HO
GENT

Faculteit Bedrijf en Organisatie

Deploying Parallel and Distributed Deep Learning methods for Luau

Rune van der Lei

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

Promotor:
Guy Dekoning
Co-promotor:
Robin Jooris

Instelling: —

Academiejaar: 2020-2021

Derde examenperiode

Faculteit Bedrijf en Organisatie

Deploying Parallel and Distributed Deep Learning methods for Luau

Rune van der Lei

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

Promotor:
Guy Dekoning
Co-promotor:
Robin Jooris

Instelling: —

Academiejaar: 2020-2021

Derde examenperiode

Woord vooraf

Tijdens het programmeren op Roblox zag ik een nieuwe update voor een bèta versie over parallellisatie van scripts. Hierdoor ontstond de opportuniteit om iets uit te werken dat nog niet bestond op Roblox. Neurale netwerken was al een hot topic op de developer forums gevolgd door veel users die erg graag deze technologie in hun spel willen gebruiken. Maar er zijn veel beperkingen in de game engine die dit moeilijk maken. Door deze nieuwe update konden veel van de beperkingen opgelost worden maar ik wist nog niet precies hoe. Het is plezant om problemen te overkomen en ideeën voor te stellen. In deze studie introduceer ik mijn methodes en conclusies.

Ik wil mijn promotor Guy Dekoning en co-promotor Robin Jooris bedanken voor hun geduld en kritische feedback op deze bachelorproef, evenals mijn ouders voor de motivatie om verder te gaan.

Samenvatting

Deep learning maakt veel applicaties mogelijk die vroeger onmogelijk waren. Om deze nieuwe technologieën te kunnen ondersteunen is een Distributed Deep Learning Systeem (DDLS) nodig. Deze systemen zijn erg afgestemd op de omgeving waarop ze worden uitgevoerd. Een veel belovende omgeving om deze technologieën toe te passen is Roblox. Roblox developers kunnen deze systemen gebruiken om nieuwe en unieke ervaringen te maken die hiervoor niet mogelijk waren. De juiste methode om een DDLS uit te voeren kan moeilijk te vinden zijn. Traditionele systemen moesten geen zorgen maken over een constant veranderende omgeving. Een speler kan op elk moment de verbinding verbreken, de calcultatie van een algorithme mag niet te veel tijd van een frame opnemen en de verbinding tussen server en speler kan onstabiel of traag zijn. Voor deze redenen hebben wij enerzijds verschillende robuuste parallellisatie methodes beschreven die rekening houden met de limieten van Roblox. Anderzijds wordt ook een overzicht gegeven van de voordelen en uitdagingen verbonden aan elke architectuur. Ook werden voor de verschillende uitdagingen van de implementatie van het systeem eventuele oplossingen opgenomen in dit onderzoek.

Men kan concluderen dat er effectieve methodes bestaan om efficiënt deep learning toe te passen binnen Roblox. Deze methodes maken gebruik van de rekenkracht van de cliënt die geconnecteerd zijn aan de server om grote versnellingen te boeken. Dit kan weinig impact hebben op de speler en de uitvoering van het spel.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

Machine learning, en in het bijzonder deep learning neemt in snel tempo verschillende aspecten over in ons dagelijkse leven. De kern van deep learning ligt het Deep Neural Network (DNN), een constructie geïnspireerd op de onderling verbonden aard van het menselijk brein. Als DNN's goed afgestemd zijn door een grote hoeveelheid gegevens te bestuderen, kunnen ze nauwkeurige oplossingen bieden voor problemen die voorheen als onoplosbaar werden beschouwd. Deep learning is met succes geïmplementeerd voor een groot aantal vakgebieden zoals: beeldclassificatie, spraakherkenning, medische diagnose, autonoom rijden en menselijke spelers verslaan in complexe spellen.

Luau is een scripttaal ontwikkeld door Roblox voor hun game engine en is gebaseerd op lua 5.1. Deze game engine is gebruikt door duizenden professionele game studio's en 42.1 miljoen dagelijkse gebruikers.

Roblox is niet één enkel spel, maar eerder een verzameling van meer dan 50 miljoen games, allemaal gemaakt door de gemeenschap van spelers. De eenvoudigste vergelijking is er met YouTube: een enorme bibliotheek met "door gebruikers gegenereerde inhoud", maar in dit geval bestaat de inhoud uit games in plaats van video's. Deze spellen zijn erg divers in hun concepten en uitvoer. Elke videogame genre bestaat op roblox en zijn meestal gratis om te spelen. Spelers downloaden de gratis Roblox-applicatie voor computers, gameconsoles, smartphones of tablets en gebruiken deze om door de catalogus met games te bladeren en deze zonder verdere installatie te spelen. Roblox werd officieel gelanceerd in 2006 en is sindsdien nog steeds aan het groeien.

Distributed deep learning systems (DDLS) trainen diepe neurale netwerken door gebruik te maken van de gedistribueerde middelen van een groep computers, ook een cluster genoemd. Hier zou de cluster het netwerk van servers en spelers die deel van een Roblox

spel zijn. Deze cluster kan tussen 2 tot 701 computers groot zijn, dit is het limiet opgesteld door de game engine.

Een recente bijwerking van de Roblox game engine heeft de mogelijkheid gegeven om taken parallel uit te voeren op meerdere processen. Hiermee bestaat nu de opportuniteit om deep learning meer efficiënt in Roblox te gebruiken.

1.1 Probleemstelling

Deep learning kan op veel manieren de spellen op Roblox veranderen en verbeteren. De impact van een goed deep learning systeem kan applicaties creëren die hiervoor niet mogelijk of erg moeilijk te implementeren waren. Een voorbeeld hiervan zijn dynamische conversaties met niet-speler karakters, zelfrijdende auto's voor politie achtervolgingen of dynamisch gegenereerde inhoud van texturen en landschappen. De taken die een neuraal netwerk kan doen binnen een video game is nog niet volledig onderzocht. De 7 miljoen developers op Roblox laten werken met deze technologie kan erg veel unieke en verrassende resultaten opleveren.

Het voorbeeld van zelfrijdende auto's voor politie achtervolgingen is al uitgevoerd in Roblox door ScriptOn en ChrisvC, maar dit is gelimiteerd op schaal en complexiteit. VirtualPixl en Kironte zijn developers die een chatbot willen maken maar vinden dat de server alleen niet krachtig genoeg is om dit tot werkelijkheid te brengen. Dit zijn problemen dat parallellisme en distributie van taken kan oplossen.

1.2 Onderzoeksvraag

1.2.1 Hoofdonderzoeksvragen

Dit onderzoek zal zich vooral focussen op parallellisatie en distributie methodes om deep learning netwerken uit te voeren.

- Is een distributed deep learning systeem mogelijk in Roblox zonder een grote performantie impact te hebben op de speler of de server?

Op deze onderzoeksvraag zal dit onderzoek een antwoord geven in de conclusie (zie Hoofdstuk 5).

1.2.2 Deelonderzoeksvragen

Daarnaast zijn er ook nog enkele ondersteunende deelonderzoeksvragen die bij dit onderzoek horen.

- Welke parallellisatie methodes en architecturen zijn mogelijk in Roblox?

- Hoe kunnen deze methodes gecombineerd worden om de uitvoering van neurale netwerken te versnellen?
- Wat zijn de voor- en nadelen van elke methode of architectuur?
- Hoe groot is de performantie impact van deze methodes en architecturen?

Op deze deelonderzoeksvragen zal gaandeweg doorheen dit onderzoek beantwoord worden. Een samenvatting hiervan is te vinden in de conclusie (zie Hoofdstuk 5).

1.3 Onderzoeksdoelstelling

Het hoofddoel van dit onderzoek is het aantonen van de werking en gevolgen van verschillende parallellisatie methodes. Deze methodes zouden een significant positief effect moeten hebben op de uitvoer van deep learning algoritmes in vergelijking met neurale netwerken zonder parallellisme. Dit doen we aan de hand van een proof-of-concept. Dit onderzoek is dan ook geslaagd wanneer dit doel behaald wordt. Een tweede doel bestaat uit het vinden van de meest performante en effectieve parallellisatie methode of architectuur. Dit wordt gedaan met een vergelijkende studie. Een derde doel is het meten van de impact dat deze systemen op de eindgebruiker hebben. Er mag geen opmerkelijk verschil zijn op de spelers als een DDLS op de achtergrond aan het werken is.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

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

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

In Hoofdstuk 4 wordt de concepten geïmplementeerd, we bespreken de testresultaten en problemen dat zijn opgekomen tijdens de implementatie.

In Hoofdstuk 5, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

2.1 Terminologie en algorithmen

2.1.1 Parallelle computerarchitectuur

We vervolgen met een kort overzicht van parallelle hardware-architecturen die worden gebruikt om algoritmes in de praktijk uit te voeren. Ze kunnen ingedeeld worden in systemen met een enkele machine (vaak gedeeld geheugen) en meerdere machines (vaak gedistribueerd geheugen).

Enkele machine parallellisme

CPU's zijn geoptimaliseerd voor algemene workloads, variërend van desktopapplicaties tot datacenter taken (bijvoorbeeld het bedienen van webpagina's en het uitvoeren van complexe zakelijke workflows). Taken voor machinaal leren zijn vaak rekenintensief, waardoor ze vergelijkbaar zijn met traditionele HPC-toepassingen (High Performance Computing). Grote taken presteren dus zeer goed op versnelde systemen zoals Graphics processing units (GPU) of field-programmable gate arrays (FPGA) die nu al meer dan tien jaar in het HPC-veld worden gebruikt. Die apparaten richten zich op rekenkracht door hun architectuur te specialiseren om gebruik te maken van hoge parallellisme in HPC opdrachten.

Meerdere machine parallellisme

Het trainen van grootschalige modellen is een zeer reken intensieve taak. Zo zijn enkele machines vaak niet in staat om deze taak in een redelijk tijdperk uit te voeren. Om de

berekening verder te versnellen, kan deze worden gedistribueerd over meerdere machines die via een netwerk verbonden zijn. De belangrijkste statistieken voor het netwerk zijn latentie (de gemiddelde tijdsduur tussen het versturen en ontvangen van een bericht), bandbreedte (gegevensoverdrachtsnelheid per tijdseenheid) en bericht hoeveelheid (aantal berichten per tijdseenheid).

Er is een nood aan communicatie tussen de machines om de calculaties vlot te laten verlopen. Als de ene machine informatie van een andere machine nodig heeft om verder te rekenen dan is het belangrijk dat:

- Latentie niet te hoog is. Vanaf een bepaald punt kan het efficiënter zijn om de berekeningen op één machine te doen, dan te wachten op het resultaat van een andere machine.
- Bandbreedte kan een effect hebben als de berichten te groot zijn. Als de bandbreedte wordt overschreden kan de latentie veel verhogen.
- De hoeveelheid berichten limiteren is belangrijk, te veel berichten kan al snel de bandbreedte beperking bereiken.

Door de limieten van een netwerk omgeving zal de architectuur van een DDLS aangepast worden. Hoe deze verandert en de technieken die we kunnen gebruiken om netwerk problemen te omzeilen zullen later duidelijk worden.

Het netwerk dat verkrijgbaar is in de Roblox omgeving is erg onzeker. We moeten rekening houden dat op elk moment een speler zijn spel kan sluiten of de connectie naar de server kan verliezen. De bandbreedte kan verschillen tussen machines. Niets hier is zeker en onze architectuur zal hier rekening mee moeten houden. Sinds dat Roblox op veel soorten apparaten werkt (Apple, Android, console, pc) kunnen we niet zeker zijn over de specificaties van de apparaten. We weten vooraf niet hoe snel een apparaat werkt omdat elke machine andere onderdelen heeft.

2.1.2 Parallel programmeren

Programmeertechnieken om parallelle algoritmen te implementeren, zijn afhankelijk van de doelarchitectuur. Ze variëren van eenvoudige implementaties met threads tot gedeeld geheugen van meerdere processen. Parallelle programma's zijn meestal geprogrammeerd met speciale talen zoals NVIDIA's CUDA, OpenCL of FPGA's die hardware ontwerpen gebruiken om efficiënt bewerkingen uit te voeren. Meestal zijn de details vaak verborgen achter libraries (bijv. cuDNN of MKL-DNN) die de tijdrovende primitieven implementeren. Roblox heeft deze talen en libraries niet, in de plaats hebben ze methodes om threads en processen aan te maken. Nieuwe processen aanmaken kon tot recent nog niet met Roblox en is nog een studio bèta toepassing. De publieke release is gepland eind 2021.(EthicalRobot, 2021)

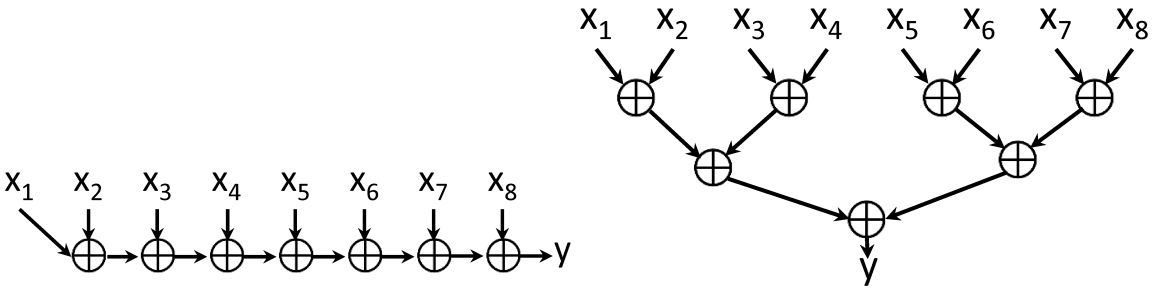
Om ons script parallel aan een ander script te laten lopen, moeten er twee voorwaarden gelden:

- De scripts moeten onder verschillende *Actor*-instanties worden geplaatst
- De thread in een script moet worden gedesynchroniseerd via *task.desynchronize()* of *Signal::ConnectParallel*

Actors zijn een nieuw instance-type waarmee je het spel in logische stukken kunt opdelen. Deze stukken markeren delen van de datamodelboom als onafhankelijk. De *Actor* is de eigenaar van zijn thread. Alles in de *Actor* kan worden uitgevoerd op de hoofd thread of op zijn eigen thread. Alles buiten de *Actor* kan niet op zijn thread uitgevoerd worden.

