KU LEUVEN

FACULTEIT INDUSTRIELE
INGENIEURSWETENSCHAPPEN

TECHNOLOGIECAMPUS GENT

Comparative Study of Single Board Computers for AI Edge Computing Purposes

Arno PLAETINCK

Promotor: Prof. Dr. Ir. Nobby Stevens

Co-promotoren: Ing. Willem Raes
Ing. Jorik De Bruycker

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: Industriële Ingenieurswetenschappen Elektronica-ICT Embedded Systems

Academiejaar 2019 - 2020

Voorwoord

In deze thesis wordt een benchmark voorgesteld die de performantie van verscheidene edge devices analyseert op vlak van latency. De uitvoering van verschillende getrainde neurale netwerken wordt vergeleken met parameters zoals processorgebruik, klokfrequentie, prijs en energieverbruik. De verworven inzichten kunnen toegepast worden in allerlei low-latency applicaties zoals zelfrijdende voertuigen.
Verder had ik graag de mensen bedankt die geholpen hebben deze thesis tot stand te brengen. Hulp van personen zoals mijn promotor Prof. Dr. Ir. Nobby Stevens en de co-promotoren Ing Willem Raes en Ing Jorik De Bruycker werd zeer geapprecieerd. Ook wil ik familie en vrienden bedanken voor hun morele steun tijdens mijn studies en eindwerk.

Arno Plaetinck
Gent
25 mei 2020

Abstract

Benchmark voor latency bij edge-devices.

Er is een hele grote verscheidenheid aan onderwerpen waarbij de duur van verkrijgen van resultaten binnen zekere tijdslimieten moeten vallen. Hiervoor wordt vaak een afweging gemaakt tussen cloud en edge computing. In de cloud kan met krachtige en snelle hardware worden gewerkt. Echter treedt hierbij een significante vertraging op, veroorzaakt door het verzenden van data over het internet. Uitvoeren van programma's in de edge hebben geen last van deze vertraging. Dit kan voordelig zijn bij latency-gevoelige applicaties zoals het zelfstandig rijden van voertuigen.

Deze scriptie bespreekt een benchmark van drie verschillende edge-toestellen en een reguliere computer. Deze benchmark biedt informatie over de latency en processor-performantie van elk device. Deze zal meer inzicht bieden wanneer men een kosten-baten analyse maakt. De onderzochte devices binnen deze thesis zijn de toestellen zoals de Google Coral Dev die over zo wel een CPU als TPU bezit en de Raspberry Pi die enkel over een CPU beschikt. Ook wordt een Nvidia Jetson Nano en een Personal Computer gebruikt die zowel een CPU als een GPU bezitten.

Om een representatieve benchmark te bekomen is het nodig om verschillende programma's op een identieke wijze te testen. De te beproeven programma's zullen verdeeld worden over de categorieën regressie en classificatie. De data die uit de benchmark afgeleid wordt bestaat uit de latency en het processor-gebruik van het runnen van elk programma. Deze worden dan herwerkt en genormaliseerd om de resultaten eenvoudiger te analyseren. Hierbij worden ook factoren zoals prijs, kloksnelheid en energieverbruik in rekening gebracht.

Uit de resultaten kan er een duidelijk verschil afgeleid worden tussen de verschillende edge-toestellen. Hierbij komt de Nvidia Jetson Nano het best naar voor. Het toestel behaalt de laagste latency voor alle toegepaste programma's. De resultaten geven ook aan dat de Coral Dev efficiënter met energie omspringt. Voor het uitvoeren van hetzelfde programma zal de Coral Dev minder energie verbruiken.

Trefwoorden: benchmark, edge-devices, latency, Jetson Nano, Coral Dev, Raspberry Pi

Abstract English

Benchmark voor latency bij edge-devices.

There is a large diversity of subjects in which the duration of receiving results must remain within certain time limits. Due to this, a consideration is often made between cloud and edge computing. In the cloud powerful and fast hardware can be used. However, a significant delay arises when sending data over the internet. Executing programs in the edge does not suffer from these delays. This can be advantageous with latency sensitive applications like autonomous driving of vehicles.

This thesis discusses a benchmark of three different edge-devices and a regular computer. This benchmark offers information about latency and processor-performance of each device. This will deliver more insight when making a cost-benefit analysis. The examined devices within this thesis are devices like the Google Coral Dev who consists of a CPU and TPU and the Raspberry Pi who only disposes only of a CPU. Also, a Nvidia Jetson Nano and a Personal Computer is used who both contain a CPU and GPU.

To obtain a representative benchmark it is necessary to test different programs in an identical way. The programs that will be tested will be divided over the category's regression and classification. The data derived from the benchmark will consist of the latency and the processor usage from running each program. These will be reworked and normalised to make an analysis easier. This will include accounting for factors like price, clock speed and energy consumption.

From the results a clear difference can be derived from the different edge devices. The Nvidia Jetson Nano proves to have the most potential. This device achieves the lowest latency for all the applied programs. The results show as well that the Coral Dev is more energy efficient. When executing the same program on different devices, the Coral Dev will consume the least amount of energy.

Keywords: benchmark, edge-devices, latency, Jetson Nano, Coral Dev, Raspberry Pi

Inhoudsopgave

Dankwoord v

Abstract vii

Abstract English ix

Inhoud xii

Figurenlijst xiii

Tabellenlijst xv

Listingslijst xvii

Afkortingenlijst xix

1 Inleiding 1

2 Literatuurstudie 3

2.1 Kadering . . . 4

2.2 Artificiële Intelligentie . . . 5

2.3 Machine Learning . . . 6

2.3.1 Leertechnieken . . . 6

2.3.2 Leeralgoritmes en -technieken . . . 7

2.3.3 Keuze voor een Machine Learning-methode . . . 13

2.4 Evolutie Single Board Computers . . . 15

2.4.1 Geschiedenis . . . 15

2.5 Assortiment aan 'off the shelf' toestellen . . . 17

2.5.1 Beaglebone AI . . . 17

2.5.2 Coral Dev Board . . . 18

2.5.3 Nvidia Jetson Nano . . . 18

2.5.4 Nvidia Jetson TX2 . . . 19

2.5.5 Raspberry Pi . . . 19

2.5.6 Personal Computer: Lenovo Legion Y520 . . . 20

2.6 Benchmarking van Machine Learning algoritmes . . . 21
2.6.1 Bestaande benchmarks . . . 21

3 Data verwerving 23
3.1 Structuur benchmark . . . 23
3.2 Uitvoeren metingen . . . 25
3.3 Opslaan van data . . . 25
3.4 Verkennen van software . . . 26
3.5 Verkennen edge-devices . . . 27
3.6 Structuur programma . . . 28
3.6.1 Regressie subprogramma's . . . 28
3.6.2 Classificatie subprogramma's . . . 32
3.6.3 Conversie naar TFLite . . . 37

4 Data verwerking 39
4.1 Data cleaning . . . 39
4.1.1 Verwerpen data . . . 39
4.1.2 Behandelen uitschieters . . . 40
4.2 Vormgeving resultaten . . . 41
4.2.1 Gebruikte grootheden . . . 41
4.2.2 Weergave resultaten . . . 41
4.3 Overzicht code . . . 42
4.3.1 Ophalen data . . . 42
4.3.2 Verwerken data . . . 44
4.3.3 Visualiseren data . . . 45

5 Resultaten 47
5.1 Ruwe benchmark gegevens . . . 48
5.2 Performantie . . . 50
5.3 Energieverbruik . . . 51

6 Conclusie 53
6.1 Toekomstig werk . . . 54

A Beschrijving van deze masterproef in de vorm van een wetenschappelijk artikel 57

B Benchmark code 59
B.1 Runnen benchmark . . . 59
B.2 Plotten resultaten . . . 70

C Poster 77

Lijst van figuren

2.1 Routine bij Reinforcement Learning. 7
2.2 Structuur van een Neuraal Netwerk. 8
2.3 Algemene structuur van een node. 9
2.4 De Sigmoid activatiefunctie. 10
2.5 Algemene structuur van een beslissingsboom. 11
2.6 Tweedimensionale Support Vector Machine. 12
2.7 Voorbeeld van Lineaire Regressie. 12
2.8 Eerste SBC: MMD-1. 15

3.1 De structuur van de benchmark. 24
3.2 Praktische betekenis van het compair-subprogramma. 28
3.3 Enkele voorbeelden uit de FashionMNIST-dataset. 33
3.4 Een voorbeeld uit de NumberMNIST-dataset. 35
3.5 Een voorbeeld van een kat uit de catsVSdogs-dataset. 36
3.6 Een voorbeeld uit de Image Recognition-dataset. 37

4.1 Een voorbeeld van uitschieters in data van de duur van het compairprogramma. . . . 40
4.2 De vorm van gebruikte variabelen. NOG OM TE ZETTEN NAAR VECTOR 43

5.1 Staafdiagram met de latencydata. 48
5.2 Staafdiagram met de genormaliseerde latencydata. 49
5.3 Staafdiagram met de CPU-verbruikgegevens. 49
5.4 Staafdiagram met de performantie van de toestellen. 50
5.5 Staafdiagram met de genormaliseerde performantie van de toestellen. 51
5.6 Staafdiagram van het energieverbruik. 52
5.7 Staafdiagram van het genormaliseerde energieverbruik. 52

Lijst van tabellen

2.1 Specificaties van gebruikte toestellen. 20

3.1 Gegevens voor verscheidene toestellen. 28
3.2 Voorbeelden van de gebruikte data voor regressiemodellen. 29

Listings

3.1 Meten van gewenste data. 25
3.2 Opslaan van de gewenste data. 26
3.3 Creëren en trainen van pyrenn-model. 29
3.4 uitvoeren van pyrenn-model. 30
3.5 Creëren en trainen van pyrenn-model voor narendra4. 31
3.6 Creëren, trainen en runnen van pyrenn-model voor gradient. 32
3.7 Creëren en trainen van sequentieel model voor FashionMNIST. 33
3.8 Runnen van sequentieel model voor FashionMNIST. 34
3.9 Structuur van het Convolutioneel Neuraal Netwerk NumberMNIST. 34
3.10 Structuur van het Convolutioneel Neuraal Netwerk catsVSdogs. 36
3.11 Converteren naar een TFLite-model. 37
3.12 Converteren naar een TFLite-programma. 38
4.1 Controleren op meetfouten. 40
4.2 Bijhouden van databestandsnamen voor elk toestel. 42
4.3 Ophalen data uit bestand voor elk toestel. 43
4.4 Data uit de specificaties voor elk toestel. 44
4.5 Verwerking van data_time naar data_time_MHzCPU. 44
4.6 Normalisatie van data_energy. 45
4.7 Pseudocode van de show_plot()-functie. 46
4.8 Voorbeeld van gebruik van de show_plot()-functie. 46

Lijst van acroniemen

LPWAN	Low-Power Wide-Area Network
IOT	Internet-of-Things
ISI	Inter-Symbol Interference
SBC	Single Board Computers
ANN	Artificiële Neurale Netwerken
ML	Machine Learning
NN	Neurale Netwerken
SOC	System On Chip
AI	Artificiële Intelligentie
EVE	embedded-vision-engine
SVM	Support Vector Machines
TPU	Tensor Processing Unit
ASIC	Application Specific Integrated Circuit
TOPS	tera operations per second
TOPW	tera operations per Watt
CPU	Central Processing Unit
RELU	rectified linear unit
DLIB	Deep Learning Inference Benchmarks
DL	Deep Learning
CNN	Convolutional Neural Networks
TF	Tensorflow
TFL	Tensorflow Lite
RNN	Recurrent Neural Network
CSV	Comma Seperated Value
RTRL	Real Time Recurrent Learning
BPTT	Back Propagation Trough Time
PCA	Principal Component Analysis
KNN	K-Nearest Neighbors

Hoofdstuk 1

Inleiding

De computerwereld maakt de laatste jaren grote stappen op vlak van Machine Learning[1]. Deze vorderingen werden gedreven door onder meer de nieuwste ontwikkelingen op vlak van computerrekenkracht en de vierde industriële revolutie[2]. Door de veelvuldige toepassingsmogelijkheden werd Machine Learning (ML) een populair en veelbesproken onderwerp. Tegenwoordig kan een bepaalde vorm van Artificiële Intelligentie (AI) in elke sector teruggevonden worden[3]. Van hartritmestoornissen herkennen in de medische sector tot commentaarherkenning in de toeristische stiel.

In deze thesis zal men trachten om een benchmark met AI op te stellen waarmee verschillende Single Board Computers (SBC)s met elkaar vergeleken kunnen worden. Dit instrument moet meer inzicht verschaffen in welke mate machineleertechnieken toepasbaar zijn. Er zal vooral gekeken worden naar performantie-parameters. Hoe groot is de latency die optreedt? Welk verbruik en complexiteit van het netwerk gaat er hier mee gepaard? Ook tussen deze hardware-opties wordt er een afweging gemaakt. Welk toestel is voordeliger in welke situatie? Is een goedkoper toestel tot evenwaardige resultaten in staat? Er zal een oplijsting gemaakt worden van de belangrijkste parameters en met behulp van de benchmark-resultaten zal er een besluit over de SBCs genomen worden.

Hoofdstuk 2

Literatuurstudie

In dit hoofdstuk zal er eerst een afgrenzing gegeven worden waarbinnen de thesis gekozen is. Vervolgens zal een korte introductie gegeven worden tot Artificiële Intelligentie en Machine Learning waarin verschillende vormen en mogelijke toepassingsdomeinen behandeld worden. De verschillende leermodellen worden bekeken en er wordt nagegaan hoe de onderdelen een werkend geheel vormen. Vervolgens wordt er een overzicht van de voor- en nadelen van de verschillende technieken gegeven. Er wordt een best bruikbare methode voor de toepassing in deze thesis gekozen. Bij deze keuze zullen de voor- en nadelen gewikt en gewogen worden. Verder wordt er ook de geschiedenis van Single Board Computers aangehaald. Hoe zijn deze toestellen ontstaan, hoe zijn ze geëvolueerd en in welke staat zijn de hedendaagse SBCs. Verder worden ook verschillende voorbeelden van SBCs besproken die in staat zijn om Machine Learning-technieken toe te passen. Deze zullen onderworpen worden aan een benchmark die in paragraaf 2.6 besproken zal worden.

2.1 Kadering

Een branche in de ML die steeds meer in de schijnwerpers staat, is de logistieke sector[4]. Door de steeds verder doorgedreven automatisatie van bedrijven, wordt er ook in bijvoorbeeld magazijnen geopteerd voor het optimaliseren van onder meer het leveren van de verschillende onderdelen en de veiligheid in het magazijn. Gebruik maken van zelfrijdende vorkheftrucks is een mogelijke optie in het verbeteren van de efficiëntie. Niet alleen in het magazijn maar ook op de weg is er een groeiende belangstelling naar zelfrijdende voertuigen, die ontwikkeld worden door grote bedrijven zoals Tesla en Uber. In beide cases zal er een zekere vorm van positiebepaling nodig zijn. Het is van groot belang dat deze bepaling zo accuraat mogelijk plaats vindt, met niet alleen een juiste locatie, maar ook een resultaat dat op zo kort mogelijke tijdsperiode geproduceerd wordt.

Deze nood aan *low latency* kan het verschil betekenen tussen een voertuig dat beslist dat hij moet vertragen of beslist dat hij veilig kan doorrijden maar toch een botsing veroorzaakt. De berekening van die cruciale locatiebepaling kan zowel in de *cloud*, als in de *edge*[5] gebeuren en gebeurt onder meer met behulp van ML. Hierbij wordt met cloud verwezen naar het verwerken van data op een locatie ver weg van het voertuig zoals serverzalen. Met edge wordt dan weer een locatie dichtbij het voertuig bedoeld. Dit kan zowel op, als vlakbij het voertuig zijn. *Cloud computing* brengt een zekere toegevoegde latency teweeg. Dit is de nodige tijd om via het internet de server te bereiken. Hierdoor zal men eerder kiezen voor *edge computing* vallen. Indien de berekeningen in de edge plaats vinden zal er bijvoorbeeld een kleinere vertraging zijn tussen het moment van vertrekken en het waarnemen door het algoritme dat er vertrokken werd. De afstand tussen de reële locatie, waar het voertuig zich fysisch ook bevindt, en de virtuele locatie, waar de computer het voertuig acht te zijn, is kleiner bij lagere latencies. Dit leidt tot een betere positiebepaling. Deze heeft dan weer tot gevolg dat meer ongevallen vermeden kunnen worden. Welke hardware men gebruikt kan variëren van applicatie tot applicatie. De berekening zelf wordt uitgevoerd met behulp van AI doordat dit verschillende baten heeft. Deze voordelen worden besproken in de volgende paragrafen.

2.2 Artificiële Intelligentie

AI verwijst naar het simuleren van menselijk intellect in machines die geprogrammeerd worden om de menselijke redenering na te bootsen. Het wordt gezien als de studie van *intelligente agents* die zijn omgeving waarnemen en acties kunnen ondernemen om de kans van het bereiken van een bepaald doel te maximaliseren[6]. Er kan een onderscheid gemaakt worden tussen zowel sterke als zwakke AI.

- **Zwakke AI:** Dit is een vorm van AI die zich bezig houdt met onderzoek in gebieden waar handelswijzen mogelijk zijn die tekenen van intelligentie vertonen, maar niet volwaardig intelligent zijn. Hier worden de meeste vorderingen in voortgebracht, zoals handschriftherkenning of zoekalgoritmen.
- **Sterke AI:** Deze vorm van AI houdt zich bezig met onderzoek dat als doel heeft om software te creëren die zelfstandig kan redeneren en problemen aanpakken.

Het gebruiken van AI kan vele voordelen hebben[7][8]. Zo is het mogelijk om data beter en sneller te gebruiken dan de mens kan. Data kan gelezen en verwerkt worden in geautomatiseerde processen zonder de tussenkomst van een persoon en in een fractie van een seconde. Verder zal er ook, indien er meer beschikbare data zijn, een nog nauwkeuriger responsie gegeven worden. AI wordt als zwakke AI veel toegepast om repetitieve taken met relatief lage complexiteit over te nemen. Een belangrijk onderdeel is Machine Learning dat verder uitgelegd zal worden in volgende paragraaf.

2.3 Machine Learning

ML is een onderdeel van AI en is de studie en modellering van de verschillende leerprocessen dat gebruikt kan worden door verschillende computersystemen[9]. Deze systemen zijn hierdoor in staat om specifieke taken te voltooien zonder rechtstreekse instructies of regels mee te krijgen van de operator. Ze steunen in de plaats op onder meer patroonherkenning om de kans op het succesvol uit te voeren van taken te maximaliseren. Hiervoor wordt er een wiskundig model gebouwd dat gebruik maakt van trainingsdata. Deze mathematische modellen en *datahandling* kan op verscheidene manieren gebeuren. In dit hoofdstuk worden eerst een aantal algemene leertechnieken uitgelegd. Vervolgens worden een aantal gebruikelijke modellen besproken. Tot slot wordt een afweging gemaakt over welk model het interessantst is voor onze toepassing.

2.3.1 Leertechnieken

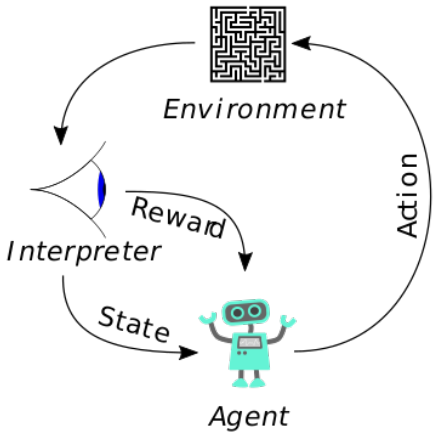
Er zijn verschillende mogelijkheden om een neuraal netwerk te laten leren. De drie belangrijkste methodes om een mathematische functie te verkrijgen worden hieronder opgesomd[3].

Supervised Learning

Deze techniek maakt gebruik van gepaarde datasets van inputobjecten en de te verwachten outputobjecten. Het doel is om een mathematische functie te creëren waarbij de gegenereerde outputs zo nauw mogelijk overeenkomen met de gelabelde outputs uit de datasets. Men optimaliseert deze mathematische functie door iteratief te trainen. De bijgeschaafde functie kan dan ook gebruikt worden voor nieuwe datasets zonder gelabelde output. Hierbij zal hij zelf outputwaardes genereren. De meest toegepaste werkwijzes zijn lineaire regressie, beslissingsbomen en Neurale Netwerken (NN). Een toepassing van Supervised Learning is bijvoorbeeld het detecteren van spam met een trainingset van al gelabelde e-mails.

Unsupervised Learning

Unsupervised learning is een techniek die gebruik maakt van Hebbian Learning om onbekende patronen te herkennen in datasets. De meest gebruikte methode onder Unsupervised Learning zijn is cluster analyse. Hierbij wordt er getracht om groep objecten te identificeren en te verdelen in clusters van gelijkaardige objecten. Deze werkwijze kan op twee voorname manier gebeuren. De eerste en meest gekende is Principle Component Analysis (PCA). PCA maakt gebruik van orthogonale transformaties om een set van mogelijke afhankelijke variabelen om te zetten in een set van lineaire onafhankelijke variabelen. Een tweede werkwijze is met behulp van K-Nearest Neighbors (KNN). Hierbij wordt er een onderscheid gemaakt tussen clusters door gebruik te maken van k nabije punten om een clusters te identificeren. Een belangrijke toepassing van Unsupervised Learning is het clusteren van gelijkaardige documenten op basis van de inhoud van de tekst.



Figuur 2.1: Routine bij Reinforcement Learning.[10]

Reinforcement Learning

Deze leertechniek heeft betrekking tot hoe agents acties moeten ondernemen in een omgeving om een bepaald attribuut te maximaliseren. Het onderscheidt zich van Supervised en Unsupervised Learning door de onafhankelijkheid van gelabelde outputdatasets. De techniek heeft als doel om een evenwicht te vinden tussen exploratie van ongekend gebied en exploitatie van de huidige kennis. In figuur 2.1 kan je een eenvoudige routine vinden van Reinforcement Learning-algoritme. Hierbij maakt een agent een bepaalde actie gebaseerd op de staat waar hij in is. Deze actie heeft in een omgeving een zekere invloed die door een Interpreter beoordeeld wordt en een score toekent. De agent kan deze verandering daarna gebruiken om zichzelf te verbeteren en zijn acties aanpassen.

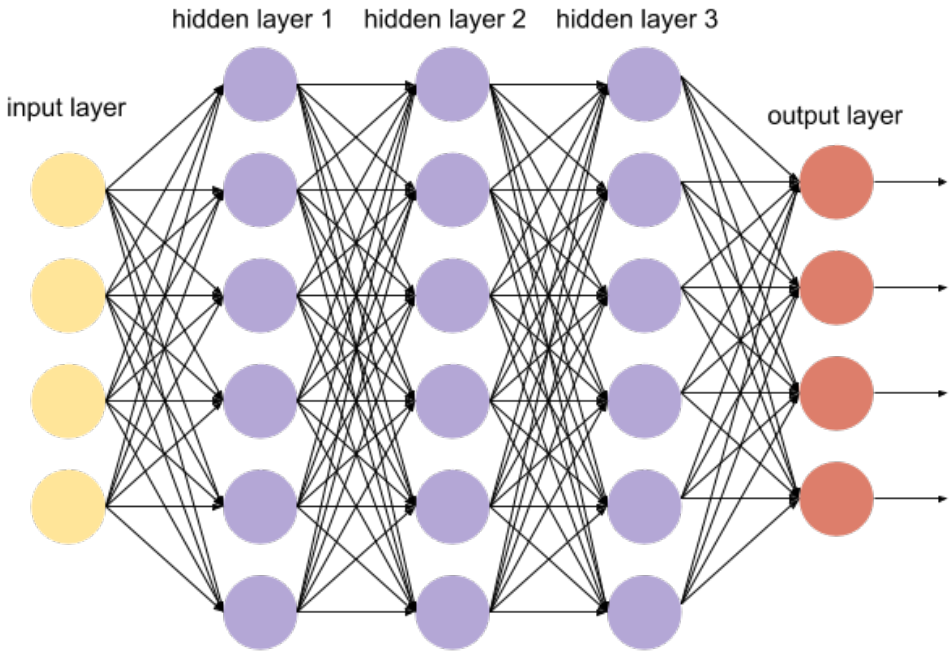
Een van de vele mogelijke toepassingen van Reinforcement Learning is het aanleren van schaken door enkel mee te geven of het algoritme gewonnen of verloren heeft.

