problem space and some possible ideas about a potential solution must exist. This includes the mathematical representation of the problem and its solution. Usually, research projects lead to new solutions of known or even totally new problems.

The second aspect; bringing a solution to production, is more of an engineering task rather than research - although the right tuning and optimizations of the processes can be research projects on their own.

Previously, we demonstrated the integration of multiple techniques for one purpose: for deep learning on top of a CDH cluster. We demonstrated, how a back propagation artificial neural network (BPANN) can be used to classify images. In other words, image classification can also be seen as symbol recognition, assuming, there is only one single thing on an image.

We used the MNIST dataset, which provides 60,000 different handwritten digits represented as 28x28 images. Transforming the PNG images into a INDArray enabled us to use the deeplearning4j library and a simple Java program to train a simple model. Our example - as shown in the following code snippet - was based on a three layer ANN with one hidden layer:

```
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();
```

Since we do not know much about the quality of the model and how well it learned its classification task, we go now a little bit deeper and vary some of the model parameters.

What we know are the following four metrics:

Multiple metrics exist, but not all make sense in each case. 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 but that won't be relevant here.

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*

On a single computer it took 129 seconds to train and to validate this particular model. Let's assume we want to vary the following parameters:

```
// Number of possible outcomes (e.g. labels 0 through 9, or all possible ASCII characters).
static int[] outputNums = { 10, 20, 128, 256};
// RNG needs a seed value. Results should be independent from those.
static int[] rngSeeds = { 123, 100, 200, 10, 5, 9, 400 };
// Learning rate defines how the individual weight-updates are scaled.
static double[] rates = { 0.0015, 0.003, 0.006, 0.012, 0.024};
```

This leads to 140 variations and if repeated 10 times each we would need 50 hours on a single computer for our model benchmark.

Although the dataset is not yet that huge, we prefer to use a CDH cluster to solve the problem which is a systematic investigation of the trained model family, regarding prediction quality and stability in the presence of parameter variation.

Later, it will also be relevant to repeat such analysis for fixed parameters, but variable input data. Especially if a system is used over a long time period, one could easily oversee hidden trends or systematic changes in the input data which leads to a wrong result because the used (and outdated) model does not reflect new features of the data which didn't exist during training time. In order to identify such conditions it seems to be valuable to keep model benchmarks beside the raw data. This allows us not only retraining of new models from old data, but also an impact analysis, which allows us to see what would had been different in the past, if the new model would have been used already earlier. An alternative could be usage of adaptive algorithms, but this is a different field of research on in our current scope.

In this article we want to demonstrate a procedure for model investigation. All steps can be done with tools available in CDH and the open source library deeplearning4j. Final results will be plotted using D3.js and Gnuplot. The motivation is to build the bridge between the „fancy lab setups“ and „real factory environments“ since this is what allows us to make and to call it a business.

Results

Using the class: `ParameterVariationExperiment1` we create several models, all stored in the folder `models` and utilise a result file named `exp1-final-clean.txt`.

# nr of classes_RNG-seed_rate	# nr of classes_RNG-seed_rate	# nr of classes_RNG-seed_rate
10_100_1.0E-5 Accuracy: 0,8641 Precision: 0,8635 Recall: 0,8613 F1 Score: 0,8624	10_200_1.0E-5 Accuracy: 0,8553 Precision: 0,8545 Recall: 0,8526 F1 Score: 0,8535	10_300_1.0E-5 Accuracy: 0,8605 Precision: 0,8604 Recall: 0,8582 F1 Score: 0,8593
10_100_1.0E-4 Accuracy: 0,9343 Precision: 0,9337 Recall: 0,9335 F1 Score: 0,9336	10_200_1.0E-4 Accuracy: 0,9347 Precision: 0,934 Recall: 0,9338 F1 Score: 0,9339	10_300_1.0E-4 Accuracy: 0,9351 Precision: 0,9344 Recall: 0,9343 F1 Score: 0,9344
10_100_0.001 Accuracy: 0,9769 Precision: 0,9769 Recall: 0,9768 F1 Score: 0,9768	10_200_0.001 Accuracy: 0,9754 Precision: 0,9756 Recall: 0,975 F1 Score: 0,9753	10_300_0.001 Accuracy: 0,9767 Precision: 0,9767 Recall: 0,9764 F1 Score: 0,9765
10_100_0.01 Accuracy: 0,9845 Precision: 0,9845 Recall: 0,9843 F1 Score: 0,9844	10_200_0.01 Accuracy: 0,9833 Precision: 0,9833 Recall: 0,9832 F1 Score: 0,9832	10_300_0.01 Accuracy: 0,9837 Precision: 0,9837 Recall: 0,9835 F1 Score: 0,9836
10_100_0.1 Accuracy: 0,9479 Precision: 0,9496 Recall: 0,9475 F1 Score: 0,9486	10_200_0.1 Accuracy: 0,1028 Precision: 0,1028 Recall: 0,1 F1 Score: 0,1014	10_300_0.1 Accuracy: 0,1135 Precision: 0,1135 Recall: 0,1 F1 Score: 0,1063
10_100_1.0 Accuracy: 0,1135 Precision: 0,1135 Recall: 0,1 F1 Score: 0,1063	10_200_1.0 Accuracy: 0,0892 Precision: 0,0892 Recall: 0,1 F1 Score: 0,0943	10_300_1.0 Accuracy: 0,1135 Precision: 0,1135 Recall: 0,1 F1 Score: 0,1063

