HO
GENT

Faculteit Bedrijf en Organisatie

Classificatie van afbeeldingen met geautomatiseerde machine learning platformen

Robbe Decorte

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
ir. Johan Decorte
Co-promotor:
dr. Kenny Helsens

Instelling: In The Pocket

Academiejaar: 2019-2020

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Classificatie van afbeeldingen met geautomatiseerde machine learning platformen

Robbe Decorte

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
ir. Johan Decorte
Co-promotor:
dr. Kenny Helsens

Instelling: In The Pocket

Academiejaar: 2019-2020

Tweede examenperiode

Woord vooraf

De keuze van mijn onderwerp heb ik snel kunnen maken. In mijn vrije tijd experimenteerde ik al eens graag met Tensorflow en later tijdens de opleiding keerde dit terug. In de lessen is het veel uitgebreider aan bod gekomen en werd mijn interesse voor wat er achter de schermen gebeurt groter. De meeste onderwerpen die iets de maken hebben met *machine learning* zijn uitdagend. De abstracte theorieën, gigantische hoeveelheid data ... Bij mij was dit niet anders, maar ik hoop om deze nieuwe kennis later te kunnen gebruiken in mijn carrière.

Deze bachelorproef werd geschreven in het kader van het voltooien van de opleiding Toegepaste Informatica en betekent ook het einde van een periode. Hierbij wil ik graag een aantal mensen bedanken die mij geholpen hebben bij het uitvoeren van dit onderzoek.

Eest en vooral zou ik graag mijn promotor, ir. Johan Decorte, bedanken. Tijdens een periode waar alles digitaal moest verlopen kon ik steeds bij hem terecht met vragen over de bachelorproef. Zijn feedback over de inhoud heeft mij vaak geholpen om alles ineen te steken.

Ook wil ik mijn co-promotor, dr. Kenny Helsens, bedanken voor het sturen van de inhoud. Tijdens de gesprekken in de kantoren van In The Pocket konden we samen brainstormen naar aanvullingen van het onderwerp. Daarnaast pushte hij me om het breder te bekijken en zo zijn er een aantal zaken naar boven gekomen waar ikzelf nooit aan gedacht zou hebben, mocht het niet door onze gesprekken zijn.

Om af te sluiten wil ik ook mijn ouders bedanken die mij van begin tot einde van de opleiding ondersteund hebben.

Samenvatting

De resultaten van dit onderzoek kunnen gebruikt worden om een keuze te maken tussen een open source of cloud platform waar geautomatiseerde *machine learning* gebruikt kan worden. Alsook in welke situaties het bruikbaar is. Het kan een hulp zijn voor bedrijven waar geen *data scientist* of ML-ingenieurs aanwezig zijn. Of waar deze gewoonweg niet genoeg tijd hebben voor de verschillende projecten. Deze technologie tracht *machine learning* toegankelijker te maken door een manier te bieden om vaak voorkomende problemen te automatiseren. Dit werd onderzocht door met AutoKeras en Google Cloud AutoML elk een prototype op te zetten dat voor een simpel maar realistisch classificatieprobleem de categorie van een afbeelding kan voorspellen. Er werden modellen getraind die katten van honden kunnen onderscheiden. Dit document beschrijft een studie naar de achterliggende gebruikte technieken, het verloop en de resultaten van beide prototypes. Er werd gevonden dat de alternatieven elk hun plaats hebben in verschillende fasen van een project. Google Cloud AutoML levert een productie waardig model terwijl AutoKeras kan dienen als hulpmiddel voor een *data scientist* of productie waardig kan zijn mits een extensieve voorbereiding van de data. De evolutie van de platformen zelf betekent enkel goed nieuws voor de toekomst. Mogelijks kan er nog onderzocht worden hoe het opschonen van de data geautomatiseerd kan worden. Dit is een grote stap binnen geautomatiseerde *machine learning* aangezien het een belangrijke factor is om *edge cases* te herkennen.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

Het informatie tijdperk centraliseert zich momenteel rond data. Bedrijven zien ook de toegevoegde waarde die het kan hebben in hun bedrijfsprocessen, kijk maar naar de grote spelers in de informaticawereld waar het begrip *Big Data* is ontstaan. In zijn ruwe vorm lijkt het op een gigantische hoop waaruit je niks kan leren, maar wanneer men deze gaat structureren zijn er plots allemaal nieuwe toepassingen beschikbaar. Eén van deze toepassingen die intensief gebruik maakt van data, is *machine learning*. Hierbij wordt geprobeerd computers zaken aan te leren met behulp van een iteratief proces zonder expliciet geprogrammeerd te zijn voor de taken die ze uitvoeren. Mensen die goed overweg kunnen met de data om zo'n model te maken (Machine Learning Engineers / Data Scientists ...) zijn vaak moeilijk te vinden. Een werkgever die zo'n probleem aan wilt pakken heeft enkele keuzes, AutoML is een mogelijke optie. Alhoewel het interesseveld ontstaan is in de jaren '50, is het nog maar sinds kort een hot topic, met dank aan de grote hoeveelheid rekenkracht in moderne systemen en doorbraken[1] binnen het onderzoeksveld die de toegangsdrempel verlagen.

Geautomatiseerde *machine learning* platformen trachten een oplossing te bieden voor *development* teams zonder een gespecialiseerde *machine learning* expert. Het platform voert alle stappen van het proces uit en uiteindelijk moeten ze het enkel in hun product integreren. Deze manier van werken verlaagt niet alleen de druk op *machine learning* experten maar het geeft hen ook de mogelijkheid om mee te werken aan uitdagende projecten of zelf onderzoek uit te voeren. Door de technische afhankelijkheid te verlagen kan de technologie sneller / meer gebruikt worden in bestaande projecten. Omdat bedrijven tot nu toe weinig contact hebben met AI en alles wat er toe behoort, zijn de meeste cases vergelijkbaar met elkaar. Zo heb je bijvoorbeeld binaire classificatie problemen, tekst

[1] Denk maar aan *open source* initiatieven zoals Tensorflow en Keras.

analyse en meer. Waarom zou het dan niet mogelijk zijn om dit te automatiseren?

Dit onderzoek bekeek hoe laag de werkelijke instapdrempel ligt bij verschillende platformen (open-source en private oplossingen), alsook hoe zo'n model werkt en welke resultaten je bekomt voor een simpel maar realistisch classificatie probleem.

1.1 Probleemstelling

Met een onschatbare hoeveelheid data die de dag van vandaag aanwezig is, is het belangrijk om deze op te schonen en features te selecteren zodat het gebruikt kan worden in een specifieke situatie. De hoeveelheid data groeit vele malen sneller dan het aantal beschikbare experten. Er moet dus naar een manier gezocht worden om *machine learning* dichter bij de man te brengen zodat mensen met een uitgebreide kennis niet vast zitten met basis problemen die ze eerder al op een gelijkaardige manier opgelost hebben. Er wordt over basisproblemen gesproken in de zin dat bedrijven een eigen toepassing willen op een probleem dat al eerder opgelost werd in andere situaties. Hierbij zijn de stappen van het trainingsproces in grote lijnen gelijk.

Het is dan ook vanzelfsprekend dat er gezocht wordt naar een manier om dit te automatiseren, zoals in elke ander aspect van ons leven. Deze platformen kunnen mogelijks een oplossing bieden voor kleine zelf organiserende *development* teams die graag een *machine learning* aspect willen toevoegen aan hun project. Dit liefst met een minimale input van de ontwikkelaars.

1.2 Onderzoeksvraag

1.2.1 Hoofdonderzoeksvraag

Met dit onderzoek werd nagegaan als deze nieuwe technologie capabel is om triviale machine learning problemen op te lossen, al dan niet met minimale input van een *developer*. AutoML is een recente ontwikkeling waardoor het in zijn huidige staat waarschijnlijk niet klaar is voor uitdagende cases, maar het kan wel toegang geven tot nieuwe functies aan *development* teams met weinig of geen *machine learning* kennis.

1.2.2 Deelonderzoeksvragen

Naast de hoofdonderzoeksvraag komen ook volgende (kleinere) vragen aan bod:

- Welke onderliggende technieken worden gebruikt bij geautomatiseerde *machine learning*.
- Kies je best voor een open source library of toch een commercieel platform.
- Is de performantie van deze platformen vergelijkbaar met die van een traditioneel gebouwd model.

1.3 Onderzoeksdoelstelling

De belangrijkste doelstelling van dit onderzoek is het reproduceren van een realistische situatie die op de traditionele manier opgelost is, maar dan met *automated machine learning*. Dit is mogelijk door de verschillende metrieken en performanties van de modellen met elkaar te vergelijken. Het experiment is geslaagd als de behaalde scores bij elkaar in de buurt liggen. Een werkend prototype is de eerste stap.

Er zijn al verschillende mogelijkheden om dit te realiseren. Bij elke implementatie werd dan ook getest hoeveel rekening er wordt gehouden met de belangrijkste aspecten[2] uit de *requirements* analyse.

Aan de hand van de prototypes en de vergelijkende studie is er een conclusie geschreven die rekening houdt met implementatie details en het standpunt van een bedrijf.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 4 en 5 is voor, respectievelijk, AutoKeras en Google Cloud AutoML een prototype opgezet en de *requirements* besproken.

In Hoofdstuk 6, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

[2] Aspecten die bepalen hoe bruikbaar de technologie is in de praktrijk

2. Stand van zaken

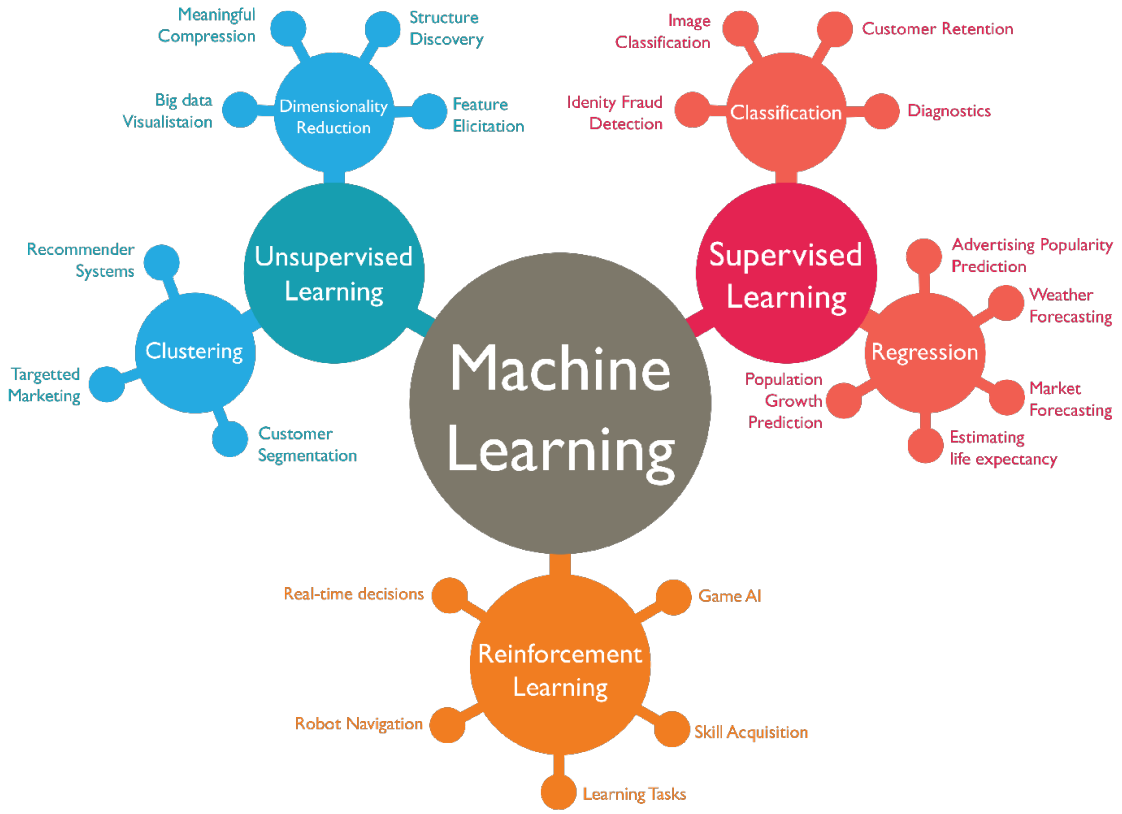
Dit onderzoek richt zich op automatische *machine learning* platformen. Maar alvorens van start te gaan met de onderliggende technieken is het belangrijk om een goed zicht te hebben op de basis waarop het gebouwd is. Zo zijn eerst een aantal belangrijke begrippen en technieken besproken, deze sectie kan overgeslagen worden als deze intro triviaal is voor de lezer. De automatisatie hiervan werd onderzocht, welke technieken zijn met elkaar gecombineerd en hoe bekom je uiteindelijk resultaten. Het is nodig om de theoretische benadering goed te begrijpen om uiteindelijk te kunnen beslissen of het resultaat van het experiment voldoet aan de normen van een goed werkend model. Tot slot werden de beschikbare cloud platformen besproken en vergeleken met een open source libraries als alternatief.

2.1 Inleiding machine learning

Deze sectie dient om mensen zonder kennis van *machine learning* te introduceren met enkele begrippen en technieken binnen het werkveld. Mensen met enige basiskennis kunnen direct doorgaan naar de volgende sectie.

Het is belangrijk dat volgende begrippen gekend zijn:

- **Data Cleaning / Cleansing**: Het detecteren, corrigeren of verwijderen van onnauwkeurige datarecords. Het resultaat is een consistente dataset die bruikbaar is om een model te trainen
- **Feature Selection**: Het selecteren van relevante kolommen uit de dataset. Het zijn de belangrijkste eigenschappen die de voorspelling bepalen
- **Software agent**: Een computerprogramma die in staat is om te leren uit eerdere



Figuur 2.1: Soorten machine learning met enklele toepassingen (Arroyo, 2017)

ervaringen

2.1.1 Soorten machine learning

Lievens (2019) schreef over 3 grote types binnen het domein van machine learning. Deze worden kort verklaard aan de hand van een praktisch voorbeeld. Figuur 2.1 is een algemeen overzicht van wat hieronder beschreven is.

Gesuperviseerd leren

Bij gesuperviseerd leren probeert de software agent een functie te leren die een voorspelling maakt voor een gegeven input. De functie die deze voorspellingen maakt evolueert door het gebruik van een trainingsdataset met input-output waarden (Peter Norvig, 1994).

Classificatie is een typisch probleem dat opgelost wordt met gesuperviseerd leren, het wordt later in dit onderzoek op een andere manier opgelost. Het doel van classificatie is om aan de hand van enkele kenmerken een voorgedefinieerde klasse te voorspellen. Er wordt gesproken van een binair classificatieprobleem als er slechts 2 klassen zijn. Spamdetectie is hier een voorbeeld van. Door de belangrijke woorden uit een bericht te halen wordt er een attribuutvector opgebouwd. Door een model te trainen met duizenden berichten en als ze al dan niet spam zijn, kan het een voorspelling maken voor de gegeven vector (Lievens,

2019).

Ongesuperviseerd leren

Met ongesuperviseerd leren is het mogelijk om in een ongelabelde dataset patronen te vinden die eerder onbekend waren. Bij elke voorspelling wordt voor elke categorie meegegeven hoe zeker het model is over zijn voorspelling (Hinton & Sejnowski, 1999).

Een ongelabelde dataset met gegevens over klanten waarvoor je te weten wilt komen als er onderliggende groepen ontdekt kunnen worden. Dit wordt ook clustering genoemd, een mogelijke oplossing is (Lievens, 2019):

- Klanten die waarschijnlijk hun contract verlengen.
- Ontevreden klanten die bijna zeker hun contract opzeggen.
- Klanten die voor een bepaalde aanbieding misschien hun contract verlengen.

Deze resultaten bekomt men door berekeningen uit te voeren op de attribuutvector zonder een outputlabel. Deze techniek is, in het kader van dit onderzoek, minder relevant.

Reinforcement Learning

Reinforcement learning focust zich op de acties die een software agent onderneemt om een zo hoog mogelijke beloning te krijgen. Er is geen behoefte aan een dataset zoals bij gesuperviseerd en ongesuperviseerd leren omdat die zelf opgebouwd kan worden. De agent probeert een balans te vinden tussen wat hij weet en wat er kan gebeuren (Kaelbling e.a., 1996). In essentie wil dit zeggen dat de agent probeert te leren welke acties leiden tot de hoogste totale beloning (Lievens, 2019).