2.3.2 Leeralgoritmes en -technieken

Om ML technieken toe te passen moet men gebruik maken van een bepaald wiskundig model dat is getraind op trainingsdata en hierdoor nieuwe data kan verwerken om voorspellingen te maken. Er bestaat een hele waaier aan mogelijke modellen. In de volgende paragrafen worden een aantal opties besproken waarna er de verschillende modellen met elkaar vergeleken worden.

Artificiële Neurale Netwerken

Artificiële Neurale Netwerken (ANN) zijn een term dat gebruikt wordt om algoritmes te omschrijven die in staat zijn om een bepaalde taak te leren, en zichzelf te verbeteren. In de meeste gevallen worden er amper richtlijnen of een omschrijving meegegeven. Het systeem ontdekt zelf hoe deze regels in elkaar zitten[9]. Hierbij blijft de interpretatie van de input en de output wel nog belangrijk. Een bekend voorbeeld is het herkennen van de cijfers 0 tot 9. Hier wordt er niet aan het systeem verteld hoe de vorm van een getal er uit ziet. Het algoritme zal dit gaandeweg ontdekken, met behulp van vele voorbeelden waar het gebruik van kan maken. Met behulp van veel data kan een algoritme zichzelf verfijnen en zo nauwkeuriger bepaalde cijfers herkennen.



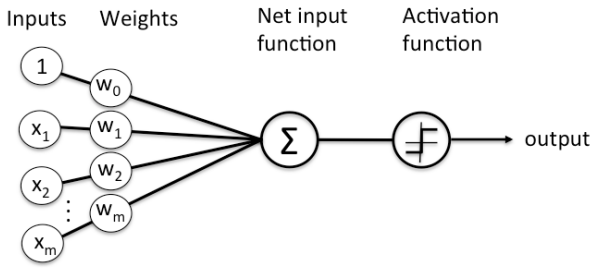
Figuur 2.2: Structuur van een Neuraal Netwerk[11].

Structuur van Neurale Netwerken Een ANN is een verzameling van nodes die met elkaar verbonden zijn zoals neuronen in de hersens van een mens. Hierbij kan elke neuron een signaal doorgeven naar het volgende neuron waar het signaal verwerkt kan worden en weer doorgegeven kan worden. Hetzelfde principe geldt ook bij NN met het verschil dat er meerdere lagen van nodes te onderscheiden zijn.

Lagen Er zijn drie soorten lagen te onderscheiden: een Input Layer, Hidden Layers en een Output Layer. Elke laag is verbonden met de volgende laag door middel van connecties tussen de verschillende nodes. In figuur 2.2 is de algemene vorm van een NN te vinden.

- **Input Layer:** De eerste laag van elke NN is de Input Layer. Deze bestaat uit een aantal inputnodes. Elke inputnode krijgt de ruwe data binnen waar er een operatie op uitgevoerd wordt en vervolgens bepaalde parameters doorgeeft aan de volgende laag. De wijze waarop data geïnterpreteerd worden, vormt een belangrijk vertrekpunt voor het NN.
- **Hidden Layers:** Na de inputlaag komen een aantal Hidden Layers. Het aantal Hidden Layers en de hoeveelheid nodes binnen één Hidden Layer kan variëren van applicatie tot applicatie en is sterk gerelateerd aan de complexiteit van de toepassing.
- **Output Layer:** Na de Hidden Layers is de laatste laag de Output Layer. Hier worden de laatste operaties uitgevoerd en worden de eindwaarden verkregen waar het resultaat uit afgeleid kan worden.

Nodes Lagen zijn opgebouwd uit meerdere nodes. Elke node krijgt een bepaald aantal inputs, verwerkt deze en geeft een bepaald aantal outputs. Deze inputs en outputs worden van node



Figuur 2.3: Algemene structuur van een node[12].

naar node doorgegeven via verbindingen. Elke node heeft met elke node in de volgende laag een connectie. Elke verbinding draagt een bepaald gewicht. Via dit gewicht kan men de invloed van de huidige node versterken of verzwakken in de volgende node.

Activatie functie De mathematische functie die een node gebruikt voor het verwerken van inputs naar outputs heet de activatie functie. In figuur 2.3 wordt de algemene vorm van een neuron besproken. Deze neuron heeft $m+1$ inputs $(x_0\ t.e.m.\ x_m)$ en bijhorende gewichten $(w_0\ t.e.m.\ w_m)$. Gebruikelijk wordt $x_0 = +1$ genomen. Hierdoor blijven er maar m echte inputs over waardoor er voor een bepaalde output de functie 2.1 opgesteld kan worden. Hierbij is ϕ een van de mogelijke transferfuncties die verder besproken zal worden.

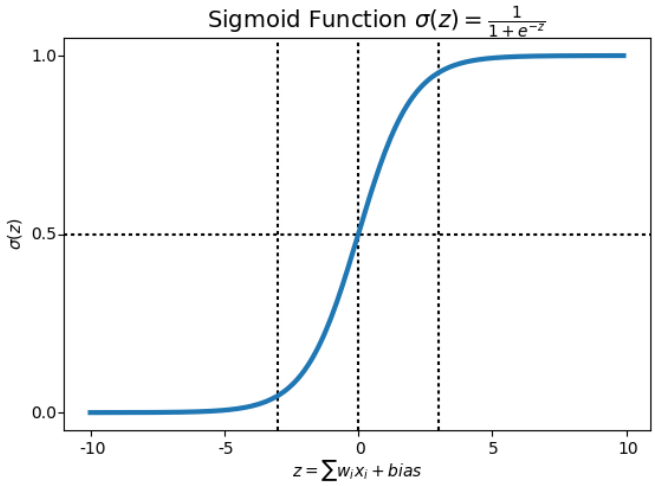
$$y = \phi\left(\sum_{j=0}^{m} w_j x_j\right) \tag{2.1}$$

Types activatiefuncties De transfer functie of activatiefunctie is een belangrijk onderdeel van een laag in een ANN. De activatiefunctie transformeert inputs uit een vorige laag en transformeert deze naar een output. Deze output zal voor een volgende laag weer als input gebruikt worden. Door gebruik te maken van de juiste activatiefuncties kunnen er niet-lineaire eigenschappen aan het netwerk toegevoegd worden. Hieronder zullen enkele transferfuncties besproken worden.

- **Lineare Combinaties:** In dit geval is de output niets minder dan de gewogen som vermenigvuldigd met een constante waarbij zoals in formule 2.2 waar er nog een tweede constante wordt opgeteld.
- **Stapfunctie:** Hier wordt er gekeken naar de verkregen waarde van de gewogen som u van $m+1$ inputs. Bedraagt deze waarde minder dan een bepaalde drempel θ, dan wordt de output gelijkgesteld aan nul, bij een hogere waarde dan weer aan 1. Dit is te zien in formule 2.3. Dit type wordt vooral gebruikt om binaire inputs te verzorgen bij de volgende laag.

$$u = \sum_{j=0}^{m} w_{kj} x_j \tag{2.2}$$

$$y = \begin{cases} 1 & \text{als } u \geq \theta \\ 0 & \text{als } u < \theta \end{cases} \tag{2.3}$$



Figuur 2.4: De Sigmoid activatiefunctie[15].

$$y = \frac{1}{1+e^{-u}} \tag{2.4}$$

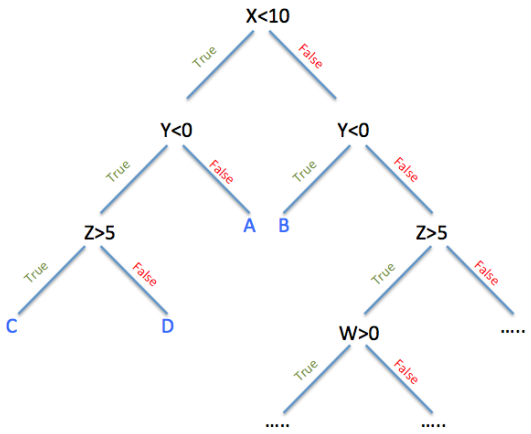
- **Sigmoid:** De Sigmoid functie[13] is een mathematische functie zoals te zien is in figuur 2.4. Het heeft de karakteristieke 'S'-vorm zoals te zien is in figuur 2.4. Door deze activatiefunctie worden inputs omgezet in een waarde tussen 0 en 1 (of -1 en 1, afhankelijk van de conventie).
- **Rectifier:** De rectifier als activatiefunctie[14] is een functie die enkel het positieve deel van zijn argument doorlaat. In vergelijking 2.5 vind je de functie weer waar x de input is van de neuron. Deze is een vector aan waarden die zowel positief als negatief kunnen zijn. Deze functie is ook gekend onder de naam *rectified linear unit (ReLU)*.

$$f(x) = x^+ = max(0,x) \tag{2.5}$$

Beslissingsboom

Het gebruik van een beslissingsboom is een leermethode die met regelmaat terugkomt in de statistiek als een voorspellend model. Men maakt gebruik van observaties rond een bepaalde uitspraak. Men kwantificeert deze observaties zodat deze leiden naar een variabele outputwaarde. Indien deze waarde valt onder te verdelen in discrete klassen spreekt men over een *classificatie boom*. Neemt de gezochte variabele eerder een continue vorm aan, dan maakt men gebruik van *regressie bomen*.

Structuur van een beslissingsboom Net zoals bij de NN bestaat een beslissingsboom uit verschillende lagen en nodes. Zoals te zien is in figuur 2.5 wordt er in elke laag een onderscheid gemaakt op basis van een statement of parameter. Deze parameter kan een enkele inputwaarde zijn of een lineaire combinatie van meerdere inputwaarden.



Figuur 2.5: Algemene structuur van een beslissingsboom[16].

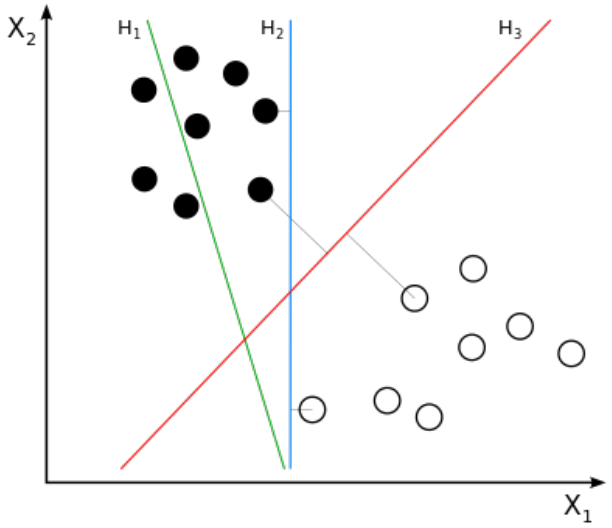
Bij de nodes kan er onderscheid gemaakt worden tussen een gewone node en een eindnode. Bij elke gewone node wordt een bepaald statement geverifieerd en wordt er naar een node overgegaan in de volgende laag op basis van dit statement. Bij een eindnode is het niet meer mogelijk om door te gaan naar een volgende laag, maar wordt er een outputwaarde gegeven.

Support Vector Machines

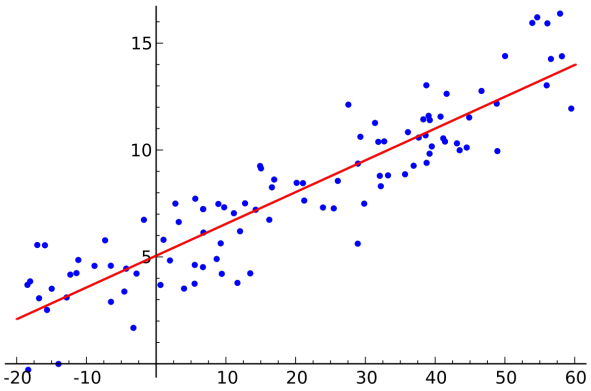
Support Vector Machines (SVMs) of ook wel gekend als support vector networks is een model dat veelvuldig gebruikt wordt in classificatie en regressie-analyse. Een belangrijke toepassing van SVMs is het verdelen van objecten in twee verschillende klassen op basis van een aantal kenmerken. Het is dus onder meer een binaire classificeerder. Om aan de hand van kenmerken een onderverdeling te maken moeten deze kenmerken eerst omgezet worden in een vectorruimte. In de trainingsfase wordt er getracht een zo optimaal mogelijke scheiding tussen beide klassen te vinden. Deze optimale scheiding wordt ook wel een *hypervlak* genoemd en ligt op een zo groot mogelijke afstand tussen de dichtstbijgelegen objecten van beide klasses of support vectors. in figuur 2.6 kan u een tweedimensionaal voorbeeld vinden. Hierin is scheidingslijn H1 geen acceptabele scheiding omdat er objecten van de zwarte klasse fout geclassificeerd worden. H2 is acceptabel maar is nog niet optimaal aangezien er weinig foutmarge is voor een nieuw object. H3 is het hypervlak omdat de foutenmarge tussen de twee klassen zo groot mogelijk is. Deze methode is niet alleen bruikbaar in toepassingen met een lineaire scheiding. Ook in niet-lineaire gevallen kan men een transformatie uitvoeren om toch een lineaire scheiding te bekomen. Deze hervorming wordt ook wel de *kernel trick*[17] genoemd.

Regressie Analyse

Regressie Analyse is een techniek uit de statistiek[19], die gebruikt wordt om gegevens te analyseren met een specifiek verband. Er bestaat vaak een relatie tussen een afhankelijke variabele en één (of meerdere) onafhankelijke variabelen. De meest voorkomende vorm van regressie analyse is de lineaire regressie, waar men op zoek gaat naar de functie die het dichtst aanleunt bij de data. De functie moet wel vervullen aan specifieke criteria, zo moet de functie bijvoorbeeld een bepaalde



Figuur 2.6: Tweedimensionale Support Vector Machine[18].



Figuur 2.7: Voorbeeld van Lineaire Regressie[20].

orde hebben. Regressie analyse wordt vooral gebruikt voor het voorspellen van nieuwe data of gebeurtenissen. In figuur 2.7 kan je een voorbeeld van lineaire regressie vinden.

Bayesian Netwerken

Bayesian Netwerken, ook wel probabilistische netwerken genoemd, zijn structuren waarin data op probabilistische wijze geanalyseerd kunnen worden. Dit wil zeggen dat men als output niet enkel de outputs maar ook de onzekerheid hierop krijgt. Men maakt gebruik van gerichte grafen. Hierin bestaan de knopen uit variabelen en de arcs beschrijven de conditionele afhankelijkheden tussen de verschillende knopen. Bayesian Netwerken worden vooral gebruikt om te analyseren wat de bepalende oorzaak is voor een zekere gebeurtenis.

Genetische Algoritmes

Genetische Algoritmes zijn een heuristiek geïnspireerd op het principe van natuurlijke selectie en zijn een klasse binnen evolutionaire algoritmes. Dit type algoritme kan gebruikt worden om

oplossingen te vinden in optimalisatie- en zoekproblemen. Door te steunen op biologische principes zoals mutatie, selectie en kruisbestuiving worden er nieuwe *chromosomen* gegenereerd die mogelijks een betere oplossing geven voor een bepaald probleem.

2.3.3 Keuze voor een Machine Learning-methode

Om de keuze voor een bepaalde ML-techniek te verantwoorden, zal dit hoofdstuk de voor- en nadelen behandelen van de voornaamste technieken die via regressie of classificatie toegepast kunnen worden[21].

Regressie

- **Neurale Netwerken:**

  Voordelen: Neurale netwerken zijn de meest gebruikte toepassing in verschillende domeinen. NN kunnen uitstekend omgaan met onder meer beeld-, audio- en tekstdata en deze verwerken. Verder kan de architectuur ook nog gemakkelijk aangepast worden aan de toepassing door te variëren in het aantal lagen of nodes. Het gebruik van de hidden layers vermindert ook het hanteren van feature engineering.

  Nadelen: Neurale netwerken zijn minder bruikbaar voor *general-purpose* algoritmes door de grote hoeveelheid data die er voor nodig zijn. In dat geval is het beter om voor beslissingsbomen te kiezen. Bovendien vragen ze veel computationeel vermogen voor het trainen van het netwerk en vragen veel expertise voor kleine aanpassingen zoals aan de architectuur of hyperparameters.

- **Regression Trees:**

  Voordelen: Beslissingsbomen met nadruk op regressie zijn in staat om niet-lineaire relaties te leren en zijn robuust voor uitschieters in de te verwerken dataset.

  Nadelen: Regression trees zijn vatbaar voor overfitting indien er te veel gebruik gemaakt wordt van branches. Bij bomen is het ook mogelijk om vertakkingen te blijven maken tot het een exacte kopie voorstelt van de trainingsdata.

- **Lineaire Regressie:**

  Voordelen: Dit is een eenvoudige methode om zowel te begrijpen als uit te leggen. Daarnaast kan er een eenvoudige bescherming tegen overfitting geïmplementeerd worden.

  Nadelen: Niet-lineaire relaties zijn een zwak punt voor lineaire regressie. Het is moeilijk om een correcte fitting te vinden voor een gegeven ingewikkelde relatie. Bovendien is het onvoldoende flexibel om complexe patronen op te vangen.

Classificatie

- **Neurale Netwerken:**

  Voordelen: NN blijven uitstekend presteren bij het classificeren van audio-, tekst- en beeldherkenning.

  Nadelen: Er is nood aan grote hoeveelheden data om het model te trainen en minder geschikt als general-purpose algoritme.

- **Classification Trees:**

  Voordelen: Verrichten zeer goed werk in praktijk. Ze zijn robuust voor uitschieters, schaalbaar voor meerdere klassen en kunnen niet-lineaire grenzen op natuurlijke wijze modelleren dankzij de hiërarchische structuur.

  Nadelen: Classification trees zijn vatbaar voor overfitting indien er te veel gebruik gemaakt wordt van branches. Bomen hebben vaak de neiging om branches aan te maken tot het een exacte kopie voorstelt van de trainingsdata.

- **Support Vector Machines:**

  Voordelen: SVMs zijn in staat om niet-lineaire beslissingsgrenzen te modelleren en hebben een sterke robuustheid tegen overfitting, vooral in hogere dimensionale vectorruimtes.

  Nadelen: SVMs zijn heel erg geheugen intensief. Ze vragen ook meer expertise in het afstemmen door het grote aanbod in mogelijke kernels. SVMs hebben de eigenschap om minder effectief te zijn bij het schalen naar grotere datasets.

- **Geregulariseerde Regressie:**

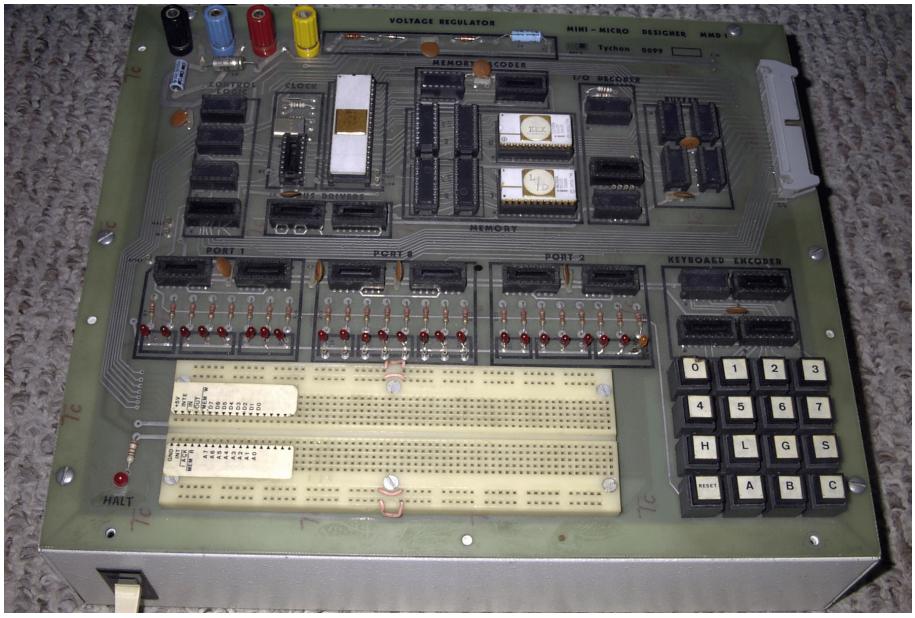
  Voordelen: Outputs hebben een gemakkelijk leesbare probabilistische interpretatie. Ook kan er bescherming tegen overfitting geïmplementeerd worden en kunnen modellen eenvoudig geüpdatet worden.

  Nadelen: Niet-lineaire relaties zijn een zwak punt voor lineaire regressie. Het is moeilijk om een correcte fitting te vinden voor een gegeven ingewikkelde relatie. Bovendien is het onvoldoende flexibel om complexe patronen op te vangen.

Besluit

Na een afweging gedaan te hebben van de voor- en nadelen van de voornaamste kandidaten bij zowel regressie als classificatie toepassingen zal in het kader van deze thesis voor NN gekozen worden. Er wordt vooral gesteund op het feit dat NN uitstekend werk levert in beide klassen in het analyseren van data. Bovendien zijn de voornaamste nadelen minder van toepassing in het kader waarin NN toegepast zal worden. Het is vooral van belang hoe de netwerken in de executiefase presteren. Het trainen van de verschillende netwerken met grote hoeveelheden data kan op aparte systemen gebeuren. De trainingsfase is dus minder van belang voor het doel van deze thesis. Bovendien is het niet nodig om een breed general-purpose netwerk te voorzien.

Een mogelijk alternatief voor NN is het gebruiken van Regressie Analyse. De bepalende factor hiervoor is dat het karakter van de verscheidene applicaties vaak uit lineaire relaties bestaat.



Figuur 2.8: Eerste SBC: MMD-1[22].

2.4 Evolutie Single Board Computers

Een SBC is een volledige computer gemaakt op 1 enkele printplaat. Het bevat onderdelen zoals een microprocessor, geheugen, inputs en outputs. De eerste SBC werd ontwikkeld als een voorstel-hulpmiddel bij educatieve doelstellingen of het gebruik als een embedded computer controller. Tegenwoordig zijn ook vele (draagbare) computers geïntegreerd op één printplaat. Het grote verschil met (draagbare) computers is dat er geen nood is aan expansion slots zoals bijvoorbeeld voor RAM-geheugen of een Graphics Processing Unit (GPU).

2.4.1 Geschiedenis

De eerste echte SBC was de zogenaamde ”dyna-micro” uit figuur 2.8 die later de naam ”MMD-1”(Mini-Micro Designer 1) kreeg[22]. Dit toestel werd uitgegeven in 1976 en werd populair doordat het werd gepresenteerd in het destijds ’BugBook’. Een andere vroege SBC was de KIM-1 (Keyboard Input Monitor 1) uit hetzelfde jaar. Beide machines werden voor ingenieurs geproduceerd en ontworpen maar vonden een breed publiek onder de hobbyisten waar het heel populair werd. Later kwamen nog andere namen zoals de Ferguson Big Board en de Nascom.

Naarmate de markt voor desktops en PC’s groeide, nam de belangstelling voor SBC in computers meer en meer af. De focus van de markt werd verlegd naar een moederbord met de belangrijkste componenten en dochterborden voor periferiecomponenten zoals seriële poorten. De voornaamste reden hiervoor was dat de componenten groot waren. Alle onderdelen op dezelfde printplaat zou zorgen voor een onpraktisch ontwerp met grote afmetingen. Deze beweging was echter tijdelijk en naarmate de vorderende technologie kleinere componenten kon leveren, werden onderdelen terug naar het mainframe verschoven. Tegenwoordig kunnen de meeste moederborden terug als SBC beschouwd worden.

