Bachelor's thesis

Bachelor of Engineering, Information and Communications Technology

2021

Martin Lehtomaa

IMAGE CLASSIFICATION USING DEEP LEARNING WITH JAVA

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

6.3.2021 | 38 sivua

Martin Lehtomaa

KUVANTUNNISTUS HYÖDYNTÄEN SYVÄOPPIMISTA SEKÄ JAVAA

Nykyajan ongelmien, kuten esim. konenäön ja luonnollisen kielen käsittelyjärjestelmien, ratkaisuissa hyödynnetään yhä enemmän kone- ja syväoppimisen menetelmiä. Tämän opinnäytetyön tavoitteena on näyttää ja tutkia, miten Javaa voidaan hyödyntää syväoppimisen sovelluksissa. Modernit syväoppimiseen pohjautuvat ratkaisut toteutetaan pitkälti Python- ja R-ohjelmointikielillä. Vaikka Java ei ole suosittu ohjelmointikieli tekoälyn ohjelmoinnissa, siinä on kattavat raamityökalut ja kirjastot kone- sekä syväoppimisen sovellusten kehitykseen. Javan tunnettuja vahvuuksia ovat ohjelmiston skaalautuvuus, tietoturvallisuus ja hyvä suorituskyky. Javaa käytetään mm. suurten yritystoimintojen järjestelmien sekä palvelinpuolen ratkaisujen kehityksessä. Suuren kehittäjäyhteisön omaavana Javalla on myös paljon avoimen lähdekoodin raamityökaluja ja kirjastoja. Yksi merkittävä tekoälyn ohjelmointiin tarkoitettu raamityökalu on DeepLearnin4J (DL4J). Tässä opinnäytetyössä keskityttiin yksinomaan Javan sekä DL4J:n käyttämiseen syväoppimiseen pohjautuvan kuvantunnistusmallin luomiseen ja hyödyntämiseen.

Demo-osiossa toteutetaan syväoppimiseen pohjautuva ratkaisu konenäön ydinongelmaan eli kuvantunnistukseen. Työssä näytetään, miten malli luodaan ja koulutetaan käyttäen Javaa ja DL4J. Tämän lisäksi näytetään, miten koulutettua mallia voidaan hyödyntää käytännössä erillisessä Java-pohjaisessa sovelluksessa. Toteutettu malli on VGG16:een pohjautuva ja koulutetaan suoriutumaan mekaanisten työkalujen kuvantunnistuksessa. Koulutettua mallia käytetään demoa varten toteutetun SpringBoot-pohjaisen REST-www-sovelluspalvelun kautta. Menetelmiä, joita hyödynnetään DL4J-raamityökalulla, ovat mm. siirto-oppiminen, ETL (Extract, Transform, Load) operaatioita (sis. Datan kasvattamista) sekä mallin koulutus.

Koulutetun mallin ja toteutetun web-palvelun tuloksista näkee kuinka vaivattomasti Java-pohjaisella ratkaisulla saa hyvin suoriutuvan mallin kuvantunnistukseen. Malli saavutti noin 85% tarkkuuden vain noin 30min:n koulutuksella ja rajatulla tietoaineistolla. Lisäksi mallin käyttämistä web-palvelun kautta näytti todenmukaisen tilanteen mallin hyödyntämisestä.

Mallin toteutuksen ja hyödyntämisen yhteydessä tehdyistä havainnoista ja saaduista tuloksista voi todeta, että Java on varteenotettava ohjelmointikieli tekoälyn ohjelmoinnissa. Java ja DL4J-raamityökalu tekevät sovellusten kehitystyön syväoppimisesta suoraviivaista ja tehokasta, varsinkin Java-pohjaisissa projekteissa.

ASIASANAT:

Java, DeepLearning4J, Kuvantunnistus, Syväoppiminen, Koneoppiminen, Siirto-oppiminen, Datan Kasvatus, Web Palvelu

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor of Engineering, Information and Communications Technology

6.3.2021 | 38 pages

Martin Lehtomaa

IMAGE CLASSIFICATION USING DEEP LEARNING WITH JAVA

Modern problems, like Computer Vision and Natural Language Processing (NLP) are nowadays tackled by leveraging Machine (ML) and Deep Learning (DL)-based solutions. This thesis strives to demonstrate the use of Java for the implementation of DL applications. While ML and DL solutions are heavily implemented with programming languages such as Python and R, there are other popular languages that provide comprehensive frameworks and libraries for these tasks. One of these languages is Java. Java provides application scalability, security and performance. Therefore it is used worldwide in enterprise grade systems and server-side applications. Java, having a huge developer community behind it, provides a bunch of open-source tools and frameworks, one of which is DeepLearning4J (DL4J) for DL applications. The work carried out in this thesis solely concentrates on the utilization of Java and DL4J for the creation and use of a DL-based image classification model.

The demonstration part includes the implementation of a solution for one of the core problems under the topic Computer Vision, which is Image Classification. The framework DL4J was leveraged in the implementation of a VGG16 -based model capable of mechanical tool image classification. The demonstration part also shows how the trained model can be utilized by an external Java-based application. The utilization of the model was implemented through a SpringBoot RESTful web service. Techniques applied with DLJ4 included Transfer Learning, ETL (Extract, Transform, Load) operations (inc. data augmentation) and model training.

The results and findings from the implementation of the DL model and web service verify how easy it is to become started and performant with a Java-based solution for Image Classification. The model reached ~85% accuracy in only ~30min of training on a limited dataset. In addition, a realistic use case scenario of the model was demonstrated by utilizing it from an external web service.

The trained model combined with the implemented web service form a complete DL and Java-based solution to tackle an Image Classification task specifically for mechanical tools. The solution demonstrates the potential Java indeed has in the field of DL. The conclusion confirms that the stack of Java and DL4J is a valid option for the development of DL based solutions. The DL4J framework makes the development process of a DL-based solution straight forward, especially for Java projects.

KEYWORDS:

Java, DeepLearning4J, Image Classification, Deep Learning, Transfer Learning, Machine Learning, Data Augmentation, Web Service

CONTENTS

FORMULAS

PICTURES

FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
CNN	Convolutional Neural Network
DL	Deep Learning
DL4J	DeepLearning4J
DTO	Data Transfer Object
ETL	Extract, Transform, Load
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
HATEOAS	Hypermedia as the Engine of Application State
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
ML	Machine Learning
NLP	Natural Language Processing
OOP	Object-Oriented Programming
ReLU	Rectified Linear Unit
REST	Representational state transfer
SGD	Stochastic Gradient Descent

1 INTRODUCTION

Over time Machine (ML) and Deep Learning (DL) solutions are leveraged more and more to solve complex problems (Maruti techlabs, n.d.). Popular programming languages like Python and R, are widely used for data analysis and statistical computing. In addition, they are also used in the fields of Machine and Deep Learning. Despite the popularity and high-level that Python and R has as programming languages, there are other valid options that should be considered (Springboard India, 2020a) many of which are developed by the open-source community. In the Java Virtual Machine (JVM) platform Java and Scala are both languages that provide comprehensive frameworks and libraries for Big Data, Machine – and Deep Learning. (Pathmind, n.d.)

There are professional developers worldwide specializing in Java. Furthermore, most of the platforms for large businesses are built with Java. Java provides application scalability, security and performance, therefore, it is used in enterprise and server-side applications. (Code Institute, n.d.) Java, having a huge developer community behind it, provides a wide range of open-source tools and frameworks, one of which is DeepLearning4J (DL4J) for DL applications (Eclipse Foundation, n.d.).

The objective of this thesis is to show that Java can contribute to the field of ML and DL. This objective is achieved by showing the implementation of a whole DL-based solution done with Java and DL4J framework. The whole solution includes the creation and training of a DL model, and how it can be utilized by an external application (note that external application means Web Service in this thesis). Often tutorials related to the implementation of a DL model only address how the model is created and trained. The work in this thesis strives demonstrate how the trained model can be utilized in practice in addition to how it is created.

The second chapter aims to unfold a comprehensive and theoretical background of image classification, DL and the techniques used and applied in image classification. It also includes a background of Java as a programming language and the introduction to frameworks available for the implementation of DL applications.

The third chapter introduces the tools and data that are leveraged in this thesis to demonstrate the use of Java and DL4J for DL applications. This is followed by the fourth chapter, which is a walkthrough of the development process and implementation of an image

classifier capable of classifying mechanical tools. It also demonstrates how the trained model can be utilized by a SpringBoot based representational state transfer (REST) web service.

In the fifth chapter the results of the implemented DL model and Web Service are viewed, in addition to some discussion on how the results were achieved. The results chapter starts by presenting what metrics are used in the creation and training phases of the model to determine the model's performance. The results also cover how the implemented Web Service can be consumed to perform a classification task by the model. In addition, it shows what data is returned to the consumer from the performed classification.

The sixth chapter, which is the conclusion, contains the author's reflections on how the result solution could be utilized in practice and how it could be further developed. It also includes core limitations of the implemented solution and discussion on how fellow developers could adopt the demonstrated techniques and tools on other projects.

The work carried out in this thesis solely concentrates on the implementation of DL applications with Java and the open-source DL4J framework. Chapter four (demonstration part of this thesis) only focuses on the creation and utilization of the DL model, so the implementation of the Web Service is outside the scope of this thesis.

