Deep learning in action – with DL4J

Sigrid Keydana
Trivadis
München

Keywords
Deep Learning, Machine Learning, Artificial Intelligence, DL4J, Deeplearning4j, Java, Anomaly Detection

Introduction

In this second decade of the 21st century, and more so every year, we see deep learning - the "neural network" version of machine learning - shaping the world we live in. Colorization of black-and-white photos, auto-generated film music... autonomous driving, medical diagnosis... product recommendations, machine translation... Deep learning (DL, henceforth) has definitely entered our everyday life.

If we want to build our own models, the way to go seems to be well-known frameworks, like TensorFlow, Keras or PyTorch, that all come with a Python API. What if our whole application runs on the JVM (and probably is written in Java)? What if we require distributed computations and high performance? This is where DL4J enters the system.

DL4J is a feature-rich, very actively developed, well-documented DL framework for the JVM, maintained by San Francisco based company SkyMind. In this text, we'll highlight important features and demonstrate an extended use case for anomaly detection.

So what *is* Deep Learning?

Before we dive into DL4J, let's first ask – what even is deep learning? How is it distinct from classical machine learning (ML)?
Put simply, we can contrast rule-based programs, ML, and DL. In rule-based computation, we input data, process them using hard-coded rules, and output a result. In ML, we don't follow pre-specified rules, instead we learn them: That is, we learn functions that applied on the input, will yield the desired result. For example, we might learn a decision tree that says „if feature 1 is < 0.5 and feature 2 is > 0.22 and feature 3 is > 55, then this is a <insert target class here>".
In ML, these functions are learned, but the features they operate on are still given as an input to the algorithm. This is what changes with DL: Deep neural networks learn to extract features, too.
An obvious example is the Convolutional Neural Network (CNN) architecture (see fig. 1).
CNNs are predominantly used for tasks like image classification and segmentation. At different layers in the multi-layer structure, features of different granularity are extracted, starting from edges via contours to object parts and whole objects. This is much more effective than the hard-coding of features required for traditional ML. For example, in terms of pixel values, how should a "bike" feature look, where a bike can be anything from a race bike to an e-bike, in the sun or in the shadow, and partially or near totally obscured by the rider standing in front?

