whitehorses

member of the human network

WHITEBOOK

Deep Java Library: AI out of the box

Juli 2022

Auteur:
Laurens van der Starre
INTEGRATIESPECIALIST

Introductie

In eerdere Whitebooks van mijn hand is de wereld van de Neurale Netwerken al een klein beetje verkend. Daar werd bijvoorbeeld een Google AI Cloud oplossing voorgesteld waarin een Neuraal Netwerk is gebruikt voor het maken van een neuraal netwerk in AutoML [1]. Ook is er sentiment analyse gedaan met een Google NLP oplossing [2].

Met name het trainen van een Neuraal Netwerk is relatief eenvoudig door gebruik te maken van Google's AutoML. Echter vereist het trainen van het netwerk toch enige expertise, met name in het prepareren van de (grote) hoeveelheden training- en testdata. Het netwerk kan alleen goed zijn werk doen als het goed is getraind met goede representatieve data, en deze training goed heeft kunnen verifiëren met goede representatieve testdata. En wat nu als het netwerk niet in Google's cloud moet draaien, maar on-premise, offline? Het is mogelijk het gegenereerde TensorFlow netwerk te downloaden, en te gebruiken. Maar is er niet een eenvoudigere manier?

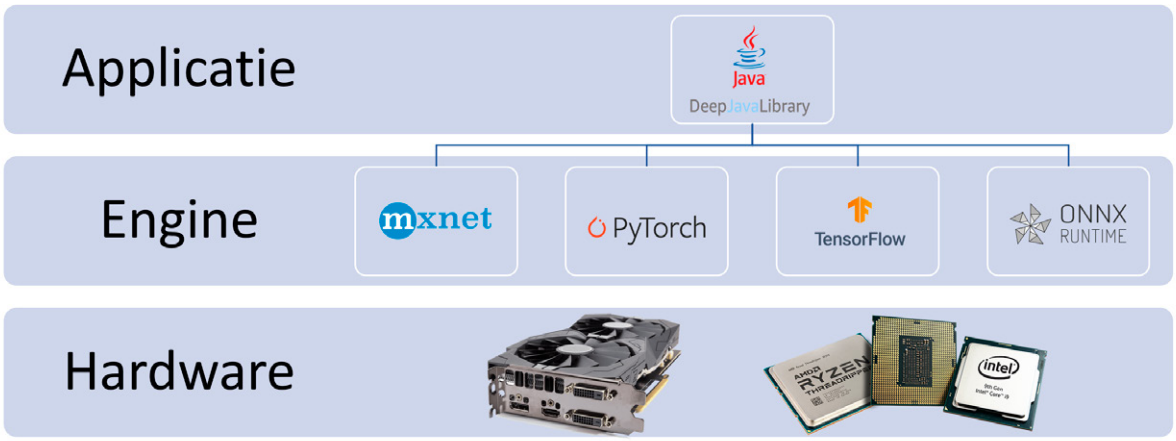
Voor heel veel AI gerelateerde usecases is het trainen van een eigen netwerk eigenlijk niet nodig. Met name op het gebied van (o.a.) beeldherkenning en object classificatie kan er gebruik gemaakt worden van bestaande netwerken. Deze zijn te gebruiken in software middels diverse (open source) libraries.

Dit Whitebook bekijkt *Deep Java Library* (DJL), en wordt een *Model Zoo* toegepast om zeer eenvoudig via beeldherkenning te kijken of een parkeerplaats vol is.

Deep Java Library

De Deep Java Library (DJL) is een high-level Java Framework voor Deep Learning. Het is open-source en engine-agnostic. Dit betekent dat verschillende deep learning engines allemaal via hetzelfde Java Framework, en API, gebruikt kunnen worden. Er hoeft dus geen keuze voor een engine worden vastgelegd, en deze engine kan worden gewijzigd en verwisseld zonder dat de bovenliggende programmatuur moet worden aangepast. De engines die DJL ondersteunt zijn Apache MXNet, Pytorch, TensorFlow en ONNX Runtime. Daarnaast kan DJL zelf de hardware configuratie bepalen om zo optimaal van de CPU en/of GPU gebruik te maken.