Elk script wordt standaard in serie uitgevoerd, maar scripts die binnen *Actors* worden uitgevoerd, kunnen overschakelen naar parallel lopen met de functie *task.desynchronize()*. Deze functie is yieldable - het pauzeert de uitvoering van de huidige thread en hervat deze bij de volgende parallelle uitvoeringsmogelijkheid.

Het is belangrijk om te begrijpen dat regio's van parallelle uitvoering (de scripts onder verschillende *Actors*) parallel lopen, maar de game engine wacht tot alle parallelle secties zijn voltooid voordat ze verder gaan met seriële uitvoering. Met andere woorden, om van deze functie te profiteren, kun je geen erg lange berekening uitvoeren die seconden duurt, parallel aan de rest van de simulatie - je moet het in kleine stukjes opbreken, maar je kunt deze stukjes op meerdere cores uitvoeren. Je mentale model zou moeten zijn 'laat me calculaties uitvoeren voor een duizend kleine dingen die mogelijk op een afzonderlijke cores draaien' in plaats van 'laat me deze langzame functie uitvoeren die opeenvolgend alle calculaties uitvoert parallel aan de verwerking van de rest van de wereld'.



(a) liniaire diepte som (b) boom som

Figuur 2.1: Som schema's (Ben-Nun & Hoefler, 2018)

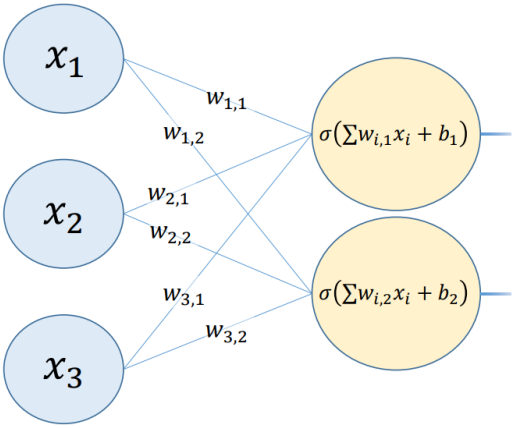
2.1.3 Parallelle algoritmen

We bespreken nu kort concepten in parallel computing die nodig zijn om parallel machine learning te begrijpen.

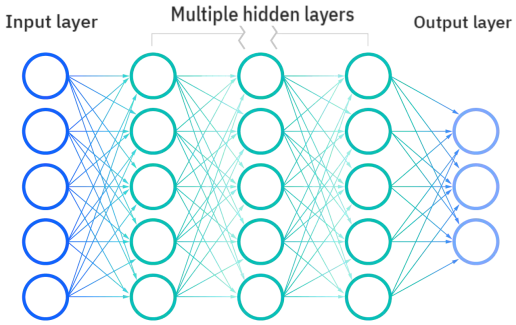
Bij een reductie (som) passen we een reeks operatoren ($\bigoplus$) toe om n waarden te combineren tot een enkele waarde, bijvoorbeeld $y = x1 \bigoplus x2 \bigoplus x3 \cdots \bigoplus xn$. We kunnen de toepassing ervan wijzigen, waardoor het verandert van een lineaire diepte calculatie zoals weergegeven in Fig.2.1a naar een logaritmische diepte calculatie zoals getoond in Fig.2.1b. In het voorbeeld van Fig.2.1a moeten we wachten totdat de $\bigoplus$ operatie 7 keer is uitgevoerd (Diepte 7). Maar voor dezelfde calculatie in Fig.2.1b moeten we wachten

voor de tijd dat een $\bigoplus$ operatie 3 keer is uitgevoerd (Diepte 3). Dit is de manier waarmee we veel van de bewerkingen van een neuraal netwerk kunnen parallelliseren en verdelen onder machines en daarmee ook de snelheid van de totale berekening verhogen.

Bij deep learning moet men vaak grote tabellen van *n*-parameters opsommen en het resultaat teruggeven aan alle processen. Dit heet *allreduce*. In clusters worden deze tabellen verdeeld over de machines die deelnemen aan de *allreduce*. Vanwege de kleine bandbreedte en hoge latentie tussen de machines (vergeleken met de lokale geheugenbandbreedtes), is deze bewerking vaak het meest kritisch voor gedistribueerd leren. Verkennen van verschillende strategieën voor communicatie, berichtplanning en topologiemapping (Hoefler & Snir, 2011) die goed bekend zijn in het HPC (High Performance Computing)-veld, zou de communicatie in gedistribueerd diep leren aanzienlijk kunnen versnellen.



(a) neuraal netwerk Operator (Ben-Nun & Hoefler, 2018)



(b) Deep Neural Network

Figuur 2.2: Deep Neural Network Architectuur

2.2 Diepe neurale netwerken

2.2.1 Neurons

De basis van een neuraal netwerk is de neuron. Gemodelleerd naar de hersenen, een kunstmatig neuron (Fig. 2.2a) verzamelt signalen van andere neuronen die verbonden zijn door synapsen. Een activeringsfunctie (of axon) wordt toegepast op de geaccumuleerde waarde, deze waarde wordt 'afgevuurd' naar de volgende neuronen. In feed-forward neurale netwerken zijn de neuronen gegroepeerd in lagen die verbonden zijn met de neuronen in volgende laag en hun resultaten van links naar rechts doorgeven.

Feed-forward operators

Net zoals en brein collecteren de neuronen de waarden van hun verbonden neuronen en activeert om zijn eigen waarde aan de volgende neuronen te geven. Neurale netwerken werken precies hetzelfde. Een neuron somt de waarden die hij krijgt van vorige neuronen.

Past een activatie functie toe en stuurt de resulterende waarde door naar de volgende neuronen. dit is voorgesteld in Fig.2.2a waar 3 neuronen hun waarde doorsturen naar de volgende 2 neuronen. Dit proces word ook *inferentie* genoemd.

2.2.2 Diepe netwerken

Wanneer we meerdere lagen van neuronen na elkaar samenstellen, creëren we diepe netwerken (zoals weergegeven in figuur 2.2b). Deze netwerken kunnen meer complexere taken oplossen.

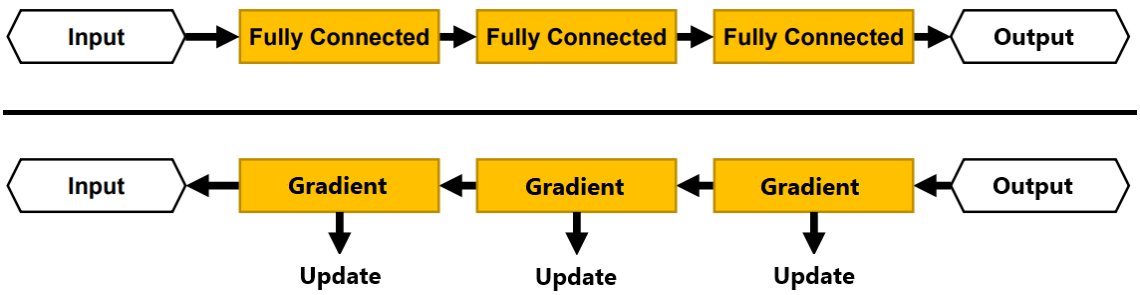
Om een netwerk iets aan te leren moeten we het resultaat van het netwerk (de laatste laag van neuronen) vergelijken met het gewenst resultaat. Als het resultaat verschilt zullen de parameters van het netwerk aangepast worden met een proces genaamd *backpropagation*. Om het verschil tussen voorspelde en verwachte resultaat te bepalen wordt een *loss function* gebruikt, hierdoor kunnen we beslissen hoe groot een verschil is.

In Fig.2.3 zien we het volledig proces nog eens. Hier wordt een netwerk voorgesteld met 5 lagen, de input laag en 3 Fully Connected lagen en de output laag. Fully Connected betekent dat elke neuron volledig verbonden is aan elke neuron van de volgende laag, zoals we zien in 2.2b. Het proces start met inferentie d.w.z. per laag en per neuron alle connecties op te sommen tot je het eindresultaat van het netwerk bekomt, de output (bovenste deel van de figuur). Hierna word de loss functie toegepast om het verschil te bereken met het verwachte resultaat. Met dit verschil kunnen we een gradiënt berekenen waarmee we nu achterwaarts door het netwerk gaan en de parameters van elke laag en neuron veranderen.

We kunnen meerdere keren inferentie uitvoeren voordat we de gradiënt berekenen. Dit maakt de update accurater en kan het neuraal netwerk sneller dingen aanleren doordat inferentie meestal veel sneller is uitgevoerd dan backpropagation. Voor deze reden gebruiken we een gedeelte van de dataset voor elke backpropagation, genaamd een $mini-batch$.

Een neural netwerk model is de collectie van alle parameters in een neuraal netwerk. Met dit model kunnen we het volledig netwerk opslaan en transporteren tussen processen.

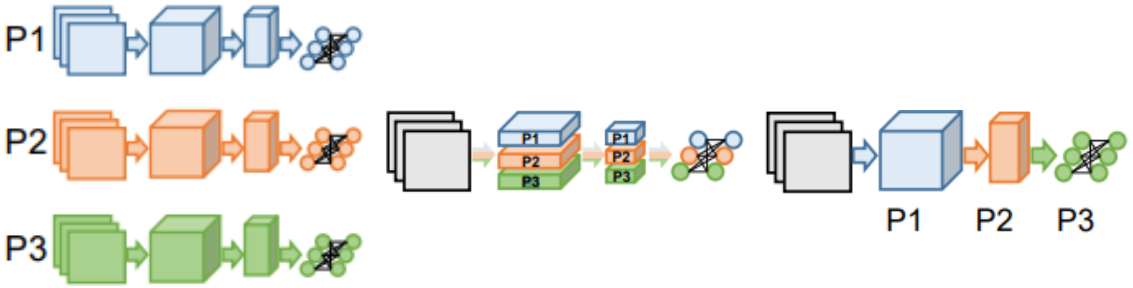
Er zijn hier veel details en formules die nodig zijn om alle soorten neurale netwerken goed uit te leggen maar dit is buiten de scope van deze batchelorproef.



Figuur 2.3: Backpropagation Algorithme

2.3 Gelijktijdigheid in netwerken

Hieronder bespreken we drie prominente partitioneringsstrategieën (geïllustreerd in 2.4): partitionering op basis van data invoer (data parallellisme), op netwerkstructuur (model parallellisme) en op model laag (pipelining). Elk van deze basis strategieën kan een DDLS versnellen en optimaliseren. We zullen later zien hoe dat deze gecombineerd kunnen worden en welke dat het best zou zijn voor onze toepassing in Roblox.



(a) Data Parallellisme (b) Model Parallellisme (c) Laag Pipelining

Figuur 2.4: Neurale netwerk parallellisme schema's (Ben-Nun & Hoefler, 2018)

2.3.1 Data parallellisme

Data parallellisme is het verhogen van de totale doorvoersnelheid van data door het model op meerdere processen te repliceren (zie Fig.2.4a), waar backpropagation parallel kan worden uitgevoerd, om sneller meer outputs en gradiënten te verzamelen. Conceptueel wordt data parallellisme als volgt bereikt. Eerst kopieert of downloadt elk proces het huidige model. Vervolgens voert elk proces backpropagation uit (fig. 2.3). Ten slotte worden de respectievelijke resultaten geaggregeerd en geïntegreerd om een nieuw model te vormen (Dean e.a., 2012).

Omdat gradiënten of de model parameters soms tussen apparaten getransporteerd moet worden, is de relatie tussen de model grootte en de netwerkbandbreedte van cruciaal belang om te bepalen of dat data parallellisme de training kan versnellen op cluster niveau. Hoe kleiner een model is in vergelijking met zijn computationele complexiteit, hoe gemakkelijker het wordt om data parallellisme te implementeren (Iandola e.a., 2015). Voor grote modellen kunnen bandbreedtegerelateerde problemen de schaalbaarheid snel beperken (Langer e.a., 2018). Zoals we echter in latere delen zullen zien, kan data parallelle DDLS verschillende trucs toepassen om de impact van bandbreedtebeperkingen te verminderen.