In het jaar 2004 werd er in Italië een nieuwe microcontroller uitgebracht onder de naam ”Arduino”. Dit ontwerp had, naast het voordeel van compact en goedkoop te zijn, ook nog eenvoudigheid

mee. Door de eenvoud werd het Arduino-platform snel populair onder techneuten van alle soorten. Twee jaar later bracht de Universiteit van Cambridge een nieuwe goedkope SBC uit. De bekende Raspberry Pi werd gelanceerd voor de prijs van $35. Het hoofddoel van dit project was een nieuw leermiddel om te programmeren maar werd door het grote aantal applicaties ook zeer populair.

De laatste jaren kende een grote explosie aan nieuwe SBCs. Een hele reeks nieuwe namen verschenen. Banana Pi, Beaglebone, Intel Galileo, Google Coral Dev en Asus Tinker Board zijn maar enkele van de vele voorbeelden. Deze toestellen hebben vaak een processor gebaseerd op de x86- of ARM-series en maken gebruik van een Linux besturingssysteem zoals Debian.

2.5 Assortiment aan 'off the shelf' toestellen

In deze sectie wordt er een kort overzicht gegeven van de te gebruiken SBCs binnen deze thesis. Er werd gekozen om met vijf verschillende toestellen te werken die in de edge toegepast kunnen worden. Daarnaast wordt er er ook met een personal Computer gewerkt. De gebruikte devices verschillen in verscheidene aspecten. Zo zijn er zowel goedkope als kostelijkere apparaten, populaire boards als minder gekende SBCs. Er zijn toestellen met hele geavanceerde processoren die specifiek voor ML zijn ontworpen, maar ook processoren die ontwikkeld zijn voor meer algemenere toepassingen. De specificaties van de verscheidene toestellen worden ook meegegeven. In tabel 2.1 kan er een samenvatting van de belangrijkste specificaties gevonden worden.

2.5.1 Beaglebone AI

BeagleBone Ai (BB AI) is een SBC dat verder bouwt op de succesvolle BeagleBoard-series[23]. Het is een open source project met een op Linux gebaseerd aanpak. De BB AI probeert het gat tussen kleinere SBC en krachtigere industriële computers te overbruggen. Met behulp van de krachtige Texas Instruments AM5729 CPU kunnen ontwikkelaars de krachtige System On Chip (SoC) gebruiken om een hele brede waaier aan toepassingen te verwezenlijken. De BB AI maakt het toegankelijker om het AI-terrein te ontdekken en te verkennen. Door gebruik te maken van onder andere embedded-vision-engine (EVE) cores die steunen op een geoptimaliseerde TIDL machine learning OpenCL API, kan je terecht in alledaagse automatisatie in industriële, commerciële en thuisapplicaties.

Specificaties

- **GPU:** Niet van toepassing
- **CPU:** Texas Instruments AM5729
- **Memory:** 16 GB on-board eMMC flash
- **Storage:** 1GB RAM + micro SD-slot
- **Power:** 5 Watt
- **Prijs:** $139.37

2.5.2 Coral Dev Board

De Coral Dev Board is een development board gemaakt door het Amerikaanse technologiebedrijf Google[24]. Het board is ontworpen om het ontwikkelen van on-device ML producten te vergemakkelijken. Hiervoor heeft het een aantal belangrijke voordelen gekregen door zijn designers. Het is vooral de aangepaste Tensor Processing Unit (TPU) AI chip die hier opvalt. De TPU is een Application Specific Integrated Circuit (ASIC) speciaal ontworpen voor NN ML, en is in staat om video in hoge resolutie te analyseren aan 30 frames per second. Deze System-on-Module (SoM) is geoptimaliseerd om Tensorflow Lite te kunnen draaien aan meerdere tera operations per second (TOPS).

Specificaties

- **GPU:** Integrated GC7000 Lite Graphics
- **CPU:** NXP i.MX 8M SOC (quad Cortex-A53, Cortex-M4F) + coprocessor Google Edge TPU
- **Memory:** 8 GB on-board eMMC flash
- **Storage:** 1GB RAM LPDDR4 + micro SD-slot
- **Power:** 0.5 watts for each TOPS - 2 Watt
- **Prijs:** $149.99

2.5.3 Nvidia Jetson Nano

De Jetson Nano is een populair bord uit de Jetson Series van Nvidia[25]. Het is een kleine maar krachtige computer ontwikkeld voor embedded applicaties en low-power AI-Internet-Of-Things (IOT). Deze SBC wordt ondersteund door meerdere bibliotheken in sectoren zoals deep learning, computer vision, beeld en multimedia. De hardware bevat zowel een GPU als een Central Processing Unit (CPU). De GPU bestaat uit een krachtige Maxwell architectuur die beeld kan decoderen aan 500 MP/sec. De CPU is van het type Cortex-A57 met 4 kernen.

Specificaties

- **GPU:** 128-core Maxwell met 128 CUDA-cores
- **CPU:** Quad-core ARM A57 @ 1.43 GHz
- **Memory:** 4 GB 64-bit LPDDR4 25.6 GB/s
- **Storage:** micro SD-slot
- **Power:** 5 - 10 W
- **Prijs:** $99

2.5.4 Nvidia Jetson TX2

De TX2, uit de zelfde Jetsonserie, is de high end versie van de hiervoor besproken Nano[26]. Het is het snelste en meest power-efficiëntst van de embedded AI toestellen die gebruikt zal worden in deze thesis. De TX2 verbruikt een 7.5 Watt en brengt het zware AI-rekenwerk naar de edge. Zijn bekwame GPU met 256 CUDA kernen en een duo CPU zijn in staat om de meest geavanceerde Machine Leertechnieken uit te voeren. De grote geheugenvoorzieningen zorgen bovendien dat de datasetgrootte geen beperkende factor meer kan spelen. Verder wordt er nog gezorgd voor een grote ondersteuning via een grote variatie aan hardware interfaces. Hierdoor wordt het integreren van producten aanzienlijk makkelijker.

Specificaties

- **GPU:** 256-core NVIDIA Pascal GPU architecture with 256 NVIDIA CUDA cores
- **CPU:** Dual-Core NVIDIA Denver 2 64-Bit & CPU Quad-Core ARM Cortex-A57 MPCore
- **Memory:** 8GB 128-bit LPDDR4 Memory 1866 MHx - 59.7 GB/s
- **Storage:** 32GB eMMC 5.1
- **Power:** 7,5 - 15 W
- **Prijs:** $399

2.5.5 Raspberry Pi

De laatste SBC die hier besproken wordt, is de Raspberry Pi 4[27]. Dit is een heel goedkoop en eenvoudig device. Het komt uit de heel gekende Raspberry-reeks zoals al besproken in paragraaf 2.4.1. In deze thesis wordt gebruik gemaakt van het model 3 B. De specificaties kunnen hieronder gevonden worden. Het heeft een behoorlijke Cortex Quad core-CPU met degelijk geheugen die uitgebreid wordt door een SD-kaart.

Specificaties

- **CPU:** Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- **Memory:** 1GB, 2GB or 4GB LPDDR4-3200 SDRAM
- **Storage:** micro SD-kaart
- **Power:** 2,8 - 5,2 W
- **Prijs:** $35

Toestel	GPU	CPU	Power [W]	Price [$]
Beaglebone Ai	n.v.t.	TI AM5729	5	139,37
Coral Dev Board	GC7000 Lite Graphics	NXP i.MX 8M + Edge TPU	2	149,99
Jetson Nano	128-core Maxwell	Quad-core ARM A57	5 -10	99
Jetson TX2	256-core Pascal	Denver 2 & Cortex-A57	7,5-15	399
Raspberri pi	n.v.t.	BCM2711, Cortex-A72	2,8 - 5,2 W	35
PC	GeForce GTX 1050 Ti	Intel Core i5-7300HQ	10 - 100 W	981

Tabel 2.1: Specificaties van gebruikte toestellen.

2.5.6 Personal Computer: Lenovo Legion Y520

De gebruikte personal computer is van het technologiemerk Lenovo. De Legion-serie biedt een breed gamma gamingcomputers aan het publiek. Het gebruikte Y520-model[28] is samengesteld uit een aantal krachtige onderdelen. Zo bevat het model een sterke GPU, de NVIDIA GTX 1050, en CPU, de Intel Core i5-7300HQ, zorgen in combinatie met een hoge kloksnelheid voor een hoge performantie. De kloksnelheid bedraagt in idle-toestand 2500 MHz en onder load kan dit door gebruik te maken van *dynamic frequency changing* opgedreven worden tot 3300 MHz. De grote hoeveelheden RAM-geheugen en opslagruimte staan toe om meer eisende programma's te runnen op het toestel. Deze grote performantie vraagt wel meer vermogen en een hogere prijs dan de tot nu toe vermelde toestellen. Bij het vermelde vermogen zitten randapparatuur zoals scherm en toetsenbord ook in verwerkt.

Specificaties

- **GPU:** NVIDIA GeForce GTX 1050 Ti Mobile
- **CPU:** Intel Core i5-7300HQ
- **Memory:** 8GB DDR4-2400
- **Storage:** 128 GB SSD, 1 TB HDD
- **Power:** 10 - 100 W
- **Prijs:** $981

2.6 Benchmarking van Machine Learning algoritmes

Om betekenisvolle resultaten te verkrijgen is het nodig om een goed bruikbare en representatieve benchmark op te stellen. Een benchmark is een onderzoek waarbij de prestaties van programma's met elkaar vergeleken worden. Dit komt tot stand als elk programma op identieke wijze wordt onderzocht. Door de prestaties van programma's met elkaar te vergelijken is het mogelijk de performantie van de verscheidene SBC in kaart te brengen. Hoe de benchmark exact in elkaar zit is gebonden aan de kwaliteitscriteria die onderzocht worden. Om ervoor te zorgen dat de benchmark onafhankelijk is van zowel het specifiek veld als toepassing, is het nodig dat er aan een aantal karakteristieken wordt voldaan[29].

- **Vergelijkbaarheid:** Benchmarks moeten zodanig opgesteld zijn, dat het evident is wat ze vergelijken en de conclusie ondubbelzinnig is.
- **Herhaalbaarheid:** Bij het herhalen van de test onder gelijkaardige omstandigheden moeten gelijkaardige resultaten gehaald worden.
- **Goed gedefinieerde methodologie:** De werkwijze, methode en aannames moeten voldoende gedocumenteerd en gestaafd worden.
- **Configureerbaar:** Benchmarks moeten beschikken over parameters die aangepast kunnen worden naar het specifieke probleem dat wordt behandeld.

2.6.1 Bestaande benchmarks

De ontwikkelaars van de Jetson Nano vermelden op de Nvidia-website[30] verschillende Deep Learning Inference Benchmarks (DLIB) waarbij de auteur verscheidene op voorhand getrainde Deep Learning (DL) modellen toepassen op het Nano bord. Deze modellen zijn gebaseerd op een brede waaier aan populaire ML frameworks zoals Tensorflow, Caffe, PyTorch en Keras. Bovendien zijn de applicaties ook gespreid over meerdere toepassingen zoals beeldherkenning, objectdetectie, positiebepaling en anderen.

Ook voor de Jetson TX2 bestaat er een benchmark zoals voorgesteld in [31]. Het gaat over een general-purpose benchmark die een aantal parameters controleert. Het gaat over variabelen zoals Frames per Second (FPS), *inference time*, GPU temperatuur en geheugen verbruik. Met inference time wordt de tijd bedoeld om een berekening te doen in GPU en CPU m.a.w. de latency veroorzaakt door de SBC. Deze gegevens worden uit de data gehaald door middel van verschillende DL modellen toe te passen op Convolutional Neural Networks (CNN). Deze modellen passen ze toe op twee verschillende datasets: Microsoft COCO dataset voor objectdetectie, een foto bank met een grote verscheidenheid aan categorieën, en de KITTI Stereo Vision databank. De KITTI-databank bestaat uit 400 foto's specifiek bedoeld als benchmark fotoset.

Er bestaan ook meer algemenere benchmarks om vergelijkingen tussen computersystemen te maken. Zo bestaat er ook de LINPACK benchmarks[32]. Dit is een programma waar de snelheid gemeten kan worden waarmee een computer in staat is om een n bij n matrix van lineaire vergelijkingen op te lossen. Ondanks dat het een veelgebruikte benchmark is, zijn er wel nog bedenkingen over de werkwijze. Zo bestaat de kerntaak maar uit een enkele computationele taak die onmogelijk de algemene performantie van een systeem kan weergeven. Desondanks geeft de LINPACK benchmark een goed karakteristiek beeld van computersystemen. De meest bekende

toepassing waar LINPACK in toegepast wordt is de ranking van de beste 500 supercomputers ter wereld[33]

Hoofdstuk 3

Data verwerving

In dit hoofdstuk wordt de verwerving van de data en de werking van de benchmark toegelicht. Als eerste wordt de structuur van de benchmark besproken. Vervolgens wordt er uitgebreid op de wijze van meten en het opslaan van data. Dan worden de gebruikte softwaretools besproken. Nadien wordt er over gegaan op de devices die gebruikt worden in deze thesis. Vervolgens worden de gehanteerde programma's doorgenomen op vlak van toegepaste data en type NN. Het hoofdprogramma wordt in de Python-programmeertaal geschreven. Deze taal werd gekozen door de veelvuldige toepassingsmogelijkheden binnen ML. En tot slot wordt er uitgelegd waarom er conversie van modellen naar TensorFlow Lite plaats vindt.

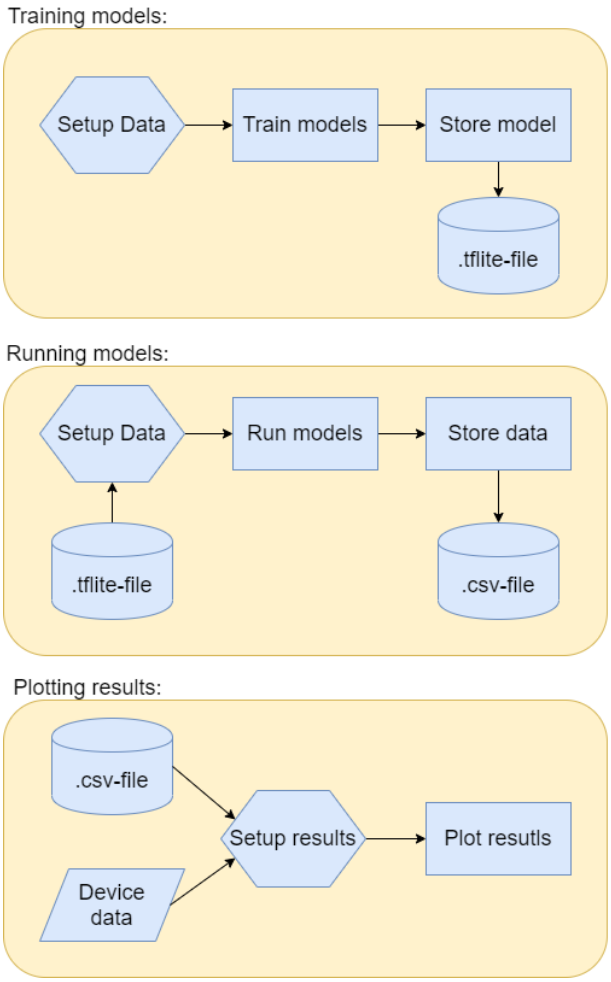
3.1 Structuur benchmark

In deze sectie wordt de structuur van de benchmark besproken. In figuur 3.1 is de structuur op schematische wijze gepresenteerd. Op de figuur valt op dat de benchmark in drie onderdelen wordt gesplitst. De onderdelen zijn het trainen van de modellen voor de verschillende te testen programma's, het uitvoeren van de modellen en het weergeven van de resultaten.

Het eerste deel is het gedeelte waar de modellen in opgebouwd en getraind zullen worden. Het proces begint hier met het voorbereiden van de data en de modellen voor de verschillende programma's. De toe te passen data krijgt de juiste vorm en de gedaante van het model wordt gedeclareerd. Vervolgens wordt het model getraind op de data en zal het geconverteerd worden naar een TFLite-model. In paragraaf 3.6.3 wordt uitgelegd waarom deze conversie gebeurd in plaats van het gebruik van een standaard TensorFlow model. Na de conversie wordt het model opgeslagen in een .tflite-bestand.

Het tweede gedeelte bespreekt het uitvoeren van de verschillende modellen. Dit is het voornaamste onderdeel van de benchmark en gebeurt op de verschillende devices zelf. Hier wordt de data gemeten tijdens het uitvoeren van elk programma. Dit gebeurt nadat de modellen vanuit de .tflite-bestanden worden geladen en de data voor verwerkt werd. Na het uitvoeren wordt de data opgeslagen in een Comma Seperated Value (CSV)-bestand.

Het derde en laatste gedeelte bestaat uit het analyseren van data uit de benchmark met behulp van data uit de specificaties zoals kostprijs, vermogen en kloksnelheid. Deze worden verwerkt en uiteindelijk gevisualiseerd in staafdiagrammen.



Figuur 3.1: De structuur van de benchmark.

3.2 Uitvoeren metingen

In de benchmark worden twee belangrijke parameters gemeten. De tijdsduur dat het programma over het uitvoeren doet en het percentage van de CPU dat gebruikt wordt tijdens het uitvoeren. Voor het meten van de tijdsduur wordt er gebruik gemaakt van de $time$-module. Met behulp van deze module kan de tijd via het commando $time.time()$ uitgedrukt worden in een floating point getal uitgedrukt in seconden. Als de tijd bij aanvang van het uitvoeren van het model opgeslagen wordt alsook bij het stoppen van het model, dan kan men de tijdsduur bekomen door het verschil te nemen tussen de stop-waarde en de start-waarde.
Om het gebruik van de CPU te meten wordt er gebruik gemaakt van de psutil-module. Deze module bevat het commando $psutil.cpu\_percent()$ waarmee het verbruik in percent opgevraagd kan worden. Het opvragen van dit commando geeft het verbruik sinds de laatste keer dat het commando werd opgevraagd weer. Dit betekent dat de functie twee keer opgeroepen moet worden. De eerste keer vlak voor het starten van het uitvoeren van het programma, de tweede keer vlak erna. Als de functie voor het eerst opgeroepen wordt, zal de waarde niet opgeslagen worden. De waarde die teruggegeven wordt bij de tweede keer opvragen wordt wel opgeslagen. Deze bevat het juiste percentage van het verbruik van de CPU sinds de start van het uitvoeren van het programma. Met de parameter $percpu$ kan er per core in de CPU gemeten worden. Aangezien alle te gebruiken devices over meerdere kernen beschikken zetten wij deze op $True$ om een beter beeld te krijgen van de werking van het programma. In listing 3.1 bevindt zich een voorbeeld van de meetmethode in pseudocode.
Om statistisch representatieve resultaten te bekomen zullen de metingen meerdere keren gebeuren. In deze thesis wordt er voor gekozen om elk programma 20 iteraties te laten voltooien. Naast de metingen van elk programma wordt er ook nog een meting gedaan over de totale duur van het uitvoeren van de verschillende programma's en het verbruik van de CPU in idle-toestand.

```
# Eerste keer opvragen van tijd en CPU-verbruik
psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()
# Uitvoeren van het model op testdata
model.run(y_test)
# Tweede keer opvragen van tijd en CPU-verbruik en opslag in gewenste variabelen
time_stop = time.time()
cpu_data = psutil.cpu_percent(interval=None, percpu=True)
time_run = time_stop - time_start
time_total += time_run
```

Listing 3.1: Meten van gewenste data.

3.3 Opslaan van data

Een belangrijk onderdeel van deze benchmark is het opslaan van de gegenereerde data. Om te voorkomen dat de data verwarrend en ongestructureerd is, wordt er getracht om data op gestructureerde wijze op te slaan in een bestand. Deze kan dan later op eenvoudige wijze verwerkt en gevisualiseerd worden.
Voor de data opgeslagen wordt, gebeurt er een controle naar de waarde van de data. Indien het zou blijken dat tijdens het meten onrealistische waarden gegenereerd worden voor een bepaalde iteratie (bijvoorbeeld 0% CPU-verbruik) dan wordt de iteratie opnieuw uitgevoerd. Indien dit niet

het geval is, wordt de data gelogd naar een CSV-bestand. Het bestand krijgt een unieke bestandsnaam gelinkt aan het moment van uitvoeren en aan het toestel waar de code op uitgevoerd wordt. In listing 3.2 kan de log-functie teruggevonden worden.

```
def logging_data(program_index, stop, start, cpu):
# Logging data
cores_avg = mean(cpu)
time_diff = stop-start
with open("unique_file_name.csv", mode='a+') as data_file:
data_writer = csv.DictWriter(data_file, fieldnames=fieldnames)
data_writer.writerow(
{'Naam': labels[program_index],
'CPU Percentage': str(cores_avg),
'timediff': str(time_diff)})
```

Listing 3.2: Opslaan van de gewenste data.

3.4 Verkennen van software

In deze thesis worden twee belangrijke libraries gebruikt om ML op applicaties toe te passen: pyrenn[1] en TensorFlow (TF)[2]. Beiden zijn een toolbox die toelaten om op heel eenvoudige manieren NN-modellen op te stellen, deze te trainen en te laten uitvoeren.
Pyrenn is de eerste bibliotheek waar gebruik van gemaakt wordt in onderafdeling 3.6.1. Het wordt toegepast op regressie-applicaties en maakt gebruik van het Levenberg-Marquardt algoritm voor het trainen van NN. Bij TF kan dit algoritme gekozen worden uit meerdere opties. De pyrenn-toolbox heeft 2 *dependencies* of afhankelijkheden in Python namelijk: pandas en numpy packages.

De tweede gebruikte bibliotheek is TF. Dit is een gratis open-source software library dat gebruikt wordt om ML-toepassingen uit te voeren, met voornamelijk NN onder de applicaties en Python als gebruikte programmeertaal. Deze bibliotheek wordt door een groot gebruikersbestand gebruikt. Binnen de TF-bibliotheek maken we gebruik van Keras. Dit is net als TF een open-source NN-bibliotheek in Python, maar is niet enkel beperkt tot TF. Ook in andere bibliotheek-omgevingen kan Keras teruggevonden worden. Onder meer in Microsoft Cognitive Toolkit, R, Theano en PlaidML kan dit teruggevonden worden[34]. In deze thesis is er voor gekozen om TF te gebruiken voor de classificatietoepassingen. Bij het opstellen van een NN wordt er gebruik gemaakt van de Keras-bibliotheek. Keras wordt gebruikt i.p.v. TF direct aan te spreken doordat Keras meer gebruiksvriendelijk is.

