Deep Learning en Python

Deep learning modellen in Java met Microsoft Cognitive Toolkit

Er is veel te doen in de wereld van artificial intelligence op het gebied van deep learning en neurale netwerken. Er zijn steeds betere frameworks beschikbaar waarmee je ook als ontwikkelaar steeds eenvoudiger een deep learning model kunt bouwen. Niet alleen de frameworks voor het trainen van netwerken worden beter. Ook de manier waarop je modellen in je productieomgeving kan toepassen, worden steeds beter. In dit artikel lees je hoe je een deep learning model kan trainen in Python en gebruiken in Java. Zo kan jij straks ook modellen van je collega *data scientists* effectief gebruiken op productie.

Microsoft Cognitive Toolkit als alternatief voor Tensorflow

Je zult ondertussen al wel gehoord hebben van Tensorflow, het deep learning framework van Google. Met dit framework zijn al veel interessante projecten bedacht, waarvan AlphaGo wel een van de bekendste en meest ambitieuze is. Naast Tensorflow bestaan nog meer frameworks om neurale netwerken te bouwen. Waaronder Microsoft Cognitive Toolkit (CNTK).

Ik wil je graag CNTK laten zien om twee redenen. De eerste reden is dat CNTK 8x sneller is dan Tensorflow. De tweede reden is complexer. Het kan heel goed dat je favoriete deep learning framework over vijf jaar niet meer bestaat. De enige oplossing die je dan hebt, is het opnieuw trainen van je model op een nieuw framework.

Een model trainen kost veel tijd. Daarnaast is het lastig om hetzelfde resultaat te krijgen, omdat je te maken hebt met een vorm van kansberekening. Je wilt dus niet je model opnieuw hoeven trainen in het nieuwe framework. In plaats daarvan wil je het model bij de eerste keer trainen opslaan in een open bestandsformaat.

CNTK ondersteund een open bestandsformaat onder de naam Open Neural Network Exchange (ONNX). Het ONNX formaat wordt gebruikt door onder andere Intel, Nvidia, Facebook en Microsoft. Er is nu al ondersteuning voor acht verschillende deep learning frameworks. Het niveau wisselt wel. Op dit moment ondersteunt CNTK het ONNX formaat volledig. InTensorflow kun je alleen modellen importeren. De belangrijkste reden waarom ik je CNTK laat zien, is het open formaat dat het ondersteund. Dat het framework daarnaast sneller is, is natuurlijk weer mooi meegenomen.

Een model trainen in Python

We gaan aan de slag met CNTK. Eerst trainen we een model in Python. Het model dat we gaan trainen, is een neuraal netwerk voor het herkennen van handgeschreven getallen. Hiervoor is op internet een open dataset beschikbaar, die je eenvoudig kan inladen (zie **Listing 1**).

Elk voorbeeld in de dataset bestaat uit een vector van 64 bytes die de originele afbeelding representeert (opgeslagen in de variabele X). Bij elk voorbeeld is aangegeven welk getal het representeert (opgeslagen in de variable y). De labels coderen we hierbij naar zogenaamde one-hot vectoren. We maken gebruik van de OneHotEncoder utility uit de scikit-learn library.

Willem Meints is een Microsoft MVP en Technical Evangelist voor Info Support.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.preprocessing import OneHotEncoder

digit_dataset = load_digits(10)

X = digit_dataset.data
y = digit_dataset.target

// Reshape, zodat we een matrix krijgen met 1 kolom
// in plaats van een vector.
y = y.reshape(-1, 1)