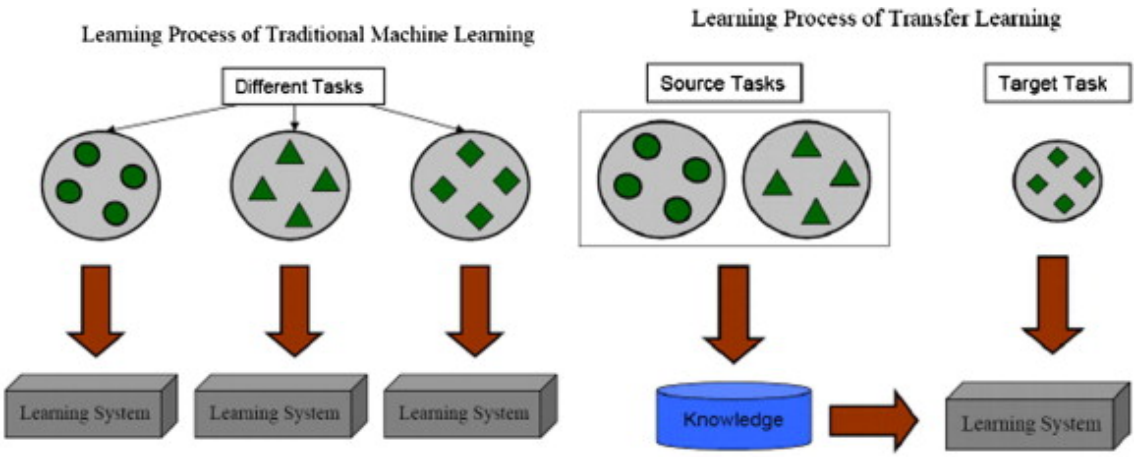
Deze techniek wordt verder besproken in het deel over geautomatiseerde *machine learning*.

In moeilijke situaties waarbij bijvoorbeeld de omgeving partieel observeerbaar is of tegen een tegenstander gespeeld wordt, kan *reinforcement learning* een zet spelen waarvoor de opbrengst voor de tegenstander zo klein mogelijk is. De voorspelling is op basis van eerder geziene situaties en eigen kennis. Door zo te werk te gaan moeten niet alle mogelijke zetten en daaraanvolgende scenario's overlopen worden, wat praktisch onmogelijk is.

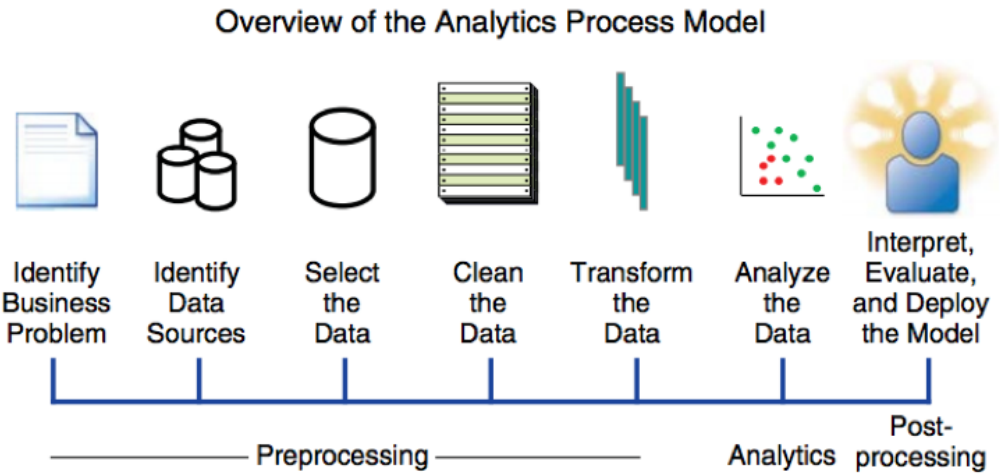
Transfer Learning

Een andere manier om een neuraal netwerk te trainen is eerst een gegeneraliseerd model maken en die later specifiek toepassen op de situatie waarin het zich bevindt. Het idee achter *transfer learning* bij afbeeldingsherkenning komt neer op een gegeneraliseerd model dat vormen, kleurveranderingen en hoeken kan herkennen. Door de laatste lagen van het model af te knippen en nieuwe meer gespecialiseerde lagen toe te voegen, zeg je in principe welke soort van de opgesomde eigenschappen het moet herkennen (Khandelwal, 2019).

Volgens Khandelwal (2019) is *transfer learning* een concept uit de *machine learning*.



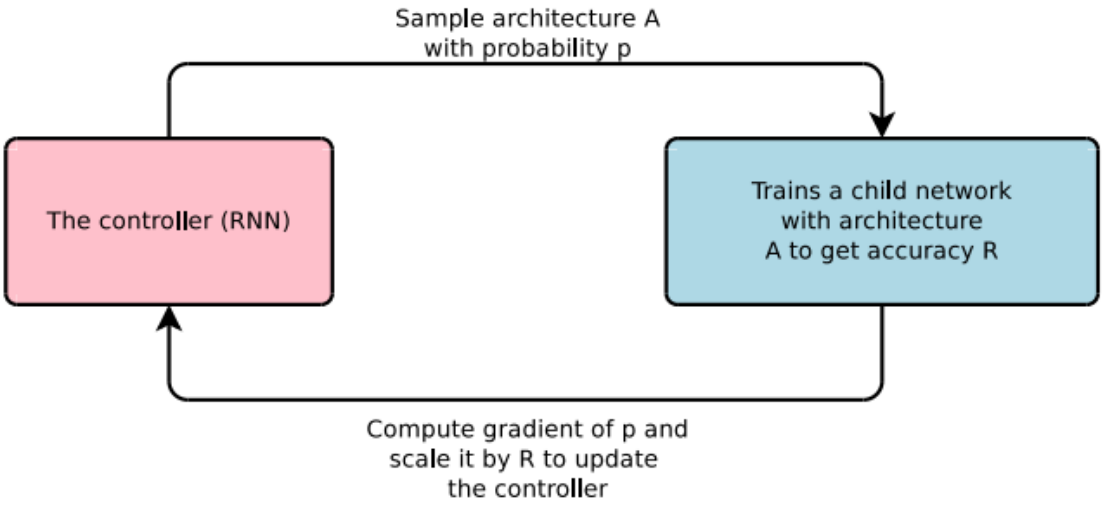
Figuur 2.2: Informatieoverdracht bij Transfer Learning (Pan & Yang, 2009).



Figuur 2.3: Verschillende stappen in een data analyse project (Lemahieu e.a., 2018)

Computer vision is een probleem dat aangepakt kan worden met traditionele *machine learning* algoritmes maar ook met *deep learning*. In het tweede geval wordt er gesproken over *deep transfer learning* waarbij het specialiseren van vooraf getrainde modellen een vaak voorkomende strategie is.

In figuur 2.2 is zichtbaar hoe een traditioneel model verschilt van een model getraind met *deep transfer learning*. Voor verschillende taken hoeft het niet volledig opnieuw getraind worden (op voorwaarde dat het binnen de generalisatie valt) en kan je de laatste lagen *fine-tunen* naar wens. Zo kan een model getraind worden om voertuigen te herkennen en later gespecialiseerd worden naar bijvoorbeeld auto's, vrachtwagens, fietsen of moto's.



Figuur 2.4: Werking van Neural Architecture Search (Zoph & Le, 2016)

2.2 Proces model bij data analyse

Volgens Lemahieu e.a. (2018) bestaat elk data analyse project uit 7 verschillende stappen, in de gegeven volgorde van figuur 2.3. Er wordt aangenomen dat 80-90% van de totale tijd gespendeerd wordt aan *preprocessing* van de data. Voordat een AutoML systeem bruikbaar is in de situatie van dit onderzoek moet het in staat zijn om zo veel mogelijk stappen van figuur 2.3 uit te voeren. Het identificeren van het business probleem en de *data sources* mogen al uitgesloten worden, de opdracht die een *development* team krijgt bevat idealiter alle informatie.

We zijn dus niet enkel op zoek naar een oplossing die analyses kan uitvoeren en interpreteren, maar ook naar iets dat data op een correcte manier kan behandelen zodat er een autonoom systeem gevormd wordt.

2.3 Neural Architecture Search

Dergelijke geautomatiseerde *machine learning* systemen gebruiken een techniek die het ontwerp van een artificieel neuraal netwerk kan automatiseren, beter bekend als *Neural Architecture Search* (Elsken e.a., 2019). De benaming verklapt al wat de kerntaak is, het zoeken naar de optimale netwerkarchitectuur optimaliseren. Uit Zoph en Le (2016) wordt vastgesteld dat deze techniek een gelijkaardige of zelfs betere performantie heeft op een vaak gebruikte dataset dan modellen die door een ML-ingenieur ontworpen zijn.

Neural Architecture Search is een kostelijk algoritme om uit te voeren op een grote dataset. De verkregen resultaten uit Zoph en Le (2016) hebben enkele weken geduurd met 800 GPU's. Daarom wordt er in Zoph e.a. (2017) voorgesteld om een model te trainen voor een (kleinere) proxy dataset en die aan de hand van *transfer learning* te extraheren naar de volledige dataset. Door op voorhand een zoekruimte te definiëren is de complexiteit van

de architectuur losstaand van de diepte van het model en de grootte van de afbeeldingen. Met andere woorden wordt er dus gezocht naar de beste lagenstructuur, want die blijven onveranderd bij het transfereren van het model. Deze aanpak komt veel sneller tot zijn resultaat want enkel de gewichten moeten aangepast worden.

De technieken uit Sectie 2.4 zijn onderliggend verwerkt en spelen een belangrijke rol binnen het domein van *machine learning* (Zoph & Le, 2016). *Reinforcement Learning* is een populaire zoekstrategie maar andere zoek- en optimalisatiealgoritmen zoals bijvoorbeeld *Bayesian optimization* zijn ook mogelijk (Elsken e.a., 2019).

Gebruik van Reinforcement Learning

Neural Architecture Search gebruikt *Reinforcement Learning* om een model te trainen. Deze manier van werken is fundamenteel anders dan gesuperviseerd / ongesuperviseerd leren omdat het model niet beter wordt door het gebruik van datasets. Als alternatief kan het neuraal netwerk beloningssignalen herkennen waardoor het kan leren welke acties leiden tot een positief resultaat (Lievens, 2019).

Op figuur 2.4 wordt gevisualiseerd hoe dit werkt. Op basis van controller structuur A (waarbij A een neuraal netwerk is) wordt een string met variabele lengte gegenereerd. Deze waarden worden gebruikt als hyperparameters om een kind-netwerk aan te maken, die getraind wordt met echte data en waarbij de accuraatheid gemeten wordt aan de hand van een validatie dataset. Het resultaat wordt gebruikt als beloningssignaal voor de controller, bij de volgende iteratie kan deze hogere kansen geven aan parameters die leiden tot accurate voorspellingen (Zoph & Le, 2016). De controller zijn zoekfunctie zal dus verbeteren met de tijd.

Verbeteringen met Network Morphism

In de tweede alinea van sectie 2.3 is al gesproken over de grote hoeveelheid resources die nodig zijn om tot de voorgestelde resultaten te komen. De afhankelijkheid van die *resources* zorgt ervoor dat er geen praktische manier is om het systeem te gebruiken als die capaciteiten niet beschikbaar zijn (Cai e.a., 2017). Cai e.a. (2017) zegt dat het verlies in performantie komt omdat elk kind-netwerk volledig vanaf nul getraind wordt, zonder kennis te hebben van vorige bewandelde paden. De technologie is bewezen, de moeilijkheid bevindt zich nu in het reproduceren met minder rekenkracht.

Denk maar aan hoe menselijke expertise tot stand komt. De herhalende taak moet kennis opleveren over de gekozen gewichten, architectuur en meer (Chen e.a., 2016). Dit zonder telkens gereset te worden.

Network Morphism tracht de bestaande kennis van een model uit te breiden door netwerk-operaties uit te voeren (lagen toevoegen, uitbreiden ...) zonder het oorspronkelijk model aan te passen (Cai e.a., 2017). Met een voorgedefinieerde transformatie functie kan bestaande kennis in een nieuw model gepompt worden zodat het dezelfde taak kan uitvoeren. Met andere parametrisatie gecombineerd met het gebruik van *reinforcement learning* (zie sectie

2.1.1) bekom je volgens Cai e.a. (2017) een *meta-controller* met verbeterde performantie.

2.4 Hyperparameter tuning

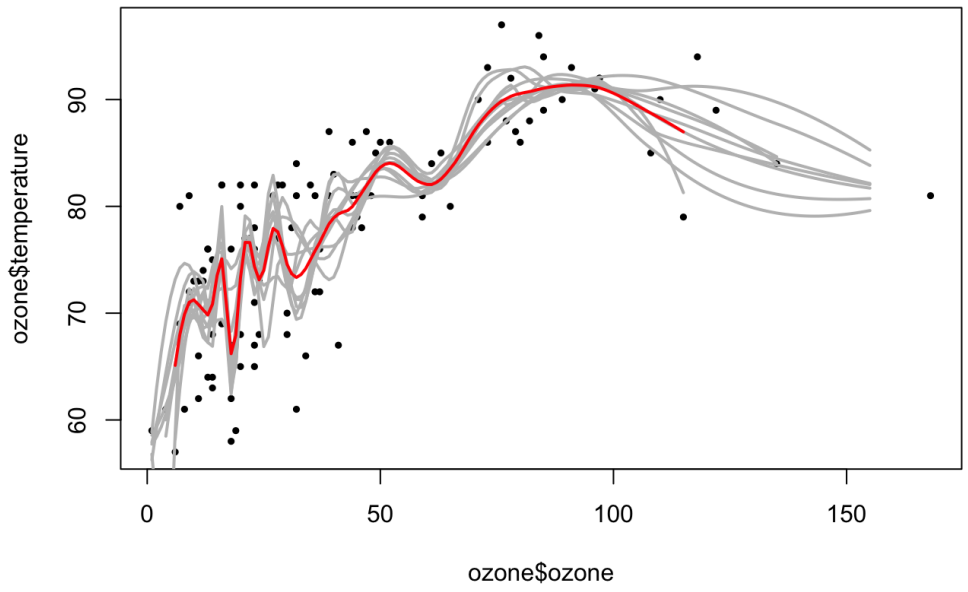
In de vorige sectie is het gebruik van parameters aan bod gekomen. *Neural Architecture Search* en *Hyperparameter tuning* zijn dan ook sterk gerelateerd aan elkaar. De normale parameters bepalen het gedrag van een neuraal netwerk en hebben een grote invloed op het eindresultaat. Het gegeven gewicht aan een parameter bepaalt hoe groot de invloed is van deze laag / neuron op de voorspelling. Google (2020c) verduidelijkt dat hyperparameters eigen zijn aan de configuratie en niet aan het model. Zo moet er bepaald worden hoeveel neuronen er in elke laag zijn. Tijdens het trainen blijven deze constant. Volgens Brust (2019) zijn er verschillende manieren om de hyperparameters te optimaliseren. *Brute force* zal elke configuratie overlopen en beslissen hoe het model vordert terwijl *feature selection* gewichten aan verschillende hyperparameters geeft. Op die manier hebben vorige simulaties een impact bij de selectie van een nieuwe set hyperparameters (Claesen & Moor, 2015). In J. S. Bergstra e.a. (2011) wordt nog gesproken over *Random Search* die in sommige gevallen dezelfde (slechte) performantie heeft als *Brute force*, en de meer gesofisticeerde methodes die verder besproken worden.

Zoph en Le (2016) stelt dat deze methoden in het algemeen minder goed werken dan Neural Architecture Search, dit is omdat de netwerk configuraties gevormd worden in een zoekruimte met vaste lengte. Bayesian optimization (J. Bergstra e.a., 2013), een variant van *hyperparameter tuning*, werkt wel met variabele zoekruimtes maar blijven minder flexibel dan wat er voorgesteld wordt bij *Neural Architecture Search*. In de praktijk bekomt men wel vaak een goed resultaat als er een initieel model bijgeleverd wordt (Zoph & Le, 2016).

2.4.1 Meta-learning

De principes achter de systemen die zelf leren en verbeteren komen uit Schmidhuber (1987). Het doel ervan is om aan de hand van randinformatie te begrijpen hoe leerproblemen flexibel worden. Die kennis kan verder ook gebruikt worden om herhalende taken te verbeteren met de tijd (Zoph & Le, 2016). De typische benaming is 'leren leren'.

Randinformatie in deze situatie wordt ook metadata genoemd. Dit is niet meer dan informatie over andere data. Voorbeelden van metadata kunnen zijn: eigenschappen van het algoritme (*metrics* die de performantie meten) of informatie over het probleem. In Schmidhuber (1993) wordt het principe getoond waarbij een zelf refererend neuraal netwerk zijn eigen gewichtenmatrix kan aanpassen. Een vaak gebruikt aanpassingsalgoritme is *gradient descent* waarbij de principes van *meta-learning* verwerkt worden.



Figuur 2.5: De grijze lijnen zijn de voorspellingen van de verschillende algoritmen, de rode lijn is het resultaat van de *ensemble*. Op de grafiek is te zien dat de uitkomst goed veralgemeniseerd is over de punten (Cen, 2016).