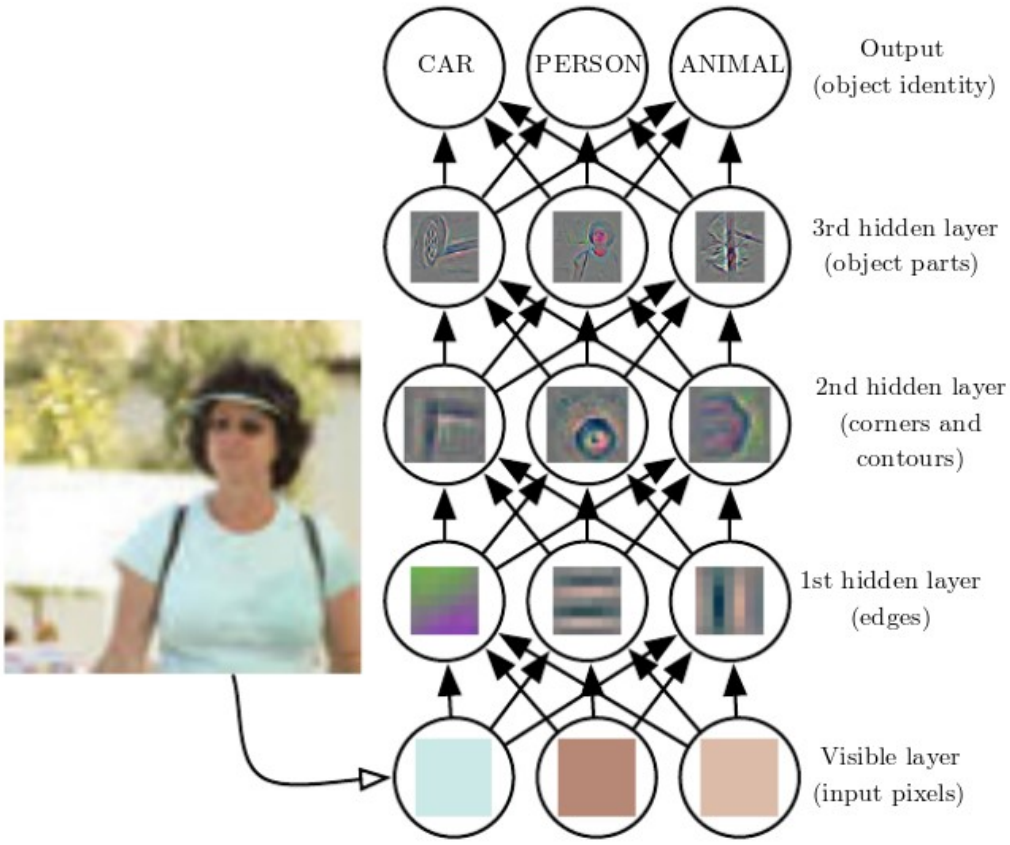

Fig. 1: Feature learning in a CNN. Source: Goodfellow et al., Deep Learning, MIT Press 2017.

How can that even work?
The next question then is, how can DL do this? In short, it relies on an algorithm called backpropagation. When training a network, we feed it the correct answer to learn from (this is called supervised learning). When told that the target is not dog, but a cat, it has to adjust its weights all throughout the network. This means that the error signal cannot just be used at the end, but must be propagated back all the way back to the very first weight matrix connecting input and the first hidden layer.
Mathematically, this is just the chain rule of calculus, but in practice, most frameworks use an optimization called “automatic differentiation” for this.

OK … but why should I care?
After the what and the how, let’s address the why. Our “DL Venn diagram” (fig. 2) shows 3 aspects. For one, DL is increasingly used in science, comprising such different disciplines as physics, finance, or pharmacology.
Secondly, in industry too we see DL being applied increasingly often and with increasing success: For example, machine translation, as of today, yields its best results when powered by neural networks. DL is state of the art in image recognition, classification and segmentation. Autonomous driving, one of

the dominant goals in AI, is deeply dependent on DL. But DL is also starting to be applied in such common fields as healthcare or manufacturing.
Thirdly, there is a lot of money involved. DL is bound to make progress as companies like Google, Microsoft, Facebook etc. are investing enormously in having the best scientists develop new algorithms and the best coders coding them up (and the best sales people selling them). Evidently, the success story of DL hast just only begun.

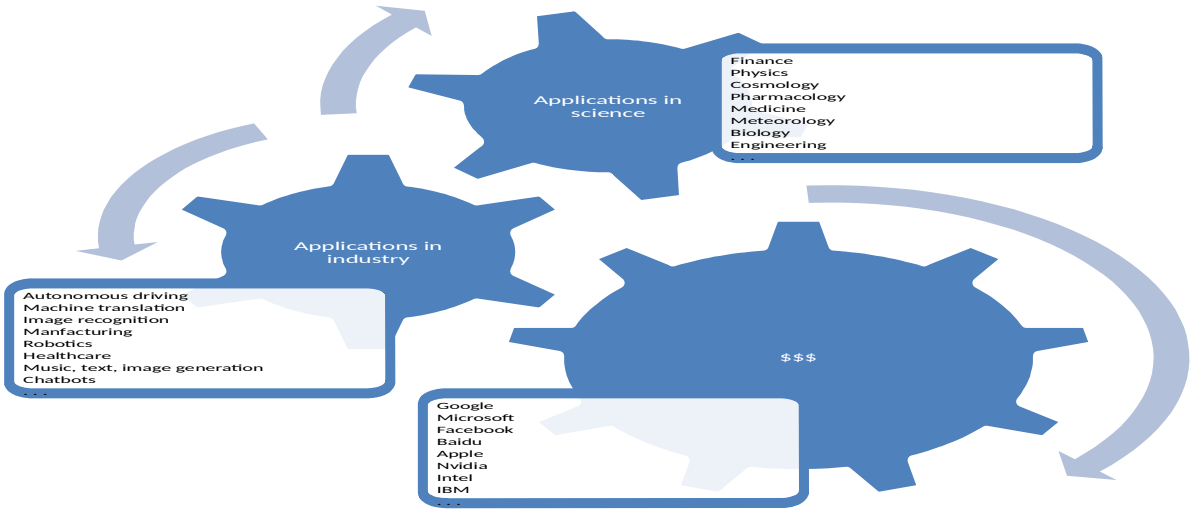

Fig. 2: A DL Venn diagram.