Hier ligt dus de grote kracht kan DJL: je hebt een enkel framework en API voor je applicatie, zonder bezig te hoeven zijn met engine specifieke details.

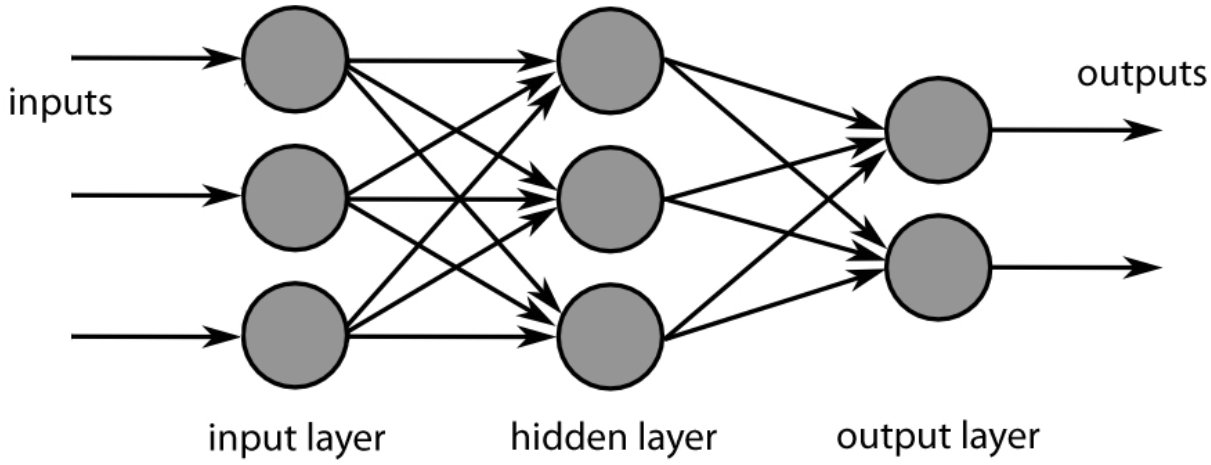


Figuur 1 DJL overzichtsplaat

Wat kan je er mee?

DJL is een deep learning framework en stelt je dus in staat neurale netwerken te definiëren, te trainen en te gebruiken. Deze netwerken kan je toepassen op een drietal toepassingsgebieden (*Application*): Computer Vision (CV), Natural Language Processing (NLP) en tabel data (*Tabular)* (voor data categoriseren en lineaire regressie voorspellingen). Binnen deze toepassingen zijn er diverse gespecialiseerde (sub) toepassingen gedefinieerd om het netwerk optimaal te kunnen toepassen, bijvoorbeeld sentiment analyse binnen het toepassingsgebied NLP.

DJL biedt een hoog over, engine-agnostic, API waarmee een neurale netwerk en trainings- en testdataset kan worden gedefinieerd, en kan worden getraind. Uiteindelijk is het resultaat een *Model*, wat kort door de bocht gezegd het neurale netwerk bevat, samen met informatieover de inputs, outputs (dus ook bijvoorbeeld data types). Dit Model kan vervolgens gebruikt worden in een applicatie via de DJL API zonder dat er engine specifieke code moet worden geschreven.



Figuur 2 voorbeeld van een multi layer perceptron netwerk

DJL biedt een uitstekende laagdrempelige API voor beginners om Models te trainen en gebruiken. Het biedt echter ook een toolkit voor zeer geavanceerd gebruik op het gebied van deep learning.

Dit Whitebook zal niet het creëren van nieuwe Models bespreken, maar zal het *gebruik* van bestaande Models toepassen. Deze Models leven in de *Model Zoo*, en die kunnen met een paar regels code worden toegepast binnen de eigen applicatie.