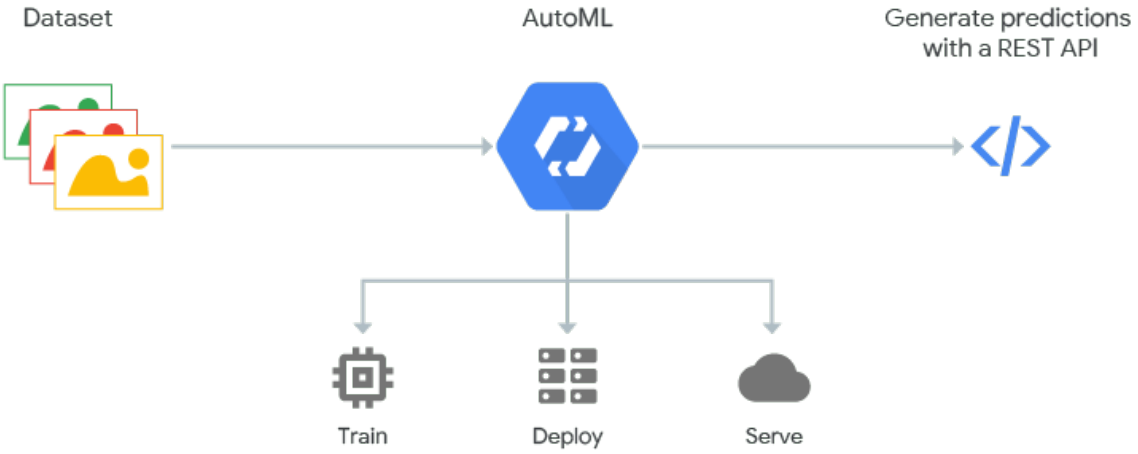
2.4.2 Ensemble construction

Met een *ensemble* bedoelt men een verzameling van *machine learning* algoritmes. Het gebruik van meerdere algoritmes die elkaar aanvullen leiden tot betere resultaten dan één enkel algoritme (Opitz & Maclin, 1999). Door de extra combinaties die nu gemaakt kunnen worden verhoogt de flexibiliteit. De simpelste manier waarop een ensemble kan werken is *bagging*[1]. Het laat elk algoritme een *overfitted* voorspelling maken, de uiteindelijke uitkomst is dan het rekenkundig gemiddelde van elke voorspelling (Decorte & De Vreese, 2019).

Zo kan je bij *random forest*, een ensemble van zoekbomen, een willekeurige factor introduceren in het trainingsproces waardoor elke zoekboom een resultaat uitkomt dat net iets van elkaar verschilt. In Decorte en De Vreese (2019) zien we dat het combineren van al deze resultaten de performantie van het finale model verhoogt zonder dat het *overfitted* is aan de trainingsdataset.

In figuur 2.5 is te zien hoe *bagging* tot zijn resultaat komt.

[1] Ook gekend als *bootstrap aggregating*.



Figuur 2.6: Stappenplan voor een geautomatiseerd model in de cloud (Google, 2019)

2.4.3 Bayesian optimization

Bayesian optimization probeert aan de hand van het probabiliteitsmodel P(score|configuratie) voorspellingen te maken over de hyperparameters (J. Bergstra e.a., 2013). De resultaten worden bekomen door aanpassingen te doen aan de vorige waarden van score en configuratie. In J. Bergstra e.a. (2013) staat dat de efficiëntie komt van het enkel overlopen van veelbelovende kandidaten uit het origineel systeem.

Zelfs met die extra abstractie laag, is het toch bruikbaar als men honderden hyperparameters moet evalueren (J. Bergstra e.a., 2013).

2.5 AutoML platformen

Het idee van geautomatiseerde toepassingen is niet nieuw, de evolutie van rekenkracht maakt het gewoon mogelijk. Er verschijnen allemaal nieuwe oplossing die runnen in de cloud of lokaal, al dan niet met grafische interfaces enzovoort. Op dit vlak is Google de koploper die het op een gebruiksvriendelijke manier aan de man brengt, maar programmeurs zijn vaak meer dan capabel om meer dan enkel een drag-en-drop systeem te gebruiken. Sommige mensen hebben een voorkeur voor *open source*, anderen gebruiken liever geen producten van een bepaald bedrijf. Het zijn allemaal zaken die de keuze van een platform kunnen beïnvloeden. Met deze platformen probeert de industrie de kloof tussen *machine learning* en een doorsnee programmeur te dichten (Gutierrez, 2019).

Een open source alternatief lijkt een goede oplossing, de interfaces zijn minder gebruiksvriendelijk dan een betalend product en er komt meer programmeerwerk aan te pas. Het resultaat is vaak commercieel bruikbaar zolang de restricties van de licentie gerespecteerd worden (Balter, 2015). AutoKeras is een voorbeeld onder de MIT licentie, die geen commerciële restricties oplegt. Samen met AutoKeras zijn tpot en Auto Sklearn de best ondersteunde libraries.

In deze sectie worden enkele mogelijkheden besproken.

2.5.1 Google Cloud AutoML

Google Cloud AutoML zorgt voor een vertrouwde interface die een gebruiker snel op weg helpt. Naast Google hebben bedrijven zoals Microsoft en Amazon een platform gebouwd op hun respectievelijke cloud infrastructuur. De AutoML service kan voordelig zijn als het bedrijf al gebruik maakt van andere producten / diensten van de leverancier, extra kosten kunnen snel de lucht in gaan zonder toegang tot andere functies (bv. van Google Cloud) als dit niet het geval is.

Zoals veel Google producten, heeft Cloud AutoML een vertrouwde interface die een gebruiker snel op weg helpt. Het proces is bijna even simpel als hun voorstelling in figuur 2.6, enkel het structureren van de dataset is niet opgenomen in de flow. De AutoML service heeft een goede kans om door te groeien in het bestaande platform van Google Cloud. De simultane werking tussen meerdere Cloud producten kan interessant zijn mocht het bedrijf gepartnerd zijn met Google.

Achterliggend zal alvast *Neural Architecture Search* gebruikt worden aangezien dat idee ontstaan is door Google Brain onderzoek (Zoph & Le, 2016). Over de manier waarop het intern gebruikt wordt is echter zeer weinig documentatie beschikbaar voor het publiek. Logisch, het blijft natuurlijk een betalende dienst. Bijgevolg, kan er enkel vanuit gegaan worden dat de abstracte concepten die eerder besproken werden, verwerkt zijn. De werkelijke combinaties en configuraties zijn puur giswerk.

Client library

De beschikbare API's in de cloud worden aangesproken door HTTP requests te sturen naar de server. Een andere manier is via *client libraries die geïntegreerd worden in de code (Google, 2020b). De interfaces bevatten voorbeeldcode die (na setup van application keys) direct bruikbaar is om datasets en modellen te managen, modellen evalueren en (batch) voorspellingen te maken (Google, 2020a). Voorspellingen van HTTP requests worden terug gestuurd als JSON, client libraries geven objecten in de gekozen programmeertaal*[2] met alle informatie terug.

2.5.2 Microsoft Azure ML Studio

Zoals bij Google Cloud AutoML zijn er ook twee manieren om aan de slag te gaan. De interface op Azure of met de Azure SDK. De *client library* is enkel beschikbaar voor Python maar de werking is gelijkaardig aan sectie 2.5.1 (Microsoft, 2020).

Fusi e.a. (2017) is de grondslag voor Azure ML Studio, het achterliggende systeem gebruikt *bayesian optimization* en leunt aan bij technieken die besproken zijn in Feurer e.a. (2015). Het bekomt een resultaat door met *collaborative filtering* te verwerken in *hyperparameter tuning*. Hierbij wordt een voorspelling gemaakt op basis van informatie die verkregen is van een vorig getraind model die gelijkaardige keuzes vertoont. De verzamelde en

[2]Enkel beschikbaar voor: C#, Go, Java, Node, PHP, Python en Ruby (Google, 2020a).

geregulariseerde gegevens worden in een matrix gegoten[3] en kan zo gebruikt worden door het model.

2.5.3 AutoKeras

Dit is een AutoML systeem gebaseerd op Keras. De bedoeling van deze library is om machine learning toegankelijk te maken voor iedereen (Jin e.a., 2019). AutoKeras is op dit moment nog in pre-release en kan nog sterk veranderen in de toekomst. Het gebruik is redelijk vanzelfsprekend en een volledige beschrijving van tekst en afbeeldingsanalyse zijn beschikbaar op de website. Van alle opgelijste mogelijkheden, vraagt deze library het meeste werk naast het voorzien van de data. Zo moet het model lokaal getraind worden en niet in de cloud, en moet de gebruiker een basis kennis Python hebben (om met de classifier van start te gaan). Een lokaal getraind model brengt ook wat voordelen met zich mee. Zo is er de mogelijkheid om het te exporteren naar een werkend Keras model dat nog verder aangepast kan worden.

Voor de Image Classifier is het voorbeeld uitgewerkt met de MNIST Hand-Written Digits, zowat de standaard dataset voor afbeeldingsherkenning. Het bevat 70000 afbeeldingen van handgeschreven letters die voorzichtig opgeschoond zijn om te gebruiken als test. Ze worden vaak gebruikt bij het schrijven van de algoritmes om verbeteringen te verifiëren. Het gevolg hiervan is dat de behaalde nauwkeurigheid niet per se representatief is op realistische voorbeelden waar afbeeldingen verschillende resoluties, kleur-schalen en aspect-ratios kunnen hebben.

Achterliggend gebruikt het *Neural Architecture Search* en het is aangewezen om het model te trainen op een machine met een externe grafische kaart die ondersteund wordt door de NVIDIA CUDA Toolkit.

2.5.4 Auto-sklearn en TPOT

Auto-sklearn is een open-source library die gebruik maakt van Bayesian optimization (Feurer e.a., 2015). Het optimalisatieproces gebruikt de secties die eerder besproken zijn in 2.4 en bestaat uit volgende stappen (Feurer e.a., 2016):

- Probabiliteitsmodel bouwen die de relatie tussen hyperparameters en de performantie meet.
- Interessante hyperparameters selecteren door een evenwicht te zoeken tussen exploratie[4] en exploitatie[5].
- Algoritme uitvoeren met de gekozen hyperparameters.

Het gegeneraliseerde proces kan gebruikt worden om algoritmes, *pre-processing* methodes en hyperparameters te selecteren.

[3] Deze techniek wordt *matrix factorization* genoemd.
[4] Zoeken op plaatsen van de zoekruimte waar het model onzeker is.
[5] Focussen op delen van de zoekruimte die leiden tot performantie.

Tree-based Pipeline Optimization Tool

Olson e.a. (2016) stelt een *data science* assistent voor. TPOT is een *machine learning pipeline* optimalisatie tool en is niet bedoeld om het proces van begin tot eind over te nemen. Het is een technologie die meer dan een basiskennis *machine learning* vergt, maar toch het vermelden waard is door het resultaat dat de gebruiker krijgt. Normaal gezien verwacht men een geëxporteerd model, bij TPOT wordt de volledige pipeline opgezet en geconverteerd naar een Python file dat nog verder geconfigureerd kan worden. Het maakt in feite een voorstel in de plaats van een definitieve keuze over de structuur van het model.

TPOT wordt samen vermeld met Auto-sklearn omdat beide gebouwd zijn op sklearn (Olson e.a., 2016).

2.6 Deployment

Voor een development team is het niet voldoende om enkel en alleen een model te trainen. Om het te kunnen gebruiken moet het ergens online staan zodat de applicatie waarin het verwerkt zit kan communiceren met het model. Bij de grote platformen zal het uitgerold worden op hun cloud services. Dat maakt nu eenmaal deel uit van hun business model. De gebruiker hangt dus gedeeltelijk vast aan zijn provider. In open source libraries is daar geen rekening mee gehouden. Een simpele oplossing is om zelf een REST API te schrijven. Er is geen beste manier omdat elke implementatie afhankelijk is van het export type van de library die je gebruikt. Om toch een representatief voorbeeld te geven wordt het deployment proces van een Keras model (export type van AutoKeras) besproken.

Een goede strategie is om je API te bouwen in de taal waarin de library geschreven is. Zo kan je de bestaande interfaces van het geëxporteerde model direct aanspreken. In dit geval is het Python + Flask framework. Om het op een productie niveau te krijgen is het een goed idee om ook Redis[6] te implementeren. Dan rest enkel nog de hosting, waardoor je soms toch uitkomt bij de grote platformen zoals Amazon AWS, Google Cloud Hosting enzovoort.

In Rosebrock (2018) is de volledige code te vinden om stap voor stap uit te voeren wat hierboven beschreven is.

[6]Redis is een gedistribueerde key-value store die volledige objecten in zijn geheugen kan opslaan en in deze situatie gebruikt wordt om wachtrijen te optimaliseren (Rosebrock, 2018).

3. Methodologie

Om tot een zo correct mogelijk resultaat te komen is het belangrijk om een grondige kennis te hebben van de geïmplementeerde algoritmen, modellen en platformen. Die zijn samen gebundeld tot een huidige stand van zaken en zijn terug te vinden in hoofdstuk 2.

Er werden meerdere prototypes opgezet die telkens volgens dezelfde criteria gequoteerd zijn. Verdere informatie over de gebruikte datasets is beschreven in sectie 3.1. De vereisten waaraan elk systeem zo veel mogelijk aan moet voldoen worden toegelicht aan de hand van een requirements analyse in sectie 3.2.

Het hoofddoel van dit onderzoek is ontdekken of de huidige staat van de technologie bruikbaar is in een bedrijfscontext zonder extreme operationele veranderingen. Dit kan afhangen van kosten om het systeem te gebruiken, extra mankracht die nodig is, eventuele omscholing enzovoort. De conclusie werd rond die vereisten geformuleerd. Omdat er verschillende implementaties getest zijn is het ook een vergelijkende studie en werden ze onderling tegen elkaar geëvalueerd. Dit niet alleen vanuit een technologisch standpunt (wat onderzocht werd in deze studie) maar ook de bijdrage die het levert aan de *business value*. Om dit zo correct mogelijk te doen is in de achtergrond van de conclusie rekening gehouden met resultaten uit een studie vanuit een economisch standpunt.

3.1 Dataset

De keuze van de dataset is belangrijk. Dit onderzoek wil de capaciteiten van AutoML systemen testen op realistische situaties. De traditionele datasets (CIFAR, MNIST ...) waarmee deze algoritmen getraind zijn vallen onmiddellijk uit de selectie zoals uitgelegd in sectie 2.5.3.


Figuur 3.1: Kat en hond zijn de focus van de afbeelding.


Figuur 3.2: Kat en hond zijn niet de focus van de afbeelding.

3.1.1 Cats vs dogs

Alle systemen zijn getraind met een online[1] dataset over katten en honden. Ooit was de dataset het onderwerp van een *data science* wedstrijd en mogen nu dus vrij gebruikt worden. Het bestaat uit 250000 foto's die gelijkaardig zijn aan de afbeeldingen in figuur 3.1. Er is niks van *preprocessing* toegepast, de foto's kunnen dus evengoed op uw smartphone staan. Door realistische foto's te gebruiken introduceren we een nieuwe moeilijkheid voor de optimalisatiealgoritmen, zo moeten ze ook rekening houden met volgende zaken:

- Foto's kunnen zeer verschillende resoluties hebben
- De kleuren zijn in 3 dimensies (rood, groen, blauw)
- Door het originele formaat te behouden en de afbeelding niet te knippen, kan het zijn dat de focus van de afbeelding niet op het dier ligt (zie afbeeldingen in figuur 3.2)

Onduidelijke foto's zijn in de minderheid maar handig om te ontdekken waaraan het model gevoelig is.

Omdat de *dataset* gebruikt is in een wedstrijd zijn er honderden inzendingen beschikbaar van mensen die een poging gedaan hebben om het probleem op te lossen. Een vergelijking van performantie met onze geautomatiseerde modellen tegenover modellen die door een persoon zijn samengesteld is op zijn plaats.

[1] https://www.kaggle.com/c/dogs-vs-cats/data

3.2 Requirementsanalyse

De verwachte functionaliteiten kunnen opgesplitst worden in twee categorieën. Enerzijds zijn er de functionele requirements die de gewenste functionaliteiten en het gedrag van een systeem beschrijven. Anderzijds de niet-functionele requirements, een oplijsting van kwaliteitseisen waaraan het moet voldoen.

Functionele requirements

- Ondersteuning voor verschillende resoluties van afbeeldingen
- Kan alle stappen uit het procesmodel uitvoeren (sectie 2.2)
- Mogelijkheid om performantie te meten
- *Batch* verwerking is ondersteund

Niet-functionele requirements

- Moet snel kunnen *deployen* naar een productieomgeving
- Performantie is vergelijkbaar met modellen die door een *data scientist* gemaakt zijn
- Als ondersteunende technologie mag het niet duurder zijn dan een *data scientist*
- Programmeurs met weinig of geen ervaring moeten het kunnen gebruiken
- Het resultaat kan ingebouwd worden in bestaande applicaties

Er is geen sprake van requirements die *nice to have* of dergelijke zijn. Om te slagen moeten alle eisen in een bepaalde mate aanwezig zijn. Ze werden dan ook één voor één besproken.

4. AutoKeras

In dit hoofdstuk wordt het volledige AutoKeras proces uitgelegd. Dit door belangrijke stukken code te nemen en expliciet toe te lichten welke keuzes gemaakt zijn en waarom. Ook *performance metrics* worden hier verzameld en vergeleken. Het model is met verschillende parameter configuraties getraind. Elke variant blijft wel hetzelfde code skelet gebruiken.

De ontwikkeling is volledig in *Jupyter notebooks*[1] uitgewerkt. Om de code makkelijk lokaal te gebruiken is *Anaconda Navigator*[2] aangeraden, de meeste *packages* worden standaard mee geïnstalleerd. De code bevindt zich ook in Appendix B en kan online bekeken worden[3].