2.3.2 Model parallellisme

Bij model parallellisme wordt het model opgesplitst. Dit betekent dat per laag de neuronen verdeelt worden onder processen om deze dan parallel uit te voeren (fig. 2.4b). Tijdens inferentie moeten neuronen hun waarden aan de volgende laag geven, dus er is

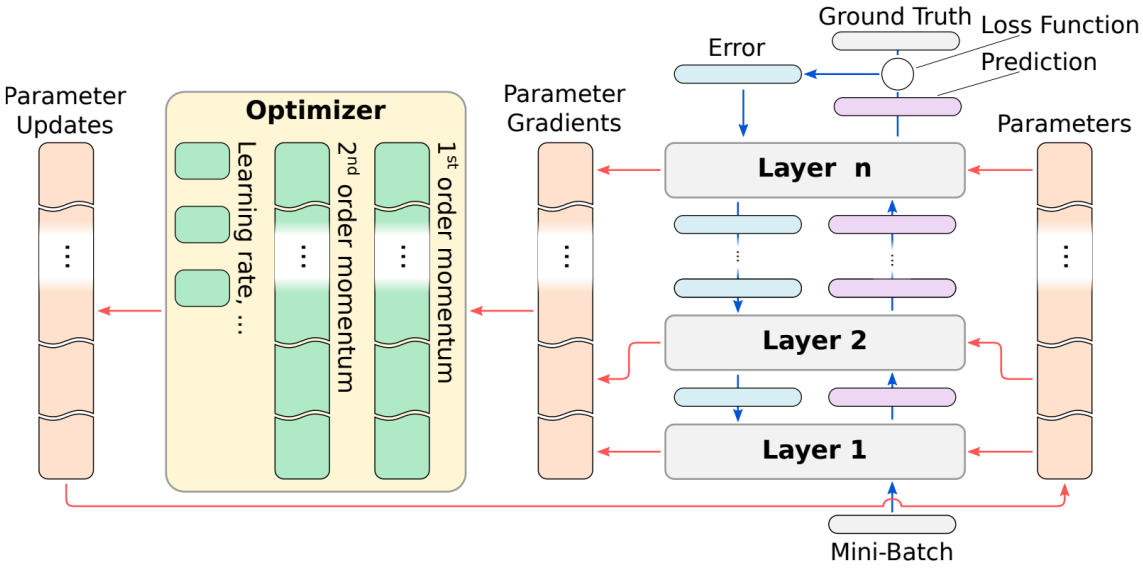
communicatie tussen processen. Het trainen van een neural netwerk met backpropagation vereist het tijdelijk onthouden van de waarden van elke laag die is waargenomen tijdens inferentie. Vaak is hun grootte veel groter dan die van het model (Iandola e.a., 2015). Het partitioneren van het model en het verdelen van de werklast over machines kan dus ook helpen als het geheugen van een individuele machine niet voldoende is om alle modelparameters op te slaan. (Xing e.a., 2015)

In model Parallellisme zijn de lagen zelf gepartitioneerd(fig. 2.4b). Daarom worden verschillende delen van elke laag parallel verwerkt met behulp van meerdere processen. Daarbij leidt model partitionering vaak tot een subset van neuronverbindingen die partitiegrenzen overschrijden. Efficiëntie hangt af van het minimaliseren van machine overschrijdende signalen. Het reorganiseren en verzenden van de output van de afzonderlijke lagen, zodat ze kunnen worden gebruikt door elke partitie van de doellaag en vice versa, is echter complex en vereist dat de DDLS gedetailleerde kennis heeft van de interne werking van de gepartitioneerde lagen, wat het implementeren van dit type model mogelijk maakt. Partitioneren over machinegrenzen heen is in de praktijk vervelend.

Ongeacht welke partitioneringsstrategie wordt gebruikt, de langzaamste route door het model bepaalt de tijd die nodig is om inferentie en backpropagation uit te voeren. Of een modeltrainingstaak kan profiteren van berekeningsstappen op clusterhardware uit te voeren met behulp van model parallellisme, is zeer situationeel (Abadi e.a., 2016). Een model partitioneren, zodat de overhead minimaal is en er geen knelpunten zijn, vereist in de praktijk geavanceerde algoritmen (T. Chen e.a., 2015). Bijzondere eigenschappen van de clusterconfiguratie en elke aanpassing van de mini-batchgrootte of het model veranderen de optimale lay-out. Daarom is een verschuiving plaatsgevonden van model- naar data parallellisme de afgelopen jaren.

2.3.3 Laag pipelining

Hier wordt elke laag aan een andere machine gegeven. Dit kan worden toegepast op elk deep learning model omdat de lagen zelf niet worden beïnvloed. De omringende logica (d.w.z. de DDLS) is verantwoordelijk voor het transporteren van de waardes van laag naar de machine die de volgende laag uitvoert (fig. 2.4c). Tijdens backpropagation worden de gradiënt in omgekeerde volgorde door deze lagen (d.w.z. van machine naar machine) geleid. De overgang tussen partities die zich op verschillende fysieke machines bevinden kunnen kostbaar zijn. De ideale partitionering hangt van veel factoren af (o.a. de mogelijkheden van de clusterhardware, groote van tussenliggende waarden, specifieke datastroom tijdens inferentie en backpropagation, etc.). Moderne model parallelisme compatibele DDLS zoals TensorFlow (Abadi e.a., 2016) gebruiken heuristieken en adaptieve algoritmen om efficiënte partitieschema's te bepalen.



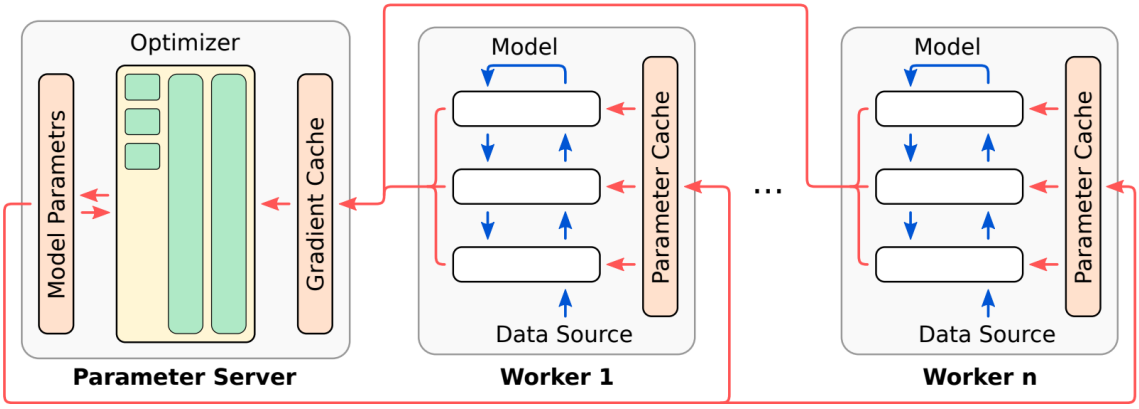
Figuur 2.5: Datastroom in deep learning modellen tijdens training (▲ = gradiëntberekeningscyclus; ▲ = modelupdate / optimalisatiecyclus). (Langer e.a., 2020)

2.4 Model optimalisatie

In Fig. 2.5 beschrijven we de gegevensstroom terwijl we een deep learning model trainen. De trainingsprocedure kan worden opgesplitst in twee verschillende cycli. Het blauwe proces (▲) berekent per parameter gradiënten op basis van de huidige modelparameters door backpropagation toe te passen op minibatches die uit de input data zijn getrokken. De optimalisatiecyclus (▲) gebruikt deze gradiënten om de modelparameters te veranderen. Hoewel dit kan lijken op een bidirectionele afhankelijkheid, is het belangrijk op te merken dat de gradiënt meerdere keren kan berekend worden met verschillende inputs om het resultaat te verfijnen, terwijl de optimizer bijgewerkte gradiënten nodig heeft om vooruitgang te boeken.

We zien twee manieren om deze cycli op een cluster van onafhankelijke machines toe te passen:

- Gecentraliseerde optimalisatie: de optimalisatiecyclus wordt uitgevoerd in een centrale machine, terwijl de gradiënt berekeningen worden uitgevoerd op de resterende apparaten.
- Gedecentraliseerde optimalisatie: beide cycli worden uitgevoerd in elk apparaat en er wordt een of andere vorm van synchronisatie toegepast waardoor de verschillende optimizers samen kunnen werken.

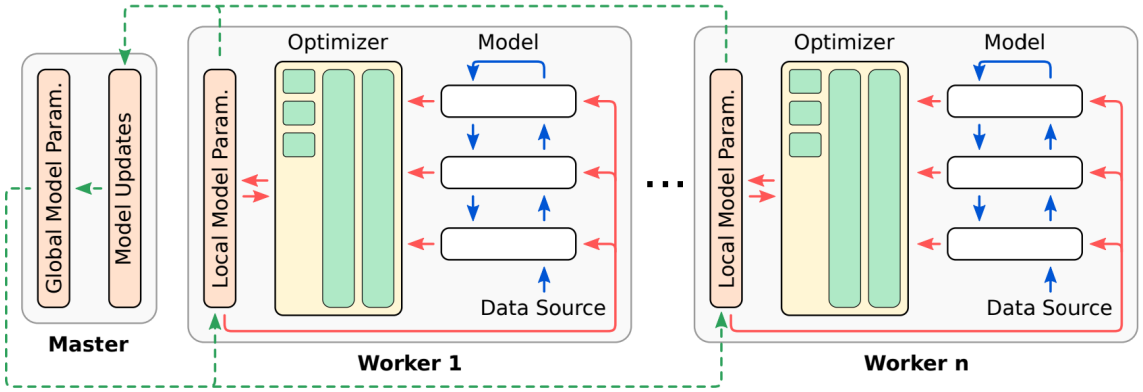


Figuur 2.6: Gecentraliseerde Optimalisatie implementatie op een cluster. Workers evalueren het model om gradiënten te genereren. De parameterserver gebruikt ze om het model bij te werken. (Langer e.a., 2020)

2.4.1 Gecentraliseerde optimalisatie

Bij een gecentraliseerde optimalisatie implementatie, is slechts één enkele optimalisatie-instantie (vaak parameterserver genoemd). Deze is verantwoordelijk voor het bijwerken van de modelparameters. Parameterservers zijn afhankelijk van de gradiënten die worden berekend door cluster nodes die backpropagation uitvoeren (workers). In Fig. 2.6 illustreren we de datastroom tijdens training. Merk op dat de termen parameter server en worker verwijzen naar softwareprocessen, in plaats van naar daadwerkelijke machines. Voor de eenvoud gaan we er voorlopig vanuit dat elk proces op een andere machine draait.

Dankzij gecentraliseerde optimalisatie kan de dure taak van het berekenen van gradiënten per parameter worden verdeeld over de cluster nodes en wordt het model op elegante wijze bijgewerkt door alle resultaten op de parameterserver te bundelen. Daardoor kunnen gradiënten per parameter voor grote hoeveelheden trainingsvoorbeelden snel worden berekend.



Figuur 2.7: Gedecentraliseerde Optimalisatie implementatie op een cluster. Het hoofd worker vormt de volgende globale model parameters door lokale modelreplica's (◀ −−) te combineren die afzonderlijk door de workers zijn getraind (⇆). (Langer e.a., 2020)

2.4.2 Gedecentraliseerde optimalisatie

Gedecentraliseerde optimalisatie DDLS behandelen hun workers als een zwerm, waarin elke worker onafhankelijk de cycli uitvoert en zoekt om de error tussen het voorspelde en verwachte resultaat te minimaliseren (S. Zhang, 2016). Dus gedecentraliseerde systemen voeren modeltraining afzonderlijk uit in elke worker. Om tot een beter gezamenlijk model te komen is enige vorm van samenkomst nodig om de verschillende visies te combineren tot en verbeterde versie van het model (Langer, 2018).

Fig. 2.7 toont de datastroom van decentraal systeem. Elke worker voert herhaaldelijk de loss functie uit en past zijn lokale modelparameters aan om het resultaat verder te verbeteren. Om collaboratieve training te bereiken, moeten de workers modelparameters met elkaar uitwisselen. In dit voorbeeld gaan we er van uit dat er een speciaal hoofd apparaat bestaat, dat de individuele parameteraanpassingen verwerkt die door de workers zijn voorgesteld en een nieuwe globale modelstatus aanmaakt die vervolgens met hen wordt gedeeld (◀ −−). Aangezien de workers vooruitgang kunnen boeken zonder enige communicatie, zijn de netwerk bandbreedtevereisten van gedecentraliseerde systemen gewoonlijk lager dan die van hun gecentraliseerde tegenhangers (Langer e.a., 2018; S. Zhang, 2016). Wat erg belangrijk kan zijn voor onze toepassing in Roblox door de onzekerheid van een goede connectie.

2.5 Model planning

DDLS kan ook worden onderscheiden in synchrone, asynchrone en begrensde asynchrone systemen.

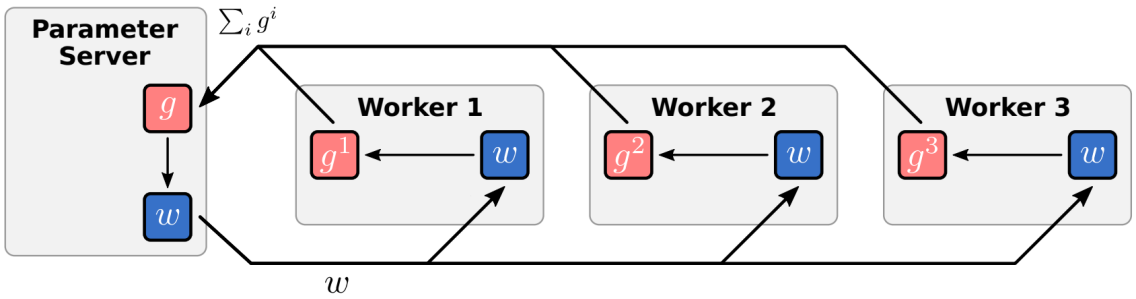
- In synchrone systemen vinden berekeningen voor alle workers tegelijkertijd plaats. Globale synchronisatiebarrières zorgen ervoor dat individuele workers niet verder gaan totdat de overige workers dezelfde status hebben bereikt.
- Asynchrone systemen hanteren een meer ontspannen benadering bij het organiseren van collaboratieve training en voorkomen dat de uitvoering van een worker wordt uitgesteld door een trage worker (d.w.z. de workers mogen in hun eigen tempo werken).

Met andere woorden, synchrone systemen realiseren efficiënte collaboratieve training door het vermijden van afwijkingen tussen workers ten koste van een mogelijke onderbenutting van middelen. Asynchrone systemen daarintegen promoten een hoog hardwaregebruik en afwijkingen tussen workers worden beschouwd als een beheersbaar neveneffect dat kan - zoals we zullen laten zien - zelfs voordelig zijn in bepaalde situaties.

Begrensde asynchrone systemen vertegenwoordigen een hybride benadering tussen deze twee archetypen. Ze werken als gecentraliseerde asynchrone systemen, maar hebben regels om op tragere of snellere workers te wachten. De workers opereren dus asynchroon ten opzichte van elkaar, maar alleen binnen bepaalde grenzen.

2.5.1 Gecentraliseerde synchrone planning

In gecentraliseerde planning wordt de modeltraining verdeeld tussen de workers (=gradiëntberekening) en de parameterservers (=modelupdate) zoals we zagen in sectie 2.4.1. Als deze planning synchroon word uitgevoerd, kan de training niet doorgaan zonder een volledige parameteruitwisseling tussen de parameterserver en zijn workers, omdat de parameterserver afhankelijk is van de gradiëntinvoer om het model bij te werken (zie (Iandola e.a., 2015; Dai e.a., 2018)). De workers zijn op hun beurt weer afhankelijk van het vernieuwde model om de verliesfunctie verder te onderzoeken. In gecentraliseerde synchrone DDLS gaat de volledige cluster dus rondgaan in fasen, waarin alle workers dezelfde bewerking uitvoeren. Fig. 2.8 toont de implementatie van de parameterserver- en worker programma's van een eenvoudig gecentraliseerd synchroon systeem.