[1] Meer info over pyrenn is te vinden op https://pyrenn.readthedocs.io/en/latest/index.html
[2] Meer info over TensorFlow is te vinden op https://www.tensorflow.org/

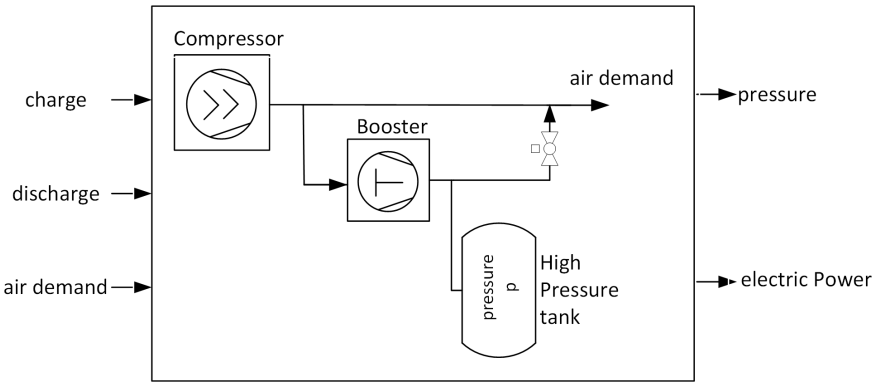
3.5 Verkennen edge-devices

In deze thesis wordt er gebruik gemaakt van drie verschillende edge-devices en een Personal Computer. De drie edge-toestellen zijn: Google Coral Dev Board, Nvidia Jetson Nano en de Raspberry Pi 3. Het zijn alle drie capabele toestellen die heel veelzijdig zijn op vlak van programma's die ze kunnen uitvoeren en randapparatuur dat kan aangesloten worden. In deze sectie worden verschillende eigenschappen van deze toestellen besproken.
De Coral Dev Board is een development board dat ML kan toepassen op de on-board Edge TPU-coprocessor. Om modellen op dit toestel te runnen is er wel nood aan TensorFlow Lite (TFLite)-compatibele modellen. In paragraaf 3.6.3 wordt er uitgelegd hoe een model naar TFLite omgezet kan worden.
Alle devices, die onderworpen worden aan de benchmark, maken gebruik van *dynamic frequency scaling*. Dit is het dynamisch veranderen van de frequentie naargelang de processor veel instructies te verwerken krijgt of niet. Op momenten dat de processor zich in een idle toestand bevindt, kan het gebruik maken van een lagere frequentie om minder vermogen te gebruiken. Door gebruik te maken van het commando $lscpu \mid grep\ MHz$ in een linux-terminal, is het mogelijk om de kloksnelheid weer te geven van het toestel in kwestie. Voor en tijdens het runnen van de benchmark werd dit toegepast om in kaart te brengen welke kloksnelheid werd toegepast op het moment van de benchmark. De Coral Dev gebruikt tijdens de benchmark de 1500 MHz frequentie, in idle toestand is dit 500 MHz. De Nano gebruikt een gelijkaardige kloksnelheid tijdens de benchmark: 1479 MHz. Voor de benchmark begon, bedroeg deze frequentie 102 MHz. De Pi gebruikt dan weer 600 MHz in ruststand en 1200 MHz gedurende de benchmark. De gebruikte Personal Computer benut een klokfrequentie van 2500 MHz in rust en 3250 MHz tijdens de benchmark.
Het verbruik van energie is een belangrijke parameter in het verkrijgen van inzicht in de resultaten. Het verbruikte vermogen voor elk toestel wordt in kaart gebracht. Er kan hiervoor de voedingsvoorziening uit datasheets beschouwd worden. Deze waarden houden echter ook vermogen voor randapparatuur zoals bijvoorbeeld een camera in. In deze thesis zal men vermogenswaarden gebruiken die representatiever zijn. Voor de Coral Dev betekent dit dat het board een vermogen van 2,65 Watt verbruikt. 2 Watt komt van de TPU die 4 TOPS uitvoert aan 2 tera operations per Watt (TOPW). De resterende 0,65 W wordt door de on-board ventilator gebruikt. Voor de Nano kan wel de datasheet voedingswaarde gebruikt worden. Dit komt neer op een verbruik van 5 Watt. De datasheet-waarde heeft hier wel al de randapparatuur in rekening gebracht. Indien er wel randapparatuur aangebracht wordt zal er op een andere manier voeding aan het board geleverd moeten worden. De Pi 3 verbruikt een 3,7 W bij het uitvoeren van programma's zonder randapparatuur. Tot slot verbruikt de Personal Computer 79.9 W bij actief gebruik. Deze waarde is inclusief peripherals zoals scherm, Wi-Fi, muis en toetsenbord aangezien deze niet los te koppelen zijn van de Personal Computer.
Een laatste belangrijke parameter is de kostprijs. Deze werd voor de verschillende toestellen al aangehaald in hoofdstuk 2.

In tabel 3.1 kan de samenvatting van deze extra data worden teruggevonden.

Toestel	Clockspeed [Mhz]	Price [$]	Power [W]
PC	3250	981	79,9
Pi	1200	41,5	3,7
Nano	1479	99	5
Coral	1500	149, 99	2,65

Tabel 3.1: Gegevens voor verscheidene toestellen.



Figuur 3.2: Praktische betekenis van het compair-subprogramma[35].

3.6 Structuur programma

In deze sectie wordt de structuur van het hoofdprogramma besproken. De belangrijkste onderdelen van de programma's worden er toegelicht. Zo wordt er weergegeven hoe de verschillende NN-modellen opgebouwd zijn en op welke data ze worden toegepast voor zowel het trainen als het uitvoeren. De benchmark bevat in totaal 10 verschillende subprogramma's. Elk van deze is een neuraal netwerk met een zekere complexiteit bedoeld voor wijde variatie aan applicaties. Van de 10 subprogramma's kunnen er zes gecategoriseerd worden als regressie en vier als classificatie. Voor elk subprogramma wordt er uitgelegd hoe het model wordt opgesteld, hoe het getraind wordt en hoe het uiteindelijk uitgevoerd wordt. Voor de benchmark is vooral het uitvoeren van de modellen van belang. Het opstellen en trainen van een NN is een eenmalige taak en wordt bijgevolg in de praktijk niet op edge-devices gerealiseerd.

3.6.1 Regressie subprogramma's

Voor de regressie subprogramma's werd er gekozen om gebruik te maken van pyrenn. Dit is een toolbox voor zowel Python als Matlab. Deze laat op een heel eenvoudige manier toe om NN te trainen en uit te voeren. De volgende subprogramma's worden opgesteld met behulp van de pyrenn-voorbeelden.

subprogramma	index	P1	P2	P3	Y1	Y2
compair	464	0	1	0.8	7	8.4
friction	14	-3			-0,29148	
narendra4	80	-0,54404			-0,45803	
pt2	208	-7,96923			-0,44761	

Tabel 3.2: Voorbeelden van de gebruikte data voor regressiemodellen.

Programma 1: compair

Het eerste subprogramma is een *compressed air storage system* of een samengedrukte lucht opslagsysteem. Het systeem heeft drie verschillende inputs en 2 gewenste outputs. De praktische werking wordt verduidelijkt in figuur 3.2 maar wordt in deze thesis niet verder op in gegaan. Hier wordt er een Recurrent Neural Network (RNN) toegepast. Dit is een regulier NN waar er een terugkoppeling bestaat tussen een node naar een vorige laag toe.

Aanmaken en trainen model Voor dit subprogramma werd er gewerkt met een dataset voorzien door Pyrenn zelf. Deze dataset levert in totaal 960 data inzendingen voor de inputs en outputs. Hiervan zijn er 480 inzendingen voorzien voor trainen en 480 voor testen van het model. In tabel 3.2 kan u een voorbeeld vinden van één datalijn. Hierbij is de inputdata P een lijst van drie features $P1$, $P2$ en $P3$. Analoog geldt dat de te verwachten outputdata Y een lijst voorstelt met twee features $Y1$ en $Y2$. Het NN wordt hier gedefinieerd door vier lagen. Een inputlaag, twee verborgen lagen en een outputlaag. Het aantal nodes voor de input- en outputlaag zijn gekend: drie en twee nodes respectievelijk. Voor de twee hidden layers werd er gekozen voor vijf nodes elk te implementeren. Het model kan gecreëerd worden met het commando *CreateNN()* zoals te zien is in listing 3.3. De variabele *net* bevat de vorm van het model. In het commando kunnen parameters toegevoegd worden. Zo wordt in een lijst de grootte en de lengte van de laag meegegeven. De parameters dIn, $dIntern$ en $dOut$ kunnen gebruikt worden om wederkerende verbindingen aan te maken. Zo wordt in dit subprogramma $dOut$ op waarde 1 gezet om van de outputlaag een verbinding met een vertraging van 1 tijdsperiode naar de vorige laag aan te brengen. Vervolgens wordt het model getraind met de data met het commando *train_LM()*. Hierbij worden parameters zoals k_max en E_stop toegepast om respectievelijk aan te duiden voor hoeveel iteraties er maximaal getraind mag worden en de minimale fout dat mag bereikt worden. Tot slot word het model ook opgeslagen in een CSV-bestand via het commando *saveNN()*. Het uitvoeren van het model kan dan op een apart device gebeuren.

```
1 # Create and train NN
2 net = pyrenn.CreateNN([3, 5, 5, 2], dIn=[0], dIntern=[], dOut=[1])
3 net = pyrenn.train_LM(P, Y, net, verbose=True, k_max=500, E_stop=1e-5)
4 # Save outputs to certain file
5 prn.saveNN(net, "./models/compair.csv")
```

Listing 3.3: Creëren en trainen van pyrenn-model.

Uitvoeren model Via het commando *loadNN()* kan het model van uit een bestand terug in een variabele worden opgeslagen. Het uitvoeren van het model op testdata kan gebeuren via de instructie *NNOut()*. Het resultaat hiervan wordt in de variabele y opgeslagen zoals in listing 3.4. In vele toepassingen is het wenselijk dat variabele y zo nauw mogelijk aansluit met de echte waarden Y. In deze thesis is de accuraatheid van het model echter niet van belang. De parameters die hier onderzocht worden zijn onafhankelijk van de accuraatheid van het model. Deze worden dus ook niet berekend en verder gebruikt.

```
# Load saved NN from file
net = prn.loadNN("./models/compair.csv")
# Calculate outputs of the trained NN for train and test data
y = prn.NNOut(P, net)
```

Listing 3.4: uitvoeren van pyrenn-model.

Programma 2: friction

Het friction-subprogramma is een voorbeeld dat een fysische grootheid berekent. Het gaat hier over de wrijvingskracht F in functie van de snelheid v. Deze grootheden voldoen aan formule 3.1.

$$F = \frac{\tanh(25 \cdot v) - \tanh(v)}{2} + \frac{\tanh(v)}{5} + 0.03 \cdot v \tag{3.1}$$

Uit deze formule kan er afgeleid worden dat we met een statisch systeem met één input, v, en één output, F werken. Voor analogie met de andere pyrenn-subprogramma's worden deze respectievelijk P en Y genoemd. De pyrenn-dataset waar we hier van gebruik maken bestaat uit 41 datapunten voor het trainen en 201 datapunten voor het testen van het model. Een voorbeeld van een datapunt kan in tabel 3.2 gevonden worden. Het model dat hier gebruikt wordt is een regulier NN en bestaat uit vier lagen. De input- en outputlaag bestaan uit één node. De twee hidden layers bestaan hier elk uit drie nodes. Zowel het creëren en trainen als het uitvoeren van het model gebeuren aan analoge wijze als in listing 3.3 en 3.4.

Programma 3: narendra4

Narendra4 is een programma dat de narendra4-functie[36] beschrijft. Dit is een voorbeeld van een dynamisch systeem met slechts één output en één input met vertraging en wordt beschreven in vergelijking 3.2. Een datapunt kan gevonden worden in tabel 3.2. Het model zal ook een RNN vormen. Hier zullen er grotere terugkoppelingen aanwezig zijn. Om een output y_{k+1} te berekenen moeten de twee vorige inputs p_{k-1} en p_k ook bekend zijn naast de huidige input. Er zal dus een vertraging van twee tijdsperiodes aanwezig zijn voor de inputnode. Dit vertaalt zich in de inputvariabele dIn uit listing 3.5 die nu gelijk is aan de waarde $[1,2]$. Op analoge wijze zijn er drie tijdsperiodes vertraging aanwezig voor de outputnode: $dOut$ is nu gelijk aan de waarde $[1,2,3]$. De twee tussenliggende verborgen lagen, die elk uit drie nodes bestaan, ondervinden zelf geen vertragingen. Het uitvoeren van het RNN gebeurt weer op analoge wijze als in listing 3.4.

$$y_{k+1} = \frac{y_k \cdot y_{k-1} \cdot y_{k-2} \cdot p_{k-1} \cdot (y_{k-2} - 1) + p_k}{1 + (y_{k-1})^2 + (y_{k-2})^2} \tag{3.2}$$

```
# Create and train NN
net = pyrenn.CreateNN([1, 3, 3, 1], dIn=[1, 2], dIntern=[], dOut=[1, 2, 3])
net = pyrenn.train_LM(P, Y, net, verbose=True, k_max=200, E_stop=1e-3)
# Save outputs to certain file
prn.saveNN(net, "./models/narendra4.csv")
```

Listing 3.5: Creëren en trainen van pyrenn-model voor narendra4.

Programma 4: pt2

Het subprogramma pt2 is een programma dat een dynamisch systeem met één input en één output beschrijft. Het te gebruiken systeem hier is een tweede order transfer functie zoals in vergelijking 3.3 is opgetekend. De gebruikte pyrenn-dataset is ook hier een set met één input feature, P, en één output feature, Y. Ook van deze set is een datapunt opgenomen in tabel 3.2. In totaal zijn er 1000 datapunten beschikbaar, waarvan 500 voor het trainen en 500 voor het testen. Voor het creëren van dit model is er gekozen om naast de input- en outputlaag, twee hidden layers te implementeren met elk twee nodes. Voor deze hidden layers wordt er een vertraging van 1 tijdsperiode voorzien. Voor de uitgang wordt er een terugkoppeling van één en twee tijdsperiodes voorzien. De waardes voor $dIntern$ en $dOut$ zijn dus respectievelijk $[1]$ en $[1,2]$ bij het aanmaken van dit model. Zowel trainen en runnen gebeuren analoog aan listing 3.3 en 3.4.

$$G(s) = \frac{Y(s)}{U(s)} = \frac{10}{0.1 \cdot s^2 + s + 100} \tag{3.3}$$

Programma 5: P0Y0-narendra4

Het P0Y0-narendra4-subprogramma is een programma dat gebruik maakt van al gekende data bij het uitvoeren van een getraind netwerk. Bij een RNN is dit een interessant gegeven voor het model. Het kan meteen de vertraagde inputs and outputs een waarde geven in plaats van deze te initialiseren op nul. Dit bevordert de accuraatheid bij de start van het uitvoeren. Dit programma wordt toegepast op de narendra4-dataset. Het model wordt dus op dezelfde wijze gecreëerd en getraind. Het verschil ligt bij het uitvoeren van het model. Hierbij worden er aan het $NNOut()$ commando drie willekeurig opeenvolgende datapunten in lijstvorm gegeven voor zowel de input als output.

Programma 6: gradient

Dit subprogramma berekent de gradiënt-vector van de foutmarge van een NN. Deze berekening is mogelijk met twee verschillende algoritmen: Real Time Recurrent Learning (RTRL) en Back Propagation Trough Time (BPTT). In deze thesis wordt er gebruik gemaakt van het RTRL-algoritme. Deze werd in de documentatie beschreven als een snellere oplossing bij het uitvoeren van het model. Dit subprogramma wordt toegepast op de pt2-dataset. Het model wordt bijgevolg op dezelfde wijze gedeclareerd als het pt2-subprogramma. De train- en run-commando's zijn te vinden in listing 3.6.

```
# Create and train NN
net = prn.CreateNN([1, 2, 2, 1], dIn=[0], dIntern=[1], dOut=[1, 2])
data, net = prn.prepare_data(P, Y, net)
# Run NN
J, E, e = prn.RTRL(net, data)
```

Listing 3.6: Creëren, trainen en runnen van pyrenn-model voor gradient.

3.6.2 Classificatie subprogramma's

Programma 7: FashionMNIST

Het FashionMNIST-subprogramma is samen met NumberMNIST een van de klassiekers voor starters die kennis met ML en NN willen maken. Bovendien worden beide programma's ook regelmatig in andere benchmarks gebruikt wat vergelijkbaarheid bevordert. Voor deze redenen zullen we beiden ook in de benchmark opnemen. FashionMNIST is een NN dat foto's van kledij-stukken probeert te classificeren volgens tien mogelijke labels.

Aanmaken en trainen model Voor we het model beschrijven worden eerst de te gebruiken data verkend. De dataset[3] van foto's en labels die voor het trainen gebruikt wordt, bestaat uit 60.000 instanties. Elke instantie uit de foto-dataset omvat een foto van 28 bij 28 pixels. Elke pixel bestaat hier uit één waarde en is dus geen RGB-pixel met drie waardes. In figuur 3.3 zijn er een aantal voorbeelden van instanties terug te vinden. Voor het model opgesteld kan worden moeten de data eerst nog verwerkt worden naar een schaal die voor de compiler van het model beter te verwerken is. De waarde van één pixel varieert tussen nul en 255. Deze worden door het maximum, 255, gedeeld zodat deze tussen nul en één komen te liggen.
Vervolgens kan het model gedeclareerd worden. In listing 3.7 wordt de declaratie, compilatie en het trainen van het model getoond. Om het model op te bouwen, werd er gebruik gemaakt van Keras. Dit is een *high level interface* die meerdere deep learning libraries kan aanspreken. Het model bestaat uit drie lagen. Aan de inputlaag wordt de verwerkte data ingegeven in matrixvorm. Vervolgens worden de 28 x 28 of 784 waarden omgezet via een hidden layer met 128 nodes en een relu-activatiefunctie naar de output. In de outputlaag wordt er de $softmax$-activatiefunctie toegepast. Deze functie zorgt voor probabilistische uitkomst voor elke outputnode. Elke node zal hierdoor een waarde krijgen die overeenstemt met de kans die het model acht aan de input om overeen te komen met een bepaald label. De som van de waarden in alle outputnodes moet gelijk zijn aan één doordat enkel de 10 gebruikte labels legitieme oplossingen zijn voor het netwerk. Na het declareren, wordt het model gecompileerd. In het $compile()$-commando worden verschillende parameters zoals optimizer en loss-functie meegedeeld aan de compiler. Deze bepalen de wijze waarop het model gecompileerd wordt. Tot slot wordt met het $fit()$-commando het trainen gestart. Hier worden de verwerkte inputdata en bijhorende labels aan toegevoegd. De volgende stap is het omzetten van het getrainde model naar een TFLite-model. Deze omzetting wordt in paragraaf 3.6.3 in detail uitgelegd. Na de conversie kan het model gebruikt worden om op data toegepast te worden.

[3] Datasets te vinden op: https://www.kaggle.com/zalando-research/fashionmnist, website geraadpleegd op 13/04/20.

Figuur 3.3: Enkele voorbeelden uit de FashionMNIST-dataset[37].

```
# Building the model
model = tf.keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation="relu"),
    # the probability for each given class (total =1)
    keras.layers.Dense(10, activation="softmax")])
# Compile the model
model.compile(optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
# training the model
model.fit(train_images, train_labels, epochs=5)
```

Listing 3.7: Creëren en trainen van sequentieel model voor FashionMNIST.

Uitvoeren model Het uitvoeren van een TFLite-model gebeurt op een andere wijze dan een standaard TF-model. Bij een gewoon model wordt de $predict()$-methode toegepast. Bij een TFLite-model wordt er eerst een *interpreter* gedeclareerd waar de verschillende tensors aan gealloceerd worden. Vervolgens kan de ingangstensor toegewezen worden met de $set\_tensor()$-methode. Daarna kan het model gerund worden op de ingangstensor, waarna de output naar de output-tensor wordt gestuurd. Met de $get\_tensor()$ kan de output opgehaald en verwerkt worden. Dit is terug te vinden in listing 3.8.

```
# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path=path_model)
interpreter.allocate_tensors()
# Run TFLite model
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
output_data = interpreter.get_tensor(output_details[0]['index'])
```

Listing 3.8: Runnen van sequentieel model voor FashionMNIST.

Programma 8: NumberMNIST

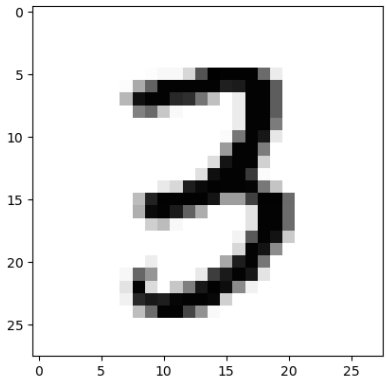
NumberMNIST is net zoals FashionMNIST een gekend voorbeeld in de ML-wereld. In dit programma wordt er een model opgesteld met als doel handgeschreven getallen van nul t.e.m. negen te herkennen. Door de eenvoud is het mogelijk om dit programma op een volledig analoge wijze te verwezenlijken zoals in FashionMNIST. Om te vermijden dat dit een exacte kopie wordt, is er voor gekozen om gebruik te maken van een ander type model dan in FashionMNIST.

Aanmaken en trainen model De te gebruiken dataset[4] voor dit model vertoont vele gelijkenissen met de dataset van FashionMNIST. Elk datapunt bestaat uit een foto van 28 bij 28 pixels. Elke pixel bestaat uit slechts één waarde i.p.v. drie zoals bij een RGB-afbeelding. De train-dataset bestaat 60.000 afbeeldingen, de test-dataset uit 10.000. Een voorbeeld van een afbeelding is te vinden in figuur 3.4. De data worden voor dit programma op dezelfde manier voor verwerkt als in FashionMNIST. De grootte van de pixels wordt door de waarde 255 gedeeld zodat de waarde van de pixels tussen nul en één liggen. Voor het opstellen van het model wordt er een andere richting uit gegaan. Voor dit model wordt er een CNN opgebouwd. De structuur ervan kan in listing 3.9 teruggevonden worden. De $Conv2D()$-methode zorgt voor de herkenning van bepaalde vormen in de afbeelding ongeacht de plaats. De opeenvolgende lagen brengen de vorm op een bepaalde plaats in verband met het juiste getal. De outputlaag is een laag met 10 nodes, één voor elk getal, waar opnieuw een $softmax$-functie op toegepast wordt. Het compilen en het trainen van het model gebeurt analoog aan FashionMNIST zoals in listing 3.7.

```
# Creating a Sequential Model and adding the layers
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(28, kernel_size=(3, 3), input_shape=input_shape))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.Flatten())  # Flattening the 2D arrays
model.add(keras.layers.Dense(128, activation=tf.nn.relu))
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.Dense(10, activation=tf.nn.softmax))
```

Listing 3.9: Structuur van het Convolutioneel Neuraal Netwerk NumberMNIST.

[4] Datasets te vinden op: https://www.kaggle.com/c/digit-recognizer/data, website geraadpleegd op 13/04/20.

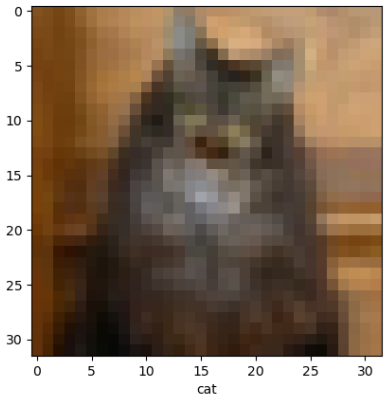


Figuur 3.4: Een voorbeeld uit de NumberMNIST-dataset.

Uitvoeren model Ook het uitvoeren van het model gebeurt op gelijkaardige wijze aan de methode in FashionMNIST. Er wordt een interpreter aangemaakt vanuit een opgeslagen model. Aan deze interpreter worden tensors toegekend die vervolgens worden ingevuld met testdata. Deze worden door het $invoke()$ commando uitgevoerd en resulteren in een tensor met de outputresultaten.

Programma 9: catsVSdogs

CatsVSdogs is het derde programma dat berust op classificatie. Het is een programma dat de inhoud van een afbeelding kan herkennen en onderverdelen volgens twee klassen: *cat* of *dog*. Dit programma is een verderzetting van een algoritme dat in staat is om tien verschillende voorwerpen te herkennen. Door de toevoeging van de laatste laag $Dense(2, activation = "softmax")$, te zien in listing 3.10, en een aanpassing van de datalabels is het mogelijk het model te vereenvoudigen naar herkennen van enkel katten en honden.



Figuur 3.5: Een voorbeeld van een kat uit de catsVSdogs-dataset.

De dataset die hier werd toegepast is de cifar10-dataset[5]. Deze bevat 60.000 afbeeldingen van 32 bij 32 pixels. Hiervan worden er standaard 50.000 afbeeldingen gebruikt worden voor het trainen van het convolutioneel model en de overige voor het testen hiervan. In deze thesis wordt er echter voor gekozen om het aantal testafbeeldingen te verlagen naar 80 afbeeldingen van katten en honden. Dit wordt gedaan om sterk uiteenlopende looptijden in de resultaten te vermijden. In figuur 3.5 is een afbeelding opgenomen. Hier is te zien dat een pixel, een RGB-pixel is met een waarde voor elke kleur. De vorm van de input is hier dus een list van $[32, 32, 3]$. Ook in dit subprogramma worden de data voorverwerkt door de grootte van de pixelwaarden te reduceren tot een getal tussen nul en één. Dit wordt gerealiseerd door alle waardes te delen door 255. De volgende stappen, zoals compileren, trainen en runnen van het model, kan nu op analoge wijze gebeuren zoals aangegeven in paragraaf 3.6.2.

```
# Creating a Sequential Model and adding the layers
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(2, activation="softmax"))
```

Listing 3.10: Structuur van het Convolutioneel Neuraal Netwerk catsVSdogs.