Het is ook mogelijk om de notebook in de cloud uit te voeren. De dataset (omgezet naar een numpy array) en de notebooks kunnen direct geüpload worden naar Google Colab of Kaggle. De gebruiker heeft dan voor een bepaalde duur gratis toegang tot grafische kaarten.

Volgende *packages* moeten zelf geïnstalleerd worden:

- tensorflow-gpu (2.1.0)
- autokeras (1.0.2)
- graphviz (0.13.2)

[1] https://jupyter.org/
[2] https://docs.anaconda.com/anaconda/navigator/
[3] https://github.com/robbedec/bachproef-automl

4.1 Voorafgaand werk

4.1.1 GPU activatie

Het verloop van de code is ook hoe je een model van nul kan trainen. Er wordt gewerkt met een *CUDA integrated GPU*, om te testen als *tensorflow* wel echt de GPU gebruikt kunnen er enkele commando's uitgevoerd worden. Als in de output een vermelding over de GPU staat mag u er vanuit gaan dat de die actief is. Meer informatie over CUDA drivers en hoe ze geïnstalleerd moeten worden is te vinden op de site van NVIDIA[4].

```
from tensorflow.python.client import device_lib

def get_available_devices():
    local_device_protos = device_lib.list_local_devices()
    return [x.name for x in local_device_protos]
print(get_available_devices())
```

Een mogelijkse output kan er als volgt uitzien: ['/device:CPU:0', '/device:GPU:0'].

Op de cloud platformen moeten geen speciale drivers geïnstalleerd worden maar enkel een *accelerator* geactiveerd zijn in de *notebook*.

4.1.2 Data transformatie

De dataset bestaat deels uit een gelabelde afbeeldingen. Het bevat ook een verzameling ongelabelde afbeeldingen die tijdens de wedstrijd gebruikt werden als verificatie. In het kader van het onderzoek kunnen deze weggelaten worden en gaan we aan de slag met de gelabelde data. Elke afbeelding wordt genormaliseerd naar een standaard resolutie en omgezet naar een *numpy array*[5]. Dit zijn de kleurwaarden van elke pixel die samen een matrix vormen. Ook het label van de afbeelding is opgenomen in de matrix (0 = hond, 1 = kat). Momenteel staat alle data nog gegroepeerd volgens categorie. Voor de volgende stap is het belangrijk dat dit niet het geval is. Daarom wordt de matrix nog eens random gesorteerd.

Om consistentie tussen de experimenten te behouden wordt de matrix opgeslagen en bij een nieuwe configuratie terug ingeladen. De data transformatie wordt zo éénmaal uitgevoerd.

```
def get_label(file):
    class_label = file.split('.')[0]
    if class_label == 'dog': label_vector = 0
    elif class_label == 'cat': label_vector = 1
    return label_vector

def get_data():
    data = []
```

[4] https://developer.nvidia.com/hpc

[5] In de code is 'np' een verwijzing naar *numpy*

```
    files = os.listdir(INPUT_PATH)

    for image in tqdm(files):
        label_vector = get_label(image)
        img = Image.open(INPUT_PATH + image).convert('L')
        img = img.resize((SIZE,SIZE))

        data.append([np.asarray(img),np.array(label_vector)])

    shuffle(data)
    return data
```

4.2 Data preprocessing

Een van de requirements is dat de *preprocessing* voor de gebruiker gedaan wordt. Bij AutoKeras is dit niet 100% het geval en moeten er zelf nog enkele zaken uitgevoerd worden. Zo moet de matrix die gemaakt is in sectie 4.1.2 opgedeeld worden in training en validatie datasets.

In *data science* projecten wordt typisch x_test en x_train gebruikt om de *features* van de afbeeldingen op te slaan, de matrix met pixel waarden. y_test y_train bevatten dan het bijhorend label. In de tweede stap wordt er dimensiereductie toegepast. De algoritmes houden geen rekening met kleuren dus die mogen we weglaten. De drie kleurendimensies (Rood, Groen, Blauw) worden omgezet naar één dimensie (grijswaarden). Deze conversie gebeurt in de laatste twee lijntjes van onderstaande code.

```
x_train = np.array([data[0] for data in train], 'float32')
x_test = np.array([data[0] for data in test], 'float32')
y_train = [data[1] for data in train]
y_test = [data[1] for data in test]

x_train = np.array(x_train).reshape(-1,SIZE,SIZE,1)
x_test = np.array(x_test).reshape(-1,SIZE,SIZE,1)

x_train /= 255
x_test /= 255
```

4.3 Model trainen en evalueren

Na alle voorbereiding is het tijd om de *classifier* in gang te steken. Hierbij moet enkel een naam (voor het uiteindelijke model) en het maximum aantal pogingen meegegeven worden. Op voorhand kan er niet ingesteld worden hoe lang er gezocht mag worden. Het is wel mogelijk dat het vroegtijdig beëindigd wordt (voor max. aantal pogingen bereikt is) omdat de loss value niet genoeg zakt over x-aantal iteraties.

Met de `clf.fit()` methode wordt het best passend model gekozen. Met de test dataset kan uiteindelijk de performantie van het model gemeten worden. `clf.evaluate()` geeft twee scores terug. Eerst de *loss value*, het is een interpretatie die kijkt hoe goed het model scoort door training en test data te nemen. Let wel op dat de test dataset die wij definiëren enkel gebruikt wordt als `clf.evaluate()` uitgevoerd wordt. AutoKeras neemt van de trainings dataset standaard 20% om te gebruiken als interne test data tijdens het trainen. De tweede waarde is het percentage van correct voorspelde afbeeldingen.

```
clf = ak.ImageClassifier(max_trials=MAX_TRIES, name=OUTPUT_NAME)
clf.fit(x_train,y_train, verbose=2)

score = clf.evaluate(x_test, y_test)
```

Momenteel zit er een bug in het *greedy* optimalisatie algoritme (dat standaard gebruikt wordt) van AutoKeras waardoor *classifiers* met een max. aantal pogingen > 5 vroegtijdig beëindigd worden. Een voorlopige oplossing is om een de API van Automodel te gebruiken en zelf de instellingen te configureren. Achterliggend gebeurt dit ook als `ak.ImageClassifier()` geïnitialiseerd wordt. Als vervanging is er gekozen om *random search* te gebruiken (zie sectie 2.4).

```
clf = ak.AutoModel(
    inputs=[ak.ImageInput()],
    outputs=[ak.ClassificationHead()],
    tuner="random",
    max_trials=MAX_TRIES,
    name=OUTPUT_NAME,
    overwrite=False
    )

clf.fit(x_train, y_train, verbose=2)
```

4.4 Model visualisatie

4.4.1 Confusion matrix

De *confusion matrix* is waarschijnlijk één van de beste visualisaties om een inzicht in het model te krijgen. Voor elke categorie wordt het aantal juiste voorspellingen getoond maar ook het aantal afbeeldingen die aan de verkeerde categorie toegekend zijn. Een voorbeeld van een *confusion matrix* is te vinden in afbeelding 4.2.

```
plt.style.use('classic')
%matplotlib inline

mat = confusion_matrix(y_test, predictions.round())
labels = ['dog', 'cat']
```

```
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                                    xticklabels=labels, yticklabels=
                                    labels)
plt.xlabel('true category')
plt.ylabel('predicted category')
```

4.4.2 Verkeerde voorspellingen

Om inzicht te krijgen in de verkeerde voorspellingen is het handig om die terug als afbeeldingen te tonen. Het kan een manier zijn om als persoon een patroon te vinden waartegen het model fouten maakt. Het zo gezegde ruis op de afbeeldingen.

De eerste stap om voorspellingen op te splitsen is het interpreteren van de waarden. Zo krijg je bij een binair classificatieprobleem altijd een getal tussen 0 en 1 terug. Het kantelpunt is dan $\frac{1}{2}$, elke voorspelling die kleiner is wordt geclassificeerd als klasse 0 en alles groter als klasse 1. Om elke verkeerde voorspelling te zoeken kunnen we simpelweg het resultaat afronden en vergelijken met het bijhorend label.

```
images = x_test.reshape(5000,64, 64)
incorrect_predictions = []

for i in range(0, len(predictions)):
    if predictions[i].round() != y_test[i]:
        incorrect_predictions.append(
            (i, images[i], predictions[i].round(4), y_test[i]))
```

In sectie 4.2 zijn alle afbeeldingen omgezet naar grijswaarden. Als deze nu opnieuw geconverteerd worden zijn ze uiteraard zwart-wit foto's. Bij elke afbeelding wordt nog een legende gegenereerd. Zo is duidelijk te zien wat de index van de afbeelding is, wat de voorspelling van het model is (afgerond tot vier cijfers) en de werkelijke waarde.

```
%matplotlib inline

figure, axes = plt.subplots(nrows=6, ncols=4, figsize=(16,16))

for axes, item in zip(axes.ravel(), incorrect_predictions):
    index, image, predicted, expected = item
    axes.imshow(image, cmap=plt.cm.gray_r)
    axes.set_xticks([])
    axes.set_yticks([])
    axes.set_title(f'index: {index}\np: {predicted}; e: {expected}'
                                        )

plt.tight_layout()
```

Het resultaat van deze code is te zien in afbeelding 4.3 en 4.4.

4.4.3 Overzicht van de lagen

Het resultaat is een combinatie van Keras lagen (inputlagen, activatielagen, normalisatielagen...). Keras heeft een ingebouwde functie `model.summary()` die volgend resultaat oplevert:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 64, 64, 1)]       0
_________________________________________________________________
normalization (Normalization (None, 64, 64, 1)         3
_________________________________________________________________
conv2d (Conv2D)              (None, 62, 62, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 60, 60, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 30, 30, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 30, 30, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 57600)             0
_________________________________________________________________
dropout_1 (Dropout)          (None, 57600)             0
_________________________________________________________________
dense (Dense)                (None, 1)                 57601
_________________________________________________________________
classification_head_1 (Sigmo (None, 1)                 0
=================================================================
Total params: 76,420
Trainable params: 76,417
Non-trainable params: 3
_________________________________________________________________
```

Het overzicht kan ook geëxporteerd worden naar een afbeelding die beter geformatteerd is. De geëxporteerde afbeeldingen bevinden zich in Appendix B.

4.5 Resultaten

Zoals eerder vermeld zijn er 2 modellen getraind met als enigste configuratieverschil, het maximum aantal pogingen. In andere woorden is dit het aantal verschillende Keras modellen dat aangemaakt mag worden. De resultaten van beide zijn samengevat in tabel 4.1.

Verder worden beide modellen vermeld als model A (max. 5 pogingen) en model B (max. 10 pogingen).

De specificaties van het host systeem zijn:

- CPU: i7-7700HG
- GPU: Nvidia GeForce GTX 1050 Ti Max-Q
- RAM: 16GB

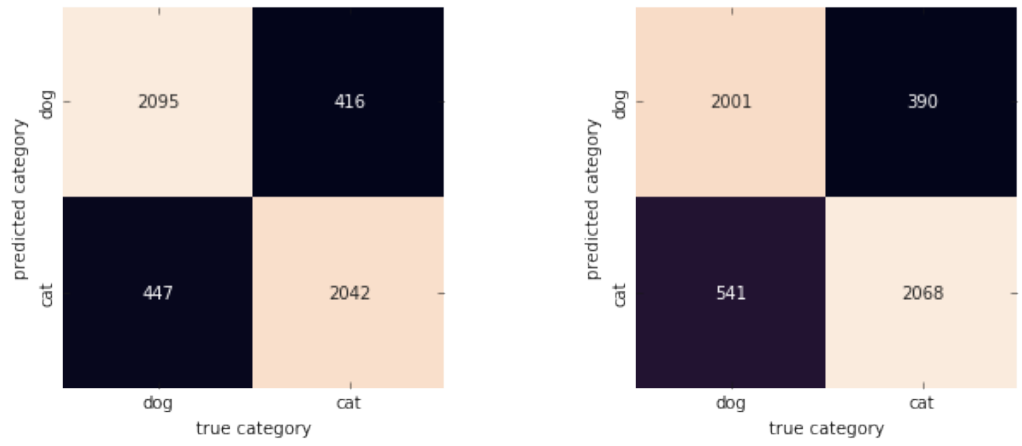
Max aantal pogingen	Accuracy	Loss	Trainingsduur	Aantal lagen
5	82.74%	0.6383	2u	10
10	81.37%	0.7110	8u	10

Tabel 4.1: Resultaten AutoKeras.

Wat meteen opvalt is dat een hoger aantal pogingen geen zekerheid is op betere prestaties. Zo wordt enkel de mogelijkheid gegeven op een uitgebreidere exploratie van de zoekruimte. Zo kan het algoritme bijvoorbeeld een extreem *overfitted* model trainen in de hoop om die later te combineren met een andere om een beter resultaat te bekomen, een toepassing van *ensemble construction* (zie sectie 2.4.2). Afbeelding 4.1 komt rechtstreeks uit de output van model B en is niet te vinden in model A.

```
Epoch 137/1000
500/500 - 137s - loss: 0.2159 - accuracy: 0.9882 - val_loss: 0.6076 - val_accuracy: 0.6705
Epoch 138/1000
500/500 - 137s - loss: 0.2144 - accuracy: 0.9886 - val_loss: 0.6074 - val_accuracy: 0.6702
Epoch 139/1000
500/500 - 137s - loss: 0.2130 - accuracy: 0.9891 - val_loss: 0.6073 - val_accuracy: 0.6710
Epoch 140/1000
500/500 - 136s - loss: 0.2115 - accuracy: 0.9895 - val_loss: 0.6071 - val_accuracy: 0.6715
Epoch 141/1000
500/500 - 136s - loss: 0.2101 - accuracy: 0.9898 - val_loss: 0.6070 - val_accuracy: 0.6712
Epoch 142/1000
500/500 - 136s - loss: 0.2087 - accuracy: 0.9900 - val_loss: 0.6068 - val_accuracy: 0.6715
Epoch 143/1000
```

Figuur 4.1: Het model expres *overfitten* in het kader van *ensemble construction*.



(a) Max. pogingen = 5

(b) Max. pogingen = 10

Figuur 4.2: Overzicht confusion matrices

De *confusion matrices* in afbeelding 4.2 tonen dat verspreiding van voorspellingen in beide gevallen redelijk overeen komt. Er is te zien dat model B meer aanleunt (en dus meer fouten maakt in die categorie) om kat te voorspellen dan hond, iets dat in model A in evenwicht is.

Van beide modellen zijn 24 verkeerd voorspelde afbeeldingen gevisualiseerd (zie Figuren 4.3 en 4.4). Bij sommige afbeelding kan duidelijk afgeleid worden wat de oorzaak is, mensen of andere onbekende objecten op de foto's. Andere zijn moeilijker te verklaren. Voor nu kan er besloten worden dat het model waarschijnlijk moeite heeft met de gespitste oren. Een typisch kenmerk van katten maar het komt soms ook voor bij honden. Maar ook het feit dat soms het volledige lichaam afgebeeld is en soms maar een deel van het hoofd.

Met een gemiddelde score van 82% bekomen we een goed resultaat als er rekening gehouden wordt met de beperkte hoeveelheid *data preprocessing*. Om dit op productie niveau te krijgen moet er wel meer moeite gedaan worden om de trainingsdataset beter te verwerken. Handgemaakte modellen uit de competitie behalen scores tot 98%[6].

Tijdens het trainingsproces heeft de aanwezigheid van een TPU geen invloed op de snelheid. Deze wordt gewoonweg niet gebruikt. Een TPU of *tensor processing unit* is een schakeling van *accelerators* die het proces vele malen sneller uitvoert. AutoKeras bied op dit moment geen ondersteuning om deze te gebruiken waardoor er teruggevallen wordt op de GPU's. De afwezigheid vormt wel een nadeel aangezien TPU's verder uitgroeien tot de nieuwe industrie standaard.

Voor een open-source library zit er zeker potentieel in voor de toekomst als u weet dat op een gelijkaardig niveau van *preprocessing* en dezelfde dataset, een pre-release versie van AutoKeras slechts 67% behaalde (Chopra e.a., 2019).

4.6 Requirements

4.6.1 Functionele requirements

Implementatie van het procesmodel

Er is geen sprake van een volledige implementatie. Als open source project dient het in de eerste plaats als een onderzoeksproject. AutoKeras is simpelweg hun contributie aan de *state of the art*. Niet commercieel onderzoek heeft weinig last van economische druk waardoor nieuwe *features* soms lang op zich laten wachten.

Bij AutoKeras moet *data preprocessing* zelf uitgevoerd worden. De optimalisatiealgoritmen gaan zelf de belangrijke *features* zoeken en selecteren. Dit houdt in dat de data getransformeerd moet worden (numpy array in de juiste dimensies), opgesplitst en aangepast volgens de intenties van de gebruiker vooraleer het gebruikt kan worden door `ImageClassifier()`.

[6] https://www.kaggle.com/c/dogs-vs-cats/leaderboard

Figuur 4.3: Verkeerd voorspelde afbeeldingen van model A.

Figuur 4.4: Verkeerd voorspelde afbeeldingen van model B.

Beschikbare metrics

De output van het algoritme wordt geëxporteerd naar een gecompileerd Keras model, onderling gebruikt het ook Keras lagen. Dit wil zeggen dat alle mogelijke visualisaties van Keras perfect mogelijk zijn met de output. In het experiment gaat het dan vooral over `model.summary()` en gelijkaardige output die ons een inkijk in het resultaat geeft.

Omdat de ruwe resultaten ook beschikbaar zijn, is de gebruiker vrij om andere *packages* te gebruiken. Zo kan met een lijst voorspellingen een *confusion matrix* gemaakt worden aan de hand van de scikit-learn *library*. De array van grijswaarden kan ook terug omgezet worden naar afbeeldingen om bijvoorbeeld, verkeerde voorspellingen te tonen. Alle verschillende mogelijkheden zijn enkel begrensd door het aantal beschikbare *packages*.

Met de wiskundige formules kunnen ook allemaal verschillende statistieken berekend worden.

Batch verwerking

Voor een AutoKeras *classifier* bestaat op de `ImageClassifier.fit()` methode een *batch_size* argument die dit kan manipuleren. Deze optie bepaalt hoeveel afbeeldingen er bij elke iteratie gepropageerd worden door het model. Door *mini batches* te gebruiken moeten niet alle afbeeldingen op hetzelfde moment in het geheugen van de computer geladen worden. Onderliggend gebeuren er meer updates aan de *gradient* die tot betere resultaten kunnen leiden.

4.6.2 Niet-functionele requirements

Deployment

De *library* biedt geen mogelijkheden aan om een model te *deployen*. Als gebruiker ben je zelf verantwoordelijk om de benodigde infrastructuur te voorzien. Ook zijn er mogelijks uitbreidingen nodig om de implementatie op productieniveau te krijgen. Meer informatie hierover in sectie 2.6.

Performantie

Het uiteindelijke resultaat in ons experiment is bruikbaar. Met 83% is de technologie op zich bewezen. Betere resultaten hangen sterk af van de hoeveelheid en kwaliteit van de *data preprocessing*.

Kostprijs

Het gebruik van de *library* zelf is volledig gratis. Ook voor het resultaat moet niet betaalt worden. Er moet zelf gekozen worden over welke investeringen gedaan worden in verband met de infrastructuur (aantal GPU's, kloksnelheid, elektriciteit ...). Op zich kan een

model getraind worden op een (gratis) online platform zoals Kaggle, of Google Colab. Om langere GPU tijden te verkrijgen kan een pro account aangeschaft worden.

Verwerking

Omdat de meeste *wrappers* geschreven zijn in Python, is het een goede keuze om er een eigen API mee te maken. Voor Flask zijn er online zeer veel *tutorials* beschikbaar. Ook hierover is meer informatie te vinden in sectie 2.6.

5. Google Cloud AutoML

Google Cloud AutoML is zoals de naam zegt, deel van het Cloud platform. In dit hoofdstuk wordt gebruik gemaakt van een *storage server* en de *vision API*. Om toegang te krijgen is uiteraard een account nodig, en als nieuwe gebruiker is er $300 beschikbaar die vrij gespendeerd kan worden op het platform (binnen de periode van één jaar).

De *vision API* is een verzameling van afbeeldings- en videoherkenning die verwerkt is in het AutoML platform. Daarnaast is er ook nog ondersteuning voor *natural language processing* en andere *beta* diensten. Alle mogelijkheden hebben een uitgebreide documentatie[1] die deels verwerkt werd in de literatuurstudie. De interfaces zijn zeer gebruiksvriendelijk en maken vaak gebruik van drag en drop systemen.

Vooraleer er gestart kan worden is een korte setup op zijn plaats. De volledige code van het gebruikte *jupyter notebook* is te vinden in Appendix C.

5.1 Voorafgaand werk

5.1.1 Google Cloud Data Storage

Om de *computer vision API* te gebruiken moeten de afbeeldingen geüpload worden. Dit kan rechtstreeks in batches van max. 30 MB. Met de hoeveelheid afbeeldingen die gebruikt worden in dit onderzoek is dit niet praktisch. Voor grote hoeveelheden data kan daarom een online bucket gebruikt worden. De afbeeldingen bevinden zich dan op een *data storage server* van Google Cloud.

[1] https://cloud.google.com/automl/docs?hl=nl

De indeling van de data moet er als volgt uitzien:

```
online bucket
├── cat
│   ├── cat.0.jpg
│   ├── cat.1.jpg
│   └── cat.2.jpg
└── dog
    ├── dog.0.jpg
    ├── dog.1.jpg
    └── dog.2.jpg
```

Dit is belangrijk voor de volgende stap. Op deze manier kunnen er makkelijk labels toegekend worden aan de afbeeldingen. Dit op een manier die het platform kan interpreteren.

5.1.2 Structuur van de data

Nu alle data online beschikbaar is, moet er nog een overzicht gecreëerd worden. Aan de hand van een *jupyter notebook* wordt een csv gemaakt die voor elke afbeelding, de locatie op Google Cloud Storage en het bijhorend label bevat. Alle niet afbeeldingen worden uit de map gefiltered

```
data_array = []

for (dict_key, files_list) in files_dict.items():
    for filename in files_list:
        if '.jpg' not in filename:
            continue # don't include non-photos

        label = dict_key
        data_array.append((base_gcs_path + dict_key + '/' +
                                            filename , label))
```

Nu moet `data_array` enkel nog ingeladen worden in een *pandas dataframe*. In de *pandas library* zit een functie die een dataframe kan opslaan als een csv bestand. Merk op dat de indexes van de kolommen en rijen niet mee opgeslagen worden. Deze kunnen als `False` geflagged worden in de functie.

```
dataframe.to_csv('all_data.csv', index=False, header=False)
```

De output van het csv bestand zou er als onderstaand uit moeten zien. Natuurlijk met een andere bucket naam.

gs://rdc-automl-catsvsdogs/dog/dog.100.jpg dog
gs://rdc-automl-catsvsdogs/dog/dog.1000.jpg dog
... ...
gs://rdc-automl-catsvsdogs/cat/cat.9995.jpg cat
gs://rdc-automl-catsvsdogs/cat/cat.9996.jpg cat

Het csv bestand moet dan ook toegevoegd worden aan de bucket.

5.2 Model trainen en evalueren

Om het trainen te starten moet nu enkel het csv bestand (dat op Google Cloud Storage staat) geselecteerd worden. Vervolgens worden alle afbeeldingen geïmporteerd en opgesplitst in training, validatie en testen. In ons geval van 22500 afbeeldingen worden ze als volgt opgesplitst.

- Training: 10000
- Validatie: 5000
- Testen: 2500

Elke dataset is gelijk verdeeld over de twee categorieën. Als er dus over 10000 afbeeldingen gesproken wordt, zijn het in feite 5000 katten en 5000 honden.

Eenmaal een model getraind is, kan het nog verbeteren door extra *node* uren te alloceren aan het model. Het trainingsproces kiest een vorige stap en hervat de exploratie in dat deel van de zoekruimte. Een bestaand model kan niet verder getraind worden met een nieuwe dataset. Het is wel mogelijk om meerdere modellen te trainen in hetzelfde project zodat er na (positieve) aanpassingen in de dataset een evolutie in de modellen is. Deze worden *edge exportable models* genoemd (Google, 2020d).

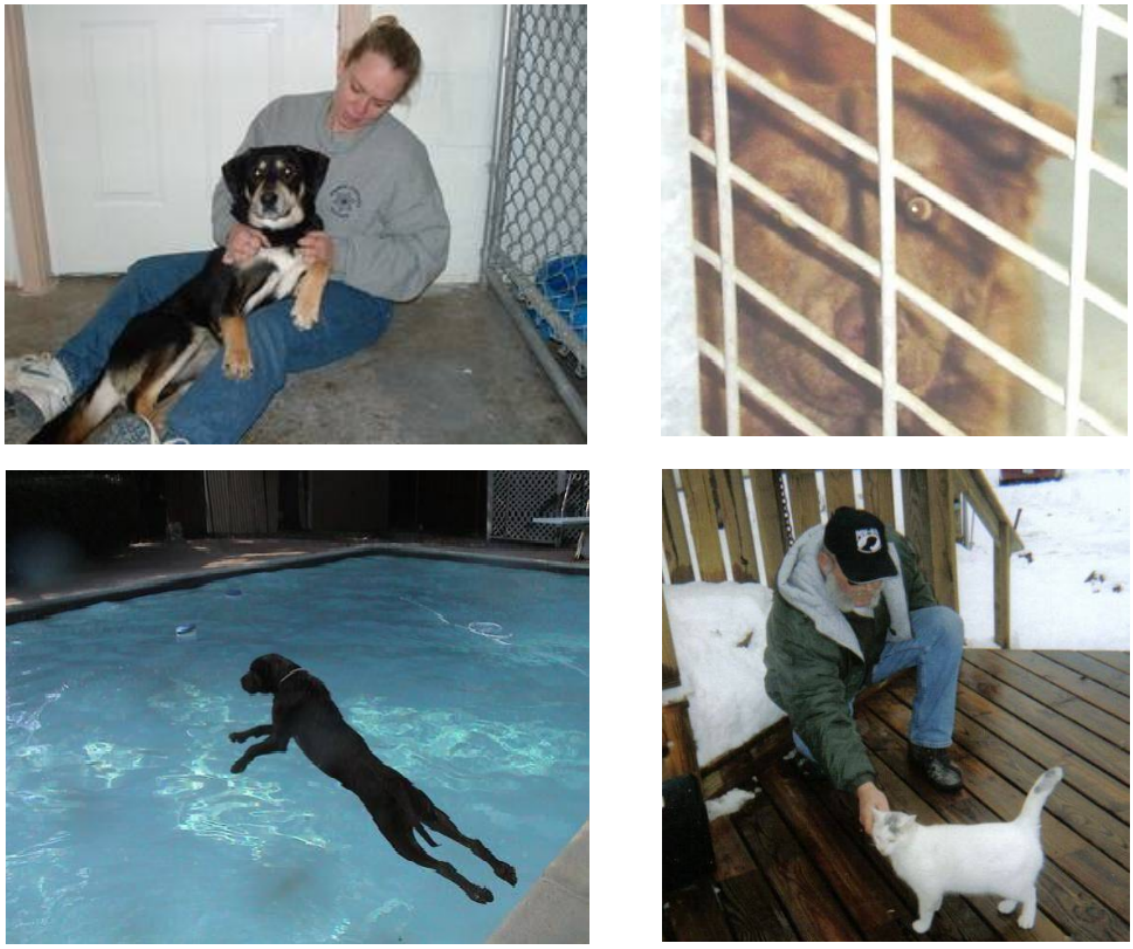
5.3 Resultaten

Het model behaalt een score van 99% en komt zo overeen met de beste inzending van de Kaggle wedstrijd. De stelling van Google, om *machine learning* makkelijk en toegankelijk te maken klopt dus. Met namelijk niks van *preprocessing* of dergelijke scoort het toch uitstekend.

Afbeelding	Voorspelling	Zekerheid	Werkelijk label
1	Hond	100%	Hond
2	Hond	66%	Hond
3	Hond	100%	Hond
4	Hond	97%	Kat

Tabel 5.1: Vier voorspelde afbeeldingen in Google Cloud AutoML (Figuur 5.1)

Op de testdataset van 2500 afbeeldingen zijn er slechts negen verkeerde voorspellingen. De *metrics* in afbeelding 5.2 zijn wel de enigste feedback over het model. Zo is het niet mogelijk om verkeerd voorspelde afbeeldingen te filteren of de lagen van het model te bekijken.



Figuur 5.1: Vier 'onduidelijke' voorspelde afbeeldingen

Alle labels

Totaal aantal images	22.494
Testitems	2.499
Precisie	99,64%
Geretourneerde relevante resultaten	99,64%

Use the slider to see which confidence threshold works best for your model on the precision-recall tradeoff curve.
Learn more about these metrics and graphs.

(a)

Correct label \ Label Voorspeld	cat	dog
cat	1.244	6
dog	3	1.246

(b) Confusion matrix

Figuur 5.2: Performantie van het model

Om het model snel zelf eens te testen, werden er vier onduidelijke foto's (figuur 5.1) voorspeld. Dit kan in de *cloud interface* voor *batches* van maximaal tien afbeeldingen. De resultaten van de voorspellingen staan in tabel 5.1.

5.4 Deployment

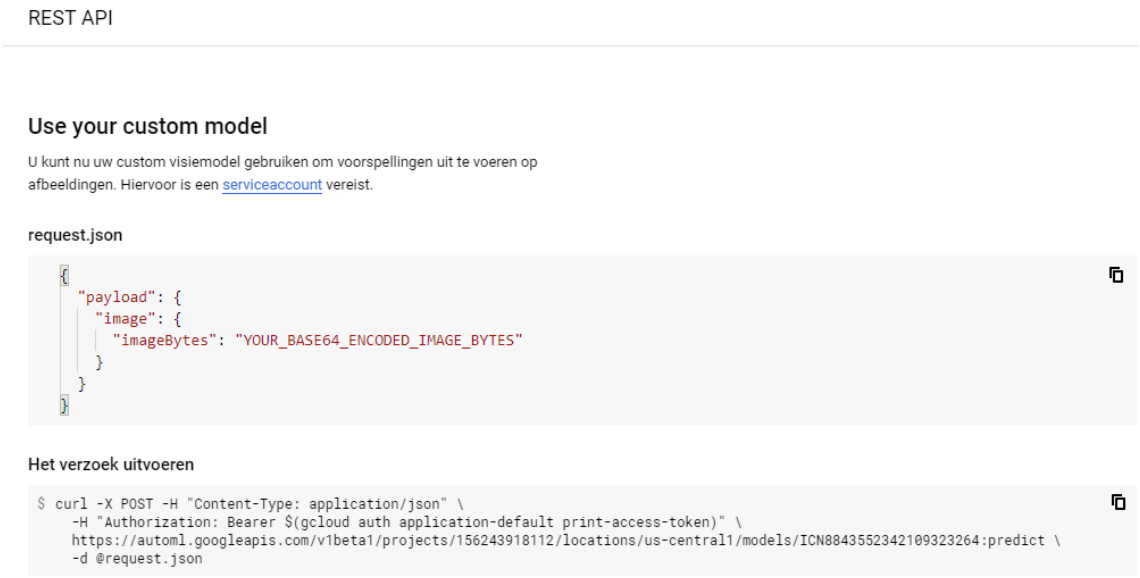
Zoals besproken in sectie 2.5.1 kunnen er interacties zijn via een *client library*. De code kan ofwel intern in de applicatie verwerkt worden (indien de code geschreven is in Python). Een tweede manier is om de het Python script extern op te slaan en op te roepen als het nodig is.

```
import sys

from google.cloud import automl_v1beta1
from google.cloud.automl_v1beta1.proto import service_pb2


# 'content' is base-64-encoded image data.
def get_prediction(content, project_id, model_id):
    prediction_client = automl_v1beta1.PredictionServiceClient()

    name = 'projects/{}/locations/us-central1/models/{}'.format(
                                        project_id, model_id)
    payload = {'image': {'image_bytes': content }}
    params = {}
    request = prediction_client.predict(name, payload, params)
    return request  # waits till request is returned

if __name__ == '__main__':
    file_path = sys.argv[1]
    project_id = sys.argv[2]
    model_id = sys.argv[3]

    with open(file_path, 'rb') as ff:
        content = ff.read()

    print get_prediction(content, project_id, model_id)
```

De functie verwacht drie argumenten. Een afbeelding (geëncodeerd in base-64), het project_id en het model_id. De ID's zijn te vinden in de eigenschappen van het project en model op Google Cloud.