PARAMETER SERVER PROGRAMMA

Require: initieel model w, aantal workers n

1: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
2: Broadcast model w
3: Wacht voor gradiënten g^i van alle workers
4: Update model w met de gradiënten g^i
5: **end for**

PROGRAMMA VAN DE i^{th} WORKER

Require: training data D^i

1: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
2: wacht voor w
3: Maak mini-batch $x \backsim D^i$
4: Calculeer alle gradiënten g^i voor mini-batch x
5: Verstuur gradiënten g^i naar parameter server
6: **end for**

Figuur 2.8: Gegevensstroom in een gecentraliseerde synchrone DDLS (boven) en minimalistische implementatie van de parameterserver en worker programma's (onder). Merk op dat globale synchronisatiebarrières worden gevormd doordat de workers en de parameterserver op elkaars resultaten wachten. (Langer e.a., 2020)

Uitgaande van geschikte en willekeurige minibatch, kunnen grotere mini-batches de data distributie beter weergeven (Keskar e.a., 2016). Als gevolg hiervan neemt de variatie in de gradiënten tussen updatestappen af, omdat de individuele updates die door de optimizer zijn bedacht, gebaseerd zijn op een bredere, beter geïnformeerde kijk op de trainingsdata.

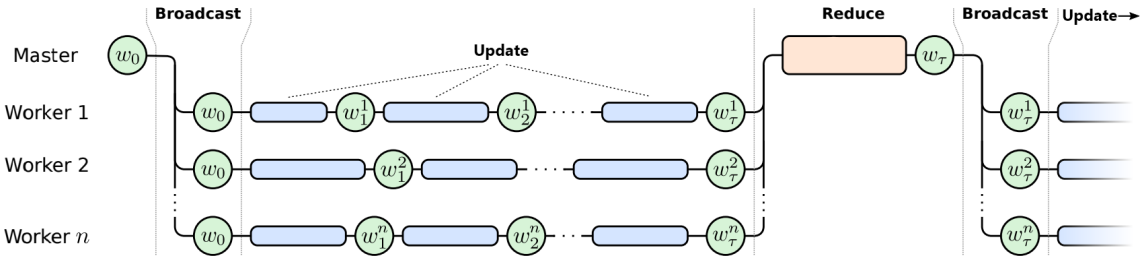
Dit kan de training aanzienlijk versnellen (Sutskever e.a., 2013; Kingma & Ba, 2014). Merk echter op dat de versnelling van de training snel kan afnemen naarmate de mini-batchgrootte toeneemt (Langer, 2018). Verder hebben Keskar e.a. (2016) ontdekte dat het optimaliseren van een model met behulp van mini-batches met een grote dekking van de data distributie de neiging heeft vast te komen zitten en niet meer resultaten te verbeteren.

Tot nu toe hebben we gesuggereerd dat de volgende trainingsstap pas kan worden uitgevoerd als alle workers hun toegewezen taak hebben voltooid en gradiënten hebben ingediend. In dergelijke configuraties moet een meerderheid van clustermachines altijd wachten op achterblijvers (J. Chen e.a., 2016). Dit is echter alleen verplicht als de update gradiënt aanzienlijk wordt gewijzigd zonder de bijdragen van bepaalde workers. Als de training data groot genoeg, redelijk evenwichtig en voldoende willekeurig over de workers zijn verdeeld, maakt het vaak niet uit of kleine delen van de training data ontbreekt. Om bijvoorbeeld te voorkomen dat rekentijd verloren gaat stelden Chilimbi e.a. (2014) voor om de trainingsperioden te beëindigen zodra 75% van alle training data is verwerkt, terwijl Abadi e.a. (2016) voorstelden om over het algemeen te veel te voorzien door meer workers toe te voegen en elke gradiëntaggregatiefase te beëindigen zodra een quorum is bereikt. Beide benaderingen resulteren in een aanzienlijk verhoogde frequentie van modelupdates, die meer dan compenseren voor de ontbrekende informatie van de vertraagde workers en de anders verloren rekenkracht van snelle workers.

2.5.2 Gedecentraliseerde synchrone planning

Synchrone DDLS met gedecentraliseerde optimalisatie voeren onafhankelijk model updates uit in elke worker en wisselen dus geen parameters uit. Maar om de onafhankelijke bevindingen van elke worker te delen met de rest van de cluster combineren ze na een tijd hun modellen om goede generalisatie te behalen (zie sectie 2.4.2). Daarbij werken ze in fasen, gescheiden door globale synchronisatiebarrières.

In Fig. 2.9 zien we implementaties van de master- en worker-programma's en illustreren we de volgorde van gebeurtenissen bij het trainen van een model in gedecentraliseerde synchrone systemen.



MAIN WORKER PROGRAMMA

Require: initieel model w_0, aantal workers n

$t \leftarrow 0$
loop
  Broadcast model w_t
  Krijg models $w^i_{t+\tau}$ van alle workers
  Combineer alle modellen $w^i_{t+\tau}$ naar een nieuw model $w_{t+\tau}$
  $t \leftarrow t + \tau$
end loop

PROGRAMMA VAN DE n^{th} WORKER

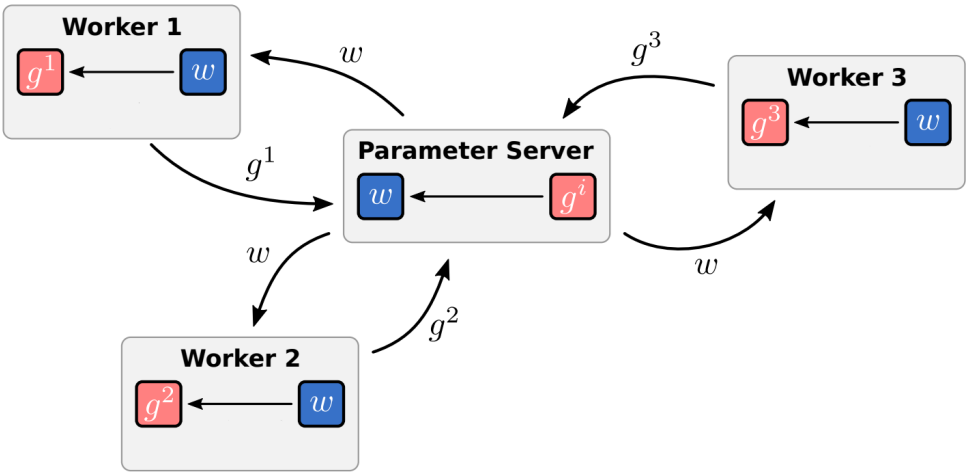
Require: training data D^i

$t \leftarrow 0$
loop
  Wacht voor w_t
  for $\tau \leftarrow 1, 2, \ldots \tau$ **do**
    Maak mini-batch $x \backsim D^i$
    Update model $w_{t+\tau}$ met mini-batch x
  end for
  Verstuur model $w^i_{t+\tau}$ naar parameter server
  $t \leftarrow t + \tau$
end loop

Figuur 2.9: Sequentie diagram (boven) en implementatie (onder) van het gedecentraliseerde synchrone planning (Langer e.a., 2020)

2.5.3 Gecentraliseerde asynchrone planning

In gecentraliseerde asynchrone DDLS (bijv. (Li e.a., 2014; Abadi e.a., 2016; T. Chen e.a., 2015; Dean e.a., 2012; Ho e.a., 2013)) werkt elke worker alleen en deelt zijn gradiënten met de parameterserver zodra een mini-batch is verwerkt. In plaats van te wachten tot andere workers dezelfde toestand bereiken, gebruikt de parameterserver de ontvangen gradiënten in het optimalisatie-algoritme om het model te trainen. Elke update van het globale model is dus alleen gebaseerd op de gradiënt van een enkele worker.



PARAMETER SERVER PROGRAMMA

Require: initieel model w

1: Verdeel w
2: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
3: **if** ontvangt gradiënt g^i van worker i **then**
4: Update model w met gradient g^i
5: Verstuur w naar worker i
6: **end if**
7: **end for**

PROGRAMMA VAN DE i^{th} WORKER

Require: training data D^i

1: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
2: Wacht voor model van de parameter server w
3: Maak mini-batch $x \backsim D^i$
4: Calculeer de gradiënt g^i met mini-batch x
5: Verstuur g^i naar parameter server
6: **end for**

Figuur 2.10: Gegevensstroom (boven) en minimalistische implementatie (onder) van een gecentraliseerde asynchrone DDLS. (Langer e.a., 2020)

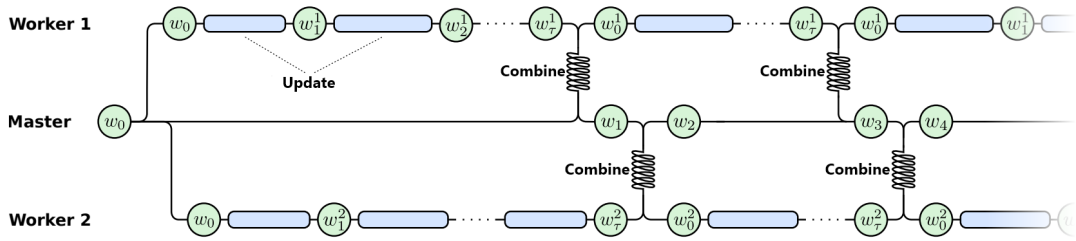
We illustreren dit idee in Fig. 2.10. Het initiatief in asynchrone systemen blijft bij de workers, die de parameterserver in hun eigen tempo aanroepen om gradiënten te sturen, waarna het globale model onmiddellijk wordt bijgewerkt en verstuurd. Op deze manier heeft elke worker een aparte parameteruitwisselingscyclus met de parameterserver. Om-

dat er geen onderlinge afhankelijkheiden tussen workers zijn, kunnen situaties waarin achterblijvers de executie van andere workers vertragen, niet voorkomen. Om dit systeem te laten werken, mag het kiezen van de resultaten van de ene worker boven de andere geen vervorming introduceren die de verliesfunctie aanzienlijk verandert (Tandon e.a., 2017). De mini-batches die voor elke worker worden samengesteld, moeten dus gemiddeld de eigenschappen en verdeling van de trainings data redelijk goed nabootsen.

2.5.4 Gedecentraliseerde asynchrone planning

De workers in gedecentraliseerde asynchrone planningen handelen onafhankelijk en blijven het model ontwikkelen dat los staat van de huidige status van de parameter server. Dus kunnen de workers hun modelparameters niet vervangen bij het voltooien van een parameteruitwisseling met de parameter server. In plaats daarvan moeten ze de respectieve asynchroon verzamelde informatie samenvoegen. De methode voor het combineren van master- en worker-modellen in een dergelijke setting kan met een verschil te berekenen en deze op beide modellen toe te passen.

Dus, bij het uitwisselen van parameters, wordt het worker model verplaatst naar de toestand van het hoofdmodel door hun verschil. En het hoofdmodel past zichzelf aan met het worker model. In Fig.2.11 laten we voorbeeld implementaties zien van respectievelijk de master- en worker-programma's in het gedecentraliseerde asynchrone systeem (S. Zhang e.a., 2015).



PARAMETER SERVER PROGRAMMA

Require: initieel model w

loop
  if krijgt download aanvraag van worker i **then**
    verstuurd w naar worker i
  end if
  if krijgt een model vershil van worker i **then**
    Update model w met model verschil
  end if
end loop

PROGRAMMA VAN DE i^{th} WORKER

Require: training data D^i, parameter deling interval τ, initieel model w

for $t^i \leftarrow 0, 1, 2, \ldots$ **do**
  Maak mini-batch $x \backsim D^i$
  Calculeer de gradiënt g met mini-batch x
  Update model w met gradiënt g
  if $t^i \mod \tau = 0$ **then**
    Download model w van parameter server
    Calculeer model verschil
    Verstuur verschil naar parameter server
    Update model w met verschil
  end if
end for

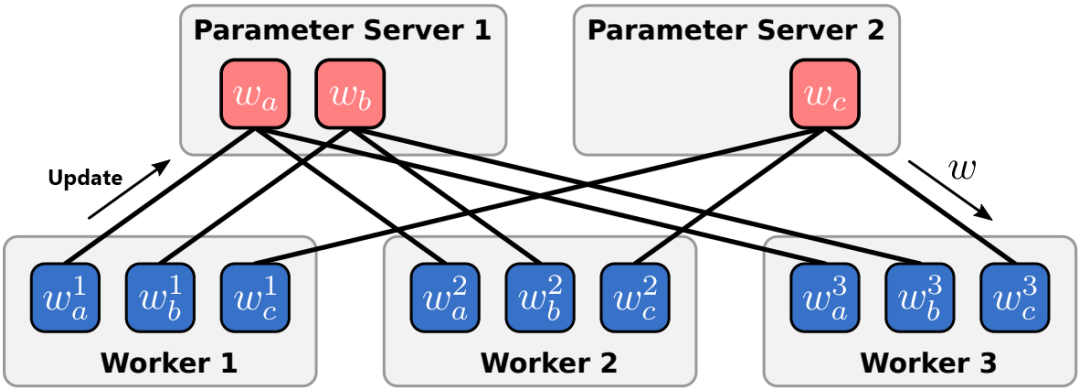
Figuur 2.11: Volgorde van uitvoering (boven) en voorbeeldimplementatie (onder) van het gedecentraliseerde asynchrone DDLS. (S. Zhang e.a., 2015; Langer e.a., 2020)

2.6 Communicatie

Tot nu toe hebben we gesuggereerd dat elke functie in het cluster wordt uitgevoerd door een afzonderlijke machine. Workers, parameterservers en masternodes zijn echter softwaretoepassingen die op dezelfde machine, op afzonderlijke machines of verspreid over meerdere machines kunnen worden uitgevoerd. Het verdelen van de rol van de worker is in essentie de parallellisme dat we in sectie 2.3 hebben beschreven. In deze sectie gaan we verder en richten we ons op communicatiepatronen die DDLS gebruiken om de cluster te organiseren en parameteruitwisselingen te versnellen.