[5]Datasets te vinden op: https://www.cs.toronto.edu/ kriz/cifar.html, website geraadpleegd op 15/04/20.

Figuur 3.6: Een voorbeeld uit de Image Recognition-dataset.

Programma 10: Image Recognition

Het laatste subprogramma dat valt onder classificatie is het Image Recognition-programma. Dit programma is in staat om 1001 verschillende voorwerpen te herkennen op alledaagse foto's. In figuur 3.6 is een voorbeeld van zo'n afbeelding te vinden. Deze grote taak wordt verwezenlijkt door te steunen op een model dat door Google werd ontwikkeld: Mobilenet[6]. Dit is een model getraind op enkele miljoenen afbeeldingen, en bestaat uit meer dan 80 lagen en 3,2 miljoen trainbare parameters. Door gebruik te maken van dit uitgebreide model zal er geen nood zijn aan het zelf compilen of trainen van een model. De data die op dit model toegepast worden, zijn afbeeldingen van 224 bij 224 pixels. Deze data worden verwerkt maar deze keer niet tussen nul en één. De range van de data wordt herleid tot het bereik $[-1, 1]$. Voor het uitvoeren van het model worden er 124 afbeeldingen voorzien.

3.6.3 Conversie naar TFLite

Doordat er in deze thesis gebruik gemaakt wordt van devices die enkel met TFLite werken worden de modellen omgezet naar een TFLite-model. Dit gebeurt door een convertor-object te creëren van het bestaande keras-model. Op deze convertor worden optimalisatietechnieken uitgevoerd waarna het model wordt omgezet naar een TFLite-equivalent model. Dit model wordt vervolgens met het commando $write()$ uitgeschreven naar het gewenste bestand. Dit nieuwe model kan vervolgens opgeroepen worden voor uitvoering op gewenste tijdstippen. In listing 3.11 kan de gebruikte code terug gevonden worden.

```
# Convert the model
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quant_model = converter.convert()
# Saving tflite model
open(path_model + "fashionMNISTmodel.tflite", "wb").write(tflite_quant_model)
```

Listing 3.11: Converteren naar een TFLite-model.

[6] Datasets te vinden op: https://ai.googleblog.com/2017/06/mobilenets-open-source-models-for.html, website geraadpleegd op 15/04/20.

Als het model gebruiksklaar is voor TFLite, wordt van het programma ook een kopie gemaakt die op de TFLite-devices zoals de Coral Dev Board uitgevoerd kan worden. Er wordt dus gewerkt met twee programma's die heel weinig van elkaar verschillen. Ze voeren dezelfde programma's uit op dezelfde wijze. De originele versie werkt voor de classificatieprogramma's met de standaard tensorflow-libraries waar TFLite een onderdeel van is. De aangepaste versie werkt op een stand-alone versie van de TFLite-module. Deze laatste bibliotheek werkt enkel op speciaal ontwikkelde hardware zoals de Coral Dev Board. De aan te brengen veranderingen zijn terug te vinden in listing 3.12.

Het betreft twee belangrijke aanpassingen. De verandering van module en toevoeging van het $libedgetpu.so.1$-bestand. Dit bestand is de Edge TPU runtime library. Deze bibliotheek helpt bij het verdelen van instructies over de CPU en TPU.

```
# Lines in original File:
import tensorflow as tf
interpreter = tf.lite.Interpreter(model_path=path_model)
# Lines in TFLite-compatible File:
import tflite_runtime.interpreter as tflite
interpreter = tflite.Interpreter(model_path=path_model,
                experimental_delegates=[tflite.load_delegate('libedgetpu.so.1')])
```

Listing 3.12: Converteren naar een TFLite-programma.

Hoofdstuk 4

Data verwerking

In dit hoofdstuk wordt de data verkregen uit vorig hoofdstuk verwerkt in visuele resultaten. De verschillende ondernomen stappen worden besproken. De eerste stap behandelt het verwerpen van bepaalde datapunten. Vervolgens wordt de data met bepaalde parameters in verband gebracht. Tot slot wordt de data op vlak van spreiding geanalyseerd en uiteindelijk in grafiekvorm weergegeven. De resultaten zelf worden in hoofdstuk 5 besproken.

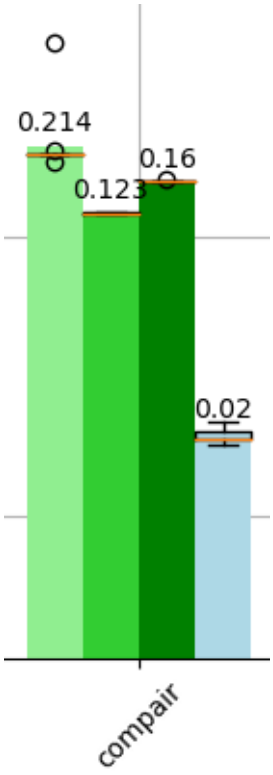
4.1 Data cleaning

Een belangrijk gebeuren voor de data verwerkt kan worden, is het schoonmaken van de data. Tijdens het uitvoeren van metingen is het mogelijk dat er meetfouten plaatsvinden. Deze kunnen door allerlei oorzaken plaats vinden en moeten zo veel mogelijk vermeden worden. In deze sectie bespreken we een aantal van deze meetfouten en hoe deze aangepakt en verwerkt worden.

4.1.1 Verwerpen data

Een eerste wijze van schoonmaken is het verwerpen van datapunten waarvan er geweten is dat deze onmogelijk correct kunnen zijn. Dit betreft meetfouten waar bijvoorbeeld de duur van het uitvoeren van een programma gelijk is aan nul seconden. Dit is uiteraard niet mogelijk, elke meting heeft namelijk een duur van een bepaalde grootte. Een andere meetfout die geregeld op trad kwam bij het meten van het CPU-verbruik voor. Indien na het uitvoeren van een programma bleek dat er geen activiteit in de CPU werd gemeten is dit als gevolg van een meetfout.

Indien deze meetfouten worden gedetecteerd zal de volledige meting worden verworpen en wordt de meting herhaald. De controle op deze meetfouten vindt dus plaats na het uitvoeren van de meting en voor het opslaan van de data. In listing 4.1 kan deze controle teruggevonden worden. Alleen nadat er geen meetfout gedetecteerd wordt de data uitgeschreven en wordt de iteratie erkend als een geldige iteratie. Door een imperfecte meting te hernemen blijft het totaal aantal bruikbare datapunten bijgevolg altijd 20.



Figuur 4.1: Een voorbeeld van uitschieters in data van de duur van het compairprogramma.

```
# Meting wordt gecontroleerd voor
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
    logging_data(10, time_stop, time_start, cores)
    iteration += 1
print("iteration: ", iteration, " mean cores: ", mean(cores), " duration: ", time_stop-
    time_start)
```

Listing 4.1: Controleren op meetfouten.

4.1.2 Behandelen uitschieters

De tweede wijze waarop de data wordt schoon gemaakt is door het behandelen van uitschieters. Tijdens het meten is het mogelijk dat een datapunt verder of dichter van het gemiddelde ligt door een externe factor. Zo kan bijvoorbeeld een subroutine van het Operating System de meting vertragen en hierdoor de meting beïnvloeden. Om deze invloeden te vermijden is het nodig om de uitschieters of outliers te herkennen en te elimineren. Datapunten worden hier beschouwd als outliers die een grotere afwijking van het gemiddelde hebben dan drie standaardafwijkingen. Om deze outliers te vinden wordt er een boxplot opgesteld met behulp van de module matplotlib. In figuur 4.1 kan een voorbeeld van data met een outlier gevonden worden. Zodra de voornaamste outliers geïdentificeerd zijn, zullen deze aangepast worden. De data wordt veranderd in het gemiddelde van de overige niet-uitschieters om het gemiddelde van de ware data niet te wijzigen.

4.2 Vormgeving resultaten

In deze sectie wordt er besproken uit welke grootheden de resultaten zullen bestaan, hoe deze gevormd worden en op welke manier de resultaten gevisualiseerd worden.

4.2.1 Gebruikte grootheden

De belangrijkste te onderzoeken parameter is de tijd. De duur van uitvoeren van een programma kan een grote factor spelen bij het maken van een kosten-baten analyse. Echter als er enkel de latency met elkaar vergeleken wordt, kan dit leiden tot een vertekend beeld. Daarom worden ook factoren zoals CPU-gebruik en kloksnelheid in rekening gebracht. De latency per percentage CPU-gebruik en per MHz kloksnelheid geeft een beter beeld van de performantie van elk toestel. Een andere variabele die in rekening gebracht kan worden is het vermogen dat elk toestel verbruikt. Door de latency te vermenigvuldigen met het vermogen bekomt men de energie die verbruikt wordt tijdens de executie. Voor de grootheden geldt:

$$time \cdot power = s \cdot \frac{J}{s} = J = Energy \tag{4.1}$$

Het is interessant om de verbruikte energie van de verschillende toestellen voor hetzelfde programma met elkaar te vergelijken. Toepassingen die energie gebonden zijn of waar een batterij de voeding voorziet kunnen een afweging maken tussen het energieverbruik en de duur van executie.

4.2.2 Weergave resultaten

De weergave van de resultaten zal op een visuele wijze in een *bar chart* gepresenteerd worden. Dit laat op eenvoudige wijze toe de verschillende boards te vergelijken. Dit wordt eveneens naast de tabelvorm regelmatig in de literatuur gebruikt. Enkel het omzetten naar een bar chart zal nog niet voldoende zijn om op een eenvoudige wijze vergelijkingen te maken. De resultaten voor verschillende programma's kunnen meerdere decades verschillen in grootte door gebruik van grotere datasets of complexere modellen. Bijgevolg zullen relatieve verschillen in grootte bij kleinere resultaten minder opvallen. Een manier om dit tegen te gaan is het gebruik van een logaritmische schaal voor de tijd-as. De resultaten die kleiner in grootte zijn, zullen hier beter gerepresenteerd worden. Een nadeel is wel dat de relatieve groottes tussen devices minder goed af te lezen valt. Als de lengte van een staaf van een eerste device twee maal groter is dan een tweede device betekent dit bijvoorbeeld niet dat het eerste device twee keer trager is dan de tweede. Een tweede methode die gebruikt wordt om vergelijkingen te vereenvoudigen, is het normaliseren van de data naar één toestel toe. Dit is het herschalen van data zodat de relatieve grootte t.o.v. een vast device gepresenteerd wordt. Er worden bijvoorbeeld dus geen werkelijke latency-waarden weergegeven maar het relatieve grootteverschillen tussen een toestel en het gekozen vaste toestel. In deze thesis werd de Personal Computer als vast toestel gekozen. Door het enige non-edge toestel te gebruiken als referentiepunt kunnen vergelijkingen tussen edge toestellen zelf en tussen edge en non-edge devices beter plaats vinden. Tot slot is er ook nog de optie aanwezig om een boxplot te tonen bovenop de figuur zelf. Indien gewenst kan de spreiding van de gebruikte data gevisualiseerd worden. Deze spreiding is typisch klein genoeg zodat deze de eenvoud van de grafiek kan belemmeren. Voor deze reden is er gekozen om boxplot als optie toe te voegen, zoals in sectie 4.3 verder uitgelegd zal worden.

4.3 Overzicht code

In deze sectie wordt de gebruikte code uitgelegd. Hoe de data verkregen uit vorig hoofdstuk verwerkt wordt en hoe deze vervolgens gevisualiseerd worden. De resultaten die hier uit leiden zullen in hoofdstuk 5 geanalyseerd en besproken worden.

4.3.1 Ophalen data

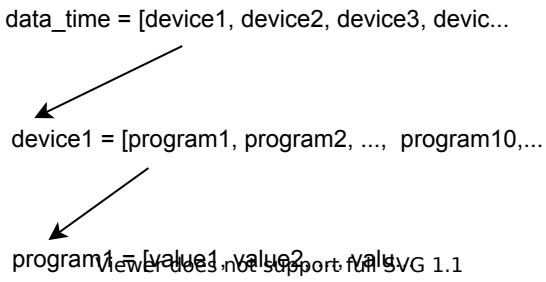
In vorig hoofdstuk werd de data gemeten tijdens het uitvoeren van het programma en vervolgens opgeslagen in CSV-bestanden. Deze zullen nu opnieuw ingelezen worden. Om de juiste bestanden op te kunnen halen wordt er eerst voor elk toestel een string aangemaakt met als inhoud de overeenstemmende titel van het databestand. De vier strings worden in een lijst opgeslagen zoals in listing 4.2. Eveneens worden er twee lijsten aangemaakt waar de tijds- en CPU-data in opgeslagen wordt.

```
# De extentie '.csv' wordt later achteraan toegevoegd
name_PC = "Benchmark_PC_Thu_Apr_2_22_05_01_2020"
name_PI = "Benchmark_PI_Fri_Apr_3_02_32_27_2020"
name_NANO = "Benchmark_NANO_Fri_Apr_3_02_11_17_2020"
name_CORAL = "Benchmark_CORAL_Wed_Apr_3_20_08_16_2020"

file_names = [name_PC, name_PI, name_NANO, name_CORAL]
data_time = [[], [], [], []]
data_CPU = [[], [], [], []]
```

Listing 4.2: Bijhouden van databestandsnamen voor elk toestel.

De lijst met opgeslagen bestandsnamen staat toe om met behulp van for-loops de data voor elk bestand sequentieel uit de bestanden te halen. Dit wordt in listing 4.3 getoond. Hier wordt er gestart met een for-loop om elk toestel te itereren. Er worden twee tijdelijke lijsten aangemaakt: $temp1$ en $temp2$. In elk van deze lijsten komt de sequentiële data van één van de features die in vorig hoofdstuk gemeten werd. In $temp2$ wordt de tijd-data van alle iteraties en alle programma's achter elkaar toegevoegd. Analoog voor $temp1$ en het CPU-percentage. De percentages worden wel nog omgezet naar een getal tussen nul en één door te delen door 100. In de volgende for-lus, die alle programma's af loopt, zal de data behorende tot het huidige programma in lijstvorm toegevoegd worden. Data met betrekking to de CPU wordt in $data\_CPU$ bijgehouden. Analoog voor tijdsdata en $data\_time$. Tot slot wordt ook de gegevens voor het totaal aan de lijst toegevoegd. Deze gegevens bestaan in het bestand maar uit één lijn. De totale duur en de CPU-gebruik in idle-stand werden maar één maal uitgeschreven naar het CSV-bestand. Om eenvoud in verdere behandeling van data te behouden wordt deze data meerdere keren aan de gepaste lijst $data\_CPU$ of $data\_time$ toegevoegd. Het aantal keer dat dit gebeurd is gelijk aan het aantal iteraties, hier dus 20.



Figuur 4.2: De vorm van gebruikte variabelen. NOG OM TE ZETTEN NAAR VECTOR

```
# Data extraction
for device in range(len(file_names)):
    temp1, temp2, = [], []

    with open('./logging/' + file_names[device] + ".csv", mode='r') as results_file:
        results_reader = csv.DictReader(results_file)
        for row in results_reader:
            temp2.append(float(row['timediff']))
            temp1.append(float(row['CPU Percentage']) / 100)

    for program in range(len(programs)):
        data_CPU[device].append(temp1[program * iterations:((program + 1) * iterations)])
        data_time[device].append(temp2[program * iterations:((program + 1) * iterations)])
    data_CPU[device].append([])
    data_time[device].append([])

    for iteration in range(iterations):
        data_CPU[device][-1].append(temp1[-1])
        data_time[device][-1].append(temp2[-1])
```

Listing 4.3: Ophalen data uit bestand voor elk toestel.

Indien het ophalen van de data is afgerond, hebben de variabelen $data\_CPU$ of $data\_time$ de vorm $(4, 11, 20)$ oftewel een lijst met drie lagen. Voor de eerste laag bestaat bijvoorbeeld de lijst $data\_time$ uit vier elementen die elk de data voor een bepaald toestel bevatten. Elke element bevat de tweede laag. De tweede laag is een lijst op zichzelf met elf elementen. Deze elementen vertegenwoordigen de tien verschillende programma's aangevuld met het totaal van de benchmark. Elk programma bevat de volgende laag. De derde laag is een lijst met 20 elementen. Deze lijst bestaat uit de fysieke datawaarden uit de benchmark. De lengte van deze lijst is bijgevolg gelijk aan het aantal iteraties die uitgevoerd werden tijdens de benchmark. In figuur 4.2 wordt deze structuur verduidelijkt. De structuur lijkt op het eerste zicht een ingewikkelde vorm te zijn, maar deze vorm maakt het aanspreken van specifieke datawaarden eenvoudig.

Als de data uit de benchmark is ingeladen kan het laatste deel van de benodigde data opgeslagen worden in variabelen. Deze datawaarden zijn specificatiewaarden die terug te vinden zijn in tabel 2.1. Deze worden in de vorm van lijsten opgeslagen waarbij de positie in de lijst overeenkomt met het device. De lijsten zijn in listing 4.4 weergegeven. Het betreft specificaties zoals de kloksnelheid, aankoopprijs en verbruikt vermogen. De volgende stap is het verwerken van de data.

```
devices = ["PC", "PI", "NANO", "CORAL"]
clockspeed = [3.25, 1.2, 1.479, 1.5]
device_price = [981, 41.5, 99, 149.99]
power = [79.9, 3.7, 5, 2.65]
```

Listing 4.4: Data uit de specificaties voor elk toestel.

4.3.2 Verwerken data

Zoals al aangegeven in subsectie 4.2.1 is het interessant om naast de latency-data ook naar grootheden zoals de duur per percentage CPU-gebruik per MHz kloksnelheid en de verbruikte energie te kijken. Om deze grootheden te visualiseren moet er eerst op de oorspronkelijke data $data\_time$ nog enkele bewerkingen gebeuren. Deze worden in listing 4.5 weer gegeven. In dit code-extract wordt met behulp van de for-lussen elke datawaarde opgeroepen, verwerkt en opgeslagen in de variabele $data\_time\_MHzCPU$. Hierbij is de variabele $labels\_time$ een lijst met de namen van alle programma's plus het totaal. De lengte van deze lijst die met behulp van het $len()$-commando wordt opgevraagd is dus gelijk aan het aantal elementen in de tweede laag van de variabele $data\_time$.

```
for device in range(len(devices)):
    for program in range(len(labels_time)):
        for iteration in range(iterations):
            data_time_MHzCPU[device][program][iteration] = data_time[device][program][
    iteration] / clockspeed[device] / mean(data_CPU[device][program])
```

Listing 4.5: Verwerking van data_time naar data_time_MHzCPU.

De verwerking van $data\_time$ om de verbruikte energie te bekomen werkt analoog aan listing 4.5 met het verschil dat elke datawaarde vermenigvuldigd wordt met het vermogen van het betreffende toestel zoals in vergelijking 4.1.

Het normaliseren van data is de volgende stap in het verwerken van de data. Met normaliseren wordt het vergelijken t.o.v. hetzelfde toestel bedoeld. Dit is een grote hulp bij het vereenvoudigen van de grafieken en bijgevolg een belangrijk onderdeel van de benchmark. In listing 4.6 kan de normalisatie verwerking gevonden worden. Aangezien er gekozen werd om met de Personal Computer te vergelijken, wordt er eerst een lijst pc_values aangemaakt. Aan deze lijst worden de gemiddelde datawaarden van de Personal Computer (dit is de index 0 in $data\_energy[0][label]$) voor elk programma toegevoegd. Vervolgens wordt elk datapunt voor elk programma in $data\_energy$ gedeeld door de corresponderende programmawaarde in pc_values. Het resultaat is een lijst met drie lagen genaamd $data\_energy\_norm$. Deze bevat voor de Personal Computer waarden rond de één met als gemiddelde voor elk programma exact één. Voor de andere devices bevat elke datawaarde nu de relatieve grootte t.o.v. het gemiddelde van de Personal Computer.

```
pc_values = []
for label in range(len(labels_time)):
    pc_values.append(mean(data_energy[0][label]))

for device in range(len(devices)):
    for program in range(len(labels_time)):
        for iteration in range(iterations):
            data_energy_norm[device][program][iteration] = data_energy[device][program][
    iteration] \ pc_values[program]
```

Listing 4.6: Normalisatie van data_energy.

4.3.3 Visualiseren data

De volgende stap is het visualiseren van de data. Om dit te verwezenlijken werd de hulpfunctie $show\_plot()$ aangemaakt. Deze functie wordt opgeroepen na het verwerken van de data en geeft een staafdiagram weer, waarop de data gevisualiseerd wordt. In listing 4.7 kan de pseudocode van de functie gevonden worden. De volledige code kan in bijlage B.2 gevonden worden. De functie beschikt over heel wat parameters die meegegeven kunnen worden. Deze worden hieronder besproken.

- **data:** Dit is de parameter die de lijst met drie lagen bevat. Er wordt verondersteld dat de vorm voldoet aan de hierboven beschreven vorm. Deze data wordt in het programma aangepast door het gemiddelde te nemen van de waarden in de onderste laag, deze waarden af te ronden tot op 3 cijfers na de komma en vervolgens op te slaan in een nieuwe variabele $data\_bar$. De nieuwe variabele beschikt dus over maar 2 lagen met groottes $(4, 11)$ of m.a.w. een matrix van 4x11.
- **ylabel:** Ylabel bevat het label dat aan de y-as wordt toegekend bij het tonen van de grafiek.
- **titel:** Deze string bevat de titel die bovenaan de grafiek wordt weergegeven.
- **labels:** Deze lijst van strings bevat de labels die onder aan de grafiek worden weergegeven. Deze zijn de namen van de programma's aangevuld met 'total' of met 'no operations' afhankelijk van het doel van de grafiek.
- **log:** Deze boolean bepaalt of de y-as al dan niet in logaritmische schaal wordt gezet. Deze staat standaard op *True*. Enkel bij het tonen van het CPU-verbruik staat deze op *False*.
- **show:** Deze boolean bepaalt of de grafiek al dan niet getoond wordt. Deze parameter aan of uitzetten vereenvoudigt het debuggen van de code.
- **index:** Deze integer houdt de volgorde van de grafieken bij. Dit wordt vooral gebruikt bij het opslaan van de grafieken volgens een unieke naam.
- **boxplot:** Deze boolean bepaalt of er bovenop de staafdiagrammen de gepaste boxplots geplaatst moeten worden.
- **normalise:** Deze boolean bepaalt of de meegegeven data op een speciale wijze wordt weergegeven of niet. Wordt de data niet genormaliseerd, dan worden alle vier de devices

naast elkaar in staafvorm getoond. Wordt de data wel genormaliseerd zullen enkel de edge-devices in staafvorm gepresenteerd worden en zal de Personal Computer getoond worden met behulp van een rode lijn.

```
def show_plot(data, ylabel, titel, labels, log, show, index, boxplot, normalise):
    if show = false: return #do nothing

    def autolabel(rects): # autolabelscript voor waarden boven elke staaf
        for rect in rects:
            height = rect.get_height()
            ax.annotate('{}'.format(height), xy=(rect.get_x() + rect.get_width() / 2, height
    ), xytext=(0, 3), textcoords="offset points", ha='center', va='bottom')

    data_bar = [[], [], [], []]
    for device in range(len(devices)):
        for program in range(len(labels)):
            data_bar[device].append(float(round(mean(data[device][program]), 3)))

    fig, ax = plt.subplots()
    x = np.arange(len(labels))  # De x-coordinaten voor de labels.

    if normalise = false: # Indien er niet genormaliseerd wordt
        // Voeg de data in staafdiagram toe voor alle vier de devices
        // Autolabel elke staaf
        if boxplot:
            // Voeg boxplot toe bovenop elke staaf.
    elif normalise:
        // Voeg de data in staafdiagram toe voor de drie edge-devices
        // Autolabel elke staaf
        // Voeg rode lijn toe ter hoogte van y = 1
        if boxplot:
            // Voeg boxplot toe bovenop elke staaf.

    // Voeg labels, titel, legende, etc. toe in gewenste grootte, vorm, rotatie
    if log: # Zet de y-as in logaritmische schaal indien gewenst.
        plt.yscale("log")
    plt.savefig(image_path + "Figure_{}".format(index)) # Opslaan van elke grafiek op
    gewenst adres volgens uniek index
    plt.show() # Plot the results
```

Listing 4.7: Pseudocode van de show_plot()-functie.