2 DEEP LEARNING AND IMAGE CLASSIFICATION

2.1 Machine- and Deep Learning

Machine Learning (ML), being a subfield under artificial intelligence (AI) is a way to use algorithms to learn by processing structured and labeled data. Decisions and predictions produced by a ML algorithm is not programmed beforehand, but the output is rather performed and improved based on the data it has been trained against. Modern applications based on ML are recommendation systems, email spam detectors and digital voice assistants. (IBM Cloud Education, 2020a)

Deep Learning (DL) is a step deeper in the subject of AI. DL is a subfield of Machine Learning, and theories of DL have been around since 1943 (Keith D. Foote, 2017). DL is a way to make a model learn and improve its performance by processing a lot of data through multiple layers of algorithms called neural networks. DL differs from ML in the way that DL is capable to process and learn from unstructured and unlabeled data in addition to structured and labeled data. At its core, a DL model identifies and classifies data on its own by extracting learning features from the data. (IBM Cloud Education, 2020b)

The applications of DL are leveraged now more than ever for problem solving. DL models are used to handle tasks like self-driving cars, Natural Language Processing (NLP) (voice-controlled systems) and Computer Vision (e.g., Image Classification, Video processing). (IBM Cloud Education, 2020b)

2.1.1 Machine Learning Pipeline

In ML, the workflow of defining a problem all the way to creating a deployable model for a ML solution has a defined pipeline. This ML pipeline is an iterative process where some steps are repeated until the algorithm is successful. The ML pipeline typically constructs of the following steps: Problem Definition, Data Ingestion, Data Preparation, Data Segregation, Model Training, Evaluation, Deployment and Performance Monitoring (Picture 1.). (Koen, 2019)

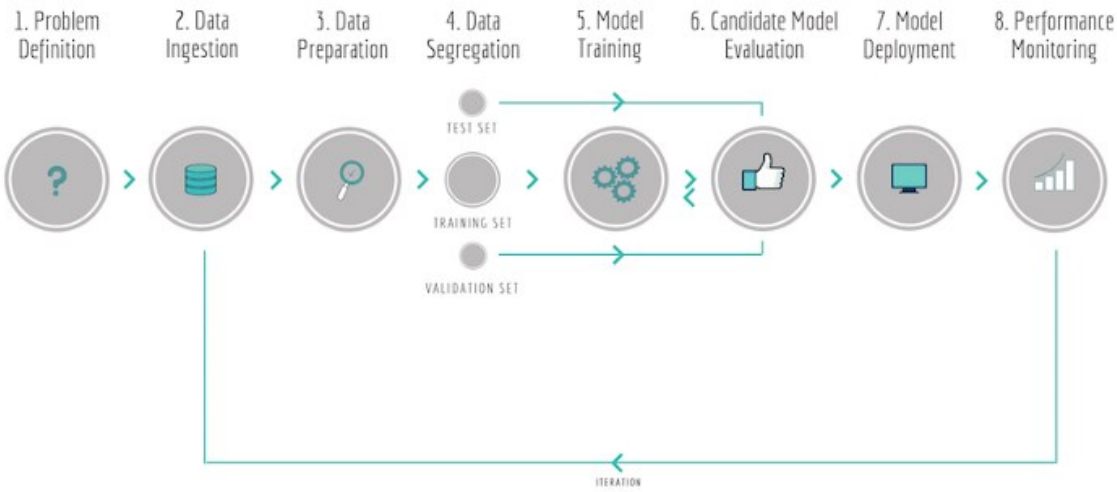


Figure 1. The ML Pipeline (Koen, 2019).

The pipeline starts with definition of the business problem at hand. This is followed by the step of Data Ingestion, which means the collection of data that is going to be used for the problem. When the data is obtained, it must be cleaned, processed and normalized before use. So, the third and most crucial step in the ML pipeline is Data Preparation. In practice this step includes removal of duplicate values, filling missing values and correcting other possible problems regarding the data. The fourth step is Data Segregation, which includes the splitting of the processed data into dedicated train, validation and test sets. Training set is used for training the model, while test and validation sets are leveraged to see how the model performs on unseen data. (Koen, 2019)

In the ML pipeline, by the fourth step, the data should be ready to use, and the fifth step of Model Training can begin. The model is trained against the training subset of the data. After training, the model's performance is evaluated iteratively against the test and validation subsets to see how it predicts unseen data. When the model is evaluated to perform as required, it can be deployed for use e.g., through an application programming interface (API) that help complete an analytics solution. Finally, when the model is deployed, it is monitored and incrementally improved on new data. (Koen, 2019)

2.1.2 Artificial Neural Networks

Artificial Neural Network (ANN), or Neural Network is a Machine Learning algorithm inspired by the way human brains function with a network of neurons (DeepAI, n.d.). A neural network consists of layers of neurons that process incoming data and passes produced outputs to other neurons. At its simplest, a neural network constructs of three layers, an input layer, one hidden layer and an output layer. This is called a “shallow” neural network (Missinglink.ai, n.d.a).

The output of a neuron is calculated based on a value called a weight and an activation function. Each neuron holds a value called a weight and calculates its output by multiplying the weight with the neurons input value. The calculated output is then passed through an activation function to the other neurons in the next layer. An activation function is used to determine the output of neural network. Furthermore, it determines if a neuron should be activated or not. In addition, it maps the output of a neuron to a value between 1 and 0 or between -1 and 1. E.g., an activation function can act based on a rule or threshold as a step function that switches the output of a neuron on and off. (Missinglink.ai, n.d.b)

There are three types of activation functions: binary step function, linear and non-linear activation functions. A binary step function activates and passes a neuron signal as is to the next layer if the input value exceeds a pre-defined threshold. Main cons are that it does not support multi-value outputs e.g., multilabel classification. (Missinglink.ai, n.d.b)

A linear activation function enables a neural network to produce multiple outputs. It produces an output proportional to the input. Linear activation functions main cons are related to cases when it is applied on multi-layer neural networks. The use of a linear activation function will make the last layer of a neural network a linear function of the first layer, so this basically makes a multi-layer network into a single-layer network. In addition, a technique called backpropagation cannot be applied on neural networks with linear activation functions. Linear activation functions restrict how complex data a neural network can handle. (Missinglink.ai, n.d.b)

Backpropagation is an algorithm applied on neural networks that enable tuning of neurons weights while training. After data has been fed through the network, backpropagation allows to go back and understand which neurons were involved in the produced prediction, and then change the weight values to generate a more accurate prediction.

Backpropagation puts the activation function on constant strain, so it is crucial that it is as efficient operation as possible. To allow a neural network to use backpropagation, it must use non-linear activation functions. (Missinglink.ai, n.d.b)

Non-linear activation functions are used in multi-layered neural networks. Because modern neural networks use backpropagation, non-linear activation functions are developed to be computationally efficient. With non-linear activation functions, a neural network can learn more complex data by creating mappings between inputs and outputs of the network. Complex data may include high dimensional data, images, video or audio. Some common non-linear activation functions are Sigmoid, TanH, Softmax and Rectified Linear Unit (ReLU). (Missinglink.ai, n.d.b)

Out of the earlier mentioned activation functions, only Softmax and ReLU will be covered, because they are leveraged in the implemented DL model, which is covered in chapter four of this thesis. ReLU is generally a computationally efficient activation function (Missinglink.ai, n.d.b). It is an activation function used in the hidden layers of a neural networks. ReLU simply outputs value of zero (0) if the input is negative, and whenever the input value is positive the value will be passed as is to the next layer. Softmax is an activation function used in the output layer of a neural network. Softmax outputs values for each output neuron, where each value represents a probability of class membership. The sum of the produced output values is always 1.0. (Brownlee, J., 2021)

2.1.3 Deep Neural Networks

As mentioned earlier, Deep Learning uses multiple layers of algorithms for data processing to produce an output of given input. DL is the utilization of a model with a structure similar to shallow ANN:s, but instead of one hidden layer, there are two or more hidden layers. These models are Deep Neural Networks (DNN) (Figure 1.) and can improve accuracy by adding more hidden layers up to 9-10. Modern DNN:s implemented have 3-10 hidden layers. (Missinglink.ai, n.d.b)

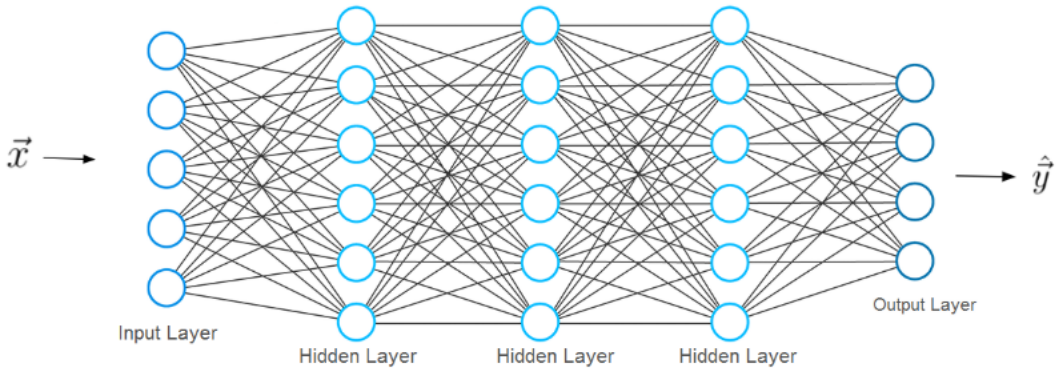


Figure 2. Deep Neural Network (Oppermann, 2019).

In DNN:s and ANN:s the role of the input layer is to receive data, while the output layers role is to produces the result. Generally, the number of neurons in an input layer corresponds to the number of features in the training data. The number of neurons in an output layer depends on the kind of problem that is to be solved. When it is a regression or a binary classification problem, the output layer has only one neuron. In cases of multi-label classification, when the output layer uses *Softmax* as an activation function, the number of neurons corresponds the amount of class labels in the model (this technique is applied in the demonstration of this thesis). As for hidden layers, the number of neurons, are commonly the same in all hidden layers. The optimal number of neurons can be calculated by dividing the amount of trading data samples with the multiplication of a tunable factor and the sum of input and output neurons (Formula 1.). The factor can be tuned in range of 1-10 to prevent over-fitting. (Malik, 2019)

$$Number\ Of\ Neurons = \frac{Trading\ Data\ Samples}{Factor * (Input\ Neurons + Output\ Neurons)}$$

Formula 1. Optimal amount of neurons in Hidden Layer (Malik, 2019).

2.2 Image Classification

Image Classification, being one of the fundamental problems under the topic Computer Vision, has different techniques in Machine and Deep Learning. Most common techniques to perform classification are Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Multi-Layer Preceptron (MLP) and Convolutional Neural Networks (CNN). Out of these techniques, classification performed by deep CNN:s have the highest accuracy and performance. (Hasabo, 2020)

Image Classification is the process where a computer model can identify an input image to a class that the image represents. E.g., when feeding an image of a cat to an image classification model, it would be able to classify and output a probability for a class label 'cat'. (ThinkAutomation, n.d.)

2.2.1 Convolutional Neural Networks for Image Classification

As mentioned earlier, one of the best ways to tackle image classification, is the utilization of CNN:s. CNN:s are generally more memory and computation heavy compared to the other techniques, but it depends on the architecture of the CNN and dataset size. To achieve great performance, it is crucial to design the CNN:s architecture according to the dataset in use. With manual configuration of hyperparameters the performance of the model can be tweaked. (Albelwi, Mahmood, 2017) CNN is a kind of Deep Neural Network, where the hidden layers are convolution, pooling, fully connected and normalization layers (Picture 2.) (Nigam, 2018).