Model zoo

Een Model Zoo is een verzameling van *pre-trained* Models. Dit betekent dat het een kant-en-klaar neuraal netwerk is, die getraind is met een grote dataset voor een specifiek doel. Zo zijn er kant en klare Models voor sentiment analyse op Engelse tekst, die getraind zijn op een grote hoeveelheid data, of bijvoorbeeld Models om objecten te herkennen in afbeeldingen. Deze Models kunnen direct worden gebruikt om andere Models te trainen of voor *inference* (directe toepassing zonder training). Voor elke engine bestaan er diverse Models die via de DJL API kunnen worden opgezocht en gebruikt. DJL stelt je in staat op basis van enkele criteria een Model te zoeken.

De zoekcriteria zijn aan de ene kant basaal. Zoals eerder besproken bevat een Model naast het neurale netwerk ook gegevens over de inputs en outputs, en is een neuraal netwerk vaak gericht op een bepaald toepassingsgebied. Dit zijn basale criteria om een Model op te selecteren. Daarnaast kunnen er meer geavanceerde filters worden toegepast, zoals het aantal lagen van de multi layer perceptron neuraal netwerk, het soort netwerk en de gebruikte dataset waarmee het is getraind.

Zo kan een expert bijvoorbeeld gericht zoeken naar een *resnet18* of *resnet50* netwerk getraind met *imagenet*, en een beginnende gebruiker -wie dat allemaal wellicht niet zo veel zegt- kan gewoon zoeken naar *een* netwerk voor Object Detection.

Een greep uit de beschikbare zoekcriteria voor de MXNet Model Zoo:

Category	Application	Model Family	Criteria	Possible values
CV	Action Recognition	ActionRecognition	backbone	vgg16, inceptionv3
			dataset	ucf101
	Image Classification	MLP	dataset	mnist
		Resnet	layers	18, 34, 50, 101, 152
			flavor	v1, v2, v1d
			dataset	imagenet, cifar10
		Resnext	layers	101, 150
			flavor	32x4d, 64x4d
			dataset	imagenet
		Senet	layers	154
			dataset	imagenet
		SeResnext	layers	101, 150
			flavor	32x4d, 64x4d
			dataset	imagenet
	Instance Segmentation	mask_rcnn	backbone	resnet18, resnet50, resnet101
			flavor	v1b, v1d

Category	Application	Model Family	Criteria	Possible values
			dataset	coco
	Object Detection	SSD	size	300, 512
			backbone	vgg16, mobilenet, resnet18, resnet50, resnet101, resnet152
			flavor	atrous, 1.0, v1, v2
			dataset	coco, voc
	Pose Estimation	SimplePose	backbone	resnet18, resnet50, resnet101, resnet152
			flavor	v1b, v1d
			dataset	imagenet
NLP	Question and Answer	BertQA	backbone	bert
			dataset	book_corpus_wiki_en_uncased

Het bovenstaande laat zien dat er een grote keuze is voor de selectie van een netwerk. Eén van deze netwerken uit deze MXNet Model Zoo zal worden toegepast in dit Whitebook.

Toepassing: hoeveel parkeerplaatsen zijn er nog vrij?

In de voorgaande hoofdstukken is met name aan bod gekomen dat DJL een hoog over API is voor deep learning, dat eenvoudig in gebruik is, en zeker met een Model Zoo snel toe te passen is. In dit hoofdstuk wordt een Model Zoo toegepast om via *Instance Segmentation* te kijken hoeveel parkeerplaatsen er vrij zijn op een parkeerterrein.

In moderne parkeergarages wordt de automobilist geholpen in het vinden van een beschikbare plek. Sensoren in de parkeerhavens zorgen er voor dat de beschikbaarheid wordt aangegeven met rode of groene lampen boven de plek en geven borden boven de weg aan hoeveel plekken er binnen een gepaalde route beschikbaar zijn. Door de aanwijzingen te volgen hoeft de automobilist niet eindeloos door een parkeergarage te rijden op zoek naar een plek.