Observation:

- Accuracy increases with increasing rate, and then suddenly it drops!
- One round is not enough to get a result, which is independent from RNG seed.
    - Need 3 rounds to get averages.
- Range 0.01 to 0.1 need to be investigated again.

We simply add more parameters and already calculated settings are not calculated again. Based on the model-id (which is used as key in a model repository) we can simplify the iterative processing approach while we keep all existing results in a consistent database.

For each individual model we should plot a matrix from this data:

>>> Re-Evaluate model....
Examples labeled as 0 classified by model as 0: 968 times
Examples labeled as 0 classified by model as 1: 1 times
Examples labeled as 0 classified by model as 3: 1 times
Examples labeled as 0 classified by model as 4: 1 times
Examples labeled as 0 classified by model as 5: 4 times
Examples labeled as 0 classified by model as 6: 1 times
Examples labeled as 0 classified by model as 8: 3 times
Examples labeled as 0 classified by model as 9: 1 times

Examples labeled as 1 classified by model as 1: 1125 times
Examples labeled as 1 classified by model as 2: 1 times
Examples labeled as 1 classified by model as 3: 2 times
Examples labeled as 1 classified by model as 6: 3 times
Examples labeled as 1 classified by model as 7: 2 times
Examples labeled as 1 classified by model as 8: 2 times

Examples labeled as 2 classified by model as 0: 3 times
Examples labeled as 2 classified by model as 1: 3 times
Examples labeled as 2 classified by model as 2: 1009 times
Examples labeled as 2 classified by model as 3: 3 times
Examples labeled as 2 classified by model as 4: 2 times
Examples labeled as 2 classified by model as 6: 1 times
Examples labeled as 2 classified by model as 7: 8 times
Examples labeled as 2 classified by model as 8: 3 times

Examples labeled as 3 classified by model as 2: 3 times
Examples labeled as 3 classified by model as 3: 990 times
Examples labeled as 3 classified by model as 5: 6 times
Examples labeled as 3 classified by model as 7: 1 times
Examples labeled as 3 classified by model as 8: 9 times
Examples labeled as 3 classified by model as 9: 1 times

Examples labeled as 4 classified by model as 0: 1 times
Examples labeled as 4 classified by model as 2: 3 times
Examples labeled as 4 classified by model as 4: 953 times
Examples labeled as 4 classified by model as 6: 6 times
Examples labeled as 4 classified by model as 7: 3 times
Examples labeled as 4 classified by model as 8: 1 times
Examples labeled as 4 classified by model as 9: 15 times

Examples labeled as 5 classified by model as 3: 9 times
Examples labeled as 5 classified by model as 5: 873 times
Examples labeled as 5 classified by model as 6: 3 times
Examples labeled as 5 classified by model as 8: 7 times

Examples labeled as 6 classified by model as 0: 3 times
Examples labeled as 6 classified by model as 1: 2 times
Examples labeled as 6 classified by model as 2: 2 times
Examples labeled as 6 classified by model as 4: 3 times
Examples labeled as 6 classified by model as 5: 5 times
Examples labeled as 6 classified by model as 6: 941 times
Examples labeled as 6 classified by model as 8: 2 times