encoder = OneHotEncoder()
encoder.fit_transform(labels)
```

Listing 1

Het model opbouwen

Het neurale netwerk dat we gaan bouwen, staat in **Figuur 1** afgebeeld met het bijbehorende aantal neuronen in elke laag. De eerste en de derde laag zijn het belangrijkste. De eerste laag moet overeenkomen met het aantal inputs (64 in ons geval). De derde en laatste laag bestaat uit tien neuronen. Dit betekent dat voor elk getal dat we willen voorspellen, er een neuron is.

Als we straks aan het neurale netwerk vragen om een plaatje te categoriseren, dan is de output neuron met de hoogste waarde het getal dat we zoeken. In **Listing 2** bouwen we het neurale netwerk met behulp van layer functies. De Dense functie definieert een zogenaamde dense layer.

Een dense layer is een laag in het neurale netwerk die alle nodes uit de vorige laag verbindt met alle neuronen in de dense layer. De Sequential functie koppelt meerdere lagen automatisch aan elkaar.

Het model trainen

Je kan het neurale netwerk zien als een complexe wiskundige formule die je probeert te optimaliseren. CNTK laat je deze formule als zodanig opschrijven. Pas als je gaat trainen, wordt deze formule omgezet naar uitvoerbare code.

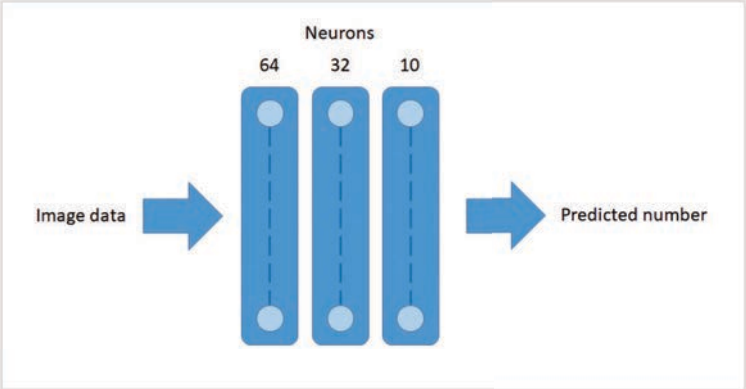
In **Listing 3** definiëren we een input variabele voor ons model. Daarna roepen we ons model aan met deze input variabele. Dit levert een nieuwe functie op, welke we opslaan in de variabele z. Om het model te kunnen trainen, heb je een loss functie nodig. Hiermee meet je of je neurale netwerk het goed doet. Met een loss function meet je hoe ver het neurale netwerk van het goede antwoord af zit.

In **Listing 4** definiëren we de loss als een categorical cross entropy functie. Deze meet het verschil tussen de categorie waarin we een plaatje verwachten te categoriseren en de werkelijke categorie. Als het neurale netwerk een goede voorspelling doet, dan is de output van deze functie laag en dicht bij de waarde 0. Als het model een onduidelijke voorspelling doet (hij valt in geen van de categorieën), dan is de output van deze functie hoog.

Naast de loss definiëren we ook een error functie. Deze gebruiken we later bij het evalueren van ons neurale netwerk. De error functie drukt uit welk percentage van de getallen we goed hebben voorspeld met ons neurale netwerk.

Voor het trainingsproces maken we een learner en een trainer aan. De learner definieert welk algoritme je wilt gebruiken voor het trainen. De trainer coördineert het leerproces. In **Listing 5** zie je hoe de code hiervoor eruit ziet.

Het trainen vindt plaats door alle voorbeelden uit een trainingset meerdere keren aan te bieden aan het neurale netwerk door middel van de trainer.



Figuur 1

```
from cntk.layers import Dense, Sequential

model = Sequential([
    Dense(64),
    Dense(32),
    Dense(10)
])
```

Listing 2

```
features = cntk.input_variable(64)

z = model(features)
```

Listing 3

```
label = cntk.input_variable(10)

loss = cntk.cross_entropy_with_softmax(z, label)
label_error = cntk.classification_error(z, label)
```

Listing 4