```
python predict.py YOUR_LOCAL_IMAGE_FILE 156243918112
                                    ICN8843552342109323264
```

Als een model gedeployed is, wordt er standaard een REST service aangemaakt. De gebruiker kiest dan zelf om een wrapper te schrijven (om een familiaire interface te maken) of om de requests met curl door te sturen, zoals op afbeelding 5.3.



Figuur 5.3: Requests sturen naar de REST API van Google AutoML.

5.5 Requirements

5.5.1 Functionele requirements

Implementatie van het procesmodel

Het platform gaat elke stap verwerken voor de gebruiker, het gaat hier over het overzicht van 2.3. Aan de hand van een drag en drop systeem wordt alles automatisch uitgevoerd. Enkel de data upload naar de cloud moet zelf gebeuren.

Beschikbare metrics

Na het trainen van het model krijgt de gebruiker een beknopt maar duidelijk overzicht. Naast het aantal afbeeldingen waarmee het model getraind en getest is, wordt ook de *precision* en *recall* getoond. Om deze begrippen te snappen is het belangrijk dat deze termen gekend zijn. *True positives* (TP) is het aantal juiste voorspellingen, de *false negatives* (FN) is het aantal voorspellingen die verkeerd geclassificeerd worden in het standpunt van één klasse. Beide worden in procent uitgedrukt.

De *precision* is dan de verhouding correcte voorspellingen tegenover het totaal aantal voorspellingen van één enkele klasse.

$$precision = \frac{TP}{TP + FP} \tag{5.1}$$

Recall is de verhouding tussen het aantal correct voorspelde afbeeldingen en het aantal afbeeldingen van die klasse die aanwezig is in de dataset.

$$recall = \frac{TP}{TP+FN} \tag{5.2}$$

Voorgaande begrippen zijn onderling verwerkt in de *confusion matrix*, die ook gegenereerd wordt voor de gebruikte dataset.

Batch verwerking

Het aantal gekozen *nodes* van een *deployment* configuratie. Elke *node* kan op zich 3.2 requests per seconde uitvoeren. Deze schalen lineair, tien *nodes* kunnen dan 32 afbeeldingen per seconde verwerken.

Voorspellingen kunnen ook in de Google Cloud Console gemaakt worden. De interface ondersteunt een *batch* van maximaal tien afbeeldingen.

Het aantal *nodes* zijn vrij te configureren. De gebruiker beslist zelf de capaciteit die het model ondersteunt. Het is perfect mogelijk om in één job 100.000 afbeeldingen door te sturen.

5.5.2 Niet-functionele requirements

Deployment

Met enkele klikken kan het volledig werken. De nodige infrastructuur wordt ook vrijgemaakt in Google Cloud. De bestaande oplossing kan makkelijk uitgebreid worden met bijvoorbeeld Redis. Een model kan ook geëxporteerd als TensorFlow model om een eigen *deployment* te voorzien of om het lokaal te gebruiken.

Performantie

Op de gekozen dataset wordt een vergelijkbare score behaald als de top scores van de competitie waarvoor het gebruikt werd (98%).

Kostprijs

Om de totale kostprijs te berekenen moet er toch rekening gehouden worden met een aantal zaken, merk op dat er mogelijks nog extra kosten bijkomen door nieuwe toepassingen te introduceren. In het algemeen kan de kostprijs opgedeeld worden in volgende groepen.

Standard storage	Nearline storage	Coldline storage
$0.026	$0.010	$0.007

Tabel 5.2: Maandelijkse kostprijs per GB

Het opslaan van data op Google Cloud is in drie verschillende categorieën. Als gebruiker kan je zelf kiezen in welke regio en in welke *storage class* je de data wilt opslaan. Eerst de standaard data, die veel gebruikt wordt en veel aangepast wordt door de gebruiker. Met Nearline bedoelt men data die af en toe gelezen wordt maar weinig veranderlijk is. Coldline wordt dan gebruikt voor data die enkele malen per kwartaal aangepast wordt, de stap naar archive is niet zo groot als het om back-ups gaat. De prijs van elke soort staat in tabel 5.2.

Het trainen van het model wordt berekend aan de hand van het aantal *node* uren. Elke *node* op zich is een NVIDIA Tesla V100 GPU, die in een schakeling staan om optimaal te werken. Acht *nodes* één uur laten werken staat gelijk aan acht trainingsuren. Per uur bedraagt de kost $3.15. Een model uitbreiden door extra te laten trainen hanteert hetzelfde tarief.

De *deployment* kosten hangen dan ook weer af van het aantal *nodes* die actief staan. Elk uur kost *$1.25* om één *node* te gebruiken, om diezelfde één maand te gebruiken kost dat $900.

Een volledig overzicht van alle kosten binnen de *vision* API is te vinden op de site van Google[2].

Verwerking

Er zijn verschillende manieren om het model te verwerken in een bestaande applicatie, zoals besproken in sectie 2.6. Het gedeployde model is ook beschikbaar en werkt als een REST API. Een gebruiker kan dus in principe een *wrapper* bouwen die alle *requests* onderschept en doorstuurt naar Google Cloud. De client library zelf kan intern in het programma gebruikt worden (indien Python) anders kan een applicatie ook *requests* sturen aan de hand van een Python script.

[2]https://www.cloud.google.com/vision/automl/pricing

6. Conclusie

In eerste instantie werd er (indirect) onderzocht als zo'n ingewikkeld proces wel geautomatiseerd kan worden. De resultaten tonen aan dat AutoML wel degelijk een plaats verdient in de wereld van *machine learning*. De hoofdvraag blijft natuurlijk de bruikbaarheid voor bedrijven en daar zijn toch enkele opmerkingen over. Zo zal er zeker een afweging plaatsvinden waarbij kwaliteit tegenover kost wordt gezet. AutoML is niet goedkoop, zeker op een cloud platform dat snel drie à vierduizend euro per maand kan kosten om operationeel te blijven. Voor AutoKeras zijn de kosten op het eerste zicht beperkt tot verbruikte elektriciteit en *deployment*. Men moet daarbij rekening houden dat elke stap zelf geprogrammeerd moet worden en er toch enige kennis voor nodig is. Het uiteindelijk resultaat wordt dan deels bepaald door de *data preprocessing* die ook handmatig moet gebeuren, deze stap is een belangrijke *trigger* om hoge scores te behalen zoals bij Google Cloud AutoML.

Beide systemen komen de verwachtingen na maar moeten op de juiste plaats ingezet worden. Zo is Google Cloud AutoML een volwaardig *drop in replacement* in bestaande applicaties. Het proces kan niet eenvoudiger zijn en de verschillende manieren om het te integreren zorgen ervoor dat het in meeste situaties past. AutoKeras, in zijn huidige staat, is niet verfijnd genoeg om productie waardig te zijn. De extra moeite om de eerste stappen van het procesmodel te verbeteren kan evengoed verwisseld worden met een ML-ingenieur die het volledige proces uitvoert. Die niche kennis blijft noodzakelijk om te slagen. Anderzijds blijkt het wel een goede *tool* te zijn in de gereedschapskist van ML-ingenieurs enzovoort. Stel een situatie voor waarbij zo snel mogelijk een MVP[1] voorgesteld moet worden. Zonder al te veel moeite kan er met AutoKeras een basis gelegd worden, ook kan het een andere inkijk over het probleem geven die misschien nog niet

[1] Minimum viable product

overwogen was.

Let wel op, Google Cloud AutoML heeft ook zijn nadelen. Naast het stevig kostenplaatje is er ook nog de *vendor lock in* op de cloud. Zo werd er voor een kleine toepassing al gebruik gemaakt van *storage servers*, *vision API* en een *deployment platform*. Als klant is het niet de bedoeling dat een applicatie volledig afhankelijk is van één platform. Het neemt deels de vrijheid af, zo is het niet mogelijk om de structuur van het model te zien.

Het aantal geautomatiseerde stappen is voor beide verschillend en bepaalt uiteindelijk hoe verfijnd het eindproduct is. AutoKeras is sterk gericht op de *core* van het probleem en zo blijven stappen vooraf en achteraf onbeantwoord terwijl die bij de werkwijze van Google Cloud AutoML een aanzienlijke rol hebben. Zo kan een model getraind en *deployed* zijn in een vijftal muisklikken, geen vooraf verwerkte afbeeldingen of andere zaken nodig. Dit terwijl AutoKeras pas gebruikt kan worden nadat de afbeeldingen omgezet zijn naar ruwe data, correct geschaald zijn en grijsfilters of andere optimalisaties toegepast worden. Achteraf is het de verantwoordelijkheid van de gebruiker om het model online te krijgen, wachtrij te optimaliseren en een interface te hebben die kan communiceren met het model.

De toekomst van geautomatiseerde *machine learning* ziet er alvast goed uit. De verbeteringen tussen versies van AutoKeras vallen op en ook steeds meer cloud platformen bieden een gelijkaardige service aan. Er is een echte *push* aan de gang, van de *community* en de bedrijven, om de toepassingen toegankelijker te maken. Verder onderzoek over dit onderwerp zou zich kunnen richten op individuele stappen van het procesmodel, bijvoorbeeld de *data preprocessing*. De automatisatie ervan is niet vanzelfsprekend omdat dit voor elke dataset anders is.

A. Onderzoeksvoorstel

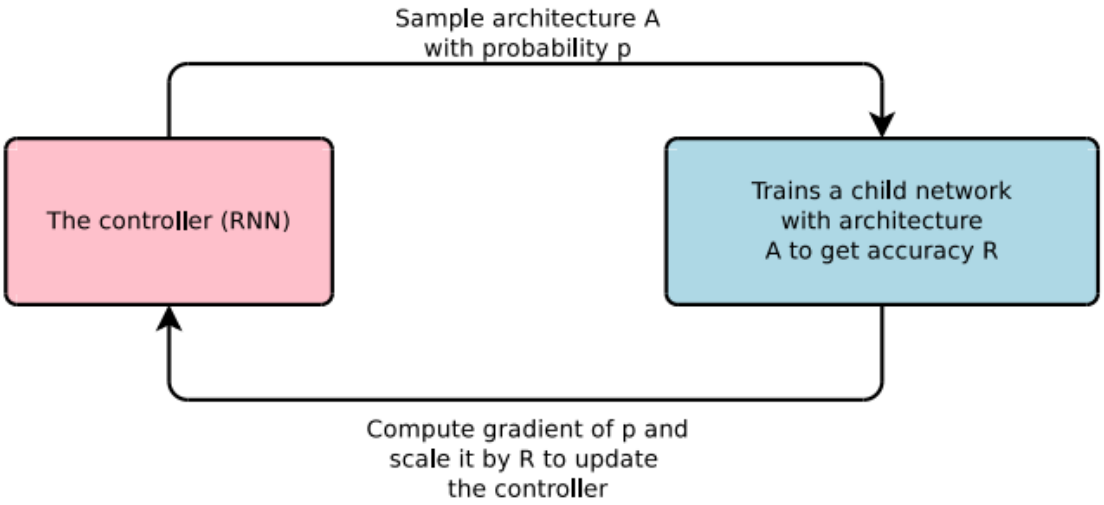
Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

Machine learning en eenvoudig in dezelfde zin gebruiken, is geen vanzelfsprekende opdracht maar wel iets dat Google probeert te realiseren. Met eigenschappen op hun site (Google, 2019) zoals: Uitstekende prestaties; Snel aan de slag; ontstaan er met Cloud AutoML toch enkele mogelijkheden om als programmeur (zonder professionele AI kennis) machine learning diensten te voorzien in een applicatie zonder dat er een data scientist bij het project betrokken wordt. Dit onderzoek, gefocust op het classificeren en herkennen van afbeeldingen, probeert aan te tonen dat deze service bruikbaar is voor bedrijven en hoe het scoort tegenover alternatieven.

A.2 Literatuurstudie

Geautomatiseerde machine learning is het automatiseren van het trainingsproces bij een artificieel neuraal netwerk. De lage toegangsdrempel zorgt ervoor dat mensen met beperkte machine learning kennis sneller en simpeler een model kunnen trainen en gebruiken.



Figuur A.1: Werking van Neural Architecture Search (Zoph & Le, 2016)

A.2.1 Neural Architecture Search

Dergelijke AutoML systemen gebruiken een techniek die het ontwerp van een artificieel neuraal netwerk kan automatiseren, beter bekend als Neural Architecture Search (Elsken e.a., 2019). Uit Zoph en Le (2016) wordt vastgesteld dat deze techniek een gelijkaardige of zelfs betere performantie heeft dan modellen die door een ML-ingenieur ontworpen zijn.

Neural Architecture Search gebruikt Reinforcement Learning om een model te trainen. Deze manier van werken is fundamenteel anders dan gesuperviseerd / ongesuperviseerd leren omdat het model niet beter wordt door het gebruik van datasets. Als alternatief kan het neuraal netwerk beloningssignalen herkennen waardoor het kan leren welke acties leiden tot een positief resultaat (Lievens, 2019).

Op figuur A.1 wordt gevisualiseerd hoe dit werkt. Op basis van controller structuur A (waarbij A een neuraal netwerk is) wordt een string met variabele lengte gegenereerd. Deze waarden worden gebruikt als parameters om een kind-netwerk aan te maken, die getraind wordt met echte data en waarbij de accuraatheid gemeten wordt aan de hand van een validatie dataset. Het resultaat wordt gebruikt als beloningssignaal voor de controller, bij de volgende iteratie kunnen er hogere kansen gegeven worden aan parameters die leiden tot accurate voorspellingen (Zoph & Le, 2016). De controller zijn zoekfunctie zal dus verbeteren met de tijd.

A.2.2 Hyperparameter tuning

In de vorige sectie is het gebruik van parameters aan bod gekomen. Ze bepalen het gedrag van een neuraal netwerk en zijn bepalend voor het eindresultaat. Volgens Brust (2019) zijn er verschillende manieren om dit te behandelen. Brute force zal elke configuratie overlopen en beslissen hoe het model vordert terwijl feature selection gewichten aan verschillende parameters geeft. Op die manier hebben vorige simulaties een impact bij de selectie van

een nieuwe set parameters (Claesen & Moor, 2015).

A.2.3 AutoML platformen

Google Cloud AutoML zorgt voor een familiaire interface die een gebruiker snel op weg helpt. Naast Google hebben bedrijven zoals Microsoft en Amazon een platform gebouwd op hun respectievelijke cloud infrastructuur. De AutoML service kan voordelig zijn als het bedrijf al gebruik maakt van andere producten / diensten van de leverancier, extra kosten kunnen snel de lucht in gaan zonder toegang tot andere functies (bv. van Google Cloud) als dit niet het geval is. Een open source alternatief lijkt een goede oplossing, de interfaces zijn minder gebruiksvriendelijk dan een betalend product en er komt meer programmeerwerk aan te pas. Het resultaat is vaak commercieel bruikbaar zolang de restricties van de licentie gerespecteerd worden (Balter, 2015). AutoKeras is een voorbeeld onder de MIT licentie, die geen commerciële restricties oplegt.

A.3 Methodologie

Eerst en vooral wordt er een image dataset gemaakt die gebruikt wordt om de modellen te trainen en te valideren. Om een model te trainen is er een grote hoeveelheid data nodig. FFmpeg is een tool waarmee je de frames van een video (in dit geval een 360 graden video van het object) kan opsplitsen in afbeeldingen. Het labelen van de images wordt verwerkt met pandas, een data-analyse library voor Python.

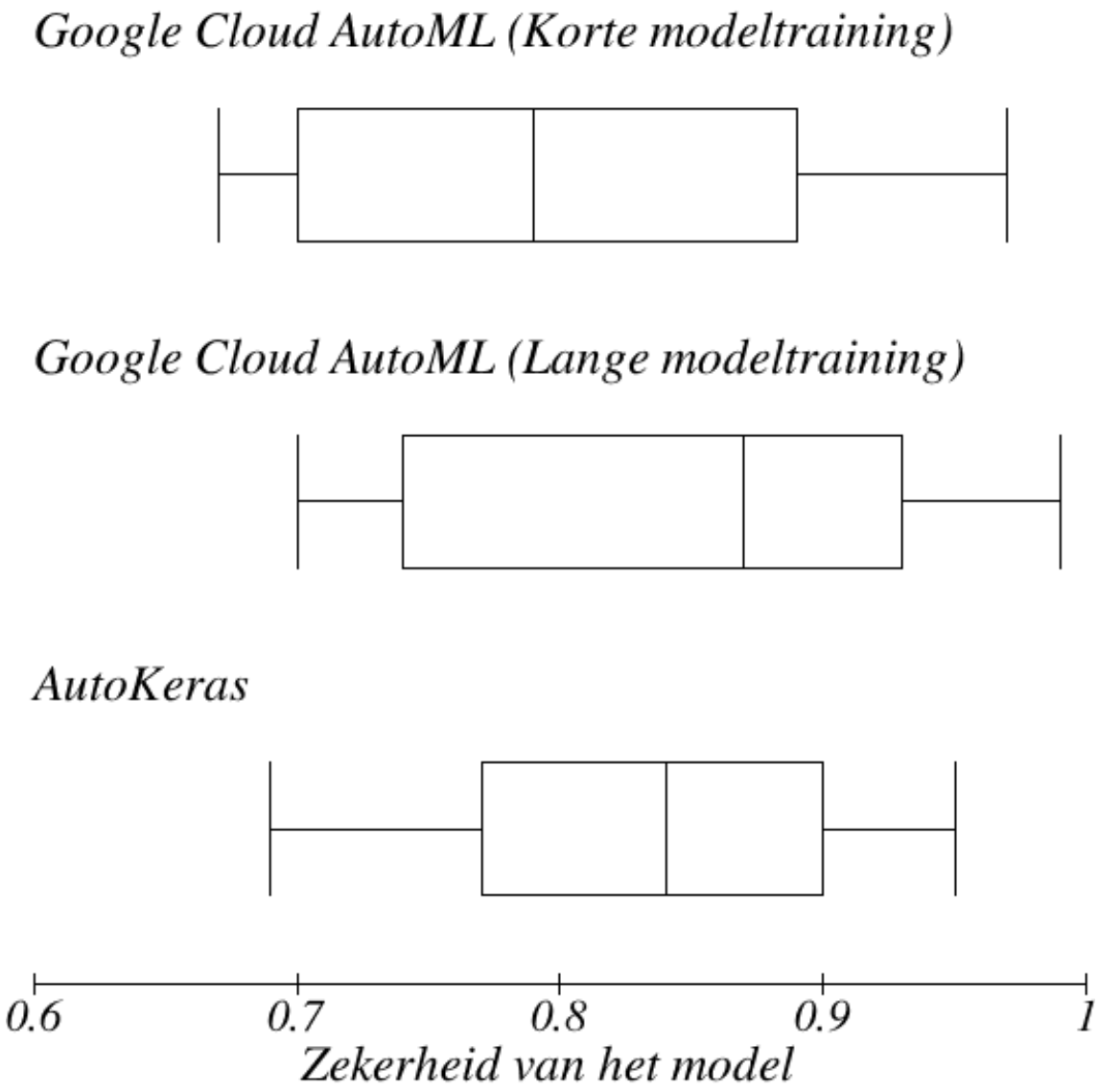
Voor het experiment worden 2 modellen getraind met Google Cloud AutoML en AutoKeras. Zo kan het resultaat vergeleken worden met een open source alternatief.

Er worden een aantal afbeeldingen geselecteerd van verschillende moeilijkheidsgraden om de modellen te toetsen.

A.4 Verwachte resultaten

Er worden goede resultaten verwacht van Google Cloud AutoML. Je betaalt voor een service en dan wil de gebruiker positieve resultaten op tafel zien. De trainingsduur van het model zal ook een zichtbare en positieve impact hebben op de gemiddelde score van een afbeelding. Uit het verleden hebben we geleerd dat *community driven development* vaak leidt tot een performant resultaat (bv. de niet automatische versie, Keras) dat door veel developers onderhouden en gebruikt wordt. Voor een bedrijf is dit zeer interessant vanwege het kostenplaatje dat volledig wegvalt voor het gebruik van de technologie.

Figuur A.2 bevat een voorspelling van de prestaties voor de verschillende modellen.



Figuur A.2: Verwachte correctheid van de modellen

A.5 Verwachte conclusies

AutoML zal zeker zijn plaats houden binnen het domein van machine learning. De Google Cloud AutoML interface zorgt voor een gebruiksvriendelijke omgeving waar zeer weinig programmatie aan te pas komt. Met AutoKeras moet de gebruiker meer kennis hebben van Python libraries (pandas, numpy, keras...) om hetzelfde resultaat te verkrijgen.

Voor bedrijven lijkt het interessanter om voor open source te kiezen als het overeen komt met hun identiteit. Zo kunnen developers een kleine machine learning implementatie voorzien terwijl een data scientist zich kan bezig houden met grotere projecten.

B. Code AutoKeras

De code in deze bijlage werd gebruikt om beide AutoKeras modellen te trainen. Verder bevat het ook alle nodige informatie om feedback op te vragen van een getraind model en een manier om data te structureren en eenmalig in te laden. AutoKeras is nog volop in ontwikkeling waardoor deze code gevoelig is voor aanpassingen. Het gebruikt de aangeraden standaarden maar kan perfect op een andere manier geïmplementeerd worden. Die keuze hangt meestal af van de andere *tools* die een ontwikkelaar gebruikt (Tensorflow dataset slices, *multi task*...) en welke taak hij moet uitvoeren.

De versie van AutoKeras die tijdens dit onderzoek gebruikt werd bevatte een *bug* die ongewenst gedrag *triggerde* in *greedy search* algoritmes wanneer het aantal pogingen groter was dan vijf. Er is een alternatieve versie voorzien dat *random search* gebruikt. Beide methodes werden besproken in Hoofdstuk 4.

autokeras

April 18, 2020

AutoKeras implementation

This notebook uses Autokeras, an automated machine learning library built upon Tensorflow and Keras. The goal of AutoKeras is to make machine learning accessible for everyone.

This example takes a real-life dataset dogs vs cat and tries to train a well working image classifier without a user who decides which layers to use.

Please make sure that your driver configuration is correct (using Nvidia CUDA and cuDNN) and that the library makes its calculations on a GPU. Note that this code is written for AutoKeras v1.0.2 and may be updated for future versions.

Thank you for reading and enjoy exploring AutoKeras!

Robbe Decorte

Global imports and declarations

[ ]:
```
# Use this when running in the cloud or if the packages aren't already installed