Als we kijken naar het openbare parkeerterrein naast mijn huis, dan is dit natuurlijk niet haalbaar. Echter kunnen we AI i.c.m. een camera gebruiken om te kijken hoe de bezetting is. Het uitzicht uit mijn werkkamer op het parkeerterrein is als volgt:

Figuur 3 het parkeerterrein

Als iedereen netjes zou parkeren, dan is er plek voor 11 auto's. We gaan DJL toepassen om te bepalen hoeveel plekken er vrij zijn.

Het toepassingsgebied voor deze casus is Computer Vision (CV). Het idee is dat de auto's worden gedetecteerd en geteld. Het zal dus gaan om Object Detection of Instance Segmentation. DJL kan bij deze toepassingsgebieden de input image verrijken met kaders om de gedetecteerde objecten (*object classes*). Instance Segmentation detecteert ook objecten en geeft tevens een mooier resultaat door de gedetecteerde objecten ook te kleuren zodat ook overlappende objecten duidelijk te zien zijn.

Een ander belangrijk criteria voor een Model Zoo is de dataset waarmee het Model is getraind. Bij de MXNet Model Zoo is daarin eigenlijk de keuze tussen imagenet en coco. De ImageNet dataset bevat bijna 15 miljoen beelden die geannoteerd zijn voor object detection en classificatie. De coco dataset is iets kleiner met 330 duizend beelden voor instance segmentation.

In dit Whitebook wordt de coco-set gebruikt. Het leuke aan coco is namelijk dat de website een goede mogelijkheid biedt deze dataset te ontdekken [3].

De volgende stappen zijn te onderscheiden:

1. Lees het invoerbestand in;
2. Kies een Model uit de Model Zoo;
3. Voer het Model uit met het gegeven invoerbestand;
4. Sla het uitvoerbestand op;
5. Tel het aantal auto's.

De complete code staat in de bijlage.

Het Model Zoo gebruiken.

Eerder is besproken dat een Model niet alleen het netwerk, maar ook o.a. de invoer en uitvoer datatypes bevat. De input voor het Computer Vision Model is een afbeeldingsbestand. Hiervoor moet een klasse uit de DJL library worden gebruikt. De uitvoer is een DetectedObjects klasse, want we gaan objecten detecteren. Deze komt tevens uit de DJL library. We willen instance segmentation uitvoeren. Dit is een subklasse van Computer Vision. Als dataset willen we "coco". De keuze voor deze dataset is arbitrair. Het had net zo goed imagenet kunnen zijn in dit specifieke geval. Dit zijn de criteria die gebruikt gaan worden om het Model te zoeken.

```
Criteria<Image, DetectedObjects> criteria =
        Criteria.builder()
                .optApplication(Application.CV.INSTANCE_SEGMENTATION)
                .setTypes(Image.class, DetectedObjects.class)
                .optFilter("dataset", "coco")
                .build();
```

Vervolgens kan een ZooModel gezocht worden o.b.v. de gespecificeerde criteria:

```
ZooModel<Image, DetectedObjects> model = criteria.loadModel()
```

Het gevonden Model kan vervolgens worden toegepast op het invoerbestand (de afbeelding). Om dit te doen moet er een Predictor worden gemaakt en aangeroepen. Het resultaat is een DetectedObjects object (ook precies zoals we in de zoekcriteria hadden gedefinieerd).

```
Predictor<Image, DetectedObjects> predictor = model.newPredictor();
```

```
DetectedObjects detection = predictor.predict(img);
```

En dan is het al klaar… DJL heeft op basis van de zoekcriteria een Model gevonden en heeft met een Predictor o.b.v. het aangeleverde afbeeldingsbestand de uitvoerklasse gevuld.