In listing 4.8 is een voorbeeld van gebruik weergegeven. Hierin worden alle parameters ingevuld op de figuur met de duur uitgedrukt per CPU-percent per MHz. In totaal zal deze functie zeven keer uitgevoerd worden en zullen we dus zeven staafdiagrammen krijgen.

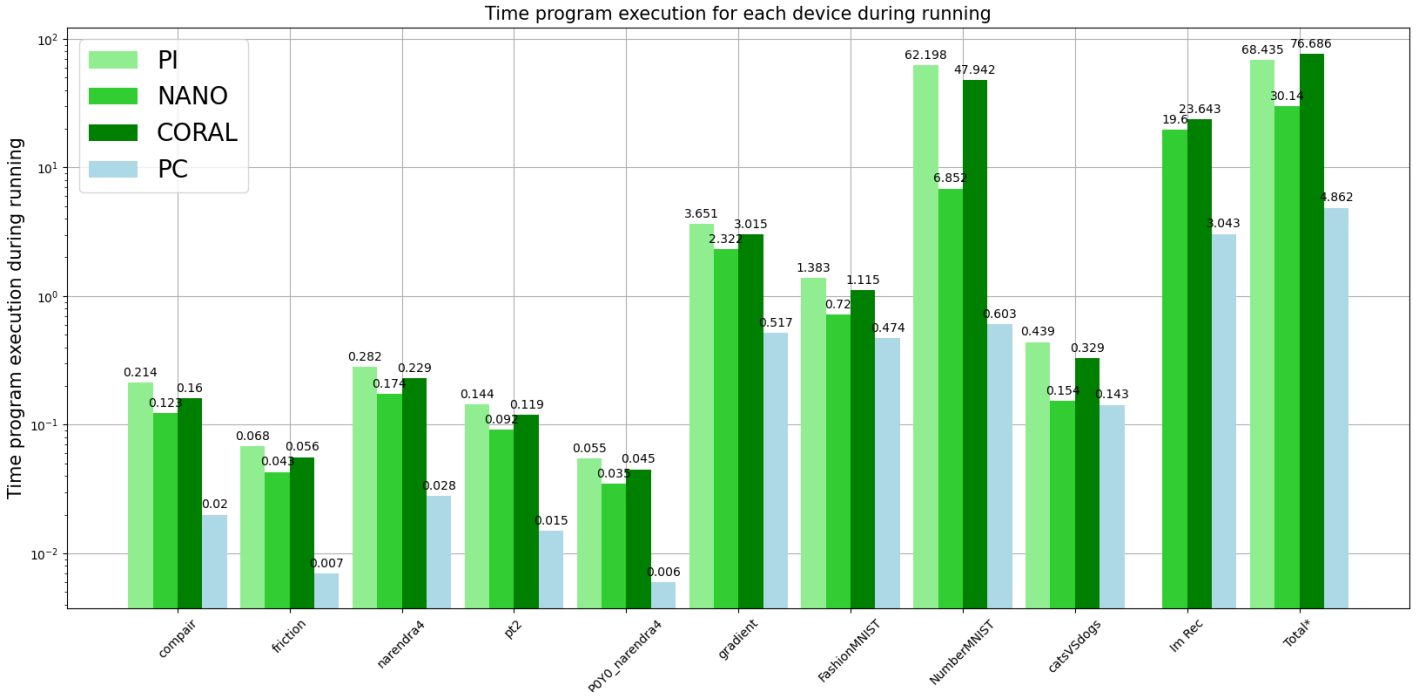
```
show_plot(data_time_MHzCPU_norm,
            ylabel="Time / CPU% / MHz.",
            titel="Time / CPU% / MHz for each device, Normalised.",
            labels=labels_time,
            log=True, show=False, index=4,
            boxplot=False, normalise=True)
```

Listing 4.8: Voorbeeld van gebruik van de show_plot()-functie.

Hoofdstuk 5

Resultaten

In dit hoofdstuk zullen de resultaten besproken worden. We bespreken elke grafiek die gegenereerd wordt uit vorig hoofdstuk. We beginnen met eerst twee kanttekeningen te maken. De eerste kanttekening betreft het programma *Image Recognition* (Im Rec) uitgevoerd op de Raspberry Pi. De Pi kon voor dit programma geen resultaten halen. Het programma kon niet op betrouwbare manier herhaaldelijk uitgevoerd worden. Er zal dus op die locatie in de grafiek geen data aanwezig zijn. Dit heeft de tweede kanttekening als gevolg. In het totaal voor de Raspberry Pi zit de tijdswaarde van dit programma niet. Het totaal zal lager liggen dan als het programma wel uitgevoerd zou kunnen worden. Men kan dus niet het totaal van de Pi rechtstreeks vergelijken met het totaal van andere devices waar de tijdswaarde van Im Rec wel in verrekend zit.



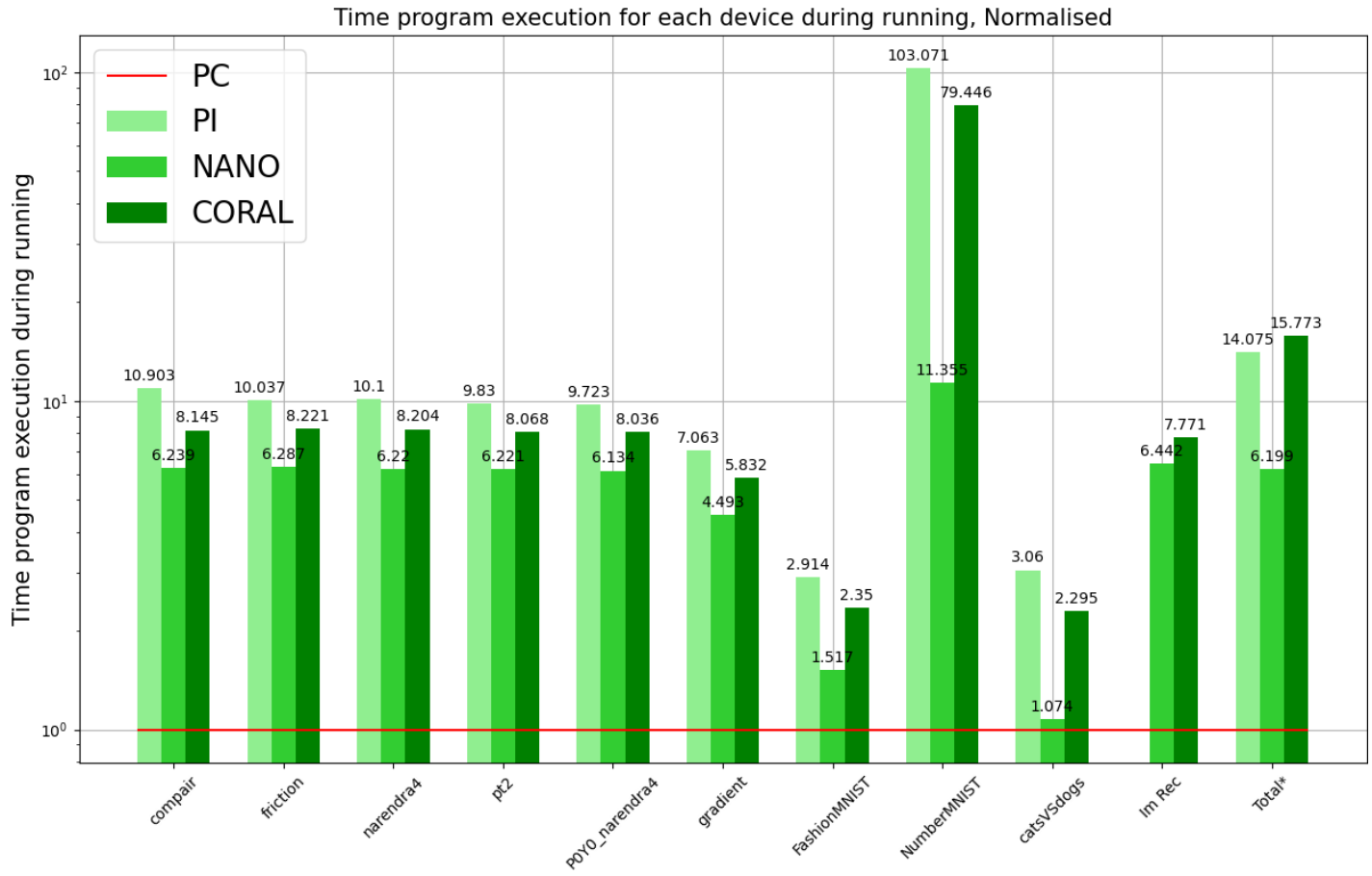
Figuur 5.1: Staafdiagram met de latencydata.

5.1 Ruwe benchmark gegevens

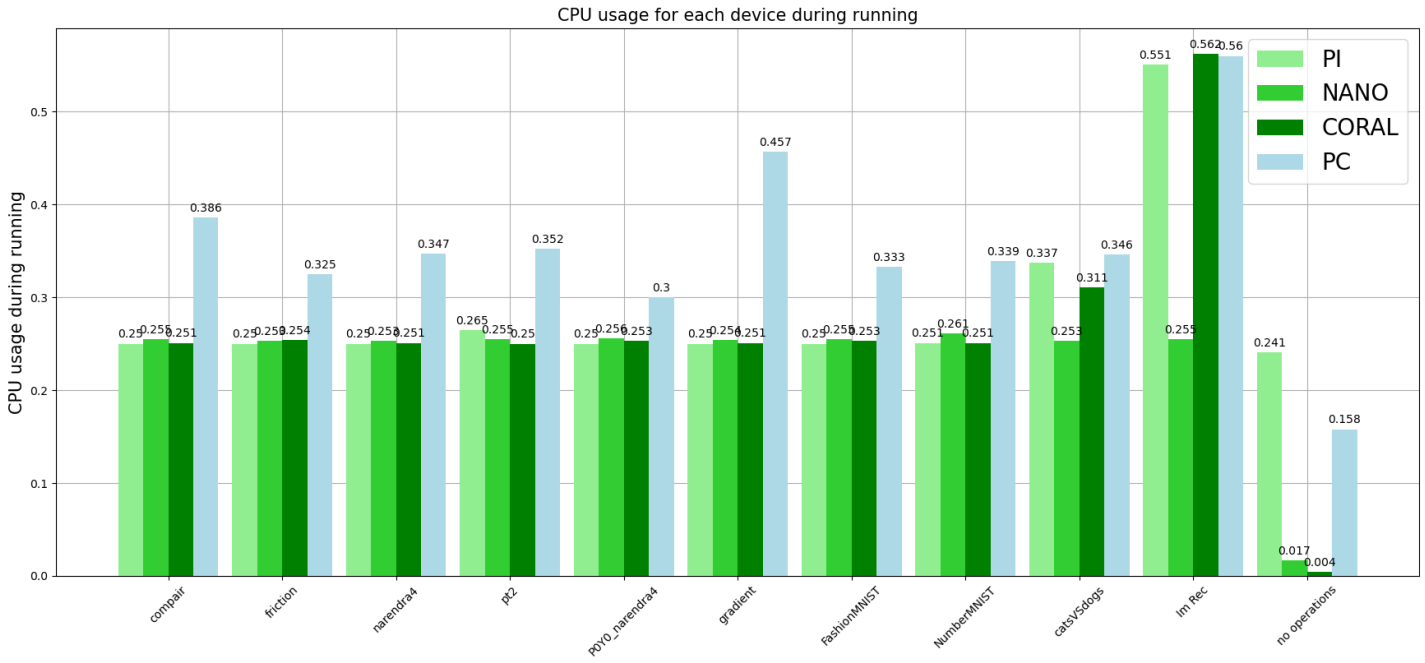
De eerste analyse behandelt de ruwe benchmark gegevens. Deze zijn terug te vinden in figuur 5.1 en 5.3 waarin respectievelijk de latency en CPU-verbruik gegevens in worden weergegeven. In figuur 5.2 kan de genormaliseerde latencydata teruggevonden worden. Op de eerste figuur kan een duidelijk verschil tussen de edge-devices en de Personal Computer opgemerkt worden over alle programma's heen. De Personal Computer heeft telkens een beduidend lagere latency. Dit ligt in lijn met de stevigere hardware waar de Personal Computer over beschikt.

Als we de edge-devices met elkaar vergelijken op de figuur met genormaliseerde latencydata valt er ook een consistent verschil op te merken. Hier haalt de Jetson Nano bij zowel regressie als classificatie lagere tijdswaarden bij elk programma. De Coral Dev behaalt grotere latencies dan de Nano en bij de Pi duurt het uitvoeren van een programma altijd het langst. Er valt bij de regressieprogramma's ook een nagenoeg constante relatieve verhouding op te merken tussen alle devices. Tussen de Personal Computer en de Nano ligt dit rond de 6.2 die enkel bij het gradient-programma afwijkt. De verhouding van de Coral t.o.v. de Nano ligt rond 1.3 en dit blijft ook behouden bij de het gradient-programma. Tussen de Pi en de Coral ligt deze verhouding over de regressie programma's rond de 1.23. Bij de classificatieprogramma's variëren deze verhoudingen veel meer.

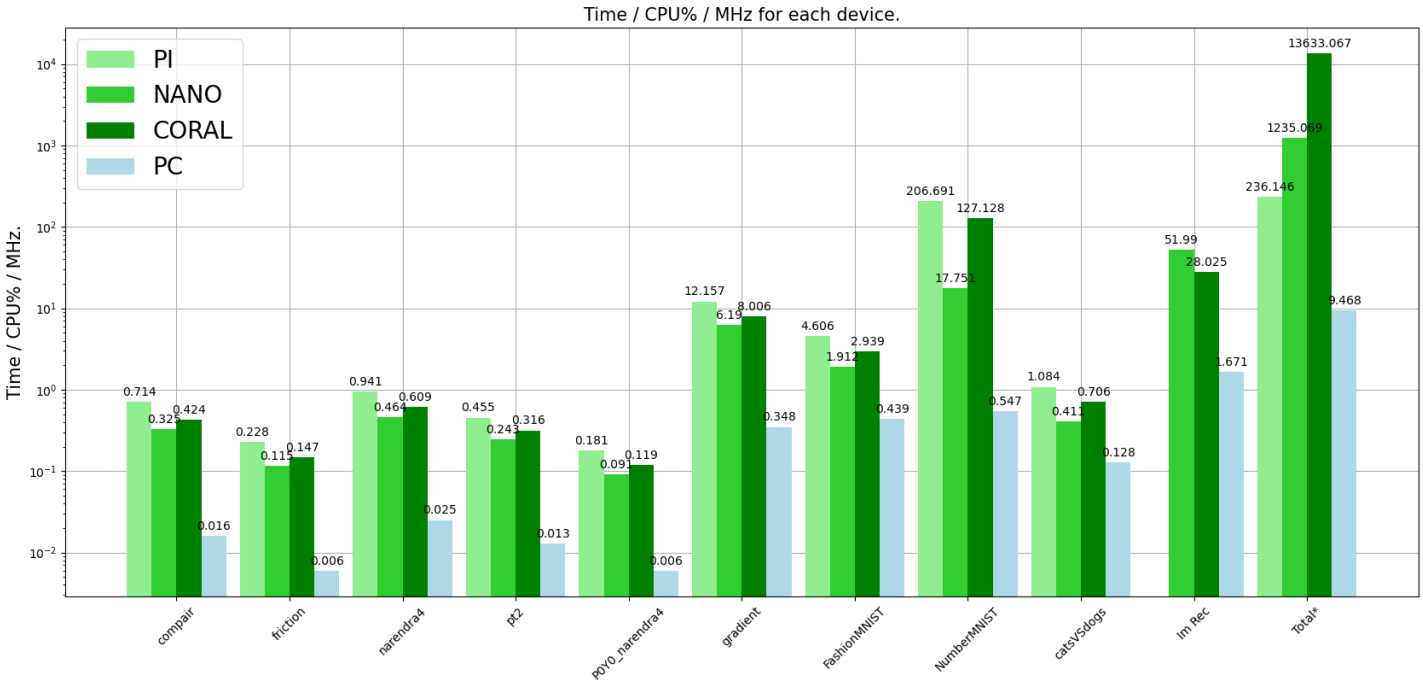
Beschouwen we figuur 5.3 dan kan er opgemerkt worden dat alle devices gebruik maken van ongeveer 25 % van hun CPU-capaciteit. Dit komt overeen met het gebruiken van één kern in de quad-core CPU's. Het hogere gebruik van de CPU bij de Personal Computer kan toegewijd worden aan de extra capaciteit die nodig is voor het aansturen van randapparatuur en -processen zoals scherm en toetsenbord, zoals ook te zien is bij *no operations* in de figuur.



Figuur 5.2: Staafdiagram met de genormaliseerde latencydata.



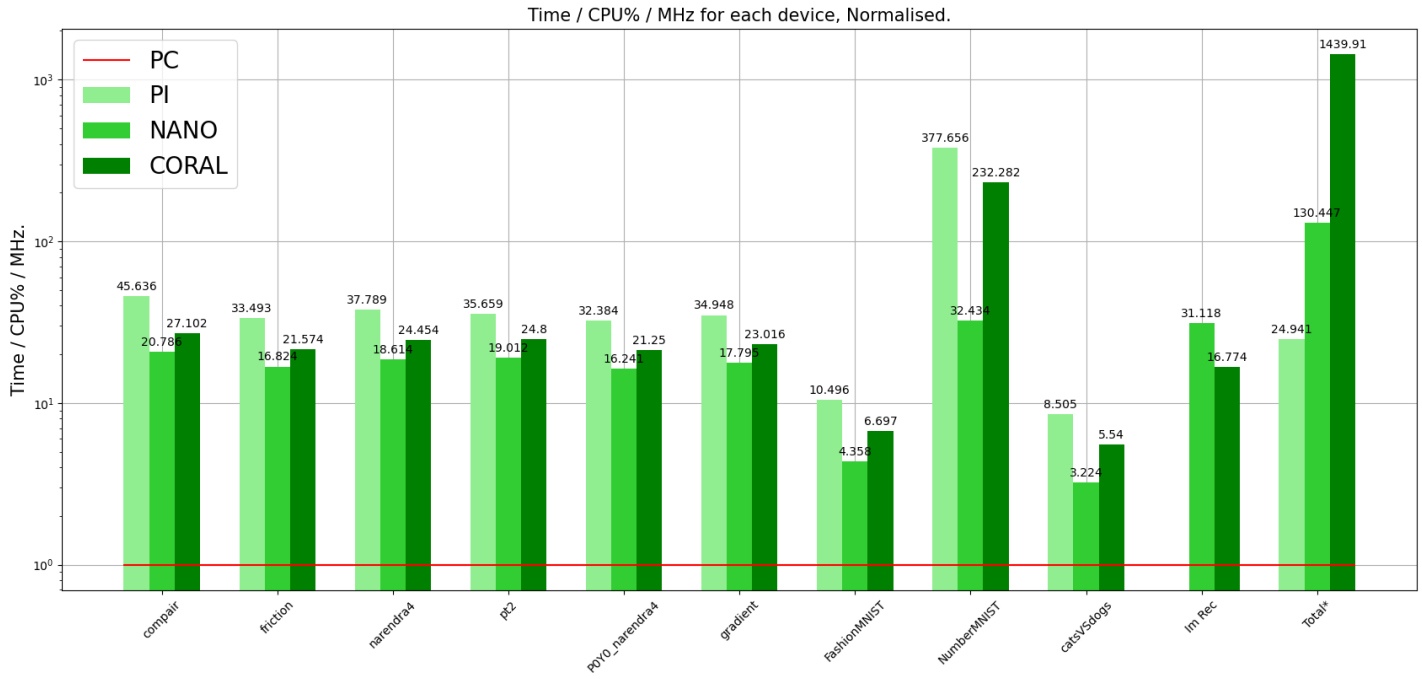
Figuur 5.3: Staafdiagram met de CPU-verbruikgegevens.



Figuur 5.4: Staafdiagram met de performantie van de toestellen.

5.2 Performantie

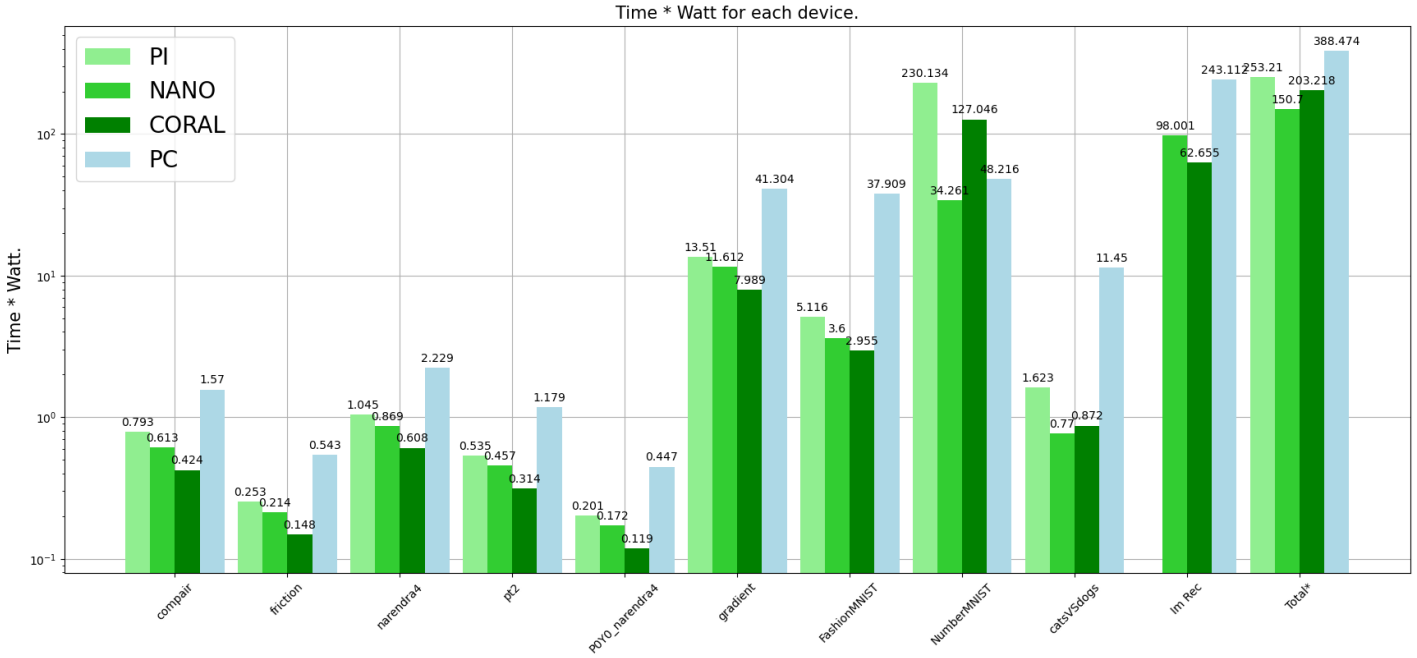
De tweede analyse die gemaakt wordt, heeft betrekking tot de performantie van de toestellen die in deze thesis onderzocht worden. Hierbij worden de gegevens uit figuur 5.1 gedeeld door de gepaste kloksnelheid en percentage CPU-gebruik zoals beschreven in hoofdstuk 4. De resultaten hiervan kan in figuur 5.4 gevonden worden. De genormaliseerde waarden zijn te vinden in figuur 5.5. In het staafdiagram met de performantie van de verscheidene toestellen kunnen een aantal zaken afgeleid worden. Net zoals in vorige sectie heeft de Personal Computer de laagste waarden. Dit ligt in lijn met de hoge en performante specificaties van het toestel. Ook heeft de Jetson Nano wederom de laagste tijdwaarden tussen de verschillende edge-devices gevolgd door de Coral en met de Pi als traagste SBC. Er kunnen gelijkaardige trends opgemerkt worden tussen figuur 5.2 en 5.5 bij de regressieprogramma's. Bij de classificatieprogramma's is een zelfde trend op te merken. Wel vinden we voor het Im Rec programma nu een andere volgorde weer. Hier presteert de Coral beter dan de Nano voor het gebruik van het mobilenet-model. Dit verschil in volgorde komt voor uit het beter gebruik van de CPU door de Coral dan de Nano tijdens het programma Image Recognition.