2.6.1 Communicatie in gecentraliseerde systemen

Ongeacht de trainingsmethode door het concentreren van de parameterserverrol in één enkele machine vereenvoudigt de volledige systeemarchitectuur aanzienlijk, omdat alle modeltraining wordt gecoördineerd in één enkel softwareprogramma. Bovendien zijn dergelijke systemen eenvoudig te configureren, te controleren en te debuggen. Maar dit kan van de parameterserver een knelpunt maken. Als de parameterserver een bottleneck is, is het zeer wenselijk om deze rol te verdelen (Dean e.a., 2012), zodat de communicatie niet gericht is op een enkel netwerkeindpunt (zie Fig. 2.12).

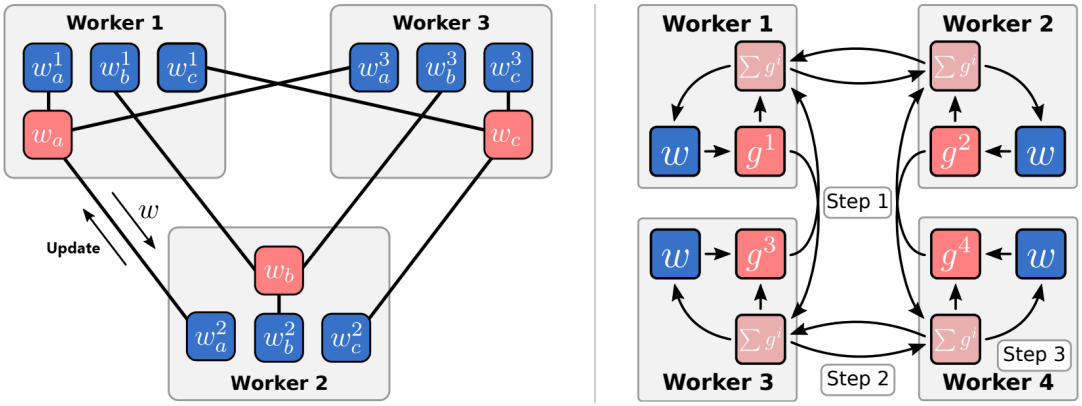


Figuur 2.12: Gecentraliseerd systeem met meerdere parameterservers. Het model is opgedeeld in scherven (a, b en c). Elke parameterserver onderhoudt een subset van deze partities. (Langer e.a., 2020)

Gewoonlijk zijn de rekenkosten van het optimalisatie algoritme laag in vergelijking met het uitvoeren van backpropagatie. Daarom is een populaire variant van de multi parameterserver systeem het migreren van de parameterserverrol naar de workers (zie (Abadi e.a., 2016; T. Chen e.a., 2015; Dai e.a., 2018)), zodat alle nodes workers zijn, maar functioneren ook als parameterservers. In Fig. 2.13 (links) illustreren we een clusterconfiguratie waarbij elke worker verantwoordelijk is voor het onderhouden en bijwerken van een fractie van de globale modelparameters. Deze lokaal onderhouden modelpartitie hoeft niet via het netwerk uitgewisseld te worden. Elke storing van een knooppunt vereist echter een volledige reorganisatie van het cluster (K. Zhang e.a., 2017).

Een alternatieve methode die dit probleem vermijdt ten koste van geheugen en computa-

tie overhead wordt geïllustreerd in figuur 2.13 (rechts). Hier wordt de volledige parameterserverfunctie in elke worker geïmplementeerd. De workers berekenen synchroon de gradiënt, die worden gedeeld tussen machines met behulp van een collectieve all-reduce bewerking. Elke machine gebruikt de daarbij lokaal geaccumuleerde identieke gradiënten om een optimalisatie te doen, die op zijn beurt exact dezelfde update toepast. Merk op hoe deze opstelling niet alleen robuust is voor worker fouten, maar ook het toevoegen en verwijderen van workers triviaal maakt.



Figuur 2.13: Clusterconfiguraties voor gecentraliseerde systemen waarbij elke worker parameterserverfuncties implementeert. Links: Elke worker is verantwoordelijk voor een deel van het model. Rechts: Alle workers werken het volledige model synchroon bij met behulp van gradiënten verzameld van alle workers. (Langer e.a., 2020)

Communicatiepatronen waarbij modelparameters vaak zonder coördinatie tussen machines worden uitgewisseld, kunnen snel moeilijk te beheren worden naarmate een cluster groter wordt. Met name in grote asynchrone systemen zijn omstandigheden die knelpunten veroorzaken onvermijdelijk. Het stellen van limieten voor asynchrone verwerkingen is een effectieve maatregel (zoals tijd of waarde limieten). Maar naarmate het cluster groter wordt, zullen strakke limieten uiteindelijk steeds meer workers tegenhouden. Een gecentraliseerde DDLS die uitbreidt naar honderden of meer workers, vermijden vaak I/O-knelpunten door de communicatie te structureren door proxyservers te introduceren die functioneren als een tijdelijke cache voor verschillende subgroepen van workers (Li e.a., 2014; T. Chen e.a., 2015). Houd er echter rekening mee dat het hebben van tijdelijke caches en/of proxyservers extra vertragingen kan veroorzaken (zie sectie 2.5.3).

2.6.2 Communicatie in gedecentraliseerde systemen

Uitgaande van geïsoleerde trainingsfasen van τ cycli, dan is de communicatie kost van elke gedecentraliseerde worker per lokale rekenstap slechts een fractie van de originele tijd. Op deze manier behouden gedecentraliseerde systemen een hoger gebruik van computerhardware, zelfs met beperkte netwerkbandbreedte, wat het trainen van grote modellen mogelijk maakt ondanks bandbreedtebeperkingen (Langer e.a., 2018). Grotere clusters kan nog steeds een knelpunt maken van de parameterserver. Natuurlijk is het mogelijk om de rol van de master zoals in gecentraliseerde systemen te splitsen om de communicatiekosten te verlagen.

3. Methodologie

In dit onderdeel zal de aanpak van het onderzoek besproken worden. Dit gaat van het zoeken naar een geschikte methodes tot de architecturen die gebruikt zijn om een antwoord te krijgen op de opgestelde onderzoeksvragen. Het ultieme doel van dit onderzoek is om te onderzoeken of het mogelijk is om met de beschikbare functionaliteit van de game engine een proof-of-concept te ontwikkelen. Aan de hand van deze proof-of-concept zal gekeken worden of deep learning mogelijk is in Roblox.

3.1 Literatuurstudie

Zoals elk onderzoek start ook dit onderzoek met een uitgebreide literatuurstudie van het onderzoeksdomein. Hier bekijken wij eerst de onderliggende architectuur en algoritmes van een neuraal netwerk. Hierna proberen we de functionaliteiten en limitaties van de Roblox game engine te ontdekken. Met deze kennis zoeken wij naar geschikte parallellisatie methodes en architecturen die compatibel zijn met Roblox. We denken tijdens het onderzoek constant hoe we deze algoritmes kunnen implementeren en stellen hypotheses op over welke effecten deze verschillende methodes hebben op de eindgebruiker. Deze literatuurstudie is te vinden in Hoofdstuk 2. Er is veel wiskunde dat wij uit dit deel hebben weggelaten omdat het onnodig is om de concepten te begrijpen, maar voor de implementatie van mijn eigen proof-of-concept was deze kennis wiskundige cruciaal.

3.2 Proof-of-concept

Wij implementeren de algoritmes en proberen deze te parallelliseren volgens de methodes die wij hebben onderzocht in de literatuurstudie. Dit verifieert dat de methodes toepasbaar zijn binnen de limitaties van Roblox. Hier is het al snel duidelijk geworden wat de problemen zijn met de game engine zijn tijdens de implementatie. In dit hoofdstuk worden onze hoofdonderzoeksvraag en deelonderzoeksvragen beantwoord.

3.2.1 Experimenten

Na het implementeren van de parallellisatie methodes kunnen we deze meten en uittesten in verschillende situaties. De variabelen van de Roblox omgeving die wij kunnen aanpassen voor de experimenten worden uitgelegd en geëxploreerd. Deze testen zijn nodig om de voordelen en nadelen van de methodes aan te tonen. De impact en performantie van elke algoritme wordt gemeten in milliseconden(ms) of seconden en het effect op de spelers wordt gemeten in frames per seconde (FPS). Om dit te meten gebruiken we de in-studio microprofiler (Roblox, g.d.) en een benchmark plugin voor meer gedetailleerdere grafieken. (boatbomber, 2020)

3.2.2 Model memory

Uit het vorige hoofdstuk is het duidelijk dat er altijd een bottleneck is in elke methode als je een groot genoeg model probeert te trainen. Hier kijken we verder hoe we deze limieten kunnen oplossen. We zien nog delen over model opslag, waar de concepten ook verdere optimalisaties kunnen betekenen tijdens het trainen en uitvoeren van een model.

3.2.3 Inferentie

Dit hoofdstuk gaat over de normale uitvoer van een model. We tonen aan dat het moeilijk is om nog veel optimalisaties te verkrijgen bij deze stap. Inferentie is het voorwaarts doorlopen van een model om van input naar output te gaan.

4. Proof-of-concept

In dit hoofdstuk bekijken we de implementatie van deze verschillende parallellisatie methodes. We bespreken hun problemen, hoe deze problemen kunnen voorkomen worden en de voor- en nadelen tegenover andere methodes.

4.1 Variabelen

Een neuraal netwerk kan in veel verschillende groottes voorkomen en de cluster omgeving kan ook veranderen. Deze verschillen kunnen impact hebben op een methode dus we proberen deze in kaart te brengen in de volgende secties. We kunnen niet alles lokaal uittesten en er kunnen verschillen bestaan tussen onze gesimuleerde omgeving en de realiteit. Hiermee houden wij rekening tijdens het evalueren van methodes. De parameters voor een neuraal netwerk zijn:

- aantal lagen
- aantal nodes per laag
- mini-batch grootte (hoeveelheid inferenties voor een update)
- aantal epochs (aantal updates per experiment)

En voor de cluster omgeving:

- aantal workers
- netwerk latentie

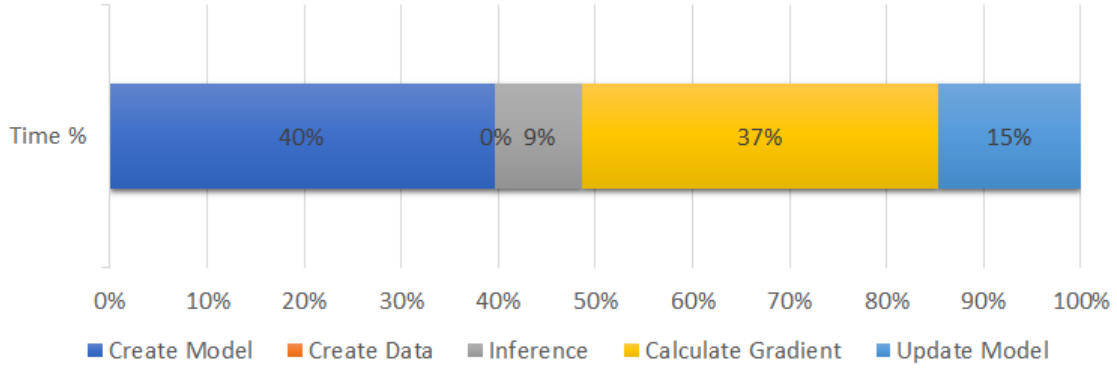
In onze gesimuleerde omgeving word alles door één computer uitgevoerd. De netwerk latentie is dus gesimuleerd en de potentiële rekenkracht door meer spelers kan niet volledig

gesimuleerd worden. We zullen deze parameters aanpassen voor elk experiment.

Tabel 4.1: Verschillende modellen gebruikt in experimenten

Model	Nodes per laag	# Parameters
Tiny	[5, 5, 5, 5, 3]	108
Small	[10, 16, 16, 16, 10]	890
Medium	[10, 128, 128, 128, 10]	19.210
Large	[10, 1024, 1024, 1024, 10]	2.120.714
Huge	[10, 4096, 4096, 4096, 10]	33.648.650
Extreme	[10, 32768, 32768, 32768, 10]	2.148.237.322

In tabel 4.1 zien we de verschillende modellen dat we hebben gebruikt tijdens de experimenten. We geven bij elk model het totaal aantal model parameters om hen te vergelijken. Moderne state-of-art modellen zijn vergelijkbaar met het huge en extreme model, maar voor de meeste toepassingen is de large of kleinere modellen goed genoeg. Deze modellen zijn specifiek gemaakt om onze methodes uit te teste. Normaal zouden deze een complexere architectuur hebben voor verschillende applicaties in productie.



Figuur 4.1: Het percentage tijd dat elk onderdeel nodig heeft om een volledige model update uit te voeren op een mini-batch van 1 en epoch 1.

De tijd dat elk onderdeel nodig heeft in vergelijking tot de andere onderdelen zien we in fig. 4.1. Dit geeft een idee over hoeveel werk dat er nodig is om iets te doen. Deze onderverdeling blijft ongeveer hetzelfde zelfs als het model groter of kleiner wordt. De inferentie hier was met een mini-batch van 1 (zie hoofdstuk 2.2.2). Dit kan verhoogd worden om een meer precieze gradiënt te krijgen en daarmee ook een betere model update. Een grotere mini-batch kan het model trainen versnellen, maar als dit te groot is kan het trainen ervan vertragen.

4.1.1 Lineaire implementatie

Om de effectiviteit van parallelle methodes te kunnen vergelijken hebben we een basismodel nodig. Dit neuraal netwerk wordt uitgevoerd op één proces en is volledig niet geparallelliseerd.