```
learning_rate = 0.1
learning_rate_schedule = cntk.learning_parameter_schedule(learning_rate)

learner = cntk.sgd(z.parameters, learning_rate_schedule)

trainer = cntk.Trainer(z, (loss,error_label), [learner])
```

Listing 5

In **Listing 6** splitsen we de dataset in een trainingset en validatieset. Met de trainingset gaan we aan de slag om te trainen. De validatieset gebruiken we later om te controleren of ons model goed werkt. Het trainingproces bestaat verder uit een tweetal loops. De eerste bepaalt hoe vaak we de hele dataset doorlopen voor het trainen. Het volledig doorlopen van de trainingset noemen we ook wel een epoch. In totaal trainen we tien epochs. De tweede loop is de minibatch loop. We sturen steeds kleine subsets van de data op naar de videokaart, zodat het netjes in het videogeheugen past.

Het daadwerkelijk trainen vindt plaats in de de train_minibatch methode. Deze accepteert een dictionary, waarin we een koppeling leggen tussen de input variabelen en de data die erbij hoort. CNTK gaat nu aan de slag en converteert het model naar uitvoerbare code en traint het model met behulp van de trainer op bijvoorbeeld je GPU. Meestal duurt dit proces enkele minuten.

Nadat het model is getraind kun je het valideren door de code uit **Lisitng 7** uit te voeren. Net als bij het trainen, testen we het model door middel van een minibatching proces. We gebruiken de test_minibatch methode voor dit proces. Aan het einde van de minibatch loop, hebben we de uiteindelijke score. Het model dat we hebben gemaakt, levert in dit geval een score op van 0.925, oftewel 92.5%. Dat is zeker niet de hoogste score voor het herkennen van handgeschreven getallen, maar ook niet heel slecht.

Modellen opslaan in een open formaat

Nu je een model hebt, kun je deze exporteren naar ONNX om hem te gaan gebruiken in Java. Het ONNX formaat is gebaseerd op protobuf, een ander bekend open source data formaat. ONNX is dan ook niet meer dan een officiële specificatie voor het opslaan van neurale netwerken en andere machine learning modellen in protobuf files. En zoals dat tegenwoordig gaat, staat ONNX als open source project op Github.

In **Listing 8** zie je hoe het model opgeslagen kan worden in het ONNX formaat. De variabele z bevat het model, zoals we dat eerder hadden gemaakt.

Het model gebruiken in jouw Java applicatie

Nu we het model hebben opgeslagen in ONNX kunnen we het gebruiken vanuit een ander deep learning framework of een andere taal zoals CNTK, die ook een implementatie in Java kent. De Java versie van CNTK is niet gepubliceerd op Maven. Je moet, om CNTK vanuit Java te kunnen gebruiken, een release downloaden van Github (**https://github.com/Microsoft/CNTK/releases**). In de release vind je de file cntk.jar, welke je met Maven kan installeren als een beschikbare dependency op je machine.

Nadat je de jar file lokaal in je Maven cache hebt gezet, kan je hem vanuit Maven gebruiken door hem als dependency op te nemen in je pom.xml. Om het model te kunnen gebruiken, moet je het ONNX bestand inladen met behulp van de Function class. In **Listing 10** zie je hoe dit werkt.

ER IS NU AL ONDERSTEUNING VOOR ACHT VERSCHILLENDE DEEP LEARNING FRAMEWORKS

```
from sklearn.model_selection import train_test_split

minibatch_size = 64

train_X, test_X, train_y, test_y =
    train_test_split(X, y, test_size=0.2)

for _epoch in range(1, 10):
    for batch_offset in range(0, train_X.shape[0], minibatch_size):
        features_batch = train_X[
            batch_offset:batch_offset + minibatch_size]
        labels_batch = train_y[
            batch_offset:batch_offset + minibatch_size]

        input_map = {
            features: features_batch,
            label: labels_batch
        }

        trainer.train_minibatch(input_map)
```

Listing 6

```
for batch_offset in range(0, len(test_X), minibatch_size):
    features_batch = test_X[
        batch_offset:batch_offset + minibatch_size]
    labels_batch = test_y[
        batch_offset:batch_offset + minibatch_size]

    input_map = {
        features: features_batch,
        label: labels_batch
    }

    score = trainer.test_minibatch(input_map)

print("Model score: {}".format(score))
```

Listing 7

```
z.save('model.onnx', cntk.ModelFormat.ONNX)
```

Listing 8

```
mvn install:install-file -Dfile=cntk.jar -DgroupId= com.microsoft.cntk -DartifactId=cntk -Dversion=2.5.1
```

Listing 9