Het resultaat verwerken

Het DetectedObjects uitvoerobject bevat alle informatie. Naast de objecten en hun classification (class) (car, person, bike etc.) bevat het ook informatie over *waar* het object is gevonden in de afbeelding. Dit zijn per object een 4-tal coördinaten die de hoekpunten van een *boundingbox* aangeven. En deze boundingboxes kunnen worden geplot op de afbeelding:

```
saveBoundingBoxImage(img, detection);
```

Het resultaat is:

Omdat DetectedObjects tevens de classes bevat van de gedetecteerde objecten, kunnen we simpelweg tellen hoe vaak “car” voorkomt:

```
for (int c = 0; c < detection.getNumberOfObjects(); c++) {
        if (“car”.equals(detection.item(c).getClassName())) {
            counter++;
        }
    }
```

Als we dit getal aftrekken van 11, dan weten we precies hoeveel plekken er vrij zijn op de parkeerplaats.

```
[INFO ] - *-*-*-*-*
[INFO ] - car is 6 keer gedetecteerd
[INFO ] - er zijn nog 5 parkeerplaatsen vrij
[INFO ] - *-*-*-*-*
```

Finetuning

Het Model zoals gebruikt in het voorgaande stuk zal veel meer objecten vinden dan eigenlijk vereist. Het is immers een Model voor algemene instance segmentation en is getraind op een dataset die duizenden geclassificeerde objecten bevat. Ook zullen hier *false positives* tussen zitten, dat het netwerk denkt een auto te detecteren, maar dat het eigenlijk de kliko van de buren is. Het DetectedObjects object bevat naast de gedetecteerde klasse ook de *waarschijnlijkheid* van de detectie. Is de waarschijnlijkheid erg laag, dan is dit ruis, en kan er voor zorgen dat er in dit geval objecten ten onrechte als auto’s worden gedetecteerd.

Een simpele *threshold* kan worden meegegeven als *argument* voor het Model, waarin we de minimale waarschijnlijkheid kunnen specificeren. Gedetecteerde objecten met een lagere waarschijnlijkheid worden niet meegenomen.

In dit voorbeeld gaan we een waarschijnlijkheid van minimaal 75% vereisen:

```
private static double MIN_PROBABILITY = 0.75d;
```

```
    Criteria<Image, DetectedObjects> criteria =
        Criteria.builder()
                .optApplication(Application.CV.INSTANCE_SEGMENTATION)
                .setTypes(Image.class, DetectedObjects.class)
                .optFilter(“dataset”, “coco”)
                .optArgument(“threshold”, MIN_PROBABILITY)
                .build();
```

Als we deze fine tuning doortrekken dan kan een nog beter resultaat worden behaald door een andere dataset te gebruiken. Als er een dataset zou bestaan met daarin alleen maar auto's (of andere voertuigen zoals vrachtwagens e.d.) dan zou de waarschijnlijkheid van de gedetecteerde objecten worden verhoogd. Ook zullen er dan geen andere classes worden gedetecteerd. Hier zien we direct de kracht dan DJL: als dit Model uit de Model Zoo niet meer zal volstaan, kan deze worden vervangen door een andere. Als er geen Model uit de Zoo volstaat, dan kan er een eigen Model worden getraind met een eigen dataset van bijvoorbeeld alleen maar voertuigen. Zo zou ik een Model kunnen trainen op een dataset van louter afbeeldingen van mijn auto, en het Model laten detecteren of mijn auto nog op de parkeerplaats staat.

Conclusie

De theorie doorgronden en het volledige potentieel van deep learning gebruiken is niet weggelegd voor de normale programmeur. Het is letterlijk hogere wiskunde en vereist een specifieke skillset.

DJL is echter een library die deep learning toegankelijk maakt voor de massa. Het abstraheert de onderliggende engines, het biedt een enkele en begrijpelijke API en levert kant-en-klare Models die direct toegepast kunnen worden. Het brede scala aan toepassingsgebieden en modellen maakt DJL een zeer geschikte library voor deep learning gerelateerde toepassingen. In dit Whitebook is dit aangetoond door een simpele toepassing te schrijven, die detecteert hoeveel parkeerplaatsen vrij zijn op een parkeerplaats. Dit past allemaal binnen een tiental regels aan code.