Tabel 4.2: Lineaire uitvoeringstijd van elk model

Model	**Time (sec)**	**Time (ms)**
Tiny	0.00007	0.07
Small	0.00148	1.48
Medium	0.00391	3.91
Large	0.43028	430.28
Huge	27.70633	27706.33
Extreme	1108.56809	1108568.09

In tabel 4.2 zien we de tijd die nodig is voor elk model om uitgevoerd te worden op één proces. Deze uitvoering is hetzelfde als in fig. 4.1, dus met 1 mini-batch en 1 epoch. Let op dat elk ander proces wacht tot dat dit gedaan is, dit betekent dat de volledige server stopt met werken totdat een model uitgevoerd was.

De moderne standaard voor frames-per-seconde (fps) is 60 fps, dit betekent dat de game engine elke 16.66ms een nieuwe frame moet aanmaken. De modellen die meer dan deze tijd nodig hebben kunnen een groot effect zijn op de eindgebruiker door fps van de server of speler te verlagen. Om dit te ontwijken kunnen we tijdens de uitvoering bijvoorbeeld elke 15ms de calculaties stopzetten om een nieuwe frame aan te maken en daarna opnieuw voor 15ms te werken. Dit is niet zo geïmplementeerd tijdens deze test maar is extreem aangeraden in productie als je grote modellen gebruikt. 15ms kan te lang zijn als er nog veel andere berekeningen moeten gemaakt worden tijdens dezelfde frame. De distributie van werk onder meerdere spelers helpt ook om dit probleem te ontwijken, door de tijdskost te verdelen onder machines.

4.2 Experimenten

Hier zien we de resultaten, problemen en conclusies van alle experimenten. We deden elke test meerdere keren en namen het gemiddelde van alle resultaten.

4.2.1 Neuraal netwerk parallellisatie

Zoals we hebben gezien in 2.3 zijn er verschillende methodes om binnen een neuraal netwerk de calculaties te verdelen. Wij proberen deze methodes uit op een lokale machine in luau.

Data parallellisme

We hebben deze methode gezien in 2.3.1. We stellen voor dat elke mini-batch op een andere proces wordt uitgevoerd. We doen inferentie met een mini-batch van 16 voor alle modellen en vergelijken dit met de lineaire methode op dezelfde test. De verwachting hier is dat het verdelen van de mini-batches een grote versnelling kan geven.

Tabel 4.3: Lineaire en data parallelle executie tijd in seconden van verschillende modellen

	Tiny	**Small**	**Medium**	**Large**	**Huge**
Linear	0.032	0.086	0.987	33.503	443.406
Data Parallel	39.642	46.581	49.401	256.958	3967.414

Zoals we zien in tabel 4.3 zijn de resultaten niet wat we verwacht hadden. De parallelle methode duurt veel langer dan de lineaire uitvoering en het extreme model werkt niet. De reden hiervoor is het feit dat elk proces geen data kan gebruiken van het hoofdproces. Al de data dat ze nodig hebben om deze calculaties uit te voeren moet naar hun getransporteerd worden. Dit kost veel tijd en maakt dit volledig inefficiënt.

Om dit probleem beter te begrijpen kijken we naar de multiprocessing implementatie die recent uitgebracht is. Hier gebruikt Roblox Virtuele Machines (VMs) om nieuwe Luau omgevingen aan te maken. Deze VMs delen de data van de hoofd Luau omgeving niet en staan volledig apart van de originele omgeving. Voor elke core wordt één VM aangemaakt en hierop kunnen dan processen apart en parallel uitgevoerd worden. Om data naar een VM te krijgen moet deze getransporteerd en opgeslagen worden en hier komt een grote kost bij.

Het transporteren van de data gaat niet snel en is vergelijkbaar met het versturen van data over een netwerk naar een andere speler. Dit moet gedaan worden voor elk proces dat je opstart en is extreem inefficiënt. Dit is een grote reden waarom parallel Luau nog steeds in bèta is. Als dit probleem opgelost was zou deze nieuwe functionaliteit niet in bèta zijn.

Het opslaan van deze getransporteerde data kan ook een groot probleem zijn. Zoals het extreme model. Dit model heeft enorm veel parameters en daardoor ook veel data. Hier-

door is het niet alleen moeizaam om te transporteren maar is ook enorm moeilijk op te slaan. Stel je voor dat je een model, dat amper in het geheugen van de server past, nog eens moet dupliceren om andere VMs, dit is onmogelijk om efficiënt te doen. Daarom zie je de uitvoeringstijd van het extreme model niet in deze tabel.

De oplossing voor deze problemen is best duidelijk. Een gedeeld geheugen tussen VMs en de hoofd Luau omgeving. Hierdoor moet de data niet getransporteerd nog gedupliceerd worden om calculaties of updates uit te voeren. Zelfs een geheugen dat je alleen kan lezen zou al veel helpen bij de optimalisatie van deze methodes. Hierover wordt al veel gepraat op de forums. Dit is exact waar Roblox op aan het werken is. Als ze deze functionaliteit uitbrengen zal de parallel Luau uit studio bèta gehaald worden en we zullen parallellisatie standaard in Roblox kunnen gebruiken. Het is verwacht dat dit gedeeld geheugen update klaar is rond eind 2021.

Model en laag parallellisatie hebben hetzelfde probleem: het transporteren van data duurt te lang om effectief te zijn en duplicatie van data kan te veel geheugen vergen. Dit kan natuurlijk ook opgelost worden met gedeelde geheugen tussen processen.

4.2.2 Gecentraliseerde training

We kijken naar de testen uitgevoerd op gecentraliseerde planningen zoals we hebben gezien in hoofdstuk 2.6.1. De cluster opstelling van onze testen is 1 server met 8 spelers. Dit wordt allemaal gesimuleerd op 1 machine dus de performantie wordt hier niet volledige correct gerepresenteerd. Maar we kunnen nog steeds conclusies trekken met deze vergelijkingen.

De experimenten worden uitgevoerd met mini-batch van 16 en 100 epochs voor elk model. We voeren dus een volledige update cyclus 100 keer uit. Elke test wordt meerdere keren uitgevoerd en we nemen het gemiddelde van alle resultaten. Bij synchrone planning wordt er gewacht totdat elke worker klaar is met het uitvoeren van hun taak. In asynchrone planning wachten de workers niet op elkaar en de parameter server stuurt een nieuwe taak als de worker klaar is. We verwachten dat het verdelen van taken tussen spelers de test sneller kan uitvoeren dan alleen op de server. Deze verdeling van taken zou ook een verminderde performantie hebben op de server en een gelimiteerde impact hebben op de spelers.

Tabel 4.4: Gemiddelde tijd in seconden dat een lineaire en gecentraliseerde methodes nodig hebben om uitgevoerd te worden op verschillende modellen met mini-batch van 16 en 100 epochs.

	Tiny	**Small**	**Medium**	**Large**
Linear	0.017	0.060	0.804	84.568
Center + Sync	0.995	0.933	1.657	86.582
Center + Async	0.641	0.684	1.046	58.826

We kunnen in tabel 4.4 zien welk effect dat deze planning heeft op de uitvoeringstijd

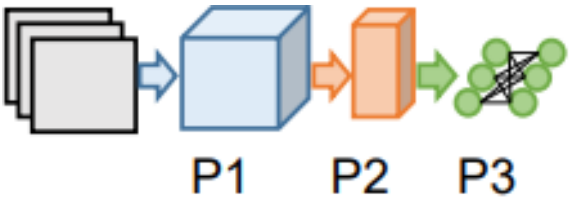
van de modellen in vergelijking met elkaar. We kunnen al direct zien dat we bij kleine modellen geen verbetering hebben geboekt en dat de huge en extreme modellen hier niet aangeduid worden. Er is een kleine versnelling bij het asynchrone uitvoering van het large model, maar dit is nog niet genoeg om echt overtuigd te zijn van het gebruik van meerdere workers. We zien maar een kleine verbetering met 8 workers. Ideaal zou het model ongeveer 8 keer zo snel moeten uitgevoerd worden tegenover een lineair model om optimaal de cluster omgeving te benutten.

Latentie

Een reden voor de slechte performantie bij kleine modellen is de netwerk latentie. We moeten wachten totdat de model data van de parameter server naar de worker transporteert en het resultaat terug wordt gestuurd naar de server. Deze vertraging kan afhangen per speler en zijn afstand tot de server. Dit geeft op synchrone uitvoering een veel grotere impact dan bij asynchrone methodes, omdat de synchrone uitvoering moet wachten op de traagste worker om verder uitgevoerd te worden. We doen later nog testen op de invloed van gesimuleerde netwerk latentie.

Bandbreedte

Zoals we kunnen zien in tabel 4.4 missen we de huge en extreme modellen. Deze zijn gecrasht tijdens uitvoering. Dit is door de bandbreedtelimieten, de grote hoeveelheid parameters die we over het netwerk moeten sturen zorgt voor problemen. Roblox heeft een soft limiet van 50 kilobytes per seconde (kbps) per worker. Hierna worden alle verdere netwerk acties in een wachtrij geplaatst. Als deze wachtrij wordt gevuld kan de latentie tussen server en gebruiker veel groter worden of zelfs de connectie verbreken. Een model die te groot is om zijn parameter over te dragen naar workers in een redelijke tijd, kan een grote impact hebben op de speler of zelf het spel volledig stopzetten. Dit was het geval bij de huge en extreme modellen.



Figuur 4.2: Laag pipelining kan gebruikt worden om de modelgrootte over het netwerk te beperken. (Ben-Nun & Hoefler, 2018)

Als we een grote hoeveelheid parameters over een netwerk willen transporteren moet dit anders aangepakt worden. Momenteel versturen we het volledig model in één stuk door naar de worker. Dan voert de worker het volledig model uit en stuurt de gradiënten terug. Dit kunnen we verder opsplitsen in delen zoals we hebben gezien in sectie 2.3.3 met laag pipelining. Hier splitsen we het model op in zijn lagen zodat we deze op aanvraag kunnen doorsturen naar de worker. Op deze manier kan de netwerklimieten ontweken worden en ons toch de optie geven om grote modellen uit te voeren op een gedistribueerde manier.

De lagen van het model dat zijn doorgestuurd worden op de worker opgeslagen. Zo heeft de worker na een iteratie het volledig model.

Omdat het model te groot is om door te sturen in één stuk zal het gradient ook te groot zijn om in één deel door te sturen. Dus deze moeten we ook verdelen in zijn lagen tijdens het doorsturen. De parameterserver combineert de stukken per worker om het volledige gradiënt te bekomen. We doen hieronder nog eens dezelfde test als in table 4.4 maar deze keer met laag pipelining geïmplementeerd.

Tabel 4.5: Gemiddelde tijd in seconden dat een lineaire en gecentraliseerde methodes nodig hebben om uitgevoerd te worden op verschillende modellen met mini-batch van 16 en 100 epochs. Hier delen we het model op in lagen en worden deze op aanvraag naar de worker verstuurd. LP is een verkorting voor Laag Pipelining.

	Tiny	**Small**	**Medium**	**Large**	**Huge**
Linear	0.017	0.060	0.804	84.568	410.809
Center + Sync	0.995	0.933	1.657	86.582	/
Center + Sync + LP	8.491	8.371	7.749	76.168	416.646

We presenteren onze bevindingen in tabel 4.5. Ons programma crashte niet meer tijdens de uitvoer van het huge model maar bij het extreme model zijn de lagen nog steeds te groot om getransporteerd te worden. Het was verwacht dat elk model meer tijd zou nodig hebben om uitgevoerd te worden. Dit is ook het geval met kleinere modellen. Roblox heeft een limiet van 30 berichten dat per seconde kunnen verstuurd worden. Hierdoor is de minimum executie tijd best hoog en maakt deze methode volledig onnodig voor kleinere modellen.

Maar bij de grote modellen hebben we een grote verbetering tegenover de Lineaire methode. Dit komt door het verspreiden van taken naar meerdere workers in plaats van alleen op de server te werken. De schakering van netwerk acties maakt ook een verbetering in het large model. Stel je voor dat de parameter server het volledig model één voor één moet doorsturen naar elke worker. Tegen de tijd dat de laatste worker kan starten met zijn werk heeft de eerste werker al veel gedaan. Door de verkleinde netwerk pakketten kan elke worker sneller beginnen met calculaties terwijl de rest van het model getransporteerd wordt.

Als de laag van een model nog steeds te groot is om comfortabel te transporteren (zoals bij het extreme model) kunnen we ook nog model parallellisme toepassen en de lagen nog eens onderverdelen tot groepen neuronen zoals we gezien hebben in sectie 2.3.2. Deze groepen kunnen zo klein zijn als we willen en hiermee zijn er geen netwerk limieten meer voor de grootte van een model. Natuurlijk is de uitvoeringssnelheid niet gegarandeerd door de invloed van netwerk latentie. Dus het is belangrijk dat je de delen niet te klein maakt. Als een generale regel proberen we de netwerk pakket grootte rond 4 miljoen parameters te houden omdat dit ongeveer het maximaal toegelaten aantal is. Het is meestal beter voor het neuraal netwerk resultaat om meer lagen te hebben dan grotere lagen. Zelfs de meest grote state-of-the-art modellen hebben geen grotere lagen dan 16,384 neuronen en gaan liever voor een 'dieper' model.

4.2.3 Gedecentraliseerde training

We voeren dezelfde testen uit op de gedecentraliseerde methodes als in de vorige sectie. Het grote verschil met gecentraliseerde planning is dat de parameter server een minder grote rol speelt en zo veel mogelijk werk overlaat aan de workers. Elke worker doet een bepaalde hoeveelheid updates voordat ze hun data opsturen naar de parameter server om gesynchroniseerd te worden met de andere workers. We hebben gezien hoe dit werkt in sectie 2.4.2.