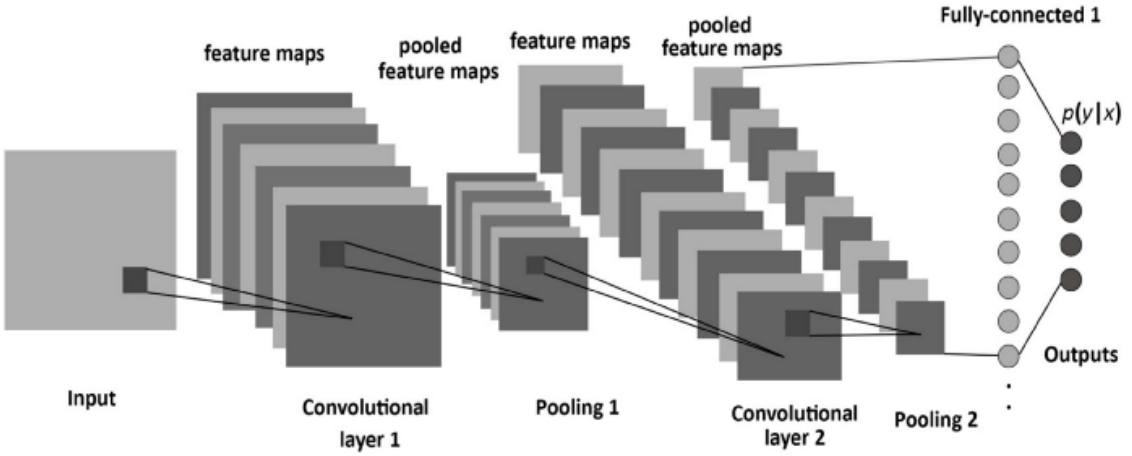


Figure 3. Structure of a CNN (Albelwi, Mahmood, 2017).

A convolutional layer acts as an activation function and uses filters (also known as kernels) on given input to perform feature extraction (Albelwi, Mahmood, 2017). As the level of convolution layers increase, the higher-level the detected features are (Chatterjee, 2019). The produced feature map of detected features is done by calculating the dot product from the input and filter (Picture 3.). In addition, a non-linear activation function is performed. Among non-linear activation functions, the most preferred is ReLU because it makes training take less time. (Albelwi, Mahmood, 2017)

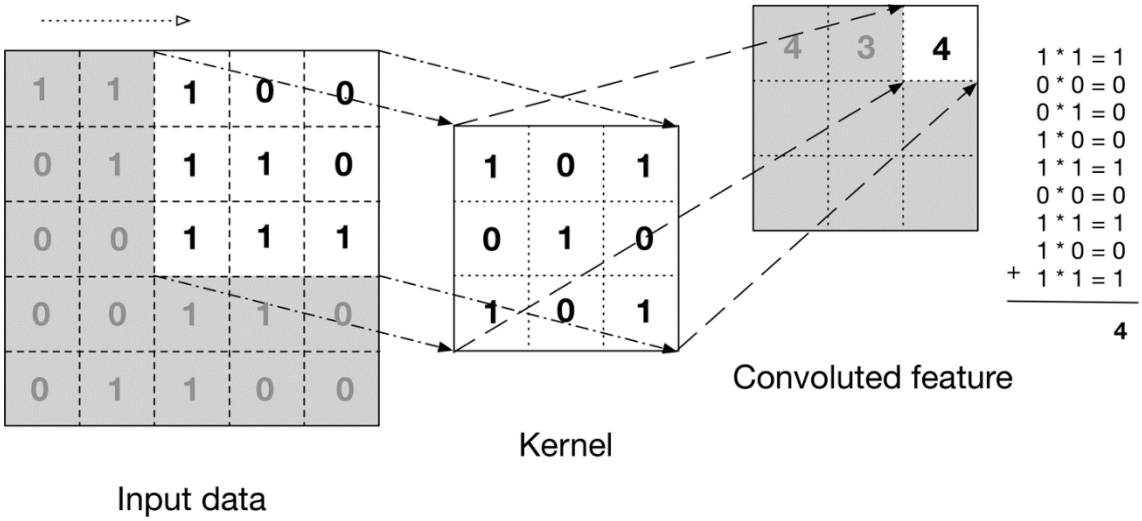


Figure 4. Convolution operation (Batista, 2018).

The pooling layer is used for dimensional reduction of the input. A pooling layer is commonly min, max or average pooling. This means that e.g., min pooling is used for picking the minimum pixel value from a region of a pixel matrix, hence the name “min pooling”. Max and average pooling works in the same manner, but as the names imply, max pooling selects the maximum pixel value and average pooling selects the average of all the pixel values in the region. Pooling operations can be visualized in the picture below (Picture 4.). (Basavarajaiah, 2019)

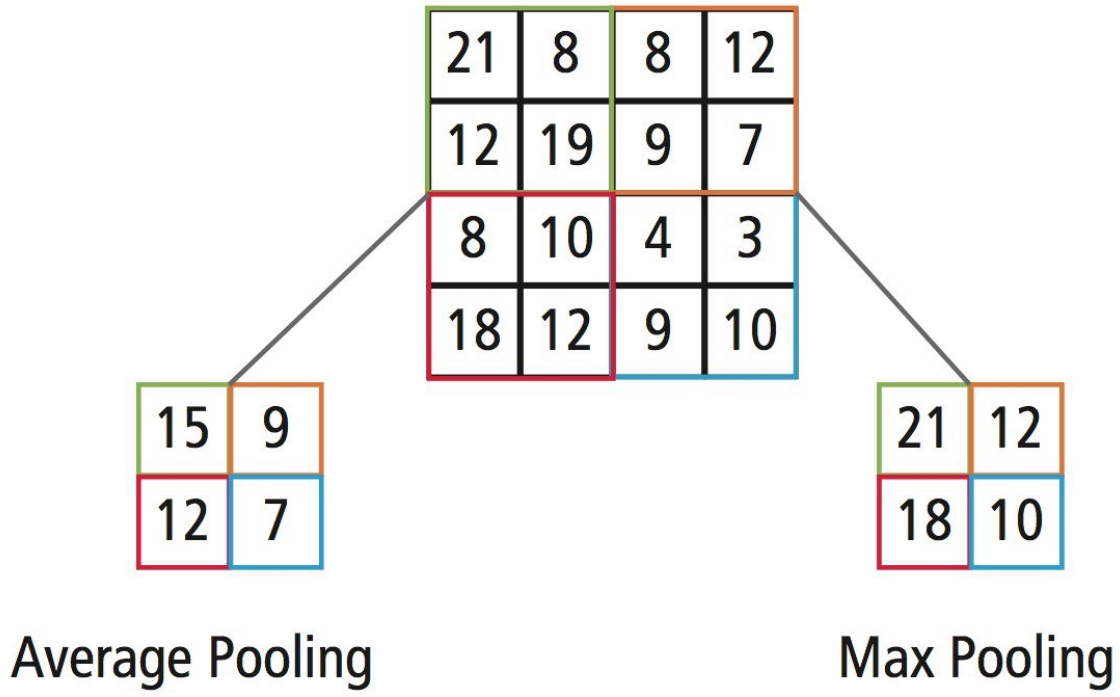


Figure 5. Average pooling and Max pooling operations (Cheng, 2017).

While CNN:s have pros like automatic feature extraction and high flexibility as the architecture can be designed according to case specific datasets and hyperparameter tuning, they also have drawbacks. Main cons of CNN:s are that they are complex, computation heavy and requires a lot of data to train from scratch to perform accurately. Fortunately, there are popular ready to use pre-trained models for classification available online that have been trained on huge datasets. There is a variety of these pre-trained models with different architectures, but most used and performant models are Xception (extension of Inception), VGG-16/VGG-19, ResNet50 and NASNet (and more). These listed pre-trained models have been trained on ImageNet dataset available from http://www.image-net.org/. (Saket, 2017) This thesis only addresses the use of VGG16 and how it is utilized for image classification with small datasets.

2.3 Pre-trained VGG16 and Transfer Learning

As mentioned before, one challenge when working with CNN:s is that training from them scratch will require a lot of data to achieve good accuracy. To train an accurate case specific image classifier with a small dataset we have pre-trained models like VGG16 available, and a concept called Transfer Learning.

VGG-16 is a deep CNN that was introduced by K. Simonyan and A. Zisserman from the University of Oxford in 2014 on a report named “Very Deep Convolutional Networks for Large-Scale Image Recognition” available from https://arxiv.org/pdf/1409.1556.pdf. In the article report one can find in-depth theory about VGG-16. VGG-16 achieved accuracy of 92.7% on the ImageNet dataset. In short, VGG-16 model is 16 layers deep with 1000 class labels and the ImageNet dataset consists of 14 million RGB images with the size of 224*224. This means that the models input tensor must be in a dimension of (224, 224, 3) and that the output layer has 1000 neurons (one for each class) as seen in the picture below of VGG-16 architecture (Picture 5.). (GeeksForGeeks, 2020)

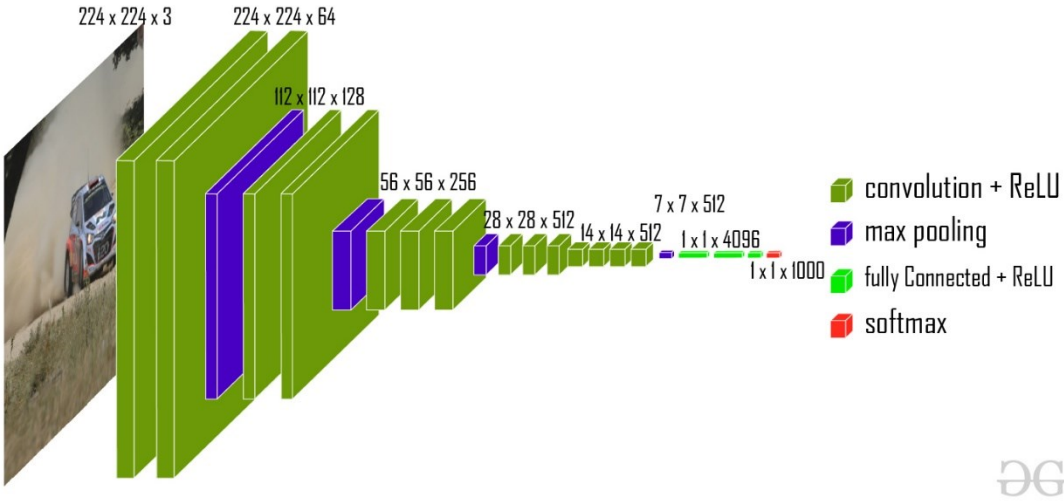


Figure 6. Architecture of VGG-16 (GeeksForGeeks, 2020).

The VGG16 model has also its own dedicated pre-processor to help reduce data complexity. According to the documentation and published report of VGG16, the VGG16 pre-processor subtracts the mean RGB value from each pixel computed on the training set (Simonyan, K., Zisserman, A., 2015).

