Sjoerd van der Heide &

Emiel de Wit

Presenteren:

Het legendarische prachtige ongelofelijke

HIRAGANABOT

MINOR VERSLAG!



**De dataset**

**Chinees:**

Ons eerste plan was om 50 Chinese tekens te laten tekenen door vrijwilligers. We kwamen er snel achter dat het moeilijk was om genoeg vrijwilligers hiervoor te motiveren in deze korte tijd.

Het tweede plan was om een bestaande dataset te gebruiken. We vonden er één. Deze was te downloaden van een universiteit in China. Helaas werkten de servers maar een paar uur per week open een halve mbps en waren deze bestanden een paar gigabyte groot. Het was te langzaam en onbetrouwbaar om te gebruiken.

**Japans - Kanji:**

Er was maar één andere dataset voor Chinese tekens die we konden vinden. Dit was een set van kuzushiji (Japans cursief) en had kanji en hiragana. Kanji zijn de Chinese tekens dus deze wilden we trainen. Deze zaten nog niet in npz bestanden en moesten dus nog klaargemaakt worden voor ze een AI zouden kunnen trainen. We namen hier de top 50 meest voorkomende tekens van om het AI op te train. Helaas bleken deze tekens niet helemaal goed geclassificeerd te zijn. Daarbij kwam dat de tekens op zich functioneel onleesbaar waren en dat een significant deel van de tekens eigenlijk verkeerd geclassificeerde hiragana tekens waren. Toch probeerden we het.

Om de dataset een beetje te helpen maakten we een nieuw document met 50 tekens. Helaas bleek dit niet te werken; de dataset was niet te redden met slechts 30 nieuwe toevoegingen per teken.

**Japans - Hiragana:**

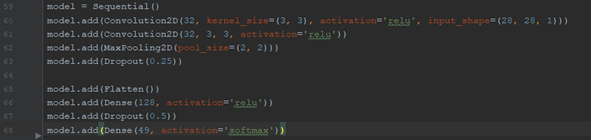
Uiteindelijk kwamen we dan uit op de kuzushiji hiragana set. Dit setje was al kant en klaar in een bruikbaar npz bestand. Deze dataset kwam uit de tijd voor de Dakuten (ﾞ) en Handakuten ( ﾟ) maar toen nog wel ゑ, ゐ en ゝgebruikten, die nu zelden nog gebruik zien. Dit hebben we opgelost in het uiteindelijke programma. We hopen in latere setten de dakuten en handakuten wel te kunnen verwerken.

**Het netwerk**

**De structuur**

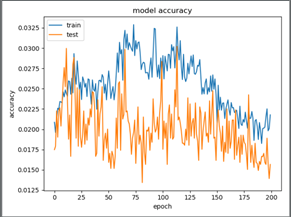
**(Legacy/Neural Network/BasicNeuralNetwork.py, Legacy/Neural Network/BasicNeuralNetworkGPU.py)**

Voor het neurale netwerk (hierna verwezen als ‘model’) hebben we besloten om Keras te gebruiken met een Tensorflow achtergrond. Het voordeel hiervan is dat Tensorflow gebruik kan maken van de GPU, wat het trainen veel sneller liet verlopen. We zijn bij de basis begonnen; een Keras tutorial volgen en zo een netwerk opzetten die werkt met de populaire MNIST dataset, gevuld met handgeschreven nummers. Hiermee hebben we een gestructureerd Convolutional Neural Network (CNN) opgezet, het meestgebruikte model voor karakterherkenning. Ons doel was om met dit netwerk een nauwkeurigheid te halen van minimaal 70%.