So… this is for Google, Microsoft and their likes, right?
Not necessarily. DL has been successfully applied by smaller companies, too. All you need is a reasonable amount of data, sufficient hardware and … the people who do the coding, of course!
So let's now zoom in on DL4J and some of its features.

Why haven't I heard about DL4J yet?
The best-known frameworks for DL are probably Keras, TensorFlow, Caffe, Torch (now PyTorch, too), Theano… most of them with a Python API and most of them developed by developers at Google or Facebook (which of course does a lot to enhance their fame). The company behind DL4J is SkyMind, a San Francisco startup founded in 2014. Skymind focuses on enterprise environments and distributed computing using Spark and Hadoop.

Developer's view
To get an concrete impression of DL4J, we're going to look at three specific scenarios. But first, let's see how to define and train a neural network in DL4J. The example will show a very simple network for linear regression.

How to code a simple network in DL4J
A basic deep neural network in DL4J is created as a *MultiLayerNetwork*, given a configuration containing parameters such as the optimization algorithm, the number of training epochs, the batchsize to use etc., as well as a list of layers with their activations. The data operated upon will be of type *NDArray*, provided by the ND4J scientific computing library also maintained by Skymind. Not surprisingly, after initialization of the network (*init*), *fit* then fits the model.

For our regression problem, we use Conjugate Gradient Descent (because it worked well on the data), ReLU activation on the hidden layer and no activation on the output layer (because it's regression). All that remains to be taken care of is getting the dimensions right (*numFeatures*, *hiddenDim*, *outputDim*).

```
public class LinearRegression {

    public static void main(String[] args) {

        int seed = 777;
        int numSamples = 64;
        int numFeatures = 1000;
        int hiddenDim = 100;
        int outputDim = 10;

        int numEpochs = 100;

        INDArray X = Nd4j.randn(numSamples, numFeatures);
        INDArray Y = Nd4j.randn(numSamples, outputDim);

        MultiLayerNetwork net = new MultiLayerNetwork(new NeuralNetConfiguration.Builder()
                .seed(seed)
                .iterations(numEpochs)
                .miniBatch(false)
                .optimizationAlgo(OptimizationAlgorithm.CONJUGATE_GRADIENT)
                .weightInit(WeightInit.XAVIER)
                .updater(Updater.SGD)
                .list()
                .layer(0, new DenseLayer.Builder().nIn(numFeatures).nOut(hiddenDim)
                        .activation(Activation.RELU)
                        .build())
                .layer(1, new OutputLayer.Builder(LossFunctions.LossFunction.MSE)
                        .activation(Activation.IDENTITY)
                        .nIn(hiddenDim).nOut(outputDim).build())
                .build());

        net.init();
        System.out.println(net.summary());
        net.setListeners(new ScoreIterationListener(10));

        net.fit(X, Y);

    }
}
```

Scenario 1: How do I quickly play around/experiment?

Now that we've seen how it basically works, let's think of the workflow. Imagine you're a developer (perhaps not a professional coder, but someone using DL for business or research questions). You've been doing DL in Python mostly for now, using Jupyter Notebook for quick prototyping. Put plainly, when you develop in a notebook, you can evaluate code blocks or even expressions by themselves, instead of running the whole program. In DL, this is incredibly useful when checking on tensor shapes.

This kind of checking on your code and quickly evaluating changes is less straightforward with Java (at least as of Java 8). Fortunately, DL4J has an import-from-Keras functionality that allows you to do model development in Keras and deploy with DL4J afterwards. (Of course, it might be that different groups in an organization do model development and productionization, respectively.)