Refenties:

1. Machine Learning made easy door Google | Whitehorses (https://www.whitehorses.nl/whitebooks/machine-learning-made-easy-door-google)
2. Sentiment analyse en categoriseren met Google NLP | Whitehorses (https://www.whitehorses.nl/whitebooks/sentiment-analyse-en-categoriseren-met-google-nlp)
3. https://cocodataset.org/#explore

Bijlage

```
package nl.whitehorses.whitebook.djl;

import ai.djl.Application;
import ai.djl.ModelException;
import ai.djl.inference.Predictor;
import ai.djl.modality.cv.Image;
import ai.djl.modality.cv.ImageFactory;
import ai.djl.modality.cv.output.DetectedObjects;
import ai.djl.repository.zoo.Criteria;
import ai.djl.repository.zoo.ZooModel;
import ai.djl.training.util.ProgressBar;
import ai.djl.translate.TranslateException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 */
public final class ParkingSpaceCounter {

    private static double MIN_PROBABILITY = 0.75d;
    private static int AVAILABLE_SPACES = 11;
    private static String CLASS_TO_DETECT = "car";


    private static final Logger logger = LoggerFactory.getLogger(ParkingSpaceCounter.class);

    private ParkingSpaceCounter() {}

    public static void main(String[] args) throws IOException, ModelException, TranslateException {

        boolean toggleOutput = true;
        String file;

        /*
         * args[1] is the input file
         * args[2] is the toggle for output segmentation file creation
        */
```

```
    if (args.length == 1) {
        file = args[0];
        toggleOutput = false;

    } else if (args.length == 2) {
        file = args[0];
        toggleOutput = ("true".equals(args[1]));
    } else {
        throw(new IOException("Usage: ParkingSpaceCounter <input file> <output
file toggle (optional)>. Number of arguments given: "+ args.length));

    }

    DetectedObjects detection = ParkingSpaceCounter.predict(file, toggleOutput);

    logger.trace("{}", detection);
  }

  public static DetectedObjects predict(String file, boolean toggleOutput) throws
IOException, ModelException, TranslateException {

    Path imageFile = Paths.get(file);
    Image img = ImageFactory.getInstance().fromFile(imageFile);

    Criteria<Image, DetectedObjects> criteria =
            Criteria.builder()
                    .optApplication(Application.CV.INSTANCE_SEGMENTATION)
                    .setTypes(Image.class, DetectedObjects.class)
                    .optArgument("threshold", MIN_PROBABILITY)
                    .optProgress(new ProgressBar())
                    .build();

    try (ZooModel<Image, DetectedObjects> model = criteria.loadModel()) {
        try (Predictor<Image, DetectedObjects> predictor = model.newPredictor()) {
            DetectedObjects detection = predictor.predict(img);

            if (toggleOutput) {
                saveBoundingBoxImage(img, detection);
            }

            countClasses(detection, CLASS_TO_DETECT);

            return detection;
```

```
        }
    }
}

private static void saveBoundingBoxImage(Image img, DetectedObjects detection)
        throws IOException {
    Path outputDir = Paths.get("build/output");
    Files.createDirectories(outputDir);

    img.drawBoundingBoxes(detection);

    Path imagePath = outputDir.resolve("instances.png");
    img.save(Files.newOutputStream(imagePath), "png");
    logger.trace("Segmentation result image has been saved in: {}", imagePath);



}

private static void countClasses (DetectedObjects detection, String className) {

    int counter = 0;

    logger.info("*-*-*-*-*");

    for (int c = 0; c < detection.getNumberOfObjects(); c++) {
        if (className.equals(detection.item(c).getClassName())) {
            counter++;
        }
    }

    logger.info(className + " is " + counter + " keer gedetecteerd.");

    if (counter <= AVAILABLE_SPACES) {
        logger.info("er zijn nog " + (AVAILABLE_SPACES - counter) + " parkeerplaatsen vrij");
    } else {
        logger.info("Er zijn geen parkeerplaatsen vrij, men staat al illegaal geparkeerd.");

    }
    logger.info("*-*-*-*-*");
}


}
```