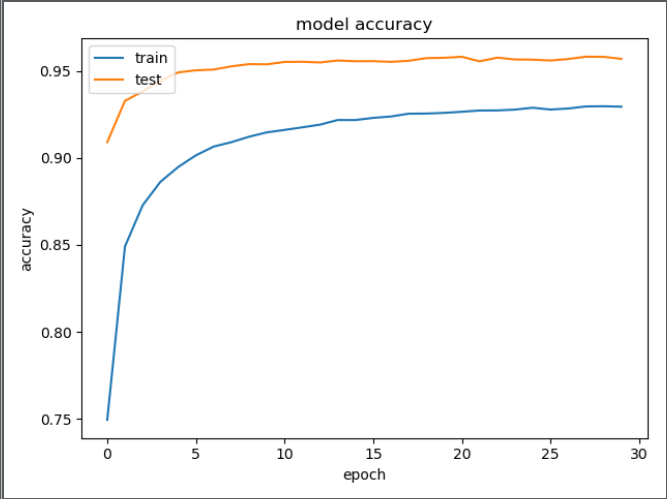
*De structuur van het model waar we mee begonnen, wat tevens het model is wat we nu gebruiken.*

**Kanji karakters**

**(**[**https://www.kaggle.com/anokas/kuzushiji**](https://www.kaggle.com/anokas/kuzushiji) **(Kkanji), Legacy/Neural Network/Kanjibot\_network.py, Legacy/Neural Network/LRFinder.py, Legacy/Neural Network/Kanjibot\_resnet.py, Legacy/Dataset/Kanjibot\_img2TFRecord.py, Legacy/Neural Network/JapaneseNeural.py, Legacy/Neural Network/resnet\_utils.py))**

We zijn begonnen met de dataset van Kanji karakters. Deze rauwe plaatjes hebben we eerst verpakt in Tensorflow Record files (.TFRecord files), om ze zo aan het netwerk te voeren. Echter kwam hier in de weken dat we eraan werkten geen goed resultaat uit. Het model bleef een nauwkeurigheid geven die niet hoger kwam dan 4,6%. Ook waren er veel tekenen van ‘overfitting’: Waar de nauwkeurigheid na het trainen 4,6% was, werd de uiteindelijke nauwkeurigheid na de validatie ongeveer 2%. Geen goed resultaat dus.

*Het resultaat met de Kanji dataset; Hier is zelfs de nauwkeurigheid niet hoger dan 3.25%.*

De weken die volgden waren gevuld met aanpassingen; Eerst probeerden we het model uit te breiden met meer Convolutional, Dropout en Batch Normalization lagen. Ook probeerden we andere optimizers, namelijk de Adam, SGD en RMSpop optimizers, zelfs met aangepaste leertempo’s. Dit hielp niet veel, hooguit een paar tiende procenten. De volgende oplossing was het gebruik van Data Generators, die bestaande plaatjes nemen, hier bewerkingen op uitvoeren zoals rotaties en translaties, en deze meegeven tijdens de training. Het enige waar dit goed voor was was het verwarmen van de tafel waar de laptop op stond. Aan de nauwkeurigheid merkten we geen verbetering.

Een laatste mogelijke oplossing die voorbij kwam was het gebruik van ‘pre-trained models’; modellen die al getraind zijn om een bepaalde taak uit te voeren, zoals classificatie van objecten of karakters. Hier hebben we onderzoek naar gedaan, en we hebben het trainen uitgevoerd met de 3 meest populaire netwerken; ResNet50, VGG-19 en InceptionV3. Alle drie maakten geen verschil tijdens de training, en soms werd het zelfs nog slechter; de laagste nauwkeurigheid die hier uit kwam zat rond de 1.7%.

*Het resultaat met de Hiragana dataset. Het trainen en testen verloopt veel soepeler, met een nauwkeurigheid die dit ook reflecteert.*

**Van Kanji naar Hiragana**

**(**[**https://www.kaggle.com/anokas/kuzushiji**](https://www.kaggle.com/anokas/kuzushiji) **(k49), Kanjibot\_Hiragana.py, run.py)**

Niks loste het probleem met de nauwkeurigheid op. Dit leidde ertoe dat we uiteindelijk de knoop hebben doorgehakt; we gingen Hiragana karakters gebruiken, een ander Japans schrift. Deze dataset was namelijk een stuk groter, en dus was er grotere kans op goede nauwkeurigheid. Dit bleek waar te zijn. Na ongeveer 26 iteraties door de dataset kwam er een nauwkeurigheid van 95% uit, met een validatie nauwkeurigheid van 92%. Dit is dus ook het uiteindelijke model geworden die we gebruiken in onze applicatie.