VGG-16 takes 528Mb of disk space, so it should be considered when estimating and allocation storage space for a project that uses it. (GeeksForGeeks, 2020)

To leverage the layers of a pre-trained model, like VGG-16, for a new classification task, a technique called Transfer Learning can be applied. Transfer Learning means reusing the weights, or in other words taking the features learned from a model that has been trained on a problem similar to the new at hand. When Transfer Learning has been applied and weights have been initialized in the new model, the only things left to do is fine-tune configuration of the weights and model training on the case specific dataset. (Brownlee, 2020) In practice fine-tuning means freezing the lower-level layers (close to input), that hold generic features (e.g., edge detection), and retrain rest of the higher-level layers on the model. It is also common to add new convolutional layers to be trained in the retraining process. This results the fine-tuning of the model's weights in the higher-level layers to learn features according to the new classification task. (Sarkar, 2018) The main benefits of Transfer Learning are the increase of generalization and less time spent on training (Brownlee, 2020). This technique is leveraged when the available dataset for solving the problem is too small.

2.4 Data Augmentation

When the dataset for a new classification task is too small, Data Augmentation can be used. Data Augmentation is a technique to generate more data from the dataset that is in use. It is possible to even 10x the dataset with Data Augmentation. In addition, a CNN may avoid learning irrelevant features and patterns by using augmentation. (Gandhi, n.d.) Common augmentation techniques used on images to generate new data are positional augmentation and color augmentation. Positional augmentation are modifications like scaling, cropping, flipping, padding, rotation, translation and affine transformation. Color augmentation includes modification of brightness, contrast saturation and hue. There are also more advanced ways to produce more data form the space of the initial dataset. Techniques like Generative Adversarial Network (GAN) and Neural Style Transfer are highly used. (Kumar, n.d.)

Data Augmentation is applied in the DL pipeline on the data before it is given as input to the model. This can be done in two ways called offline augmentation or online augmentation. Offline augmentation means that the transformations are performed on the whole dataset before feeding it to the model. Online augmentation means that the transformations are made on the fly to images right before being fed to the model. (Gandhi, n.d.)

2.5 Java for Deep Learning

The programming languages used to develop Machine and Deep Learning applications are mostly done with Python or R. Python is even more used than R because of its simplicity as a programming language and huge community built around it in the field of Machine Learning. Despite the popularity of Python, there is still other valid options that developers should consider to use more. (Springboard India, 2020a) Java Virtual Machine (JVM) languages like Java and Scala are also highly used in the field of Big Data, ML and DL. Furhtermore, Java has a huge developer community behind it and as a programming language brings great performance, application scalability and security. (Pathmind, n.d.)

Both Java and Python have their places when considering to leverage DL for solving a problem. Python is more used for experimental use and to quickly get started on developing the system. With Python, less resources are spent on the software engineering aspects of the implementation and rather consentrated on the heuristics related to DL. (Bhatia, 2018) Python has a bunch of libraries and frameworks related to Deep Learning and the most popular ones are TensorFlow, Keras and PyTorch (Bhattacharyya, 2020).

Java on the otherhand, is used for enterprise applications. It is not as much used in DL, because of its steep learning curve and verboseness as a programming language compared to Python. These characteristics of Java makes people to draw assumptions that it would be unnecessarily complex to use it for DL. Java has great libraries for DL to ease the implementation process and still achieve great performance. Most common libraries used in Java for DL are DeepLearning4J (DL4J, target of this thesis) and Deep Java Library (DJL) (Baeldung, 2020).

2.5.1 Java

Java is a general-purpose Object-Oriented programming (OOP) language developed by James Gosling at Sun Microsystems in 1990s. Java is cross-platform which implies, that a Java source code file (.java) can be compiled once into byte code (.class) then and ran on any platform with JVM. (Thereaderwiki, 2020) Java is used mostly on enterprise applications for large businesses that require ease of scalability, high performance and application security. Java also supports concurrent processing with a concept named multithreading. Furthermore, these aspects make Java a preferred choice for the implementation of server-side applications.

Generally, Java is faster in runtime compared to Python because it is a compiled language while Python is interpreted. Java is a statically typed language and therefore known as a very verbose language. (Sayantini, 2020) Statically typed means that all type-checking is performed by the compiler and possible errors are found before runtime. In contrast, Python is a dynamically typed language, which means that type-checking is made at runtime on the fly by CPU. This makes Python and other similar languages less performant and more error prone. (JBallin, 2017).

One of the reasons why Python is in favor over Java, is because one can simply start writing executable code starting from the first line with Python. Java requires definition of classes and methods in-order to get started. This is one of the main reasons why Java it is not considered as a good choice for complex Deep Learning applications. (Springboard Inidia, 2020b) Fortunately, there is an open-source library named DeepLearning4J (DL4J), that makes the implementation of such applications more convenient and straight forward (Eclipse Foundation, n.d.).

2.5.2 DeepLeanring4J Framework

DeepLearning4J is a commercial-grade open-source framework that was released under Apache license 2.0 in 2014 by contributors from San Francisco and Tokyo. DL4J became a part of Eclipse Foundations property in October 2017 and then was contributed to the open-source Java Enterprise Edition library ecosystem. (Rao, 2017) DL4J was developed and intended for JVM based deep learning applications. It can be used with all the

popular JVM based languages like Java, Scala, Kotlin and Clojure. (Eclipse DL4J Repository, 2015)

DL4J provides a developer the tools to handle the whole Deep Learning pipeline from data preparation to the construction of simple or complex CNN:s. The great performance of DL4J is achieved mainly because it uses C, C++ and Cuda as a backend for the underlying computations. DL4J supports the utilization of GPU:s to decrease time spent on training a model. For distributed computing it leverages Apache Spark and Hadoop. DL4J is also highly flexible because it is possible to import Keras models that are saved in h5 format. As of from 1.0.0-beta7 version of DL4J, it also supports importing of tf.keras models. (Eclipse DL4J Repository, 2015)

The main stack of libraries available in the framework are DL4J, ND4J, SameDiff, DataVec, Arbiter and LibND4J. From this collection of libraries, DL4J contains the high-level API (Application Programming Interface) to build Deep Learning models. The main classes used for the creation of a neural network are *MultiLayerNetwork* and *ComputationGraph*. All the scientific computations like mathematical, linear algebra and deep learning operations are in the ND4J library, which is based on LibND4J. LibND4J is a highly optimized C++ engine that handles the processing of n-dimensional arrays from ND4J for Java. The handling of ETL for data used on a Deep Learning project is done by DataVec library. DataVec supports file formats like HDFS, Spark, Images, Video, Audio, CSV, Excel and many others. (Eclipse DL4J Repository, 2015)

2.5.3 RESTful Web Services with Java

The strengths of Java as a server-side language are utilized in the work done in this thesis by creating a representational state transfer (REST) web service, from which an image classification model can be used.

A REST web service is also known as a REST application programming interface (API). An API allows to integrate application software without having to know how the implementation of the communication between two systems (or more) is made. API:s ease the designing and integration of new features into existing systems, and help to maintain security and control of a resource. (RedHat, n.d.a)

A REST API is a standardized way for information exchange. It is a specification where an API has REST architectural constraints. The core constraints include having Client-Server architecture, being stateless, utilization of caching, having a layered system and being a uniform interface. (RedHat, n.d.a) A REST API supports transferring of many different formats through HTTP, including JavaScript Object Notation (JSON), HTML, XLT, Python, PHP or plain text. Out of these formats, JSON is the most popular one. (RedHat, n.d.b)

A REST web service can be made Hypermedia as the Engine of Application State (HATEOAS) driven. As a small improvement to a typical REST application architecture, HATEOAS enables dynamic navigation between resources of a web service. Navigation is done through hypermedia links that are sent back to the client in the HTTP response body. In addition of returning the requested data, the resource returns links to other related actions. With HATEOAS the service consumer does not need prior knowledge of the service because of the guidance provided by the hypermedia links. (Karanam, 2019)

There is a variety of frameworks available for Java that allow the building of a REST API. Some frameworks available include Spring Boot, Spring (MVC), Quarkus, Apahe CFX, Jersey, RESTEasy, Restlet, Micronaut, TomEE and Dropwizard. (Brannan, 2021)

2.5.4 Picocli Command Line Framework

In this thesis, a VGG16 based image classification model is created and trained via a command line tool to demonstrate the use of DL4J components. The command line tool is implemented with a lightweight framework available for Java called Picocli.

Main benefits of Picocli, is within its flexibility, because it does not need to be included as an external dependency. Picocli source is all in one file, so it can be included as source. In addition, it supports different types of command line syntax styles like POSIX, GNU, MS-DOS and others. (Picocli, 2021)

2.5.5 Maven for Java Projects

Apache Maven Project is a tool created to simplify a Java-based projects build process, keep the build system uniform and provide quality project information. A maven project is built according to a project object model (POM) file that a maven projects holds. The POM file (POM.xml) contains information of the project. The POM includes information like what dependencies the project uses, possible build plugins that are leveraged and cross-referenced sources (among other beneficial information). Another important aspect of Maven is that it provides guidelines on how a project's layout and directory structure should be. This enables the ease of navigation in new projects that use Maven. (Apache Maven Project, n.d.)

3 DATASET AND TOOLS

To demonstrate the use of Java and DL4J for a DL solution, an image classification model capable of classifying mechanical tools is implemented. The dataset used for the training of the image classification model is available at Kaggle https://www.kaggle.com/salmaneunus/mechanical-tools-dataset. Classifiable objects are Gasoline Can, Hammer, Pebbles, Rope, Screwdriver, Toolbox, Wrench and Pilers. The model is deployed and used through a Java Representational state transfer (REST) web service.

3.1 Initial dataset setup

The Mechanical Tool dataset fetched from Kaggle is split according to a common split percentage, which is 80% for training and 20% for test sets (80/20). All data in both training and test sets are organized into dedicated subdirectories according to tool type (pictures of hammers are in a directory named “Hammer” etc.). In addition of having a separate training and test set, the training set is split into actual training set and validation set (80/20). The splitting of training and validation set is done internally by the implemented Java application, which performs the model training. Data augmentation is applied only for the actual training set that is fed to the model.

3.2 Frameworks used for the RESTful Web Service

A Java EE framework named Spring Framework is leveraged in the implementation of the REST web service. To make the process of setting up the server application simple, SpringBoot is used to auto-configure the application setup.

The created Java application is a Maven project. This eases the handling of dependencies, like the use of SpringBoot and other libraries. All the dependencies are configured in the *pom.xml* file, which is located under project root. Maven projects have clear folder structure as default and helps to organize the architecture of the project.