```
DeviceDescriptor device = DeviceDescriptor.getCPUDevice();
Function modelFunction = Function.load(
    "model.onnx", device, ModelFormat.ONNX);
```

Listing 10

Net als in Python kun je CNTK modellen in Java op je CPU of GPU trainen of gebruiken. In **Listing 10** laden we het model in, zodat het op de CPU kan draaien.

Nadat het model is ingeladen, kan je het voeden met data. In **Listing 11** zie je dat we een batch maken die bestaat uit een FloatVectorVector met daarin een FloatVector die de pixels van het plaatje met het getekende getal bevat.

```
FloatVectorVector batch = new FloatVectorVector();
FloatVector vector = new FloatVector();

for (int index = 0; index < pixels.length; index++) {
    vector.add((float)pixels[index]);
}

batch.add(vector);
```

Listing 11

Belangrijk is dat je zorgt dat de afbeelding 8 bij 8 pixels groot is, gelijk aan de grootte van de input van je neurale netwerk. Ik heb hiervoor een losse functie gemaakt die dit regelt met behulp van de BufferedImage API in Java.

Om de voorspelling te kunnen doen, moet je de data doorgeven aan het model. Hiervoor leggen we in **Listing 12** een mapping aan tussen input variabelen en de bijbehorende data. De output geven we in dit geval wel door, maar laten we leeg.

```
Value inputValue = Value.createDenseFloat(
    features.getShape(), batch, device);

UnorderedMapVariableValuePtr inputMap =
    new UnorderedMapVariableValuePtr();

UnorderedMapVariableValuePtr outputMap =
    new UnorderedMapVariableValuePtr();

inputMap.add(features, inputValue);
outputMap.add(predictedDigit, null);

modelFunction.evaluate(inputMap, outputMap, device);
```

Listing 12

De evaluatie methode vertaalt de input vanuit Java naar native C++ code en voert de werkelijke voorspelling uit op het device dat we eerder hebben gekozen. Om vervolgens de output uit het netwerk terug te halen naar Java, moeten we het extra stukje code toevoegen uit **Listing 13**.

```
FloatVectorVector outputBuffer = new FloatVectorVector();

outputMap.getitem(predictedDigit).copyVariableValueToFloat(
    predictedDigit, outputBuffer);

FloatVector outputRecord = outputBuffer.get(0);

// Om een waarde van een output neuron te bepalen
Float neuronValue = outputRecord.get(1);
```

Listing 13

De FloatVector kan je vervolgens uitlezen, om te bepalen welke neuron de hoogste waarde had. De index van de neuron is dan het getal dat in de afbeelding wordt gerepresenteerd.

Conclusie

Er is veel gaande op het gebied van artificial intelligence. Het wordt steeds eenvoudiger om modellen te maken en te hosten in je programma's. Ontwikkelingen, zoals ONNX, dragen hier flink aan bij. De gebruiksvriendelijkheid van libraries, zoals CNTK in Java, is dan misschien nog niet het beste wat je ooit hebt gezien. Maar het laat wel goed zien wat nu al mogelijk is met deep learning in Java.

En terwijl het nu nog primair een feestje is voor *Data Scientists*, binnenkort kan ook jij als ontwikkelaar de vraag krijgen of je niet even kan helpen om een model te maken en in productie te zetten. Nu je gezien hebt dat het met een combinatie van CNTK en ONNX goed mogelijk is om een model te trainen in Python en deze vervolgens in Java te gebruiken, merk je direct dat ook jij als ontwikkelaar prima een deep learning model kan bouwen.

Ben je geïnteresseerd in de code? Deze staat op Github: **https://github.com/wmeints/cntk-deeplearning-java/.** ■

JE KAN HET NEURALE NETWERK ZIEN ALS EEN COMPLEXE WISKUNDIGE FORMULE DIE JE PROBEERT TE OPTIMALISEREN