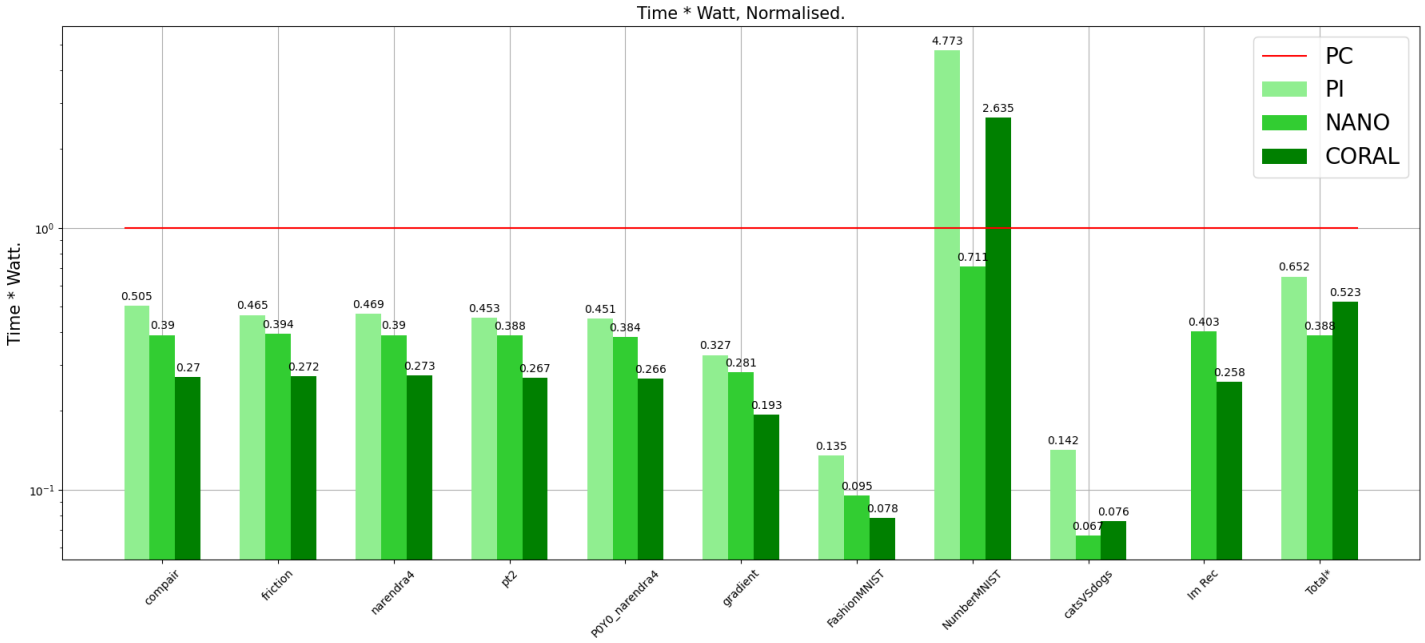
Figuur 5.5: Staafdiagram met de genormaliseerde performantie van de toestellen.

5.3 Energieverbruik

Tot slot kan er ook nog een analyse gemaakt worden met betrekking tot het energieverbruik tijdens het uitvoeren van de verscheidene programma's. De resultaten hiervan kunnen in figuur 5.6 gevonden worden, de genormaliseerde waarden in figuur 5.7. Wat opvalt bij deze figuren is dat tussen de edge devices de Nano ondanks de kortere tijdswaarden in vorige resultaten niet efficiënter is met energie dan de andere edge-devices. De Coral Dev is over een groot deel van de programma's energie efficiënter. Enkel bij het programma NumberMNIST en catsVSdogs verbruikt de Nano nog minder energie dan de Coral. Verder vinden we ook dat de Personal Computer door de extra randapparatuur bij alle programma's behalve NumberMNIST meer verbruikt dan de verschillende SBCs. Bij dit programma is het vooral de lange executieduur van de toestellen Raspberry Pi en de Coral Dev waardoor er meer energie verbruikt wordt.



Figuur 5.6: Staafdiagram van het energieverbruik.



Figuur 5.7: Staafdiagram van het genormaliseerde energieverbruik.

Hoofdstuk 6

Conclusie

In deze studie is onderzocht hoe een benchmark opgesteld kan worden om verschillende toestellen te vergelijken met elkaar en zo inzicht te bieden bij een kosten-baten analyse. Met behulp van de packages Pyrenn en Tensorflow werden tien verschillende programma's opgesteld, waarvan zes regressie en vier classificatieprogramma's. Vervolgens werden deze uitgevoerd op vier toestellen: de Google Coral Dev, Nvidia Jetson Nano, Raspberry Pi en de Personal Computer. Hierbij werden twee metrics gemeten: de duur van executie en het verbruik van de CPU. Deze data werd uitgeschreven naar CSV-bestanden. Als alle data verzameld werd kon de verkregen data verwerkt worden met behulp van extra data zoals kloksnelheid, prijs en energieconsumptie. Tot slot werd de data gevisualiseerd in staafdiagrammen en genormaliseerd om vergelijkbaarheid te bevorderen.

Uit de resultaten blijkt dat de Personal Computer de laagste tijdswaarden blijft halen t.o.v. de edge-toestellen. Dit werd verwacht door de krachtige specificaties. Tussen de verschillende edge-toestellen is de Nano de SBC met de laagste tijdwaarden. Het board word gevolgd door de Coral en de Pi. Deze bevindingen zijn onafhankelijk van het type uitgevoerde programma. Verder werden deze bevindingen ook terug gevonden indien er gecompenseerd werd voor de performantie van elke CPU. Hier haalde de Coral wel lagere waarden dan de Nano bij het programma *Image Recognition*.

Tot slot werd er gekeken naar het laagste energieverbruik van de verscheidene toestellen. Hier werd er gevonden dat vooral de Coral Dev minder energie verbruikt dan andere toestellen. De Coral werd eerst gevolgd door de Nano en daarna de Pi. De Personal Computer verbruikt het meest energie, voornamelijk door de extra randapparaten. De Nano had wel een lager verbruik dan de Coral bij classificatieprogramma's zoals NumberMNIST en catsVSdogs.

Uit de benchmark blijkt dus dat de Nano een betere keuze is bij tijdsduurgevoelige applicaties in de edge waar de duur van executie van een programma zo laag mogelijk dient te zijn. De Coral is een betere optie bij applicaties waar energieverbruik van groter belang is. Hier kan wel de keuze voor Nano bij bepaalde programma's beter uitkomen.

6.1 Toekomstig werk

Toekomstig werk aan deze benchmark berust zich vooral op het uitbreiden van de huidige benchmark. Deze uitbreiding kan op meerdere vlakken gebeuren. Het eerste vlak waar gewerkt kan worden is het uitbreiden naar andere boards toe. Het kan interessant zijn om verschillende edge-devices te testen en te vergelijken met de huidige geteste SBCs. Boards die hiervoor in aanmerking zouden komen zijn bijvoorbeeld: Raspberry Pi 4, Nvidia Jetson TX2 en Qualcom DragonBoard.

Een tweede optie voor het uitbreiden van de benchmark is het toevoegen van meerdere programma's eventueel over meerder packages. Tot nu toe werden 2 packages voor classificatie- en regressiemodellen gebruikt. Het kan een interessant zijn om ook andere onderdelen van Machine Learning toe te passen en de resultaten hiervan te analyseren.

Vervolgens is het ook mogelijk om de code verder te optimaliseren. Door gebruik te maken van klasses voor de boards en metingen is het eventueel mogelijk de code te vereenvoudigen. Bovendien zijn er nog mogelijkheden om de code zodanig aan te passen om het makkelijker te maken om meerdere toestellen en programma's toe te voegen aan de benchmark.

Tot slot is het mogelijk om aanpassingen te maken om cloud-devices en edge-devices rechtstreeks met elkaar te vergelijken voor de executieduur. Hiervoor moet er een oplossing worden gevonden om de latency veroorzaakt door het verzenden van data over het internet in te calculeren.

Bibliografie

[1] M. R. Minar and J. Naher, “Recent advances in deep learning: An overview,” 02 2018.

[2] J. Bloem, M. Van Doorn, S. Duivestein, D. Excoffier, R. Maas, and E. Van Ommeren, “The fourth industrial revolution,” *Things Tighten*, vol. 8, 2014.

[3] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[4] L. Barreto, A. Amaral, and T. Pereira, “Industry 4.0 implications in logistics: an overview,” *Procedia Manufacturing*, vol. 13, pp. 1245–1252, 2017.

[5] H. Li, K. Ota, and M. Dong, “Learning iot in edge: Deep learning for the internet of things with edge computing,” *IEEE Network*, vol. 32, no. 1, pp. 96–101, Jan 2018.

[6] D. I. Poole, R. G. Goebel, and A. K. Mackworth, *Computational intelligence*. Oxford University Press New York, 1998.

[7] https://www.frankwatching.com/archive/2017/02/06/artificial-intelligence-6-redenen-\
waarom-je-juist-nu-moet-starten/, accessed on 27.1.2020.

[8] https://www.sas.com/en_us/insights/analytics/what-is-artificial-intelligence.html, accessed on 27.1.2020.

[9] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell, “An overview of machine learning,” in *Machine learning*. Springer, 1983, pp. 3–23.

[10] http://webindream.com/reinforcement-learning/, accessed on 23.12.2019.

[11] https://github.com/drewnoff/spark-notebook-ml-labs/tree/master/labs/DLFramework, accessed on 24.12.2019.

[12] https://pathmind.com/wiki/neural-network, accessed on 23.12.2019.

[13] J. Han and C. Moraga, “The influence of the sigmoid function parameters on the speed of backpropagation learning,” in *International Workshop on Artificial Neural Networks*. Springer, 1995, pp. 195–201.

[14] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[15] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6, accessed on 23.12.2019.

[16] https://medium.com/machine-learning-bites/machine-learning-decision-tree-classifier-9eb67cad263e, accessed on 24.12.2019.

[17] F. Sun, “Kernel coherence encoders,” 2018.

[18] https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788994170/5/ch05lvl1sec36/support-vector-machines, accessed on 20.12.2019.

[19] I. Sieben and L. Linssen, “Logistische regressie analyse: een handleiding,” *Geraadpleegd via www. ru. nl/publish/pages/525898/logistischeregressie. pdf*, 2009.

[20] https://en.wikipedia.org/wiki/Regression_analysis, accessed on 19.12.2019.

[21] https://elitedatascience.com/machine-learning-algorithms, accessed on 21.12.2019.

[22] https://nl.wikipedia.org/wiki/Singleboardcomputer, accessed on 23.12.2019.

[23] https://beagleboard.org/ai, accessed on 23.12.2019.

[24] https://venturebeat.com/2019/10/22/googles-coral-ai-edge-hardware-launches-out-of-beta/, accessed on 23.12.2019.

[25] https://developer.nvidia.com/embedded/jetson-nano, accessed on 23.12.2019.

[26] https://developer.nvidia.com/embedded/jetson-tx2, accessed on 23.12.2019.

[27] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/, accessed on 23.12.2019.

[28] https://www.notebookcheck.net/Lenovo-Legion-Y520-15IKBN-7300HQ-GTX-1050-Ti-FHD-Laptop-Review.256682.0.html, accessed on 20.04.2020.

[29] L. A. Libutti, F. D. Igual, L. Pinuel, L. De Giusti, and M. Naiouf, “Benchmarking performance and power of usb accelerators for inference with mlperf.”

[30] https://devblogs.nvidia.com/jetson-nano-ai-computing/, accessed on 17.02.2020.

[31] L. P. Bordignon and A. von Wangenheim, “Benchmarking deep learning models on jetson tx2.”

[32] https://en.wikipedia.org/wiki/LINPACK_benchmarks, accessed on 27.02.2020.

[33] https://www.top500.org/project/linpack/, accessed on 27.02.2020.

[34] https://keras.io/backend/, accessed on 16.04.2020.

[35] https://pyrenn.readthedocs.io/en/latest/examples.html, accessed on 10/04/2020.

[36] K. S. Narendra and K. Parthasarathy, “Identification and control of dynamical systems using neural networks,” *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990.

[37] https://becominghuman.ai/how-to-create-a-clothing-classifier-fashion-mnist-program-on-google-colab-99f620c24fc accessed on 13/04/2020.

Bijlage A

Beschrijving van deze masterproef in de vorm van een wetenschappelijk artikel

KU LEUVEN

Ghent Technology Campus
2019-2020

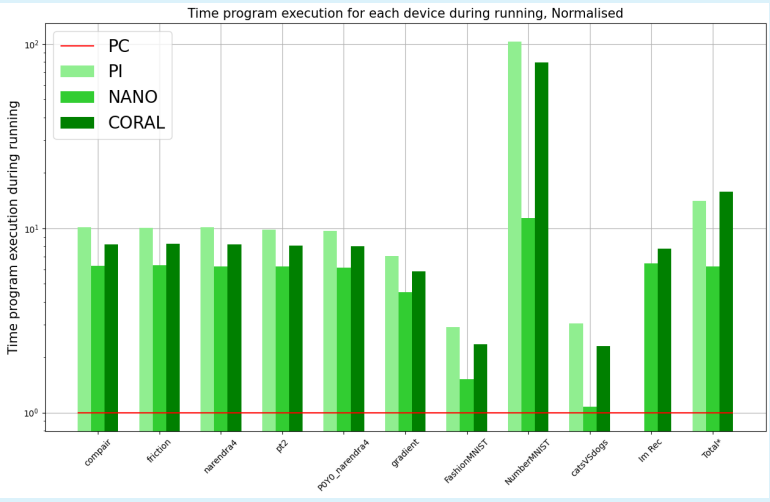
Comparative Study of Single Board Computers for AI Edge Computing Purposes

Introduction

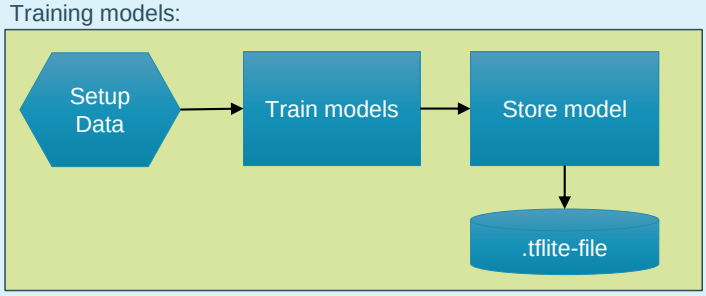
The deployment of self-driving vehicles is one of many new and interesting low-latency applications. To aid to lower latency during localization in position fixing, a benchmark is proposed to examine the potential of executing a trained Neural Networks (NN) or Machine Learning (ML) algorithm on edge devices like the Nvidia Jetson Nano, Google Coral Dev and Raspberry Pi 3 compared to regular computer.

Data acquisition

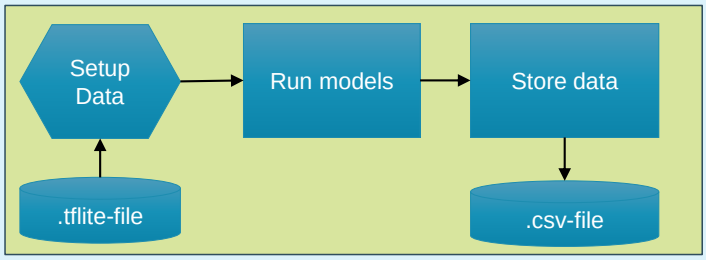
In order to achieve a representative benchmark, it is necessary to investigate the performance of different programs on each edge device in an identical manner. In this benchmark several different programs are tested spread across categories of NN as regression and classification. For each program the duration of execution and utilization of the processor unit is measured and stored. To achieve statistically results, every program will be run for 20 different iterations.



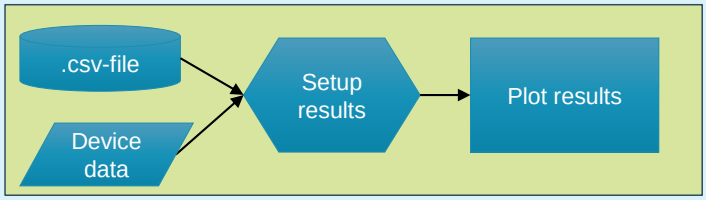
Training models:



Running models:



Plotting results:



Data analysis

To compare the performance of different edge devices, latency data will be adjusted for variables like processor usage, clock speed, price and energy usage. The results are normalised to ease a comparison.

Conclusion

The results show that among edge-devices, the Jetson Nano is the Single Board Computer with the lowest latency. When comparing power consumption for the given latency, the Coral Dev is the most power efficient.

Student: Arno Plaetinck
Promotor: Prof. Dr. Ir. Nobby Stevens
Co-promotors: Ing. Willem Raes
Ing. Jorik De Bruycker

KU Leuven
Ghent Technology Campus
Gebroeders de Smetstraat 1
9000 Ghent

Master in Electronics and ICT Engineering
Technology option Embedded Systems
Academic year: 2019-2020
Date: 6/04/2020

Bijlage B

Benchmark code

B.1 Runnen benchmark

```
from numpy import genfromtxt
import pyrenn as prn
import csv
import time
from statistics import mean
import psutil
from PIL import Image
import tensorflow as tf
import numpy as np
import gzip

import nvidia_smi

nvidia_smi.nvmlInit()
handle = nvidia_smi.nvmlDeviceGetHandleByIndex(0)
# card id 0 hardcoded here, there is also a call to get all available card ids, so we could iterate
res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

sess = tf.compat.v1.Session(config=tf.compat.v1.ConfigProto(log_device_placement=True))

cores = []
cpu_percent = []
virtual_mem = []
time_start = []
time_stop = []

time_total = 0
iterations = 2
labels = ["compair", "friction", "narendra4", "pt2", "P0Y0_narendra4", "P0Y0_compair", "gradient",
"FashionMNIST", "NumberMNIST", "catsVSdogs", "Im Rec", "Totaal"]

###
# Creating a filename
seconds = time.time()
local_time = time.ctime(seconds)
```

```
naam2 = local_time.split()
naam = "Benchmark_PC"
for i in range(len(naam2)):
naam += "_" + naam2[i]
naam = naam.replace(':', '_')

with open('./logging/' + naam + ".csv", mode='w') as results_file:
fieldnames = ['Naam', 'CPU Percentage', 'timediff']
file_writer = csv.DictWriter(results_file, fieldnames=fieldnames)
file_writer.writeheader()


def logging_data(program_index, stop, start, cpu):
# Logging data
cores_avg = mean(cpu)
time_diff = stop-start

with open('./logging/' + naam + ".csv", mode='a+') as data_file:
data_writer = csv.DictWriter(data_file, fieldnames=fieldnames)
data_writer.writerow(
{'Naam': labels[program_index],
    'CPU Percentage': str(cores_avg),
    'timediff': str(time_diff)
})


# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# first time calling cpu percent to get rid of 0,0
psutil.cpu_percent(interval=None, percpu=True)

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# example_compair.py
# This is an example of a dynamic system with 2 outputs and 3 inputs
print("compair")
iteration = 0
while iteration < iterations:
# Read Example Data
df = genfromtxt('example_data_compressed_air.csv', delimiter=',')
df = df.transpose()

P = np.array([df[1][1:-1], df[2][1:-1], df[3][1:-1]])
Y = np.array([df[4][1:-1], df[5][1:-1]])
Ptest = np.array([df[6][1:-1], df[7][1:-1], df[8][1:-1]])
Ytest = np.array([df[9][1:-1], df[10][1:-1]])

# Load saved NN from file
net = prn.loadNN("./SavedNN/compair.csv")

psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()

# Calculate outputs of the trained NN for train and test data
y = prn.NNOut(P, net)
ytest = prn.NNOut(Ptest, net)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')
```

```

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(0, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# example_friction.py
print("friction")
iteration = 0
while iteration < iterations:    # Read Example Data
df = genfromtxt('example_data_friction.csv', delimiter=',')
df = df.transpose()

P = df[1][1:-1]
Y = df[2][1:-1]
Ptest = df[3][1:-1]
Ytest = df[4][1:-1]

# Load saved NN from file
net = prn.loadNN("./SavedNN/friction.csv")

psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()

# Calculate outputs of the trained NN for train and test data
y = prn.NNOut(P, net)
ytest = prn.NNOut(Ptest, net)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(1, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# example_narendra4.py
print("narendra4")
iteration = 0
while iteration < iterations:
# Read Example Data
df = genfromtxt('example_data_narendra4.csv', delimiter=',')
df = df.transpose()

P = df[1][1:-1]
Y = df[2][1:-1]
Ptest = df[3][1:-1]
Ytest = df[4][1:-1]

```

```
# Load saved NN from file
net = prn.loadNN("./SavedNN/narendra4.csv")

psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()

# Calculate outputs of the trained NN for train and test data
y = prn.NNOut(P, net)
ytest = prn.NNOut(Ptest, net)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(2, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# example_pt2.py
print("pt2")
iteration = 0
while iteration < iterations:
# Read Example Data
df = genfromtxt('example_data_friction.csv', delimiter=',')
df = df.transpose()

P = df[1][1:-1]
Y = df[2][1:-1]
Ptest = df[3][1:-1]
Ytest = df[4][1:-1]

# Load saved NN from file
net = prn.loadNN("./SavedNN/pt2.csv")

psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()

# Calculate outputs of the trained NN for train and test data
y = prn.NNOut(P, net)
ytest = prn.NNOut(Ptest, net)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(3, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# example_using_P0Y0_narendra4.py
```

```
print("using_P0Y0_narendra4")
iteration = 0
while iteration < iterations:
# Read Example Data
df = genfromtxt('example_data_narendra4.csv', delimiter=',')
df = df.transpose()

P = df[1][1:-1]
Y = df[2][1:-1]
Ptest = df[3][1:-1]
Ytest = df[4][1:-1]

# define the first 3 timesteps t=[0,1,2] of Test Data as previous (known) data P0test and
    Y0test
P0test = Ptest[0:3]
Y0test = Ytest[0:3]
# Use the timesteps t = [3..99] as Test Data
Ptest = Ptest[3:100]
Ytest = Ytest[3:100]

# Load saved NN from file
net = prn.loadNN("./SavedNN/using_P0Y0_narendra4.csv")

psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()

# Calculate outputs of the trained NN for test data with and without previous input P0 and
    output Y0
ytest = prn.NNOut(Ptest, net)
y0test = prn.NNOut(Ptest, net, P0=P0test, Y0=Y0test)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(4, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# example_gradient.py
print("gradient")

iteration = 0
while iteration < iterations:
df = genfromtxt('example_data_pt2.csv', delimiter=',')
df = df.transpose()

P = df[1][1:-1]
Y = df[2][1:-1]

# Load saved NN from file
net = prn.loadNN("./SavedNN/gradient.csv")

psutil.cpu_percent(interval=None, percpu=True)
```

```
time_start = time.time()

# Prepare input Data for gradient calculation
data, net = prn.prepare_data(P, Y, net)

# Real Time Recurrent Learning
J, E, e = prn.RTRL(net, data)
g_rtrl = 2 * np.dot(J.transpose(), e)  # calculate g from Jacobian and error vector

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(6, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# FashionMNISTREAD.py
print("Fashion MNISTREAD")

f = gzip.open('./datasets/fashionMNIST/t10k-images-idx3-ubyte.gz', 'r')
image_size = 28
num_images = 10_000
f.read(16)
buf = f.read(image_size * image_size * num_images)
data = np.frombuffer(buf, dtype=np.uint8).astype(np.float32)
data = data.reshape(num_images, image_size, image_size)
f.close()

f = gzip.open('./datasets/fashionMNIST/t10k-labels-idx1-ubyte.gz', 'r')
f.read(8)
test_labels = []
for i in range(0, 10_000):
buf = f.read(1)
inhoud = np.frombuffer(buf, dtype=np.uint8).astype(np.int64)
test_labels.append(inhoud)
f.close()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'coat',
'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
path_model = "./SavedNN/FashionMNIST/fashionMNISTmodel.tflite"

# Data Preprocessing
test_images = data / 255

# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path=path_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