The trained model is uploaded in a directory named “saved” under the resources folder. Because it is a Maven project, the default resources folder is found under the source folder that is in root (project_name/src/main/resources/saved).

The Tool Classification REST web service is made HATEOAS driven. Spring HATEOAS is used in the implementation of HATEOAS support.

3.3 Command Line Tool for Image Classification Model Training

The creation and training of the VGG16 based tool classification model is performed and executed via a Picocli based command line tool created in this thesis for the demonstration of DL4J. To execute the command line model trainer, it can be provided with an optional argument (-p, -P or --path) to specify where the trained model will be saved. As default, the model will be saved under *C*:/*<userPath>/image_recognition_model/.* The model will be saved as a compressed .zip file. After obtaining the compressed model, it is ready to use and can be deployed/uploaded to the tool classification web service.

This command line tool is a Maven project, which helps the dependency management of Picocli and DL4J components. In addition, it leverages a maven-assembly build plugin, which enables the building of the project into an executable Java Archive (JAR) file containing all necessary dependencies. The build plugin allows to use the command line tool as a exportable and executable JAR file.

4 DEVELOPMENT WITH DL4J

The walkthrough of the development process with Java and DL4J framework is done by introducing the components used in the creation of the image classification model trainer. This includes showing the actual implementation and use of different DL4J components for the creation of the image classification model. In addition, this section shows the implementation on how the web service leverages the trained model with DL4J components.

The workflow for the demonstration is going as follows. First the image classification model trainer is utilized to create a compressed .zip file of the new VGG16 based model. When the model is obtained, the SpringBoot application can be executed to start the web service. Before starting the web service, the SpringBoot application uploads the compressed model into the projects *resources* folder, where it can then be used by the web service.

4.1 Creation of the Tool Classification Model

Most of the implementation regarding DL4J framework is done withing the command line tool project "*Model_train_tool"*. The projects main application to run the Picocli command line tool is in a java file named "*ModelTrainerTool.java"*. (Figure 2.)

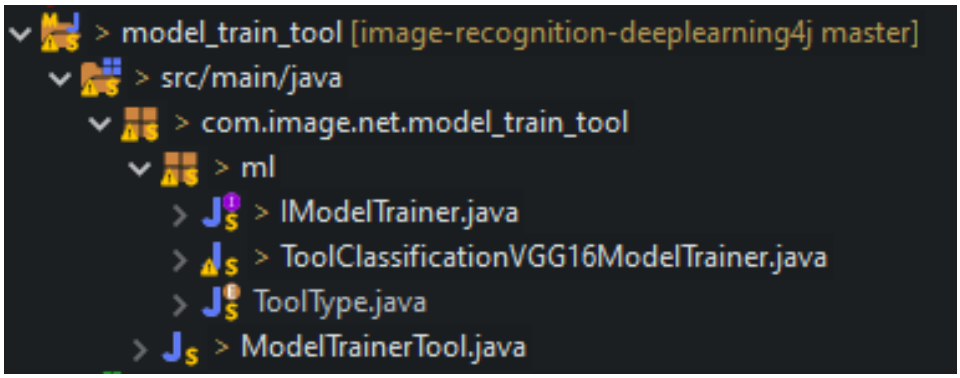


Figure 7. Project structure of Model_Train_Tool.

The main application uses an interface named *"IModelTrainer.java"* to perform the creation and training of the compressed model. (Figure 6.)

All the source code regarding the implementation of the DL pipeline can be found under the package "*com.image.net.model_train_tool.ml*". This package holds a java class file named "*ToolClassificationVGG16ModelTrainer.java*" that implements and is used through the java interface *"IModelTrainer.java"* (Figure 5.). The class "*ToolClassificationVGG16ModelTrainer"* contains the implementation of creation, training and saving of the tool classification model by utilizing DL4J components.

4.1.1 Main application

The main application class *ModelTrainerTool* is decorated with an annotation *@Command*, which indicates that it is Picocli command line executable application.

```
ModelTrainerTool.java
 1  package com.image.net.model_train_tool;
 2
 3  import java.io.IOException;
11
12
14  * A command line tool to train a VGG16 based Image classification model.
27  @Command(name = "ModelTrainerTool",
28          description = "Trains and saves a Mechanical tool classification model to desired location."
29                  + " <br>Default location: C:/users/userPath/image_recognition_model."
30                  + " <br>Optional argument: -p, -P or --path can be used to specify location.")
31  public class ModelTrainerTool implements Runnable {
32
```

Figure 8. ModelTrainerTool main application decorated with Picocli @Command annotation.

With the *@Command* annotation, the class can also utilize an annotation *@Option* that allows the application to use command line arguments. The *@Option* annotation maps an optional command line argument to its dedicated class field variable (Figure4.). In this demonstration it enables the user to specify a desired location to which the application will read and write data from/to regarding the DL pipeline (location for reading datasets and model saving). The arguments can be set with optional argument *--path, -p or -P*. *As* default the fields value is initialized to hold the path to directory named *image_recognition_model* under local user folder (*C:/users/userPath/image_recognition_model*).

```
30
32      * Option to set custom path to dataset and model data saving location
36      @Option(names = {"-p", "-P", "--path"})
37      public static String dataSaveLocation = System.getProperty("user.home") + "\\image_recognition_model";
38
```

Figure 9. Optional command line arguments enabled with @Optional.

The main class *ModelTrainerTool* implements *Runnable* interface so that Picocli CommandLine object can be leveraged to execute the program in a new thread. When executed by Picocli, a implemented method named *run()* is called (Figure 5.). The method *run()* acts as the driver of the program and is implemented from the *Runnable* interface.

```
38
39      public static void main(String[] args) {
40          int exitCode = new CommandLine(new ModelTrainerTool()).execute(args);
41          System.exit(exitCode);
42      }
43
44      @Override
45      public void run() {
46          IModelTrainer modelTrainer = new ToolClassificationVGG16ModelTrainer(dataSaveLocation);
47          try {
48              modelTrainer.initPreTrainedModelWithTransferLearning();
49              modelTrainer.trainPretrainedModel();
50          } catch (IOException e) {
51              e.printStackTrace();
52          }
53      }
```

Figure 10. Picocli executes ModelTrainerTool, which calls *run()* -method driver code.

4.1.2 Transfer Learning and creating a model

As mentioned earlier, the Java class used to perform the model's creation and training with DL4J is named *"ToolClassificationVGG16ModelTrainer.java"*. It has two main methods that can be used through its interface *"IModelTrainer.java": initPreTrainedModelWithTransferLearning()* and *trainPretrainedModel()* (Figure 6.).

IModelTrainer.java

```
1  package com.image.net.model_train_tool.ml;
2
3  import java.io.IOException;
4
5  public interface IModelTrainer {
7       * Initializes the pre-trained ZooModel.
11     public void initPreTrainedModelWithTransferLearning() throws IOException;
12
14      * Performs the training of the model.
21     public void trainPretrainedModel() throws IOException;
```

Figure 11. IModelTrainer -interface.

The method *initPreTrainedModelWithTransferLearning()* initializes the pre-trained model (Figure 7.) and performs the transfer learning to the new model (Figure 9.). In addition, it fine-tunes the new model. DL4J core library contains a dedicated class "*VGGBuilder*" (used through *org.deeplearning4j.zoo.model.VGG16.builder()*) that is used to create an object instance of a pre-trained VGG16 model. All pre-trained model instances from DL4J are class type of *ZooModel* (org.deeplearning4j.zoo.ZooModel).

The ZooModel class object itself is not meant to be programmatically manipulated and therefore needs to be converted to a neural network object representation. To do so it can be instantiated to an object instance type of *ComputationGraph* (org.deeplearning4j.nn.graph.ComputationGraph) which indeed is the programmable neural network class object.

```
137         ZooModel zooModel = VGG16.builder().build();
138         preTrainedNet = (ComputationGraph) zooModel.initPretrained(PretrainedType.IMAGENET);
139
```

Figure 12. Initializing a pre-trained VGG16 model with DL4J.

The fine-tuning of the new model is performed by applying a configurable object instance type of *FineTuneConfiguration* (org.deeplearning4j.nn.transferlearning.FineTuneConfiguration) to it (Figure 8.) (Figure 9.). Again DL4J has a dedicated builder class (used through *org.deeplearning4j.nn.transferlearning.FineTuneConfiguration.Builder.Builder()*) that makes the usage of the object straight forward. In the demonstration the *FineTuneConfiguration* object is used to define the optimization algorithm to be SGD (Stochastic Gradient Descent) and then to leverage Nesterov's momentum to keep the gradient updated.

```
144             FineTuneConfiguration fineTuneConf = new FineTuneConfiguration.Builder()
145                     .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
146                     .updater(new Nesterovs(5e-4, Nesterovs.DEFAULT_NESTEROV_MOMENTUM))
147                     .seed(seed)
148                     .build();
```

Figure 13. Fine-tuning configuration for the new model.

Transfer Learning is performed by constructing and initializing a new *ComputationGraph* object based on the pre-trained model. To apply the defined fine-tuning for the new model, the *FineTuneConfiguration* object is passed to its builder in the object construction phase. DL4J has a dedicated API for Transfer Learning and through it a

GraphBuilder class (*org.deeplearning4j.nn.transferlearning.TransferLearning.GraphBuilder*) is leveraged to build the new *ComputationGraph*. Again, DL4J has a good implementation of the common builder pattern, which makes the construction of such complex and highly configurable object effortless and readable.

```
149             vgg16Transfer = new TransferLearning.GraphBuilder(preTrainedNet)
150                     .fineTuneConfiguration(fineTuneConf)
151                     .setFeatureExtractor(FREEZE_UNTIL_LAYER)
152                     .removeVertexKeepConnections("predictions")
153                     .addLayer("predictions",
154                             new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
155                                     .nIn(4096)
156                                     .nOut(OUTPUT_LABELS)
157                                     .weightInit(WeightInit.XAVIER)
158                                     .activation(Activation.SOFTMAX)
159                                     .build(),
160                             FREEZE_UNTIL_LAYER)
161                     .build();
```

Figure 14. Applying Transfer Learning and constructing the new model with *TransferLearning.GraphBuilder* object.

The new model is given a custom output layer holding the predictable class labels for all eight mechanical tools (Gasoline Can, Hammer, Pebbles, Rope, Screwdriver, Toolbox, Wrench and Pilers). All layers up to the last fully connected layer (fc2) are frozen on the model to keep the weights fixed while training. The activation function used on the output layer is defined to be *Softmax* (Figure 9.).