**Input**

**Foto’s**

**(legacy/herkenners/\*)**

Ons eerste plan was om de AI foto’s te laten herkennen. Het bleek echter erg lastig om de tekens uit een plaatje te kunnen halen als de maat, kleur en achtergrond niet standaard waren. De tekens bestonden ook uit meerdere delen, wat steeds nieuwe entries gaf in de herkenning. Als we meerdere tekens zouden willen kunnen herkennen van een foto, dan zou het herkennings algoritme moeten zoeken op de tekens zelf in plaats van ‘wat mogelijk een teken kan zijn’. Dit is niet wat we met ons project wilden doen. In de map ‘herkenners’ is een aantal slordige pogingen te vinden.

**Tekenen**

**(static/js/draw.js)**

Het tweede plan was om de input rechtstreeks op de computer te tekenen. Dit bleek makkelijk binnen Javascript gedaan te kunnen worden op een html5 canvas en is ook wat we uiteindelijk gebruiken. Als op ‘upload’ geklikt wordt, wordt het plaatje bijgeknipt tot de randen van de tekening binnen het canvas; wordt de kleur omgedraaid; en wordt het plaatje verkleind tot 28px bij 28px voor hij het model in gaat.

**Website**

**Backend**

**(run.py)**

Voor het backend gebruikten we Flask. We kozen hiervoor omdat het relatief simpel was in gebruik, en omdat het model ook gemaakt was in python. Bij invoer bewerkt het backend de tekening zoals beschreven bij ‘input - tekenen’ en slaat hij het plaatje als png op. Het url van dit opgeslagen plaatje, wat gebaseerd is op een cookie van de gebruiker, wordt dan ingeladen in het model. Het model geeft dan een array terug met kans, wat dan naar het frontend wordt gestuurd.

Dit plaatje wordt later in de juiste map gezet als het teken door de gebruiker geverifiëerd is. Deze worden opgeslagen om later een nieuw en beter model mee te trainen.

**Frontend**

**(static/\*, templates/form.html)**

Voor het frontend gebruikten we een combinatie van html, css, javascript en jquery. Het meeste van css komt uit bootstrap. Javascript en jquery werden gebruikt om het mogelijk te maken om te tekenen op het canvas. Bij het uploaden wordt het data url van dit canvas verstuurd naar het backend via Ajax.

In de array die teruggestuurd wordt, wordt de hoogste waarde opgezocht. De indices van deze array corresponderen met de indices van de hiragana, dus zet Javascript het juiste hiragana in de uitkomst met de uitspraak en mogelijke footnotes. Deze verschijnen met een javascript toggle. Omdat niet alle moderne hiragana in de gebruikte dataset staan, zijn er mogelijk alternatieven bij de tekens. Deze verschijnen in blauwe knoppen.

Het is mogelijk dat de tekens niet kloppen. De dataset is immers een hele oude set van een twijfelachtig handschrift. Als het verkeerde teken wordt herkend wordt naar het eerstvolgend hoogste getal in de array gezocht. Als het teken wel klopt, of als er op één van de blauwe knoppen wordt geklikt, wordt het juiste teken met Ajax terug naar het backend gezet om opgeslagen te worden.

**Formulieren inlezen**

**( legacy/formulier/\* )**

Eerder hadden we verteld over twee formulieren die we hadden gemaakt. Voor het tweede (formulier.docx) hadden we een programma gemaakt dat dit kon inlezen (template.py). Het werd uiteindelijk niet in het eindresultaat gebruikt, maar het is toch onderdeel van het project. Het tweede formulier was gemaakt met de voorbeeldtekens in lichtgrijs. Dit maakte het mogelijk om binnen het programma een threshold te bepalen die de voorbeeldtekens zwart maakte, maar de achtergrond van de tekens wit hield. Nu was het makkelijk om de formulieren uit te lezen door contouren te tellen, te filteren op grootte en vierkant uit te knippen. De coördinaten van de X en Y werden gebruikt om de karakters in het juiste mapje te plaatsen. Aangezien er geen bruikbaar patroon was in de tekens, werd de map aangewezen door een dubbele dictionary (dirdict.py). Elk formulier gaf tussen de 110 en 140 tekens met een lage false positive rate. Deze zijn te vinden onder ‘output/’.

Delall.py was een programma om snell de output te wissen om het opnieuw te proberen in het geval dat het niet werkte.