```

```
# Get input and output shapes
floating_model = input_details[0]['dtype'] == np.float32
height = input_details[0]['shape'][1]
width = input_details[0]['shape'][2]

# Test model on  input data.
input_shape = input_details[0]['shape']

# Preprocessing data 2
index = 0
input_data_array = []

for image in test_images:
input_data = np.expand_dims(test_images[index], axis=0)
index += 1
if floating_model:
input_data = np.float32(input_data)
input_data_array.append(input_data)

iteration = 0
while iteration < iterations:
output_data_array = []

psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()

for index in range(10000):
interpreter.set_tensor(input_details[0]['index'], input_data_array[index])
interpreter.invoke()
output_data = interpreter.get_tensor(output_details[0]['index'])
output_data_array.append(output_data)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(7, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# NumberMNISTREAD.py
print("NumberMNISTREAD")

f = gzip.open('./datasets/numberMNIST/t10k-images-idx3-ubyte.gz', 'r')
image_size = 28
num_images = 10_000
f.read(16)
buf = f.read(image_size * image_size * num_images)
data = np.frombuffer(buf, dtype=np.uint8).astype(np.float32)
data = data.reshape(num_images, image_size, image_size)
f.close()

f = gzip.open('./datasets/numberMNIST/t10k-labels-idx1-ubyte.gz', 'r')
```

```
f.read(8)
test_labels = []
for i in range(0, 10_000):
buf = f.read(1)
label = np.frombuffer(buf, dtype=np.uint8).astype(np.int64)
test_labels.append(label)
f.close()

class_names = ["zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine
    "]
path_model = "./SavedNN/NumberMNIST/mnist_model.tflite"

# Data Preprocessing
test_images = data / 255

# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path=path_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Get input and output shapes
floating_model = input_details[0]['dtype'] == np.float32
height = input_details[0]['shape'][1]
width = input_details[0]['shape'][2]

# Test model on  input data.
input_shape = input_details[0]['shape']

# Preprocessing data 2
index = 0
input_data_array = []
for image in test_images:
input_data = np.expand_dims(test_images[index], axis=0)
index += 1
if floating_model:
input_data = np.float32(input_data)
input_data_array.append(input_data)

iteration = 0
while iteration < iterations:
output_data_array = []

psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()

for index in range(10_000):
interpreter.set_tensor(input_details[0]['index'], input_data_array[index])
interpreter.invoke()
output_data = interpreter.get_tensor(output_details[0]['index'])
output_data_array.append(output_data)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = time.time()
```

```
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(8, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
     time_stop-time_start)

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# catsVSdogs10Read.py
# 124 images from COCO val2017 dataset
print("catsVSdogs10Read")

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
'dog', 'frog', 'horse', 'ship', 'truck']
path_image = "./images/catsVSdogs/"
extention = ".jpg"
path_model = "./SavedNN/catsVSdogs/"
labels_array = np.arange(0, 39)
# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path=path_model + "catsVSdogsmodel.tflite")
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Test model on  input data.
input_shape = input_details[0]['shape']
floating_model = input_details[0]['dtype'] == np.float32
height = input_details[0]['shape'][1]
width = input_details[0]['shape'][2]
print(height, width)
input_data2 = []

for label in labels_array:
path = path_image + "/Cat/" + str(label) + extention
img = Image.open(path).resize((width, height))

input_data = np.expand_dims(img, axis=0)
if floating_model:
input_data = (np.float32(input_data)) / 255
input_data2.append(input_data)
for label in labels_array:
path = path_image + "/Dog/" + str(label) + extention
img = Image.open(path).resize((width, height))
# img.show()
img.save("./images/test/" + str(label) + ".png")
input_data = np.expand_dims(img, axis=0)

if floating_model:
input_data = (np.float32(input_data)) / 255
input_data2.append(input_data)


iteration = 0
while iteration < iterations:
psutil.cpu_percent(interval=None, percpu=True)
```

```
time_start = time.time()

for data in input_data2:
interpreter.set_tensor(input_details[0]['index'], data)

interpreter.invoke()
output_data = interpreter.get_tensor(output_details[0]['index'])
npa = np.asarray(output_data[0], dtype=np.float32)
# print(class_names[np.argmax(npa)])

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(9, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# ImRecKerasRead.py
# 124 images from COCO val2017 dataset
print("ImRecKerasRead")

path_image = "./images/KerasRead/"
extention = ".jpg"

imagenet_labels = np.array(open(path_image+"ImageNetLabels.txt").read().splitlines())

# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path="./SavedNN/ImRecKerasModel/ImRecKerasModel.
    tflite")
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

floating_model = input_details[0]['dtype'] == np.float32
height = input_details[0]['shape'][1]
width = input_details[0]['shape'][2]
print("height: ", height, " width", width)

input_data2 = []
with open("./images/KerasRead/labels.txt", mode='r') as label_file:
for label in label_file:
path = path_image + label.rstrip() + extention
img = Image.open(path).resize((width, height))
input_data = np.expand_dims(img, axis=0)
if floating_model:
input_data = (np.float32(input_data) - 127.5) / 127.5
input_data2.append(input_data)

iteration = 0
while iteration < iterations:
```

```
psutil.cpu_percent(interval=None, percpu=True)
time_start = time.time()

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

for data in input_data2:
interpreter.set_tensor(input_details[0]['index'], data)

interpreter.invoke()
res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')
output_data = interpreter.get_tensor(output_details[0]['index'])

decoded = imagenet_labels[np.argsort(output_data)[0, ::-1][:5] + 1]
print("Result AFTER saving: ", decoded)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(handle)
print(f'gpu: {res.gpu}%, gpu-mem: {res.memory}%')

time_stop = (time.time())
cores = psutil.cpu_percent(interval=None, percpu=True)
if (mean(cores) != 0.0) and (time_stop-time_start != 0):
logging_data(10, time_stop, time_start, cores)
iteration += 1
time_total += time_stop - time_start
print("iteration: ", iteration, " mean cores: ", mean(cores), " time_stop-time_start: ",
    time_stop-time_start)

cores = psutil.cpu_percent(interval=2, percpu=True)
with open('./logging/' + naam + ".csv", mode='a+') as data_file:
data_writer = csv.DictWriter(data_file, fieldnames=fieldnames)

data_writer.writerow(
{'Naam': labels[-1],
    'CPU Percentage': str(mean(cores)),
    'timediff': str(time_total/iterations)
})
```

B.2 Plotten resultaten

```
import csv
from statistics import mean
import matplotlib.pyplot as plt
import numpy as np
from tabulate import tabulate

iterations = 20
boxplot_bool = False

name_PC = "Benchmark_PC_Thu_Apr_2_22_05_01_2020"
name_PI = "Benchmark_PI_Fri_Apr_3_02_32_27_2020"
name_NANO = "Benchmark_NANO_Fri_Apr_3_02_11_17_2020"
name_CORAL = "Benchmark_CORAL_Wed_Apr_3_20_08_16_2020"

file_names = [name_PC, name_PI, name_NANO, name_CORAL]

data_CPU = [[], [], [], []]
data_time = [[], [], [], []]
data_CPU_avg = [[], [], [], []]
data_time_avg = [[], [], [], []]

cpu_percent = []
virtual_mem = []
time_diff = []

programs = ["compair", "friction", "narendra4", "pt2", "P0Y0_narendra4",
            "gradient", "FashionMNIST", "NumberMNIST", "catsVSdogs", "Im Rec"]
labels_cpu = programs + ["no operations"]
labels_time = programs + ["Total*"]

ylabel_time = 'Time program execution during running'
ylabel_cpu = 'CPU usage during running'
title_time = 'Time program execution for each device during running'
title_cpu = 'CPU usage for each device during running'

devices = ["PC", "PI", "NANO", "CORAL"]
clockspeed = [3.25, 1.2, 1.479, 1.5]
device_price = [981, 41.5, 99, 149.99]
power = [79.9, 3.7, 5, 2.65]

width = 0.22
image_path = "./images/figures/"


def show_plot(data, ylabel, titel, labels, log, show, index, boxplot, normalise):
    if not show:
        return

    def autolabel(rects):
        """Attach a text label above each bar in *rects*, displaying its height."""
        for rect in rects:
            height = rect.get_height()
            ax.annotate('{}'.format(height),
                        xy=(rect.get_x() + rect.get_width() / 2, height),
                        xytext=(0, 3),  # 3 points vertical offset
```

```
                        textcoords="offset points",
                        ha='center', va='bottom')

    data_bar = [[], [], [], []]
    for device in range(len(devices)):
        for program in range(len(labels)):
            data_bar[device].append(float(round(mean(data[device][program]), 3)))
            for iteration in range(iterations):
                data[device][program][iteration] = round(data[device][program][iteration],
    5)
    fig, ax = plt.subplots()

    # the label locations
    x = np.arange(len(labels))

    # variables to be used for broken PC normalised line
    xmin = x - 3 * width / 2
    xmax = x + 3 * width / 2
    y = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

    if not normalise:
        rects1 = ax.bar(x - 3 * width / 2, data_bar[1], width, label='PI', color='lightgreen
    ')
        rects2 = ax.bar(x - width / 2, data_bar[2], width, label='NANO', color='limegreen')
        rects3 = ax.bar(x + width / 2, data_bar[3], width, label='CORAL', color='green')
        rects4 = ax.bar(x + 3 * width / 2, data_bar[0], width, label='PC', color='lightblue'
    )
        autolabel(rects1)
        autolabel(rects2)
        autolabel(rects3)
        autolabel(rects4)
        if boxplot:
            ax.boxplot(data[1], positions=x - 3 * width / 2, widths=width, showfliers=True)
            ax.boxplot(data[2], positions=x - width / 2, widths=width, showfliers=True)
            ax.boxplot(data[3], positions=x + width / 2, widths=width, showfliers=True)
            ax.boxplot(data[0], positions=x + 3 * width / 2, widths=width, showfliers=True)
    elif normalise:
        rects2 = ax.bar(x - width, data_bar[1], width, label='PI', color='lightgreen')
        rects3 = ax.bar(x, data_bar[2], width, label='NANO', color='limegreen')
        rects4 = ax.bar(x + width, data_bar[3], width, label='CORAL', color='green')

        autolabel(rects2)
        autolabel(rects3)
        autolabel(rects4)

        ax.hlines(y=1,
        xmin=xmin[0],
        xmax=xmax[-1],
        colors='r', linestyles='solid', label='PC')
        if boxplot:
            ax.boxplot(data[1], positions=x - width, widths=width)
            ax.boxplot(data[2], positions=x, widths=width)
            ax.boxplot(data[3], positions=x + width, widths=width)

    # Add some text for labels, title and custom x-axis tick labels, etc.
    ax.set_ylabel(ylabel, fontsize=15)
    ax.set_title(titel, fontsize=15)
    ax.set_xticks(x)
```

```python
    ax.set_xticklabels(labels)
    plt.xticks(rotation=45)
    ax.legend(prop={'size': 20})

    fig.tight_layout()
    plt.grid()
    ax.set_axisbelow(True)
    mng = plt.get_current_fig_manager()
    mng.window.state('zoomed')
    if log:
        plt.yscale("log")
    plt.savefig(image_path + "Figure_{}".format(index))
    plt.show()


def tabel(data):
    table = [["PC"], ["PI"], ["NANO"], ["CORAL"]]

    for program in range(len(programs)):
        for device in range(len(devices)):
            table[device].append(round(mean(data[device][program]), 3))
    print()
    print(tabulate(table, headers=["Device", "compair", "friction", "narendra4", "pt2", "P0Y0_narendra4",
    "P0Y0_compair", "gradient", "FashionMNIST", "NumberMNIST", "catsVSdogs", "Im Rec"]))
    print()


# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Data extraction
for device in range(len(file_names)):
    temp1, temp2, = [], []

    with open('./logging/' + file_names[device] + ".csv", mode='r') as results_file:
        results_reader = csv.DictReader(results_file)
        for row in results_reader:
            temp2.append(float(row['timediff']))
            temp1.append(float(row['CPU Percentage']) / 100)

    for program in range(len(programs)):
        time_diff_avg = round(mean(temp2[program * iterations:(program * iterations + iterations)]), 5)
        cpu_avg = round(mean(temp1[program * iterations:(program * iterations + iterations)]), 5)
        data_CPU_avg[device].append(cpu_avg)
        data_time_avg[device].append(time_diff_avg)

        data_CPU[device].append(temp1[program * iterations:(program * iterations + iterations)])
        data_time[device].append(temp2[program * iterations:(program * iterations + iterations)])
    data_CPU[device].append([])
    data_time[device].append([])

    for iteration in range(iterations):
        data_CPU[device][-1].append(temp1[-1])
        data_time[device][-1].append(temp2[-1])
    data_CPU_avg[device].append(temp1[-1])
```

```
164     data_time_avg[device].append(temp2[-1])
165
166 # Adding new total value to PI
167 total = 0
168 for program in range(len(programs)-1):
169     total += mean(data_time[1][program])
170 data_time[1][-1] = []
171 for iteration in range(iterations):
172     data_time[1][-1].append(total)
173
174 # Plotting figures
175 show_plot(data_time, ylabel_time, title_time, labels_time, log=True, show=True, index=0,
        boxplot=boxplot_bool, normalise=False)
176 show_plot(data_CPU, ylabel_cpu, title_cpu, labels_cpu, log=False, show=True, index=1,
        boxplot=boxplot_bool, normalise=False)
177
178 # making sure variables have right shape, content of data_time will be ignored
179 data_time_norm = []
180 data_energy = []
181 data_energy_norm = []
182 data_time_MHzCPU = []
183 data_time_MHzCPU_norm = []
184 data_time_MHzCPUprice = []
185 data_time_MHzCPUprice_norm = []
186
187 for device in range(len(devices)):
188     data_time_norm.append([])
189     data_energy.append([])
190     data_energy_norm.append([])
191     data_time_MHzCPU.append([])
192     data_time_MHzCPU_norm.append([])
193     data_time_MHzCPUprice.append([])
194     data_time_MHzCPUprice_norm.append([])
195
196     for program in range(len(labels_time)):
197         data_time_norm[device].append([])
198         data_energy[device].append([])
199         data_energy_norm[device].append([])
200         data_time_MHzCPU[device].append([])
201         data_time_MHzCPU_norm[device].append([])
202         data_time_MHzCPUprice[device].append([])
203         data_time_MHzCPUprice_norm[device].append([])
204
205         for iteration in range(iterations):
206             data_time_norm[device][program].append([])
207             data_energy[device][program].append([])
208             data_energy_norm[device][program].append([])
209             data_time_MHzCPU[device][program].append([])
210             data_time_MHzCPU_norm[device][program].append([])
211             data_time_MHzCPUprice[device][program].append([])
212             data_time_MHzCPUprice_norm[device][program].append([])
213
214 # Rescaling to lowest nr of each program
215 pc_values = []
216 for program in range(len(labels_time)):
217     pc_values.append(mean(data_time[0][program]))
218     for device in range(len(devices)):
219         for iteration in range(iterations):
```

```
220             data_time_norm[device][program][iteration] = \
221                 data_time[device][program][iteration] / pc_values[program]
222
223 show_plot(data=data_time_norm,
224             ylabel=ylabel_time,
225             titel=title_time + ", Normalised",
226             labels=labels_time,
227             log=True, show=True, index=2,
228             boxplot=boxplot_bool, normalise=True)
229
230 # plotting time/MHz/cpu%
231 for device in range(len(devices)):
232     for program in range(len(labels_time)):
233         for iteration in range(iterations):
234             data_time_MHzCPU[device][program][iteration] = \
235                 data_time[device][program][iteration] / clockspeed[device] / data_CPU_avg[
    device][program]
236
237 show_plot(data=data_time_MHzCPU,
238             ylabel="Time / CPU% / MHz.",
239             titel="Time / CPU% / MHz for each device.",
240             labels=labels_time,
241             log=True, show=False, index=3,
242             boxplot=boxplot_bool, normalise=False)
243
244 # plotting normalised time/MHz/cpu%
245 pc_values = []
246 for label in range(len(labels_time)):
247     pc_values.append(mean(data_time_MHzCPU[0][label]))
248
249 for device in range(len(devices)):
250     for program in range(len(labels_time)):
251         for iteration in range(iterations):
252             data_time_MHzCPU_norm[device][program][iteration] = data_time_MHzCPU[device][
    program][iteration] /      pc_values[program]
253
254 show_plot(data_time_MHzCPU_norm,
255             ylabel="Time / CPU% / MHz.",
256             titel="Time / CPU% / MHz for each device, Normalised.",
257             labels=labels_time,
258             log=True, show=False, index=4,
259             boxplot=boxplot_bool, normalise=True)
260
261 # plotting time*watt
262 for device in range(len(devices)):
263     for program in range(len(labels_time)):
264         for iteration in range(iterations):
265             data_energy[device][program][iteration] = \
266                 power[device] * data_time[device][program][iteration]
267
268 show_plot(data=data_energy,
269             ylabel="Time * Watt.",
270             titel="Time * Watt for each device.",
271             labels=labels_time,
272             log=True, show=False, index=5,
273             boxplot=boxplot_bool, normalise=False)
274
275 # plotting normalised time/watt
```

```
pc_values = []
for label in range(len(labels_time)):
    pc_values.append(mean(data_energy[0][label]))

for device in range(len(devices)):
    for program in range(len(labels_time)):
        for iteration in range(iterations):
            data_energy_norm[device][program][iteration] = data_energy[device][program][
    iteration] / pc_values[program]

show_plot(data_energy_norm,
            ylabel="Time * Watt.",
            titel="Time * Watt, Normalised.",
            labels=labels_time,
            log=True, show=False, index=6,
            boxplot=boxplot_bool, normalise=True)

# plotting time/$
for device in range(len(devices)):
    for program in range(len(labels_time)):
        for iteration in range(iterations):
            data_time_MHzCPUprice[device][program][iteration] = device_price[device] /
    data_time[device][program][iteration]

show_plot(data=data_time_MHzCPUprice,
            ylabel="Time * dollar.",
            titel="Time * dollar for each device.",
            labels=labels_time,
            log=True, show=True, index=7,
            boxplot=boxplot_bool, normalise=False)

# plotting normalised time/MHz/cpu%/$
pc_values = []
for label in range(len(labels_time)):
    pc_values.append(mean(data_time_MHzCPUprice[0][label]))

for device in range(len(devices)):
    for program in range(len(labels_time)):
        for iteration in range(iterations):
            data_time_MHzCPUprice_norm[device][program][iteration] = data_time_MHzCPUprice[
    device][program][iteration] / pc_values[program]

show_plot(data=data_time_MHzCPUprice_norm,
            ylabel="Time * dollar.",
            titel="Time * dollar for each device, Normalised.",
            labels=labels_time,
            log=True, show=True, index=8,
            boxplot=boxplot_bool, normalise=True)

tabel(data_time)
tabel(data_time_norm)
tabel(data_time_MHzCPU)
tabel(data_time_MHzCPU_norm)
tabel(data_time_MHzCPUprice)
tabel(data_time_MHzCPUprice_norm)
```

Bijlage C

Poster

KU LEUVEN

Ghent Technology Campus
2019-2020

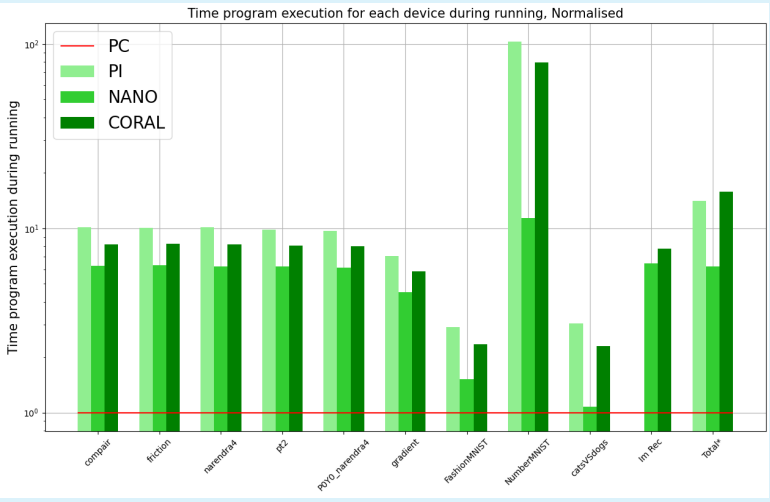
Comparative Study of Single Board Computers for AI Edge Computing Purposes

Introduction

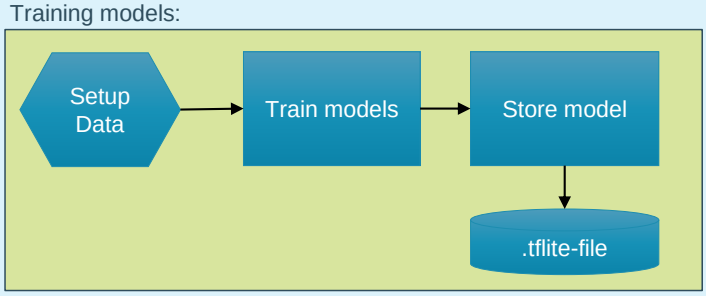
The deployment of self-driving vehicles is one of many new and interesting low-latency applications. To aid to lower latency during localization in position fixing, a benchmark is proposed to examine the potential of executing a trained Neural Networks (NN) or Machine Learning (ML) algorithm on edge devices like the Nvidia Jetson Nano, Google Coral Dev and Raspberry Pi 3 compared to regular computer.

Data acquisition

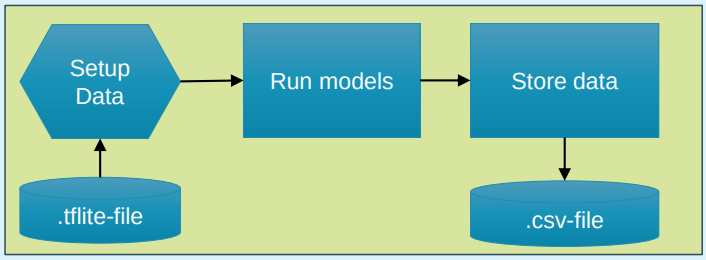
In order to achieve a representative benchmark, it is necessary to investigate the performance of different programs on each edge device in an identical manner. In this benchmark several different programs are tested spread across categories of NN as regression and classification. For each program the duration of execution and utilization of the processor unit is measured and stored. To achieve statistically results, every program will be run for 20 different iterations.



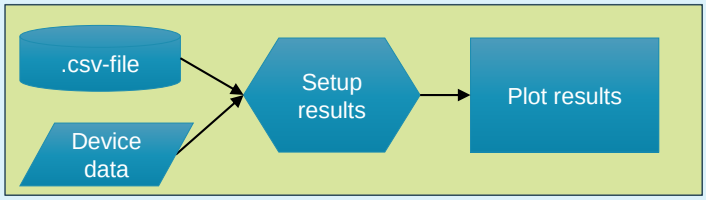
Training models:



Running models:



Plotting results:



Data analysis

To compare the performance of different edge devices, latency data will be adjusted for variables like processor usage, clock speed, price and energy usage. The results are normalised to ease a comparison.

Conclusion

The results show that among edge-devices, the Jetson Nano is the Single Board Computer with the lowest latency. When comparing power consumption for the given latency, the Coral Dev is the most power efficient.

Student: Arno Plaetinck
Promotor: Prof. Dr. Ir. Nobby Stevens
Co-promotors: Ing. Willem Raes
Ing. Jorik De Bruycker

KU Leuven
Ghent Technology Campus
Gebroeders de Smetstraat 1
9000 Ghent

Master in Electronics and ICT Engineering
Technology option Embedded Systems
Academic year: 2019-2020
Date: 6/04/2020

FACULTEIT INDUSTRIELE INGENIEURSWETENSCHAPPEN
TECHNOLOGIECAMPUS GENT
Gebroeders De Smetstraat 1
9000 GENT, België
tel. + 32 92 65 86 10
fax + 32 92 25 62 69
iiw.gent@kuleuven.be
www.iiw.kuleuven.be