We voeren deze test opnieuw uit met een mini-batch van 16 en 100 epochs. Maar gedecentraliseerde methodes hebben nog een extra parameter, het aantal updates per synchronisatie. Dit aantal beslist wanneer de workers hun data naar de parameter server sturen. Hoe groter het aantal, hoe beter omdat we dan praktisch gezien meerdere lineaire modellen ter gelijk laten lopen. Maar als het aantal te groot is kan het moeilijk worden voor de parameter server om de worker modellen terug te synchroniseren en een slecht effect hebben op de update kwaliteit. Voor deze test hebben we gekozen voor een redelijk lage aantal: 10 updates per synchronisatie. Dit garandeert nog steeds een goede updatekwaliteit zoals gezien in (Langer, 2018; Moritz e.a., 2016). Het is verwacht dat we een grotere versnelling kunnen behalen dan met de gecentraliseerde methode door de vermindering van netwerk communicatie.

Tabel 4.6: Gemiddelde tijd in seconden dat een lineaire en gedecentraliseerde methodes nodig hebben om uitgevoerd te worden op verschillende modellen met mini-batch van 16, 100 epochs en 10 updates per synchronisatie.

	Tiny	**Small**	**Medium**	**Large**
Linear	0.017	0.060	0.804	84.568
Decenter + Sync	0.078	0.086	0.242	17.325
Decenter + Async	0.051	0.063	0.195	13.901

In tabel 4.6 zien we de resultaten van onze testen. We krijgen een grote versnelling bij grotere modellen, maar bij kleinere modellen zorgt het verdelen van taken nog steeds voor een vertraging. De grootste reden hiervoor is netwerk latentie, het is veel sneller om iets kleins lokaal uit te voeren. De grote versnellingen komt doordat worker zelfstandig updates kunnen doen voordat ze hun resultaten naar de parameter server sturen. We zien nu een grote verbetering met 8 workers. Ideaal zou het model ongeveer 8 keer zo snel moeten uitgevoerd worden tegenover een lineair model om optimaal de cluster omgeving te benutten. Hier benaderen we dit ideaal met ongeveer 6 maal sneller dan de lineaire uitvoering. Opnieuw zien we dat de huge en extreme modellen niet op deze manier kunnen uitgevoerd worden, meer uitleg hierover in sectie 4.2.2.

Vanaf 2000 of meer parameters kan het al interessant worden om van lineair over te schakelen naar gedecentraliseerd methode om een model te trainen. De tiny en small modellen zijn moeilijk te optimaliseren. Ze kunnen al uitgevoerd worden binnen één frame zonder invloed te hebben op de fps van de speler.

Met dit resultaat is ons hoofdonderzoeksvraag opgelost. Het is mogelijk om een DDLS

in Roblox uit te voeren zonder een grote impact te hebben op de speler of server.

Gedecentraliseerde asynchrone voordelen

De gedecentraliseerde asynchrone aanpak is de snelste methode bij grotere modellen. Hier zijn de voordelen van deze aanpak:

- Bij elk model is de gedecentraliseerde aanpak sneller uitgevoerd dan elke andere niet lineaire methode.
- De implementatie van de asynchrone gedecentraliseerde planning is één van de makkelijkste om te programmeren. De grootste complexiteit is het model combinatie algoritme.
- De asynchrone handeling van workers is enorm robuust. Ze kunnen dynamisch toegevoegd en verwijderd worden. De server geeft taken aan workers en luistert voor resultaten zoals we kunnen zien in Fig. 2.11. Als een worker crasht, heeft dit geen effect op de parameter server of andere workers.

Netwerk latentie

We kijken verder naar het effect van latentie op onze meest performante methode. Elk bericht de tussen server en worker ondergaat een hoeveelheid latentie, dit wordt gemeten in milliseconden. Hier wordt latentie gedefinieerd als de tijd die nodig is om een bericht van server naar speler te sturen. We testen uit welk effect gesimuleerde latentie heeft op onze methode.

Tabel 4.7: Gemiddelde tijd in seconden dat de gedecentraliseerde asynchrone methode nodig heeft om uitgevoerd te worden op verschillende gesimuleerde latentie en modellen met mini-batch van 16, 100 epochs en 10 updates per synchronisatie.

Latentie	Small	Medium	Large
0 ms	0.0628	0.1947	13.9008
10 ms	0.0765	0.2062	14.8759
50 ms	0.1844	0.3063	15.1364
100 ms	0.2661	0.4113	15.1500
300 ms	0.6679	0.7978	15.7934
1000 ms	2.0794	2.2312	18.4471

De gedecentraliseerde asynchrone methode heeft de minste hoeveelheid communicatie tussen server en worker van alle andere methodes. Daardoor is dit ook minder aangetast door netwerk vertragingen. Zoals we zien in tabel 4.7 is impact van latentie groter op kleinere modellen. Dit is omdat de hoeveelheid communicatie tussen server en worker niet verschilt met modellen. Het wordt dus nog duidelijker dat gedistribueerde training van kleine modellen niet efficiënt is. Bij deze test was de latentie even groot voor elke worker, maar dit kan in realiteit verschillen. Dit heeft op synchrone uitvoering een veel grotere impact dan bij asynchrone methodes, omdat de synchrone uitvoering moet wachten op de traagste worker om verder uitgevoerd te worden.

4.2.4 Grote modellen updaten

Nu wij een idee hebben van de snelheid van elke methode kunnen we verder kijken naar de mogelijke groottes van een model. Wij hebben tot nu toe nog niet het extreme model kunnen uittesten op een parallelle manier door de netwerk en geheugen restricties. Maar wij kennen manieren om deze restricties te ontwijken zoals we in sectie 4.2.2 hebben gezien. Dit zijn de netwerk limitaties die wij hebben ondervonden tijdens het testen tussen de parameter server en de workers:

- 50kbps per worker
- maximum 30 berichten per seconde
- maximum ongeveer 4 miljoen parameters per bericht

Om deze limieten te ontwijken, proberen we methodes zoals laag pipelining en model parallellisatie toe te voegen om ons model op te splitsen in transporteerbare delen.

Het extreme model heeft in totaal 2.148.237.322 parameters. Om dit volledig te kunnen transporteren moet minstens 538 berichten verzonden worden als we het maximum parameters per bericht in het oog houden. De grootste laag in het model is 1.073.741.824 parameters groot. Dus alleen laag pipelining zal niet genoeg zijn om het model te verdelen. We moeten dus ook model parallellisme toevoegen. Hiermee kunnen we elke laag nog eens onderverdelen in groepen van neuronen. We zullen dit nu uittesten, de verwachting is dat we het extreme model parallel succesvol kunnen trainen en dat dit sneller is dan de lineaire methode die alleen op de server uitgevoerd wordt.

Tabel 4.8: Gemiddelde tijd in seconden dat een lineaire en gedecentraliseerde methode nodig heeft om uitgevoerd te worden op verschillende modellen met mini-batch van 16, 100 epochs en 10 updates per synchronisatie. De server was geconnecteerd met 8 spelers tijdens deze testen. De DC, AS, LP, MP zijn verkort voor Decenter, Async, Laag Pipelining en Model Parallellisme.

	Tiny	**Small**	**Medium**	**Large**	**Huge**	**Extreme**
Linear	0.017	0.060	0.804	84.568	410.809	145676.434
DC + AS + LP	0.516	0.571	0.588	17.195	64.252	/
DC + AS + LP + MP	0.875	0.878	0.868	16.607	66.838	26143.993

We zien in tabel 4.8 dat onze hypothese succesvol was. We kunnen het extreme model parallel trainen en dit heeft een enorme versnelling tegenover de lineaire methode. de verschillen tussen laag pipelining (LP) en de combinatie met model parallellisme (MP) bij grotere modellen is minimaal en verwaarloosbaar. De deviatie kan aan toevallige netwerk- en workeromstandigheden geattribueerd worden. Bij de kleinere modellen is er een duidelijke invloed van MP. De minimum tijd die nodig is om alle data door te sturen is verhoogd naar 8 seconden.

Het extreme model kan nu uitgevoerd worden op een parallelle manier die ongeveer 5.5 keer sneller is. Dit is een kleiner versnelling dan bij het huge model waar het 6.1 keer sneller is dan de lineaire methode. De grootste reden hiervoor is geheugen limieten. Het

extreme model heeft 2 biljoen parameters en kan niet volledig in RAM geheugen opgeslagen worden. Het volledig model één maal in geheugen opslaan gaat nog net maar elke worker moet het apart opslaan en updaten om dan terug te sturen. Dit neemt vele malen meer geheugen op dan het model één maal opslaan. Om de gelimiteerde RAM te vergroten gebruikt Roblox de solid state drive (ssd) of hard disk drive (hdd) van de computer. Deze opslag methodes hebben een veel lagere bandwidth en daardoor gaan de computaties trager.

4.3 Model memory

Modellen zoals huge en extreme kunnen problemen hebben om op de server of worker in geheugen opgeslagen te worden door de grote hoeveelheid parameters. In dit hoofdstuk zien we wat die problemen zijn en hoe we daarmee om kunnen gaan.

4.3.1 Model persistentie

Na of tijdens het trainen van een model moeten de parameters opgeslagen worden zodat we het model op een later tijdstip kunnen oproepen en uitvoeren. In Roblox kunnen we DataStores gebruiken om de modellen op te slaan, maar deze functie heeft bepaalde limieten:

- alle data wordt opgeslagen als een string. voor tabellen gebruiken we JSON.
- max 4.000.000 karakters per sleutel
- oneindige hoeveelheid sleutels
- oproepen van data max $60 + numSpelers \times 10$ sleutels per minuut
- schrijven van data max $60 + numSpelers \times 10$ sleutels per minuut

We kunnen opnieuw ons model opsplitsen voor opslag met de concepten die we al hebben besproken in hoofdstuk 2.3, dus per laag en groep neuronen opdelen met een maximum grootte van 4 miljoen karakters. We kunnen al snel aan deze 4 miljoen karakters komen omdat elke parameter een dubbele precisie drijvende-kommagetal (64-bit, 8 bytes) is en daarmee maximaal 15 karakters heeft. Als we nog steeds problemen hebben met deze limieten kunnen we nog andere methodes toepassen zoals tekst compressie (1waffle1, 2018). Met deze DataStores kunnen we de model parameters volledig opslaan voor later gebruik.

We kunnen voorstellen dat de Datastore als gedeeld geheugen kan gebruikt worden, maar hier zijn er nog problemen mee. Workers kunnen niet aan de Datastore, data moet via de server doorgestuurd worden. Als je lokaal op de server probeert te paralleliseren heeft het data ophalen nog steeds een latentie en dit maakt het niet efficiënt zoals we al hebben gezien in hoofdstuk 4.2.1.

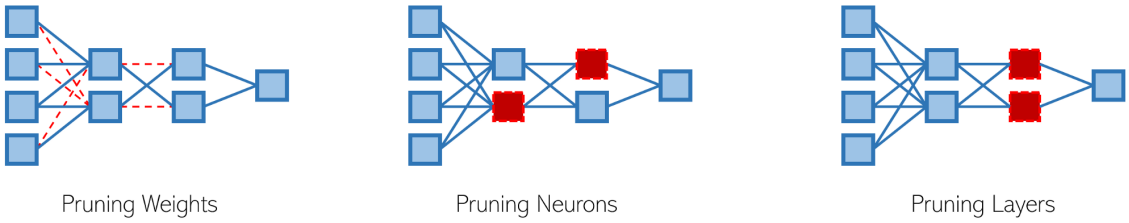
4.3.2 Model optimalisatie

Sommige modellen zijn gewoon te groot om praktisch uit te voeren. Een voorbeeld hiervan is het extreme model. De 2 biljoen parameters zijn te veel om in het RAM geheugen van een gsm of andere apparaten op te slaan. Soms kan het model kleiner gemaakt worden zonder een grote impact te hebben op het eindresultaat. Het is altijd interessant om model compressie toe te passen en dit is standaard geworden in de industrie om deep learning met gelimiteerde middelen uit te voeren. We gaan hier over een paar van de belangrijkste compressie methodes:

- Pruning
- Quantization
- Knowledge distillation

Pruning

Pruning reduceert het aantal parameters door overbodige, onbelangrijke verbindingen te verwijderen die niet prestatiegevoelig zijn. Dit helpt niet alleen de totale modelgrootte te verkleinen, maar bespaart ook op rekentijd en energie.



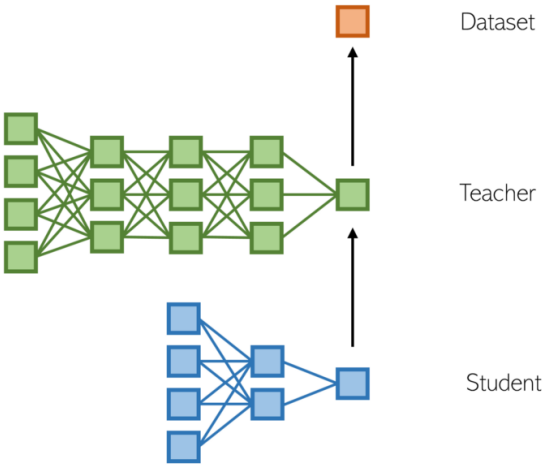
Figuur 4.3: Momenteel is pruning de meest populaire methode voor modelcompressie. (Ye, 2021)

Quantization

In DNN worden gewichten opgeslagen als 32-bits getallen met drijvende komma. Quantization is het idee om deze gewichten weer te geven door het aantal bits te verminderen. De gewichten kunnen worden gekwantiseerd naar 16-bit, 8-bit, 4-bit of zelfs met 1-bit. Door het aantal gebruikte bits te verminderen, kan de omvang van het diepe neurale netwerk aanzienlijk worden verminderd.

Knowledge distillation

Bij Knowledge distillation wordt een groot, complex model getraind op een grote dataset. Wanneer dit grote model kan generaliseren en goed kan presteren, wordt het overgebracht naar een kleiner netwerk. Het grotere model wordt ook wel het 'leerkracht model' genoemd en het kleinere netwerk wordt ook wel het 'student netwerk' genoemd. We gebruiken dus de resultaten van het grote model om een klein model te trainen. We zien een voorstelling hiervan in Fig. 4.4.