All parameters regarding Transfer Learning and fine-tuning are defined as static class field variables to make the accessing of these parameters efficient.

4.1.3 ETL operations

The model is trained in three epochs, which means that the whole training dataset has been fed through the network three times. The dataset is not passed as whole, but rather fed in small batches with the size of 16. All ETL operations like defining and splitting datasets, including data augmentation are done with components provided from the DL4J DataVec library.

The paths for the datasets are first instantiated into object instances type of *FileSplit* (*org.datavec.api.split.FileSplit*). Furthermore, the *FileSplit* objects are used for getting the

datasets as *InputSplit (org.datavec.api.split.InputSplit)* instances that hold the actual lists of loadable file locations. To get the datasets as traversable objects, they are constructed into instances *DataSetIterator (org.nd4j.linalg.dataset.api.iterator.DataSetIterator).* The iterator is initialized according to the provided *InputSplit* paths. (Figure 10.)

```
176             File trainData = new File(TRAIN_FOLDER);
177             File testData = new File(TEST_FOLDER);
178             FileSplit train = new FileSplit(trainData, NativeImageLoader.ALLOWED_FORMATS, RAND_NUM_GEN);
179             FileSplit test = new FileSplit(testData, NativeImageLoader.ALLOWED_FORMATS, RAND_NUM_GEN);
180
181             InputSplit[] trainDataSample = train.sample(PATH_FILTER, TRAIN_SIZE, 100 - TRAIN_SIZE);
182             DataSetIterator trainIterator = getDataSetIterator(trainDataSample[0], true);
183             DataSetIterator validIterator = getDataSetIterator(trainDataSample[1], false);
184             DataSetIterator testIterator = getDataSetIterator(test.sample(PATH_FILTER, 1, 0)[0], false);
```

Figure 15. Defining dataset file paths and dataset iterators.

A method named *getDataSetIterator()* (*com.image.net.model_train_tool.ml.ToolClassificationVGG16ModelTrainer.getDataSetIterator(InputSplit, boolean))* is implemented to construct and initialize the iterator (Figure 11.). The class instance of the constructed *DataSetIterator* is type of *RecordReaderDataSetIterator (org.deeplearning4j.datasets.datavec.RecordReaderDataSetIterator.RecordReaderDataSetIterator()).* The iterator is equipped with a image pre-processor dedicated for a VGG16 model *(org.nd4j.linalg.dataset.api.preprocessor.VGG16ImagePreProcessor)*.

```
218    public DataSetIterator getDataSetIterator(InputSplit sample, boolean withDataAugmentation) throws IOException {
219        ImageRecordReader imageRecordReader = new ImageRecordReader(HEIGHT, WIDTH, CHANNELS, LABEL_GENERATOR_MAKER);
220        imageRecordReader.initialize(sample);
221        if (withDataAugmentation) {
222            List<ImageTransform> transforms = dataAugmentation();
223            for (ImageTransform transform : transforms) {
224                imageRecordReader.initialize(sample, transform);
225            }
226        }
227
228        DataSetIterator iterator = new RecordReaderDataSetIterator(imageRecordReader, BATCH_SIZE, 1, OUTPUT_LABELS);
229        iterator.setPreProcessor(new VGG16ImagePreProcessor());
230        return iterator;
231    }
```

Figure 16. Constructing a DataSetIterator.

Because the images in training and test datasets are organized into subdirectories by their tool type, the label for an image is generated according to the parent directory it is in. A class named *ParentPathLabelGenerator* (*org.datavec.api.io.labels.ParentPathLabelGenerator)* is used to perform the label generation. In this *ToolClassificationVGG16ModelTrainer* class the label generator is assigned to a dedicated static field named *LABEL_GENERATOR_MAKER* (seen in the picture above).

Data augmentation is also applied at this point by the method *getDataSetIterator()* depending on the given *boolean* flag argument (Figure 11.). The augmented transforms applied on the data include simple flipping, rotation and wrapping (Figure 12.).

```
235    private List<ImageTransform> dataAugmentation() {
236        ImageTransform flipTransform1 = new FlipImageTransform(RNG);
237        ImageTransform flipTransform2 = new FlipImageTransform(RNG_2);
238        ImageTransform warpTransform = new WarpImageTransform(RNG, 42);
239        ImageTransform rotationTransform = new RotateImageTransform(RNG, 90);
240
241        return Arrays.asList(new ImageTransform[] { flipTransform1, warpTransform, flipTransform2, rotationTransform });
242    }
```

Figure 17. Data augmentation transforms.

4.1.4 Training the model

The training of the model is performed in a while loop by incrementing the number epochs after each time the whole training set has been fed. There is also an inner while loop which feeds the dataset in the earlier defined 16 size batches to the model. The model is evaluated every ten iterations against the validation set and after every epoch against the test set. Finally, after three epochs the model is saved as a .zip file. (Figure 13.)

```
190            while (iEpoch < EPOCH) {
191                while (trainIterator.hasNext()) {
192                    DataSet trained = trainIterator.next();
193                    vgg16Transfer.fit(trained);
194                    if (i % EVAL_INTERVAL == 0 && i != 0) {
195                        evalModelOn(validIterator, i);
196                    }
197                    i++;
198                }
199                trainIterator.reset();
200                iEpoch++;
201                evalModelOn(testIterator, iEpoch);
202            }
203            ModelSerializer.writeModel(vgg16Transfer, new File(MODEL_VGG16_SAVING_PATH + "/tool_classification_model_vgg16.zip"),
204                    false);
```

Figure 18. Training and evaluation of the model.

4.2 Running the Model Trainer Tool

4.2.1 Initial setup

Before running the model trainer, there is some setup required to be done regarding the dataset and folder structure. The dataset is already properly split into training and test sets, they need to be uploaded to the right dedicated location. As default the application will read and write data from/to a folder named *image_recognition_model* under user path (*C:/Users/userPath/image_recognition_model*). The training and test sets (directories *train_all* and *test_all*) are placed under a directory named *tool_data* (*../image_recognition_model/tool_data*) as seen in the figure below (Figure 14.).

```
C:\Users\Saitamas PC\image_recognition_model>dir /ad /b /s
C:\Users\Saitamas PC\image_recognition_model\tool_data
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Gasoline Can
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Hammer
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\pebbel
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Pilers
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Rope
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Screw Driver
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Tool box
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Wrench
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Gasoline Can
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Hammer
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\pebbel
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Pliers
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Rope
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Screw Driver
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Tool box
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Wrench

C:\Users\Saitamas PC\image_recognition_model>
```

Figure 19. Initial folder structure.

After setting up the directories for the datasets the initial setup is complete, and the command line tool can be executed. The model trainer tool will create new directories named *saved* and *log* under the base directory *image_recognition_model*. The directory *image_recognition_model/saved* is where the compressed model is saved by the tool. The directory *image_recognition_model/log* contains a log file with all logging entries produced while executing and running the model trainer tool. The image below shows the complete folder structure after the tool is executed (Figure 15.).

```
C:\Users\Saitamas PC\image_recognition_model>dir /ad /b /s
C:\Users\Saitamas PC\image_recognition_model\log
C:\Users\Saitamas PC\image_recognition_model\saved
C:\Users\Saitamas PC\image_recognition_model\tool_data
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Gasoline Can
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Hammer
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\pebbel
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Pilers
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Rope
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Screw Driver
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Tool box
C:\Users\Saitamas PC\image_recognition_model\tool_data\test_all\Wrench
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Gasoline Can
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Hammer
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\pebbel
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Pliers
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Rope
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Screw Driver
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Tool box
C:\Users\Saitamas PC\image_recognition_model\tool_data\train_all\Wrench

C:\Users\Saitamas PC\image_recognition_model>
```

Figure 20. Folder structure after model trainer is executed.

4.2.2 Executing the model trainer

Before execution, the project is packaged into a JAR file containing all necessary dependencies, which enables to use it as an exportable tool. To build the JAR file of the project, the following maven commands are executed through command line in the project root (*model_train_tool/*); ***mvn clean*** followed by ***mvn package***. As a result, a JAR file (among other build related files and directories) named ***model-train-tool-jar-with-dependencies.jar*** is produced and located under project root in a folder */target (model_train_tool/target/model-train-tool-jar-with-dependencies.jar).*

The JAR can be executed from project root through command line with default path location for the dataset reading and model saving:

java -jar target/model-train-tool-jar-with-dependencies.jar

Or with specified location by providing ***-p*** argument:

java -jar target/model-train-tool-jar-with-dependencies.jar -p C:/users/myUser/files/recognition_model

When the program has finished training, it produces the compressed model *image_recognition_model/saved/**tool_classification_model_vgg16.zip*** (or in other specified location). The compressed model is ready for deployment and use for the REST web service.

4.3 Tool Classification REST Web Service

The SpringBoot server application for the classification REST web service is implemented as a basic Rest Controller. It contains two *GET* endpoints for the consumer: ***/api/tools/detect*** and ***/api/tools/labels***.

The endpoint ***/api/tools/detect*** performs the classification. The HTTP request sent to the endpoint has to contain an image file (key: *imageFile*, formats: tif, jpg, png, jpeg, bmp, JPEG, JPG, TIF, PNG) that the classification is performed on. As a response the web service returns a Data Transfer Object (DTO) containing the label which the image is classified as and rest of the label predictions. The predictions are returned as *double* data type to represent percentage, so the value for a prediction is between 0-1 (1.0 = 100%). In addition of returning the classification and prediction, the response contains HATEOAS links to the ***/api/tools/labels*** endpoint and the base entry point for the web service.

The endpoint ***/api/tools/labels*** simply returns a list of all classifiable labels.

4.3.1 Using the compressed model

The Rest Controller uses a *ToolRecognitionService* -interface to get the predictions. The class implementation *ToolRecognitionServiceImpl* deserializes and initializes the compressed model as a *ComputationGraph* into a dedicated class field (Figure 17.). The predictions are then be obtained as a vectorized representation holding a prediction for each output label (eight elements). The predictions are returned from the service as a

Map data structure containing the tool type as a key and the prediction as a value (*Map<ToolType, Double>*). (Figure 16.)