!pip install autokeras
!pip install keras
!pip install tensorflow==2.1.0
```

[1]:
```
import pandas as pd
import numpy as np
import autokeras as ak

MAX_TRIES=5
SIZE=64
OUTPUT_NAME="autokeras-model-5"
```

Verify GPU status

[70]:
```
# Use the Keras GPU package
# conda install -c anaconda keras-gpu
from tensorflow.python.client import device_lib
```

[71]:
```
# Show detailed information about the usable devices for Tensoflow
print(device_lib.list_local_devices())
```

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 7502211016056982511
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 3142752665
locality {
  bus_id: 1
  links {
  }
}
incarnation: 12990106037986299612
physical_device_desc: "device: 0, name: GeForce GTX 1050, pci bus id:
0000:01:00.0, compute capability: 6.1"
]
```

[72]:
```
# Only request device names
def get_available_devices():
    local_device_protos = device_lib.list_local_devices()
    return [x.name for x in local_device_protos]
print(get_available_devices())
```

```
['/device:CPU:0', '/device:GPU:0']
```

Shuffle data and convert to numpy array

[18]:
```
import os
from tqdm import tqdm
from PIL import Image
from random import shuffle

INPUT_PATH="data/train/"
```

[20]:
```
def get_label(file):
    class_label = file.split('.')[0]
    if class_label == 'dog': label_vector = 0
    elif class_label == 'cat': label_vector = 1
    return label_vector
```

[21]:
```
def get_data():
    data = []
```

```
    files = os.listdir(INPUT_PATH)
    for image in tqdm(files):

        label_vector = get_label(image)

        img = Image.open(INPUT_PATH + image).convert('L')
        img = img.resize((SIZE,SIZE))

        data.append([np.asarray(img),np.array(label_vector)])

    shuffle(data)
    return data
```

[22]:
```
data = get_data()
```

```
100%|
| 25000/25000 [01:54<00:00, 217.91it/s]
```

[23]:
```
# Save the data as a numpy array
np.save('data/images_shuffled.npy', data)
```

Data preprocessing and split to test and train

[5]:
```
data = np.load('data/images_shuffled.npy', allow_pickle=True)

# Split the labled data in train (20000 images) and test (5000 images)
train = data[:20000]
test = data[20000:]

print("Training dataset contains %d items" % len(train))
print("Testing dataset contains %d items" % len(test))
```

```
Training dataset contains 20000 items
Testing dataset contains 5000 items
```

[6]:
```
x_train = np.array([data[0] for data in train], 'float32')
x_test = np.array([data[0] for data in test], 'float32')
y_train = [data[1] for data in train]
y_test = [data[1] for data in test]
```

[7]:
```
x_train = np.array(x_train).reshape(-1,SIZE,SIZE,1)
x_test = np.array(x_test).reshape(-1,SIZE,SIZE,1)

x_train /= 255
x_test /= 255

y_train = np.array(y_train)
```

```
y_test = np.array(y_test)

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(20000, 64, 64, 1)
(20000,)
(5000, 64, 64, 1)
(5000,)
```

Training and model evaluation

[ ]:
```
# Init image classifier.
clf = ak.ImageClassifier(max_trials=MAX_TRIES, name=OUTPUT_NAME)
# Feed the image classifier with training data.
clf.fit(x_train,y_train, verbose=2)
```

[ ]:
```
# This is a temporary workaround for a bug in the eager search algorithms
# https://github.com/keras-team/autokeras/issues/1045

clf = ak.AutoModel(
    inputs=[ak.ImageInput()],
    outputs=[ak.ClassificationHead()],
    tuner="random",
    max_trials=MAX_TRIES,
    name=OUTPUT_NAME,
    overwrite=False
)

clf.fit(x_train, y_train, verbose=2)
```

[53]:
```
score = clf.evaluate(x_test, y_test)
```

```
157/157 [==============================] - ETA: 13s - loss: 0.6545 - accuracy:
0.718 - ETA: 5s - loss: 0.6889 - accuracy: 0.796 - ETA: 4s - loss: 0.5824 -
accuracy: 0.81 - ETA: 3s - loss: 0.5444 - accuracy: 0.82 - ETA: 3s - loss:
0.6013 - accuracy: 0.82 - ETA: 3s - loss: 0.6256 - accuracy: 0.82 - ETA: 3s -
loss: 0.6035 - accuracy: 0.82 - ETA: 2s - loss: 0.6514 - accuracy: 0.82 - ETA:
2s - loss: 0.6221 - accuracy: 0.83 - ETA: 2s - loss: 0.6244 - accuracy: 0.83 -
ETA: 2s - loss: 0.6180 - accuracy: 0.82 - ETA: 2s - loss: 0.6278 - accuracy:
0.83 - ETA: 2s - loss: 0.6278 - accuracy: 0.83 - ETA: 2s - loss: 0.6330 -
accuracy: 0.83 - ETA: 2s - loss: 0.6378 - accuracy: 0.83 - ETA: 2s - loss:
0.6155 - accuracy: 0.83 - ETA: 1s - loss: 0.6196 - accuracy: 0.83 - ETA: 1s -
loss: 0.6450 - accuracy: 0.83 - ETA: 1s - loss: 0.6463 - accuracy: 0.83 - ETA:
1s - loss: 0.6567 - accuracy: 0.82 - ETA: 1s - loss: 0.6539 - accuracy: 0.82 -
```

```
ETA: 1s - loss: 0.6547 - accuracy: 0.82 - ETA: 1s - loss: 0.6527 - accuracy:
0.82 - ETA: 1s - loss: 0.6561 - accuracy: 0.82 - ETA: 1s - loss: 0.6449 -
accuracy: 0.82 - ETA: 1s - loss: 0.6329 - accuracy: 0.83 - ETA: 1s - loss:
0.6329 - accuracy: 0.83 - ETA: 1s - loss: 0.6454 - accuracy: 0.82 - ETA: 1s -
loss: 0.6401 - accuracy: 0.83 - ETA: 0s - loss: 0.6320 - accuracy: 0.83 - ETA:
0s - loss: 0.6352 - accuracy: 0.83 - ETA: 0s - loss: 0.6421 - accuracy: 0.82 -
ETA: 0s - loss: 0.6450 - accuracy: 0.82 - ETA: 0s - loss: 0.6423 - accuracy:
0.82 - ETA: 0s - loss: 0.6376 - accuracy: 0.82 - ETA: 0s - loss: 0.6380 -
accuracy: 0.82 - ETA: 0s - loss: 0.6375 - accuracy: 0.82 - ETA: 0s - loss:
0.6410 - accuracy: 0.82 - ETA: 0s - loss: 0.6458 - accuracy: 0.82 - ETA: 0s -
loss: 0.6505 - accuracy: 0.82 - ETA: 0s - loss: 0.6439 - accuracy: 0.82 - ETA:
0s - loss: 0.6378 - accuracy: 0.82 - ETA: 0s - loss: 0.6356 - accuracy: 0.82 -
ETA: 0s - loss: 0.6418 - accuracy: 0.82 - ETA: 0s - loss: 0.6398 - accuracy:
0.82 - 3s 17ms/step - loss: 0.6409 - accuracy: 0.8274
```

[64]:
```
print('Accuracy: {accuracy}\nLoss: {loss}'.format(accuracy=score[1],␣
↪loss=score[0]))
```

```
Accuracy: 0.8274000287055969
Loss: 0.6383044676065445
```

[65]:
```
model = clf.export_model()
model.save(OUTPUT_NAME + '/model.h5')
print('Model succesfully exported')
print(type(model))
```

```
Model succesfully exported
<class 'tensorflow.python.keras.engine.training.Model'>
```

Model visualization

[2]:
```
import matplotlib.pyplot as plt
import seaborn as sns;sns.set()
from tensorflow.python.keras.models import load_model
from tensorflow.keras.layers.experimental.preprocessing import Normalization
from sklearn.metrics import confusion_matrix
from tensorflow.keras.utils import plot_model
from sklearn.metrics import accuracy_score
```

[3]:
```
# Load model
cust = ak.CUSTOM_OBJECTS
cust['Normalization'] = Normalization

model = load_model(OUTPUT_NAME+'/model.h5', custom_objects=cust)

print("Model succesfully loaded")
```

```
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your
```

```
model is starting with a freshly initialized optimizer.
Model succesfully loaded
```

[8]:
```
score = model.evaluate(x_test, y_test)
```

```
5000/5000 [==============================] - ETA: 10:23 - loss: 0.6545 -
accuracy: 0.718 - ETA: 1:41 - loss: 0.6116 - accuracy: 0.822 - ETA: 54s - loss:
0.6069 - accuracy: 0.8153 - ETA: 36s - loss: 0.6256 - accuracy: 0.820 - ETA: 27s
- loss: 0.6313 - accuracy: 0.825 - ETA: 21s - loss: 0.6260 - accuracy: 0.828 -
ETA: 18s - loss: 0.6213 - accuracy: 0.829 - ETA: 16s - loss: 0.6188 - accuracy:
0.830 - ETA: 14s - loss: 0.6388 - accuracy: 0.828 - ETA: 13s - loss: 0.6346 -
accuracy: 0.831 - ETA: 12s - loss: 0.6330 - accuracy: 0.833 - ETA: 11s - loss:
0.6372 - accuracy: 0.834 - ETA: 10s - loss: 0.6289 - accuracy: 0.836 - ETA: 9s -
loss: 0.6172 - accuracy: 0.838 - ETA: 8s - loss: 0.6184 - accuracy: 0.83 - ETA:
8s - loss: 0.6518 - accuracy: 0.83 - ETA: 7s - loss: 0.6552 - accuracy: 0.82 -
ETA: 7s - loss: 0.6558 - accuracy: 0.82 - ETA: 6s - loss: 0.6532 - accuracy:
0.82 - ETA: 6s - loss: 0.6547 - accuracy: 0.82 - ETA: 5s - loss: 0.6525 -
accuracy: 0.82 - ETA: 5s - loss: 0.6561 - accuracy: 0.82 - ETA: 4s - loss:
0.6431 - accuracy: 0.83 - ETA: 3s - loss: 0.6344 - accuracy: 0.83 - ETA: 3s -
loss: 0.6454 - accuracy: 0.82 - ETA: 3s - loss: 0.6401 - accuracy: 0.83 - ETA:
2s - loss: 0.6320 - accuracy: 0.83 - ETA: 2s - loss: 0.6365 - accuracy: 0.83 -
ETA: 2s - loss: 0.6492 - accuracy: 0.82 - ETA: 2s - loss: 0.6423 - accuracy:
0.82 - ETA: 1s - loss: 0.6383 - accuracy: 0.82 - ETA: 1s - loss: 0.6353 -
accuracy: 0.82 - ETA: 1s - loss: 0.6375 - accuracy: 0.82 - ETA: 1s - loss:
0.6410 - accuracy: 0.82 - ETA: 1s - loss: 0.6490 - accuracy: 0.82 - ETA: 0s -
loss: 0.6459 - accuracy: 0.82 - ETA: 0s - loss: 0.6395 - accuracy: 0.82 - ETA:
0s - loss: 0.6341 - accuracy: 0.82 - ETA: 0s - loss: 0.6418 - accuracy: 0.82 -
ETA: 0s - loss: 0.6389 - accuracy: 0.82 - ETA: 0s - loss: 0.6394 - accuracy:
0.82 - 6s 1ms/sample - loss: 0.6383 - accuracy: 0.8274
```

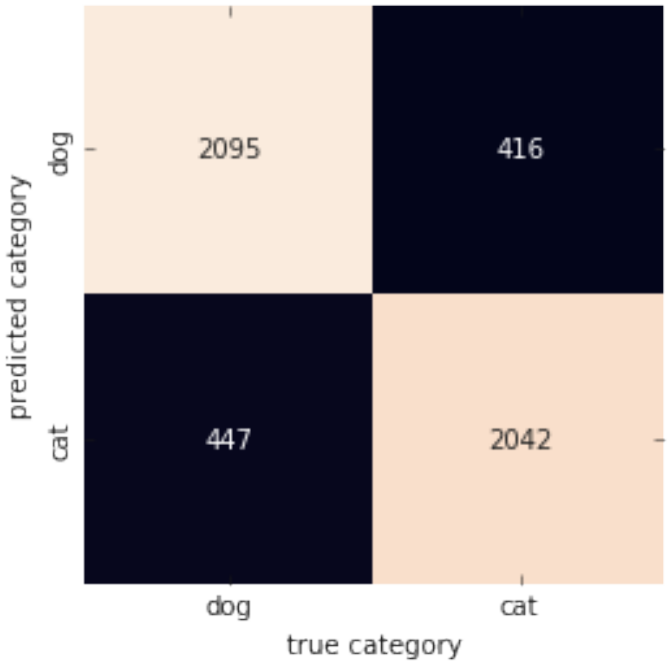
[9]:
```
print('Accuracy: {accuracy}\nLoss: {loss}'.format(accuracy=score[1],␣
↪loss=score[0]))
```

```
Accuracy: 0.8274000287055969
Loss: 0.6383046453475952
```

[12]:
```
predictions = model.predict(x_test)
```

[10]:
```
plt.style.use('classic')
%matplotlib inline

mat = confusion_matrix(y_test, predictions.round())
labels = ['dog', 'cat']

sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,␣
↪xticklabels=labels, yticklabels=labels)
plt.xlabel('true category')
plt.ylabel('predicted category')
```

[10]: Text(90.80000000000001, 0.5, 'predicted category')