Importing a model (including its weights) is as simple as

```
MultiLayerNetwork network = KerasModelImport.importKerasSequentialModelAndWeights(modelPath);
```

From here on, *network* can be used as any other *MultiLayerNetwork*.

Scenario 2: How do I use pretrained models?

DL4J comes with a *zoo* of pretrained models (mostly pretrained on ImageNet).

These include VGG16, VGG19, ResNet, GoogleNet and more.
To instantiate e.g. VGG16 – including the weights – and classify some image is straightforward:

```
ZooModel vgg16 = new VGG16();
ComputationGraph pretrainedNet = (ComputationGraph)
vgg16.initPretrained(PretrainedType.IMAGENET);

File file = new File(path);
NativeImageLoader loader = new NativeImageLoader(224, 224, 3);
INDArray image = loader.asMatrix(file);

DataNormalization scaler = new VGG16ImagePreProcessor();
scaler.transform(image);

INDArray[] output = pretrainedNet.output(false,image);
System.out.println(TrainedModels.VGG16.decodePredictions(output[0]));
```

The DL4J example code on github, which is extensive and fast growing, has several examples how to use and fine tune a pretrained network for your own image classes.

Scenario 3: Using a Variational Autoencoder (VAE) for anomaly detection
In DL, it's hard for the software to keep up with advances in algorithms proposed.
Recently, DL has also been applied to the challenging task of anomaly detection.
Anomaly detection is difficult, because it's inherently a poor fit for supervised learning: The nature of anomalies is such that you can't know them before.

In DL, a (formerly) popular, semi-supervised model for anomaly detection are *autoencoders*.
Autoencoders compress the input – due to smaller capacity of the central hidden layer(s) - and decompress it again. The logic then goes as follows: Anomalies are those data points which have high reconstruction error, because the model was trained on the majority and thus, has developed a compression-decompression algorithm that fits the common cases but fails on the outliers.

A more sophisticated model is the *variational autoencoder*, following Kingma and Welling's paper *Autoencoding Variational Bayes* (https://arxiv.org/abs/1312.6114).
This model has successfully been used for anomaly detection, as described in An & Cho's *Variational Autoencoder based Anomaly Detection using Reconstruction Probability* (http://dm.snu.ac.kr/static/docs/TR/SNUDM-TR-2015-03.pdf).
In constrast to the usual procedure of using *reconstruction error* to evaluate outliers/anomalies, An & Cho use *reconstruction probability*.
As far as I'm aware (and as of today, naturally), this reconstruction probability based method is implemented in no other DL framework besides DL4J. (Keras for example has sample code for VAE, but does not make VAE available as a special layer).
In DL4J though, adding a VariationalAutoencoder layer with a ReconstructionDistribution (that has to match the data) is all it takes:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
                    .learningRate(learningRate)
                    .updater(Updater.ADAM)
                    .weightInit(WeightInit.XAVIER)
                    .list()
                    .layer(0, new VariationalAutoencoder.Builder()
                            .activation(Activation.TANH)
                            .encoderLayerSizes(encoderSizes)
                            .decoderLayerSizes(decoderSizes)
                            .pzxActivationFunction(latentActivation)
                            .reconstructionDistribution(new
BernoulliReconstructionDistribution(Activation.Sigmoid))
                            .nIn(inputSize)
                            .nOut(latentSize)
                            .build())
                    .pretrain(true).backprop(false).build();
```

```
net = new MultiLayerNetwork(conf);
```

DL4J comes with example code demonstrating VAE on MNIST, the (in)famous handwritten digits dataset used abundantly in machine learning. We have tried applying VAE to a totally different kind of dataset – logs of network intrusions -, with mixed results (so far).

Conclusion

In this talk, the main motivation is to provide a feel for how practical deep learning may look with DL4J. In conclusion, I'd like to add that my personal experience with the DL4J ecosystem – including github issues and gitter chats – has been a really positive one so far.

Kontaktadresse:
Sigrid Keydana
Trivadis
Lehrer-Wirth-Strasse 4
D-81829 München

E-Mail sigrid.keydana@trivadis.com
Internet: www.trivadis.com