The dimensions of an image file sent in the request must match the models input layers dimensions (224, 224, 3). Before using the image to get predictions, it is converted to a vectorized representation with the required dimensions of height 224, length 224 and channels 3 for RGB values (224, 224, 3). In addition, a dedicated VGG16 pre-processor used on it. (Figure 16.)

```
44      @Override
45      public Map<ToolType, Double> detectTool(File file) throws IOException {
46          if (computationGraph == null) {
47              computationGraph = loadModel();
48          }
49
50          computationGraph.init();
51          LOGGER.info(computationGraph.summary());
52          INDArray image = imageLoader.asMatrix(new FileInputStream(file));
53          scaler.transform(image);
54
55          INDArray output = computationGraph.outputSingle(false, image);
56          return getPredictions(output);
57      }
```

Figure 21. Using the model for a classification task.

```
51      private ComputationGraph loadModel() throws IOException {
52          computationGraph = ModelSerializer.restoreComputationGraph(new File(TRAINED_MODEL_PATH));
53          return computationGraph;
54      }
```

Figure 22. Method used for restoring the compressed model as *ComputationGraph*.

5 EVALUATION AND RESULTS OF TOOL CLASSIFICATION MODEL

5.1 Metrics and evaluating the model

Before getting into the actual evaluation of the new VGG16 based model, one needs to verify the architecture of the new model. This includes checking that it has the eight outputs, and that all layers excluding the output and input layers are frozen (as defined earlier in chapter four). The architecture of the new model is verified through auditing a log entry of the model's structure before the actual training process. A comprehensive summary of the models architecture is logged with *summary()* -method provided by the *ComputationGraph (org.deeplearning4j.nn.graph.ComputationGraph.summary()).*

```
Jan 27, 2021 12:10:55 AM com.image.net.model_train_tool.ml.ToolClassificationVGG16ModelTrainer initPreTrainedModelWithTransferLearning
INFO:
================================================================================================================
VertexName (VertexType)                  nIn,nOut     TotalParams   ParamsShape                  Vertex Inputs
================================================================================================================
input_1 (InputVertex)                    -,-          -             -                            -
block1_conv1 (Frozen ConvolutionLayer)   3,64         1,792         W:{64,3,3,3}, b:{1,64}       [input_1]
block1_conv2 (Frozen ConvolutionLayer)   64,64        36,928        W:{64,64,3,3}, b:{1,64}      [block1_conv1]
block1_pool (Frozen SubsamplingLayer)    -,-          0             -                            [block1_conv2]
block2_conv1 (Frozen ConvolutionLayer)   64,128       73,856        W:{128,64,3,3}, b:{1,128}    [block1_pool]
block2_conv2 (Frozen ConvolutionLayer)   128,128      147,584       W:{128,128,3,3}, b:{1,128}   [block2_conv1]
block2_pool (Frozen SubsamplingLayer)    -,-          0             -                            [block2_conv2]
block3_conv1 (Frozen ConvolutionLayer)   128,256      295,168       W:{256,128,3,3}, b:{1,256}   [block2_pool]
block3_conv2 (Frozen ConvolutionLayer)   256,256      590,080       W:{256,256,3,3}, b:{1,256}   [block3_conv1]
block3_conv3 (Frozen ConvolutionLayer)   256,256      590,080       W:{256,256,3,3}, b:{1,256}   [block3_conv2]
block3_pool (Frozen SubsamplingLayer)    -,-          0             -                            [block3_conv3]
block4_conv1 (Frozen ConvolutionLayer)   256,512      1,180,160     W:{512,256,3,3}, b:{1,512}   [block3_pool]
block4_conv2 (Frozen ConvolutionLayer)   512,512      2,359,808     W:{512,512,3,3}, b:{1,512}   [block4_conv1]
block4_conv3 (Frozen ConvolutionLayer)   512,512      2,359,808     W:{512,512,3,3}, b:{1,512}   [block4_conv2]
block4_pool (Frozen SubsamplingLayer)    -,-          0             -                            [block4_conv3]
block5_conv1 (Frozen ConvolutionLayer)   512,512      2,359,808     W:{512,512,3,3}, b:{1,512}   [block4_pool]
block5_conv2 (Frozen ConvolutionLayer)   512,512      2,359,808     W:{512,512,3,3}, b:{1,512}   [block5_conv1]
block5_conv3 (Frozen ConvolutionLayer)   512,512      2,359,808     W:{512,512,3,3}, b:{1,512}   [block5_conv2]
block5_pool (Frozen SubsamplingLayer)    -,-          0             -                            [block5_conv3]
flatten (FrozenVertex)                   -,-          -             -                            [block5_pool]
fc1 (Frozen DenseLayer)                  25088,4096   102,764,544   W:{25088,4096}, b:{1,4096}   [flatten]
fc2 (Frozen DenseLayer)                  4096,4096    16,781,312    W:{4096,4096}, b:{1,4096}    [fc1]
predictions (OutputLayer)                4096,8       32,776        W:{4096,8}, b:{1,8}          [fc2]
----------------------------------------------------------------------------------------------------------------
            Total Parameters:  134,293,320
        Trainable Parameters:  32,776
           Frozen Parameters:  134,260,544
================================================================================================================
```

Figure 23. New VGG16 based model architecture to which Transfer Learning is applied, logged with summary() -method.

The audited log entry of the model summary above (Figure 18.) verifies that the new model has frozen layers until the last fully connected layer (fc2 layer). The column *VertexType* contains information about what type a layer is and if it is frozen or not. The summary also shows that the new model has a new custom layer for predictions containing eight outputs. The output layer can be identified by locating the layer with *VertexType* of *OutputLayer* (layer name *predictions*). Other beneficial metrics displayed by the summary, are the dimensions that each layer has and in what dimensions the input data for each layer has to be (columns *nIn,nOut* and *ParamsShape*).

When the model's architecture is verified, the evaluation can begin. The model is evaluated mostly according to its prediction accuracy calculated against the validation and test datasets. To get comprehensive log entry of the evaluation metrics, the *ComputationGraph* has a *evaluate()* -method *(org.deeplearning4j.nn.graph.ComputationGraph.evaluate(DataSetIterator)),* which enables evaluation against provided dataset (Figure 19.). In addition of the prediction accuracy, it also displays a confusion matrix that help visualize the distribution of predictions performed on the dataset (Figure 19.).

```
222     private void evalModelOn(DataSetIterator iterator, int iter) throws IOException {
223         LOGGER.info("Evaluate model at iteration " + iter + " ....");
224         Evaluation eval = vgg16Transfer.evaluate(iterator);
225         LOGGER.info(eval.stats());
226         iterator.reset();
227     }
```

Figure 24. Model evaluation method.

```
INFO: Evaluate model at iteration 3 ....
Jan 27, 2021 12:32:14 AM com.image.net.model_train_tool.ml.ToolClassificationVGG16ModelTrainer evalModelOn
INFO:

========================Evaluation Metrics========================
 # of classes:    8
 Accuracy:        0.8667
 Precision:       0.8704
 Recall:          0.8667
 F1 Score:        0.8640
Precision, recall & F1: macro-averaged (equally weighted avg. of 8 classes)


=========================Confusion Matrix=========================
  0  1  2  3  4  5  6  7
-------------------------
 11  0  0  0  1  1  1  1 | 0 = Gasoline Can
  0 13  1  0  1  0  0  0 | 1 = Hammer
  0  2 10  0  0  0  2  1 | 2 = Pilers
  0  0  0 15  0  0  0  0 | 3 = Rope
  0  2  2  0 11  0  0  0 | 4 = Screw Driver
  0  0  0  0  0 15  0  0 | 5 = Tool box
  0  0  0  0  1  0 14  0 | 6 = Wrench
  0  0  0  0  0  0  0 15 | 7 = pebbel

Confusion matrix format: Actual (rowClass) predicted as (columnClass) N times
==================================================================
```

Figure 25. Evaluation metrics and confusion matrix on test dataset at the end of epoch 3.

The model's performance is monitored by following the metrics logged from the evaluation performed on both validation and test sets. This data was leveraged to prevent overfitting and some tweaking of the hyperparameters. Tweaked hyperparameters include such as learning rate of Nesterov's updater, different weight initialization techniques (Xavier vs Normal distribution), effect of batch size and number of epochs.

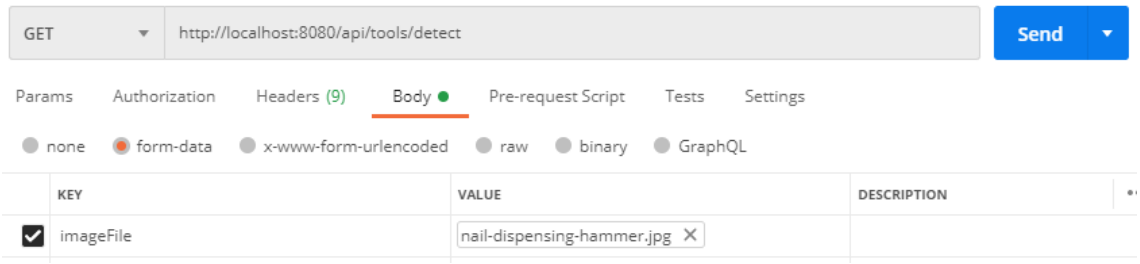
The above figure (20.) shows the final statistics evaluated of the model's performance on the test set. In three epochs the model has a reached accuracy of ~86%, which is quite good considering that the training on CPU took only ~30min and the dataset available was limited.

The training of a model from scratch against the same dataset would take longer time and the accuracy would not reach more than ~40-50% (tested without proper tweaking of hyperparameters).

5.2 Testing the Tool Classification Web Service

To test the implemented Tool Classification web service, a HTTP GET request containing an image file is sent. The following request is sent with Postman to the Tool Classification REST endpoint with a picture of a nail dispensing hammer in request body (Picture 6.).

Picture 1. A nail dispensing hammer to test the classification.



Picture 2. Postman GET request to test classification of a hammer.

The web service returns a HTTP response body in JSON format (Figure 21.).


```
{
    "classifiedAs": {
        "type": "HAMMER",
        "probability": 0.7750087380409241
    },
    "predictions": {
        "WRENCH": 0.1983461230993271,
        "PEBBELS": 4.062337029608898E-5,
        "TOOLBOX": 1.84110103873536E-4,
        "ROPE": 2.4799152015475556E-5,
        "SCREW_DRIVER": 4.824756761081517E-4,
        "GASOLINE_CAN": 4.0155587157642E-6,
        "PILERS": 0.025909164920449257,
        "HAMMER": 0.7750087380409241
    },
    "_links": {
        "self": {
            "href": "http://localhost:8080/api"
        },
        "all-classification-labels": {
            "href": "http://localhost:8080/api/tools/labels"
        }
    }
}
```


Figure 26. Hammer classification response body returned from Tool Classification Web Service.

The goal is that the model could classify the image as a hammer. It can be confirmed that the classification model and web service indeed work and is able to classify the nail dispensing hammer as a HAMMER (Figure 21.). Rest of the predictions gives some in-

sight data of other possible classifications. The web service classifies an item as a specific tool type if the prediction is > 0.5 (50%). Here is also displayed how the leveraged Softmax activation function of the model returns the values of each output as a represented prediction of class membership (output values sum is 1.0).