[13]:
```
instances = 0
correct_ones = 0

images = x_test.reshape(5000,64, 64)
incorrect_predictions = []

for i in range(0, len(predictions)):
    if i < 5:
        print("Prediction: ", predictions[i], ", Actual: ", y_test[i])
    if predictions[i].round() == y_test[i]:
        correct_ones += 1
    else:
        incorrect_predictions.append((i, images[i], predictions[i].round(4),␣
 ↪y_test[i]))
    instances += 1

print('\nMade {amountPred} predictions. {amountCorrect} of those are correct.␣
 ↪Approximately {percentage}%.'
      .format(amountPred=instances, amountCorrect=correct_ones,␣
 ↪percentage=round((correct_ones/instances) * 100)))
```

```
Prediction:  [0.5145127] , Actual:  0
Prediction:  [0.56603503] , Actual:  0
```

```
Prediction:  [0.99702567] , Actual:  0
Prediction:  [4.35466e-07] , Actual:  0
Prediction:  [0.98181605] , Actual:  1

Made 5000 predictions. 4137 of those are correct. Approximately 83%.
```

[14]:
```
# Show images that are classified to the wrong class
# Dog = 0, Cat = 1
%matplotlib inline

figure, axes = plt.subplots(nrows=6, ncols=4, figsize=(16,16))

for axes, item in zip(axes.ravel(), incorrect_predictions):
    index, image, predicted, expected = item
    axes.imshow(image, cmap=plt.cm.gray)
    axes.set_xticks([])
    axes.set_yticks([])
    axes.set_title(f'index: {index}\np: {predicted}; e: {expected}')
plt.tight_layout()
```

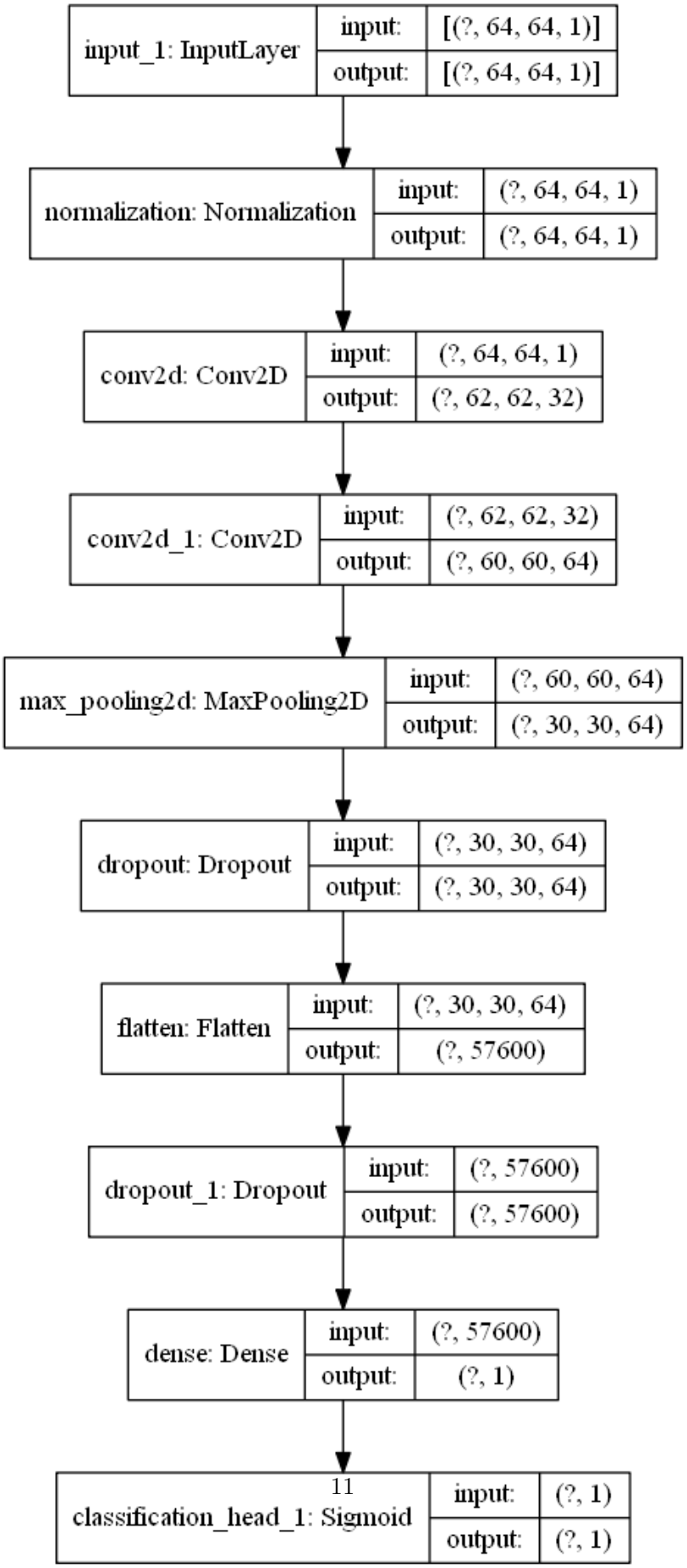
[67]:
```
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 64, 64, 1)]       0
_________________________________________________________________
```

```
normalization (Normalization (None, 64, 64, 1)         3
_________________________________________________________________
conv2d (Conv2D)              (None, 62, 62, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 60, 60, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 30, 30, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 30, 30, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 57600)             0
_________________________________________________________________
dropout_1 (Dropout)          (None, 57600)             0
_________________________________________________________________
dense (Dense)                (None, 1)                 57601
_________________________________________________________________
classification_head_1 (Sigmo (None, 1)                 0
=================================================================
Total params: 76,420
Trainable params: 76,417
Non-trainable params: 3
_________________________________________________________________
```

[68]:
```
plot_model(model, to_file=OUTPUT_NAME + '/model.png', show_shapes=True,␣
↪show_layer_names=True)
```

[68]:


input_1: InputLayer
input: [(?, 64, 64, 1)]
output: [(?, 64, 64, 1)]
normalization: Normalization
input: (?, 64, 64, 1)
output: (?, 64, 64, 1)
conv2d: Conv2D
input: (?, 64, 64, 1)
output: (?, 62, 62, 32)
conv2d_1: Conv2D
input: (?, 62, 62, 32)
output: (?, 60, 60, 64)
max_pooling2d: MaxPooling2D
input: (?, 60, 60, 64)
output: (?, 30, 30, 64)
dropout: Dropout
input: (?, 30, 30, 64)
output: (?, 30, 30, 64)
flatten: Flatten
input: (?, 30, 30, 64)
output: (?, 57600)
dropout_1: Dropout
input: (?, 57600)
output: (?, 57600)
dense: Dense
input: (?, 57600)
output: (?, 1)
classification_head_1: Sigmoid
input: (?, 1)
output: (?, 1)

C. Code Google Cloud AutoML

De onderstaande code is een aangepaste versie van een algemeen script dat voor andere *cloud services* binnen de *vision API* gebruik kan worden. De integratie met online opslag is de makkelijkste manier om modellen te trainen, vaak worden ze in dezelfde formule aangeboden. Het code-skelet van Guo (2018) is herbruikbaar, volgende zaken moeten aangepast worden:

- De namen van de mappen die de data bevat, merk op dat de namen van de mappen ook gebruikt worden om het label van elke afbeelding te bepalen
- Het pad naar de gebruikte *bucket* op Google Cloud Storage
- Het formaat van de afbeeldingen

Deze stappen voldoen op voorwaarde dat volgende mappenstructuur gebruikt wordt:

```
online bucket
├── cat
│   ├── cat.0.jpg
│   ├── cat.1.jpg
│   └── cat.2.jpg
└── dog
    ├── dog.0.jpg
    ├── dog.1.jpg
    └── dog.2.jpg
```

Dit voorbeeld is gebruikt voor binaire classificatie maar kan zonder extra stappen (in de code) gebruikt worden voor meerdere klassen. Gewoon een nieuwe map toevoegen en dezelfde regels toepassen. Het platform zal zelf het aantal gebruikte klassen herkennen.

google-automl-prep

March 29, 2020

[2]:
```
import os
import pandas as pd
```

[3]:
```
# hardcoded
data_folders = ["dog", "cat"]
data_folders
```

[3]:
```
['dog', 'cat']
```

[4]:
```
# array of arrays, containing the list files, grouped by folder
filenames = [os.listdir(f) for f in data_folders]
[print(f[1]) for f in filenames]
[len(f) for f in filenames]
```

```
dog.1.jpg
cat.1.jpg
```

[4]:
```
[12500, 12500]
```

[5]:
```
files_dict = dict(zip(data_folders, filenames))
```

[6]:
```
base_gcs_path = 'gs://rdc-automl-catsvsdogs/'
```

[7]:
```
# What we want:
# gs://rdc-automl-catsvsdogs/dog/dog.0.jpg, 'dog'
# base_gcs_path + dict_key + '/' + filename

data_array = []

for (dict_key, files_list) in files_dict.items():
    for filename in files_list:
        # print(base_gcs_path + dict_key + '/' + filename)
        if '.jpg' not in filename:
            continue # don't include non-photos

        label = dict_key

        data_array.append((base_gcs_path + dict_key + '/' + filename , label))
```

[9]:
```
dataframe = pd.DataFrame(data_array)
```

[10]:
```
dataframe
```

[10]:

	0	1
0	gs://rdc-automl-catsvsdogs/dog/dog.0.jpg	dog
1	gs://rdc-automl-catsvsdogs/dog/dog.1.jpg	dog
2	gs://rdc-automl-catsvsdogs/dog/dog.10.jpg	dog
3	gs://rdc-automl-catsvsdogs/dog/dog.100.jpg	dog
4	gs://rdc-automl-catsvsdogs/dog/dog.1000.jpg	dog
…	…	…
24995	gs://rdc-automl-catsvsdogs/cat/cat.9995.jpg	cat
24996	gs://rdc-automl-catsvsdogs/cat/cat.9996.jpg	cat
24997	gs://rdc-automl-catsvsdogs/cat/cat.9997.jpg	cat
24998	gs://rdc-automl-catsvsdogs/cat/cat.9998.jpg	cat
24999	gs://rdc-automl-catsvsdogs/cat/cat.9999.jpg	cat

[25000 rows x 2 columns]

[11]:
```
dataframe.to_csv('all_data.csv', index=False, header=False)
```

Bibliografie

Arroyo, A. (2017, februari 13). *Business Intelligence and its relationship with the Big Data, Data Analytics and Data Science*. https://www.linkedin.com/pulse/business-intelligence-its-relationship-big-data-geekstyle

Balter, B. (2015, maart 9). *Open source license usage on GitHub.com*. GitHub. Verkregen 8 december 2019, van https://github.blog/2015-03-09-open-source-license-usage-on-github-com/

Bergstra, J., Yamins, D. & Cox, D. D. (2013). Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures, In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, Atlanta, GA, USA, JMLR.org.

Bergstra, J. S., Bardenet, R., Bengio, Y. & Kégl, B. (2011). Algorithms for Hyper-Parameter Optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira & K. Q. Weinberger (Red.), *Advances in Neural Information Processing Systems 24* (pp. 2546–2554). Curran Associates, Inc. http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

Brust, A. (2019, maart 11). *AutoML is democratizing and improving AI*. Verkregen 8 december 2019, van https://www.zdnet.com/article/automl-democratizing-and-improving-ai/

Cai, H., Chen, T., Zhang, W., Yu, Y. & Wang, J. (2017). Reinforcement Learning for Architecture Search by Network Transformation. *CoRR*, *abs/1707.04873*arXiv 1707.04873. http://arxiv.org/abs/1707.04873

Cen, A. (2016). *Example Ozone data*. Verkregen 8 maart 2020, van https://rpubs.com/mp43ily/90520

Chen, T., Goodfellow, I. J. & Shlens, J. (2016). Net2Net: Accelerating Learning via Knowledge Transfer (Y. Bengio & Y. LeCun, Red.). In Y. Bengio & Y. LeCun (Red.), *4th International Conference on Learning Representations, ICLR 2016, San*

Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. http://arxiv.org/abs/1511.05641

Chopra, R., England, A. & Alaudeen, M. (2019). *Data Science with Python: Combine Python with Machine Learning Principles to Discover Hidden Patterns in Raw Data*. Packt Publishing. https://books.google.be/books?id=VWrOwQEACAAJ

Claesen, M. & Moor, B. D. (2015). Hyperparameter Search in Machine Learning. *CoRR*, *abs/1502.02127*arXiv 1502.02127. http://arxiv.org/abs/1502.02127

Decorte, J. & De Vreese, S. (2019). *Machine learning with Python*.

Elsken, T., Metzen, J. H. & Hutter, F. (2019). Neural Architecture Search: A Survey. *Journal of Machine Learning Research*, *20*(55), 1–21. http://jmlr.org/papers/v20/18-598.html

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M. & Hutter, F. (2015). Efficient and Robust Automated Machine Learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama & R. Garnett (Red.), *Advances in Neural Information Processing Systems 28* (pp. 2962–2970). Curran Associates, Inc. http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf

Feurer, M., Klein, A. & Frank, H. (2016). *Contest Winner: Winning the AutoML Challenge with Auto-sklearn*. KDnuggets, University of Freiburg. Verkregen 15 maart 2020, van https://www.kdnuggets.com/2016/08/winning-automl-challenge-auto-sklearn.html

Fusi, N., Sheth, R. & Elibol, H. M. (2017). Probabilistic Matrix Factorization for Automated Machine Learning.

Google. (2019). *Cloud AutoML - Aangepaste modellen voor machine learning*. Verkregen 7 december 2019, van https://cloud.google.com/automl/?hl=nl

Google. (2020a, maart 13). *AutoML Vision Client Libraries*. Google. Verkregen 14 maart 2020, van https://cloud.google.com/vision/automl/docs/client-libraries

Google. (2020b, februari 18). *Client Libraries Explained*. Google. Verkregen 14 maart 2020, van https://cloud.google.com/apis/docs/client-libraries-explained

Google. (2020c, februari 27). *Overview of hyperparameter tuning*. Google. Verkregen 29 februari 2020, van https://cloud.google.com/ai-platform/training/docs/hyperparameter-tuning-overview

Google. (2020d, april 23). *Training Edge Exportable Models*. https://cloud.google.com/vision/automl/docs/train-edge

Guo, Y. (2018, juli 10). *AutoML Data Preparation*. Google, Github. Verkregen 30 januari 2020, van https://gist.github.com/yufengg/984ed8c02d95ce7e95e1c39da906ee04

Gutierrez, D. (2019, januari 31). *Automated Machine Learning: Myth Versus Reality*. Verkregen 24 februari 2020, van https://opendatascience.com/automated-machine-learning-myth-versus-realty/

Hinton, G. & Sejnowski, T. J. (1999). *Unsupervised Learning: Foundations of Neural Computation*. The MIT Press. https://doi.org/10.7551/mitpress/7011.001.0001

Jin, H., Song, Q. & Hu, X. (2019). Auto-Keras: An Efficient Neural Architecture Search System, In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM.

Kaelbling, L. P., Littman, M. L. & Moore, A. W. (1996). Reinforcement Learning: A Survey. *CoRR*, *cs.AI/9605103*. https://arxiv.org/abs/cs/9605103

Khandelwal, R. (2019, augustus 29). *Deep Learning using Transfer Learning*. Verkregen 24 februari 2020, van https://towardsdatascience.com/deep-learning-using-transfer-learning-cfbce1578659

Lemahieu, W., Broucke, S. V. & Baesens, B. (2018). *Principles of Database Management: The Practical Guide to Storing, Managing and Analyzing Big and Small Data*. USA, Cambridge University Press.

Lievens, S. (2019, september 19). *Artificiële Intelligentie, lesnota's*.

Microsoft. (2020, februari 28). *What is automated machine learning?* Verkregen 16 maart 2020, van https://docs.microsoft.com/en-us/azure/machine-learning/concept-automated-ml

Olson, R. S., Bartley, N., Urbanowicz, R. J. & Moore, J. H. (2016). Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science, In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, Denver, Colorado, USA, ACM. https://doi.org/10.1145/2908812.2908918

Opitz, D. & Maclin, R. (1999). Popular Ensemble Methods: An Empirical Study. *J. Artif. Int. Res.*, *11*(1), 169–198.

Pan, S. J. & Yang, Q. (2009). A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, *22*(10), 1345–1359. https://doi.org/10.1109/TKDE.2009.191

Peter Norvig, S. J. R. (1994, december 11). *Artificial Intelligence: A Modern Approach*.

Rosebrock, A. (2018, januari 29). *A scalable Keras + deep learning REST API*. Verkregen 29 februari 2020, van https://www.pyimagesearch.com/2018/01/29/scalable-keras-deep-learning-rest-api/

Schmidhuber, J. (1993). A 'Self-Referential' Weight Matrix (S. Gielen & B. Kappen, Red.). In S. Gielen & B. Kappen (Red.), *ICANN '93*, London, Springer London.

*Schmidhuber, J. (1987). *Evolutionary Principles in Self-Referential Learning. On Learning now to Learn: The Meta-Meta-Meta...-Hook* (Diploma Thesis). Technische Universitat Munchen, Germany. http://www.idsia.ch/~juergen/diploma.html

Zoph, B. & Le, Q. V. (2016). Neural Architecture Search with Reinforcement Learning. *CoRR*, *abs/1611.01578*arXiv 1611.01578. http://arxiv.org/abs/1611.01578

Zoph, B., Vasudevan, V., Shlens, J. & Le, Q. V. (2017). Learning Transferable Architectures for Scalable Image Recognition. *CoRR*, *abs/1707.07012*arXiv 1707.07012. http://arxiv.org/abs/1707.07012