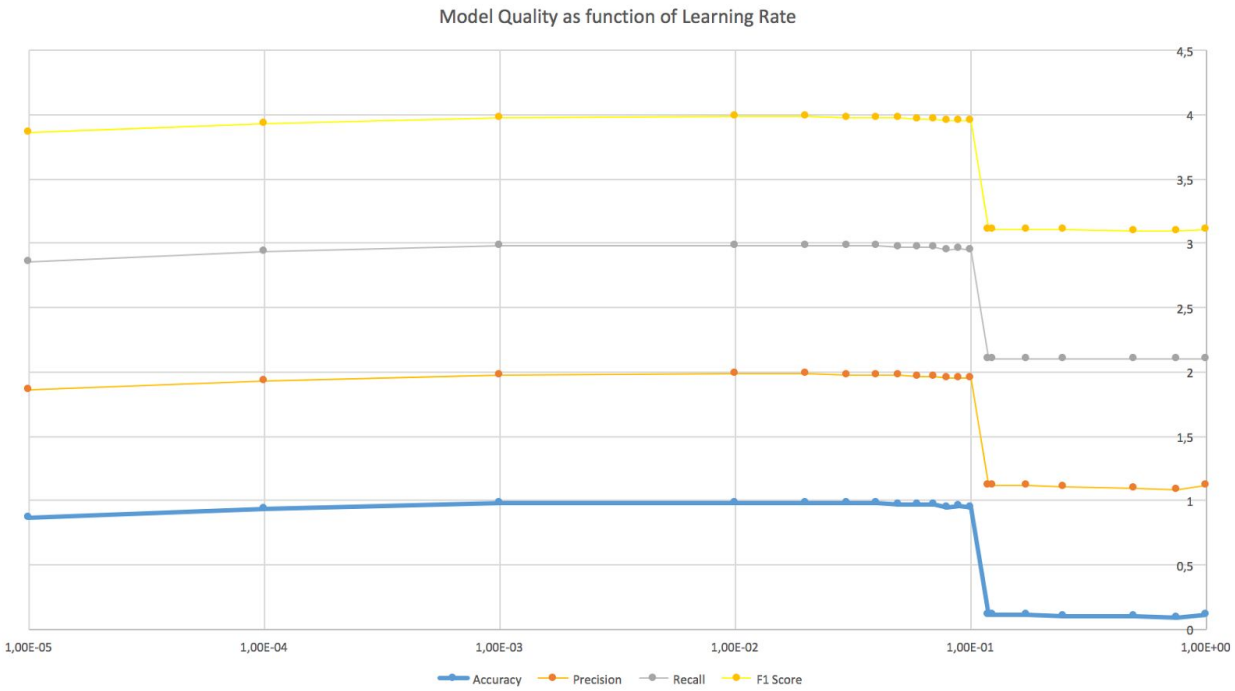
Examples labeled as 7 classified by model as 1: 3 times
Examples labeled as 7 classified by model as 2: 4 times
Examples labeled as 7 classified by model as 3: 3 times
Examples labeled as 7 classified by model as 5: 1 times
Examples labeled as 7 classified by model as 7: 1008 times
Examples labeled as 7 classified by model as 8: 2 times
Examples labeled as 7 classified by model as 9: 7 times

Examples labeled as 8 classified by model as 2: 1 times
Examples labeled as 8 classified by model as 3: 1 times
Examples labeled as 8 classified by model as 4: 1 times
Examples labeled as 8 classified by model as 5: 1 times
Examples labeled as 8 classified by model as 7: 3 times
Examples labeled as 8 classified by model as 8: 965 times
Examples labeled as 8 classified by model as 9: 2 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: 5 times
Examples labeled as 9 classified by model as 5: 5 times
Examples labeled as 9 classified by model as 6: 1 times
Examples labeled as 9 classified by model as 7: 2 times
Examples labeled as 9 classified by model as 8: 12 times
Examples labeled as 9 classified by model as 9: 974 times

Observation:

- Not all possible false positives exist. 3 is never classified as 6, but 3 times as two.



What can we learn from such a benchmark (Conclusions)?

=> More tests with more parameters needed.

Outlook

Next steps:

- Bring the model to production with real images (5 groups of photos)
- Show the limits of the trained model in the presence of changing input

TODO LIST:

- In a later edition we can illustrate the use of GPUs for training
- Implement the Flume-Interceptor
- Define a SOLR Schema and create the indexer-pipeline for Flume
- Implement the DL4J-DataSetIterator for Kudu
- Implement the ClassificationWrapper for usage in parallel in a Dataframe in Spark Shell.

[1] https://prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-network