6 CONCLUSION

The objective of this thesis was to demonstrate how and why Java and DL4J framework should be considered as a good option for the implementation of an image classifier and other DL related tasks. The core concepts of DL and image classification were introduced as a starting point. The theoretical part was followed with an introduction to Java's strengths as a programming language and the core libraries that DL4J offers for developers to implement DL projects.

The application of these powerful tools was demonstrated with an implementation of a command line tool that trains a VGG16 based mechanical tool image classifier with DL4J. The creation of the mechanical tool classifier included techniques such as transfer learning, fine-tuning, ETL operations (inc. data augmentation), evaluation and practical ways on how to tweak the model hyperparameters. The usage of the model was demonstrated through a SpringBoot RESTful web service that utilizes the model to perform the tool classification task on a given image.

The demonstration part demonstrates the ease of use of DL4J libraries due to their high-level implementations. The trained model reached accuracy of ~85% within ~30min of training on a limited dataset. The model was capable of classifying tools through the web service and give the right predictions as response. The main limitations regarding the application are that it is only capable of detecting a small variety of tools. A more scaled version of this classifier and service could be utilized e.g., as a customer service helper. A consumer could use it to query the availability of some product by providing an image through camera and the helper would recognize and response with the availability.

All techniques applied with DL4J on the result can be implemented on any Java-based project. Java experts or developers may find it easier to integrate state-of-the-art DL-based solutions into a project when the programming language stays the same. This benefits also further development so that any fellow Java developer on the project can make changes and additions if needed.

To further develop the classifier, a GAN could be implemented to handle data augmentation and boost the training with a limited dataset. Also, some modifications and additions could be made regarding the layers of the CNN. Layer modifications would include

the addition of a dense layer, the change of the feature extraction layer and unfreezing different layers to enable later layers to be tweaked properly during training.

REFERENCES

Albelwi, S. and Mahmood, A., 2021. 'A Framework for Designing the Architectures of Deep Convolutional Neural Networks' *Entropy*, [Online]. 19 (6), 242. Available from: https://www.mdpi.com/1099-4300/19/6/242/htm [Accessed 9 January 2021]

Apache Maven Project, n.d., *'What is Maven?'*, Available from: https://maven.apache.org/what-is-maven.html [Accessed 4 March 2021]

Baeldung, 2020, '*Overview of AI Libraries in Java*', Available from: https://www.baeldung.com/java-ai [Accessed 29 December 2020]

Basavarajaiah, M., 2019, '*Maxpooling vs minpooling vs average pooling*', Available from: https://medium.com/@bdhuma/which-pooling-method-is-better-maxpooling-vs-minpooling-vs-average-pooling-95fb03f45a9 [Accessed 9 January 2021]

Batista, D. S., 2018, '*Convolutional Neural Networks for Text Classification*', Available from: http://www.davidsbatista.net/blog/2018/03/31/SentenceClassificationConvNets/

Bhatia, R., 2018, *'Why Do Data Scientists Prefer Python Over Java?*', Available from: https://analyticsindiamag.com/why-do-data-scientists-prefer-python-over-java/ [Accessed 6 January 2021]

Bhattacharyya, J., 2020, *'Popular Deep Learning Frameworks: An Overview'*, Available from: https://analyticsindiamag.com/deep-learning-frameworks/ [Accessed 18 February 2021]

Brannan, J., 2021, *'Top 10 Best Java REST and Microservice Frameworks (2021)'*, Available from: https://rapidapi.com/blog/top-java-rest-frameworks/ [Accessed 4 March 2021]

Brownlee, J., 2020, *'How to Improve Performance With Transfer Learning for Deep Learning Neural Networks*', Available from: https://machinelearningmastery.com/how-to-improve-performance-with-transfer-learning-for-deep-learning-neural-networks/ [Accessed 10 January 2021]

Brownlee, J., 2021, *'How to Choose an Activation Function for Deep Learning'*, Available from: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/ [Accessed 1 March 2021]

Chatterjee, C. C., 2019, '*Basics of the Classic CNN*', Available from: https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add [Accessed 9 January 2021]

Cheng, A.J., 2017, '*Convolutional Neural Network*', Available from: https://medium.com/@Aj.Cheng/convolutional-neural-network-d9f69e473feb [Accessed 10 January 2021]

Code Institute, n.d., *'What is Java and why is it important?'*, Available from: https://codeinstitute.net/blog/what-is-java/ [Accessed 17 February 2021]

DeepAI, n.d., *'What is a Neural Network?'*, Available from: https://deepai.org/machine-learning-glossary-and-terms/neural-network [Accessed 27 February 2021]

Eclipse DL4J Repository, 2015, '*DeepLearning4J*' (Repository README.md file), Available from: https://github.com/eclipse/deeplearning4j [Accessed 12 January 2021]

Eclipse Foundation, n.d., *'Deep Learning for Java'*, Available from: https://deeplearning4j.org/ [Accessed 17 February 2021]

Foote, K. D., 2017, '*A Brief History Of Deep Learning',* Available from: https://www.dataversity.net/brief-history-deep-learning/ [Accessed 28 December 2020]

Gandhi, A., n.d., '*Data Augmentation | How to use Deep Learning when you have Limited Data—Part 2*', Available from: https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/ [Accessed 11 January 2021]

GeeksForGeeks, 2020 '*VGG-16 | CNN model*', Available from: https://www.geeksforgeeks.org/vgg-16-cnn-model/ [Accessed 10 January 2021]

Hasabo, I., 2020, '*Image Classification using Machine Learning and Deep Learning*', Available from: https://medium.com/swlh/image-classification-using-machine-learning-and-deep-learning-2b18bfe4693f [Accessed 9 January 2021]

IBM Cloud Education, 2020a, *'Machine Learning'*, Available from: https://www.ibm.com/cloud/learn/machine-learning [Accessed 22 February 2021]

IBM Cloud Education, 2020b, *'Deep Learning'*, Available from: https://www.ibm.com/cloud/learn/deep-learning#toc-deep-learn-md_Q_Of3 [Accessed 24 February 2021]

JBallin, 2017, *'I Finally Understand Static vs. Dynamic Typing and You Will Too!*', Available from: https://hackernoon.com/i-finally-understand-static-vs-dynamic-typing-and-you-will-too-ad0c2bd0acc7 [Accessed 29 January 2021]

Karanam, R., 2019, '*REST API – What is HATEOAS?*', Available from: https://dzone.com/articles/rest-api-what-is-hateoas [Accessed 14 January 2021]

Koen, S., 2019, *'Not yet another article on Machine Learning!'*, Available from: https://towardsdatascience.com/not-yet-another-article-on-machine-learning-e67f8812ba86 [Accessed 6 March 2021]

Kumar, H., n.d., '*Data augmentation Techniques'*, Available from: https://iq.opengenus.org/data-augmentation/ [Accessed 11 January 2021]

Malik, F., 2019, '*What Are Hidden Layers?*', Available from: https://medium.com/fintechexplained/what-are-hidden-layers-4f54f7328263 [Accessed 29 December 2020]

Maruti Techlabs, n.d., *'9 Real-World Problems that can be Solved by Machine Learning'*, Available from: https://marutitech.com/problems-solved-machine-learning/ [Accessed 17 February 2021]

Missinglink.ai, n.d.a, *'The Complete Guide to Artificial Neural Networks: Concepts and Models'*, Available from: https://missinglink.ai/guides/neural-network-concepts/complete-guide-artificial-neural-networks/ [Accessed 24 February 2021]

Missinglink.ai, n.d.b, *'7 Types of Neural Network Activation Functions: How to Choose?'*, Available from: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/ [Accessed 27 February 2021]

Nigam, V., 2018, *'Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning*', Available from: https://medium.com/analytics-vidhya/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90 [Accessed 9 January 2021]

Oppermann, A., 2019, '*What is Deep Learning and How does it work?',* Available from: https://towardsdatascience.com/what-is-deep-learning-and-how-does-it-work-2ce44bb692ac [Accessed 29 December 2020]

Pathmind, n.d., *'Java Tools for Deep Learning, Machine Learning and AI'*, Available from: https://wiki.pathmind.com/java-ai [Accessed 17 February 2021]

Picocli, 2021, '*picocli - a mighty tiny command line interface*', Available from: https://picocli.info/ [Accessed 16 January 2021]

Rao, V., 2017, '*Get started with Deeplearning4j*', Available from: https://developer.ibm.com/articles/cc-get-started-deeplearning4j/ [Accessed 12 January 2021]

RedHat, n.d.a, '*What is an API?*', Available from: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces [Accessed 4 March 2021]

RedHat, n.d.b, '*What is an REST API?*', Available from: https://www.redhat.com/en/topics/api/what-is-a-rest-api [Accessed 4 March 2021]

Saket, 2017, '*7 Best Models for Image Classification using Keras*', Available from: https://www.it4nextgen.com/keras-image-classification-models/ [Accessed 10 January 2021]

Salinas, D., 2020, '*Deep Learning Use Cases: Separating Reality from Hype in Neural Networks',* Available from: https://*towardsdatascience*.com/deep-learning-use-cases-separating-reality-from-hype-in-neural-networks-9d31cc1bc746 [Accessed 28 December 2020]

Sarkar, D., 2018, '*A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning*', Available from: https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a [Accessed 11 January 2021]

Sayantini, N. 2020, '*Java vs Python : Comparison between the Best Programming* Languages', Available from: https://www.edureka.co/blog/java-vs-python/ [Accessed 6 January 2021]

Simonyan, K., Zisserman, A., 2015, '*VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION*', Available from: https://arxiv.org/pdf/1409.1556.pdf [Accessed 26 January 2021]

Springboard India, 2020a, *'Best language for Machine Learning: Which Programming Language to Learn'*, Available from: https://in.springboard.com/blog/best-language-for-machine-learning/ [Accessed 17 February 2021]

Springboard Inidia, 2020b, *'Which is Better for AI Java or Python? Which Programming Language Should I Learn?'*, Available from: https://in.springboard.com/blog/which-is-better-for-ai-java-or-python/ [Accessed 18 February 2021]

Thereaderwiki, 2020, '*Java (programming language)',* Available from: https://thereaderwiki.com/en/Java_(programming_language) [Accessed 29 December 2020]

ThinkAutomation, n.d., *'ELI5: what is image classification in deep learning?'*, Available from: https://www.thinkautomation.com/eli5/eli5-what-is-image-classification-in-deep-learning/ [Accessed 3 March 2021]