Figuur 4.4: De output van de leerkracht wordt getraind om overeen te komen met de gegevens, de output van de student wordt getraind om overeen te komen met die van de leerkracht. (Ye, 2021)

4.4 Inferentie

Figuur 4.5: Inferentie gaat één maal door het volledige neuraal netwerk. Van input naar output.

Tot nu toe hebben we het alleen over training van een model gehad. Inferentie is het uitvoeren van een neuraal netwerk, van input naar output zoals we hebben besproken in sectie 2.2.2 en nog eens zien in Fig. 4.5. Inferentie heeft veel minder calculaties nodig dan het berekenen van een gradiënt of een volledige update, we zien dit in Fig. 4.1. In de vorige experimenten zagen we dat parallellisatie minder goed werkt als elk deel niet veel werk moet uitvoeren, zoals bij kleine modellen is het sneller om alles lineair uit te voeren. De redenen hiervoor was de kost aan transport van informatie naar de verschillende geparallelliseerde delen tegenover de hoeveelheid calculaties dat kan gedaan worden met deze informatie. Als dit ratio te laag is kan het niet efficiënt zijn om parallellisatie toe te voeren. Natuurlijk wordt dit probleem opgelost als er een gedeeld geheugen is tussen processen, zodat er geen kost meer is voor het transporteren van data. Bij inferentie is dit wel nog een groot probleem, het ratio tussen model parameters en de hoeveelheid calculaties is erg slecht om parallellisatie uit te voeren.

Tabel 4.9: Gemiddelde tijd in seconden dat een lineaire methode nodig heeft om inferentie uitgevoerd te worden op verschillende modellen. De server was geconnecteerd met 8 spelers tijdens deze testen.

	Tiny	**Small**	**Medium**	**Large**	**Huge**	**Extreem**
Lineair Inferentie	0.00002	0.00008	0.00056	0.04047	0.52554	35.51900

Meestal wordt de inferentie stap lokaal op één machine uitgevoerd. Vaak zijn modellen niet groot genoeg of moeten niet vaak genoeg uitgevoerd worden om een grote versnellingen te boeken met parallellisatie over meerdere computers. Hierdoor stellen we voor dat elke worker het model van de parameter server aanvraagt en deze lokaal opslaat. Daarna kan deze lokaal uitgevoerd worden op een Lineaire methode. We kunnen data parallellisatie niet toepassen bij inferentie opdat het vaak niet nodig is om het volledig model meerdere keren uit te voeren om aan een bruikbaar resultaat te komen.

We kunnen natuurlijk wel laag en model parallellisatie toe passen. Maar hier zal de tijd om gegevens naar verschillende processen te transporteren veel te lang zijn in vergelijking met de nodige berekeningen. Het is ook complex om te implementeren en zal niet efficiënt zijn zolang dat er geen gedeeld geheugen verkrijgbaar is.

In tabel 4.9 zien we de tijd die nodig is om inferentie uit te voeren. Dit is extreem kort in vergelijking met een model updaten. Hierdoor zijn veel van de methodes niet efficiënt net zoals we hebben gezien voor kleinere modellen.

We kunnen ons model niet opsplitsen tussen workers om daarmee inferentie uit te voeren.

Sinds dat het model maar één keer moet uitgevoerd worden, zouden de workers met elkaar moeten communiceren en dit zal ook veel latentie veroorzaken. Uiteindelijk zal er een belachelijk groot model nodig zijn om echte performantie verbeteringen te krijgen van distributie van taken onder workers.

Om parallellisatie uit te voeren op een lokale machine kunnen we de methodes gebruiken dat we hebben gezien in hoofdstuk 2.3. Maar in sectie 4.2.1 zagen we dat hier nog problemen mee waren en stellen we ook oplossingen voor.

5. Conclusie

In dit onderzoek werd een antwoord gegeven op de onderzoeksvraag: 'Is een distributed deep learning systeem mogelijk in Roblox zonder een grote performantie impact te hebben op de speler of de server?'. Hiervoor werd een proof-of-concept applicatie opgesteld. Hiermee werd er met verschillende distributie methodes geëxperimenteerd. We beantwoorden deze vraag met succes. Het is mogelijk en bij grote modellen zelfs aangeraden dat je gedistribueerd omgaat met neurale netwerken. Verder bespreken we nog het effect op performantie van verschillende methodes en verdere optimalisaties.

Uit de resultaten voor lokale distributie van taken tussen processen zijn wij problemen tegen gekomen. Het transporteren van data kost te veel tijd om efficiënt te zijn. Als we meer berekeningen moeten doen met dezelfde hoeveelheid data kan het wel terug efficiënt worden. Het verwijderen van de data transportkosten door gebruik van een gedeeld geheugen tussen processen is een oplossing die Roblox momenteel aan het implementeren is. Als deze update uitkomt zal parallel Luau ook promoveren uit bèta en geïntegreerd worden in alle spellen.

Wij hebben gecentraliseerde methodes uitgetest waarbij alle workers hun resultaten naar een parameter server sturen. Deze server zal dan de resultaten gebruiken om een model te verbeteren. Hier vonden we al snel problemen met de netwerk bandbreedte. De grotere modellen konden niet uitgevoerd worden en de kleinere modellen waren veel trager dan de lineaire methode. We hadden wel een klein succes in het versnellen van één model door de asynchrone handeling van workers.

Gedecentraliseerde training werkt nog steeds met een parameter server maar elke worker kan tot een bepaald punt volledig op zichzelf werken. De worker verbetert modellen lokaal en hoeft zijn resultaten niet vaak naar de server sturen. Hier zagen we grote suc-

cessen tot 6 maal sneller met 8 spelers in vergelijking tot de lineaire methode. Maar de grotere modellen konden nog steeds niet uitgevoerd worden door netwerk limitaties.

We kijken hoe we grote modellen kunnen uitvoeren. We delen het model in lagen en nodes. Deze kunnen we dan doorsturen naar de workers zonder aan de limieten van het netwerk te komen. De workers kunnen al berekeningen maken met deze delen en moeten zo niet wachten totdat het volledig model is verstuurd. Dit maakt het mogelijk om extreem grote modellen uit te voeren en terug een versnelling van 6 maal te behalen met 8 spelers.

We bespreken technieken om modellen te optimaliseren en verkleinen. Deze methodes worden niet verder geïmplementeerd. We kijken naar de limitaties van Datastores en hun mogelijke toepassingen. We discussiëren hoe inferentie parallel kan worden toegepast en hoe dat dit niet praktisch is.

Dit onderzoek biedt een meerwaarde aan Roblox developers die grote neurale netwerken willen toepassen in Roblox. We geven een beeld van hoe zij dit kunnen structureren en implementeren. Deze technieken kunnen gebruikt worden om grote modellen uit te voeren en optimalisaties aan te tonen voor al bestaande neurale netwerk systemen. Wij hopen dat de verdere uitwerking van parallel Luau nog meer mogelijkheden kan betekenen voor de optimalisatie van lokale neurale netwerken. Het is duidelijk dat het uitvoeren van extreem grote neurale netwerken niet praktisch is zonder te paralleliseren en verdere optimalisaties.

A. Onderzoeksvoorstel

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

A.1 Introductie

Luau is een scripttaal ontwikkeld door Roblox voor hun game engine en is gebaseerd op lua 5.1. Deze game engine is gebruikt door duizenden professionele game studio's en 42.1 miljoen dagelijkse gebruikers (Clement, 2021).

Roblox is niet één enkel spel, maar eerder een verzameling van meer dan 50 miljoen games, allemaal gemaakt door de gemeenschap van spelers. De eenvoudigste vergelijking is er met YouTube: een enorme bibliotheek met "door gebruikers gegenereerde inhoud", maar in dit geval bestaat de inhoud uit games in plaats van video's. Deze spellen zijn erg divers in hun concepten en uitvoer. Elke videogame genre bestaat op roblox en zijn meestal gratis om te spelen. Spelers downloaden de gratis Roblox-applicatie voor computers, gameconsoles, smartphones of tablets en gebruiken deze om door de catalogus met games te bladeren en deze te spelen. Roblox werd officieel gelanceerd in 2006 en is sindsdien nog steeds aan het groeien.

Machine learning, en in het bijzonder deep learning (LeCun e.a., 2015), neemt in snel tempo verschillende aspecten over in ons dagelijkse leven. De kern van deep learning ligt het Deep Neural Network (DNN), een constructie geïn-spireerd op de onderling verbonden aard van het menselijk brein. Als DNN's goed afgestemd zijn door een grote hoeveelheid gegevens te bestuderen, kunnen ze nauwkeurige oplossingen bieden voor problemen

die voorheen als onoplosbaar werden beschouwd. Deep learning is met succes geïmplementeerd voor een groot aantal vakgebieden zoals: beeldclassificatie (Huang e.a., 2016), spraakherkenning (Amodei e.a., 2015), medische diagnose (Cirean e.a., 2013), autonoom rijden (Bojarski e.a., 2016) en menselijke spelers verslaan in complexe spellen (Vinyals e.a., 2019)

Distributed deep learning systems (DDLS) trainen diepe neurale netwerken door gebruik te maken van de gedistribueerde middelen van een cluster. Hier zou de cluster het netwerk van servers en spelers die deel van een Roblox spel zijn.

Een recente bijwerking (EthicalRobot, 2021) van de Roblox game engine heeft de mogelijkheid gegeven om taken parallel uit te voeren. Hiermee bestaat nu de opportuniteit om deep learning in Roblox te gebruiken. De toepassingen hiervan zijn bijvoorbeeld realistische interactieve gesprekken en geavanceerde tegenstanders.

De doelstelling van dit onderzoek is het uitleggen van een methode om een parallel en gedistribueerd systeem te bouwen met de servers en clients als deel van een cluster. Er wordt onderzocht hoe performant zo een systeem zou zijn en de mogelijke toepassingen worden beschreven.

A.2 state-of-the-art

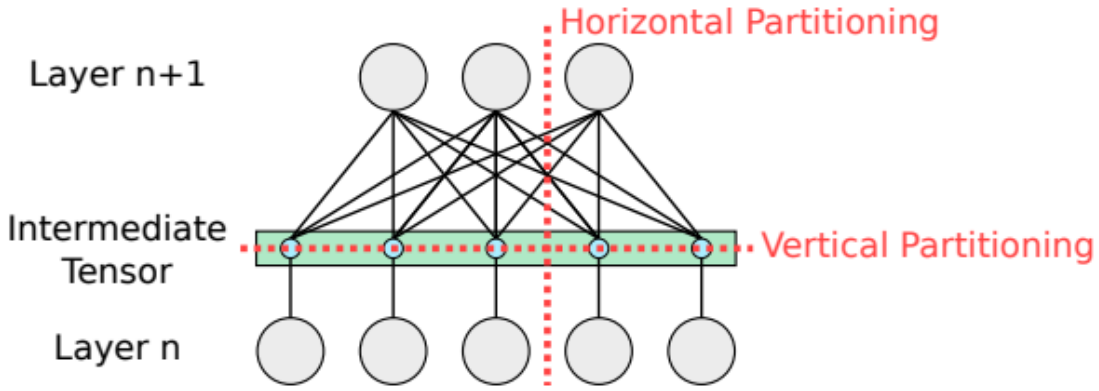
Luau is een gelijktijdige programmeertaal. Dit betekent dat elke Luau omgeving alleen op 1 processor core werkt en is niet gebouwd om meerdere dingen tegelijkertijd uit te voeren. De nieuwe parrallele functionaliteit gebruikt virtual machines (VMs) om meerdere Luau omgevingen op te starten, elk met hun eigen processor core. Hierdoor kan je op een hoofd luau omgeving taken sturen naar de andere omgevingen en dan wachten op een resultaat, dit kan veel programma's veel sneller en efficiënter maken maar heeft een paar problemen zoals:

- Omgevingen delen geen data of variabelen die niet doorgestuurd zijn.
- Alleen de hoofd omgeving kan de game wereld en game data aanpassen.
- De hoofd omgeving moet een resultaat terug krijgen van de andere omgevingen voordat het continueert.

Deze functonaliteit is nog in studio beta, de publieke release is gepland eind 2021. (EthicalRobot, 2021)

Deep Neural Networks (DNN's) worden een belangrijk hulpmiddel in moderne computertoepassingen. Hun training versnellen is een grote uitdaging en technieken variëren van gedistribueerde algoritmen tot het ontwerpen van circuits op hardware niveau. Model- en gegevensparallellisme zijn twee technieken voor het opschalen van grote deep learning-workloads. Het fundamentele verschil tussen beide benaderingen is dat modelparallellisme probeert de modeluitvoeringsstappen op de clusterhardware in kaart te brengen, terwijl dataparallelle benaderingen collaboratieve modeltraining aanpakt als een gelijktijdigheids-/synchronisatieprobleem. Omdat het orthogonale concepten zijn, kun-

nen model- en dataparallellisme gelijktijdig worden toegepast om hybride parallellisme te bereiken. Li e.a. (2014)



Figuur A.1: Model parallellisme kan worden bereikt door horizontale of verticale opdeling van het model

A.2.1 Model parallellisme

Bij model parallellisme wordt het model opgesplitst in partities, die vervolgens in afzonderlijke machines worden verwerkt. Om berekeningen uit te voeren of het model te trainen, moeten signalen tussen beide worden verzonden, zodat elke partitie kan worden geëvalueerd. Model partitionering kan worden uitgevoerd door splitsingen tussen neurale netwerklagen toe te passen (=verticale partitionering) of door de lagen te splitsen (=horizontale partitionering), zoals weergegeven in Fig.A.1.

A.2.2 Data parallellisme

Het basisprincipe van gegevens parallellisme zorgt ervoor dat de totale doorvoersnelheid van data verhoogt door het model op meerdere machines te repliceren. Conceptueel wordt data parallellisme als volgt bereikt. Eerst downloadt elk cluster node het huidige model. Vervolgens voert elk node een berekening uit op hun toegewezen data. Ten slotte worden de respectievelijke resultaten geaggregeerd en geïntegreerd om een nieuw model te vormen. (Dean e.a., 2012)

A.2.3 Gerelateerde werken

K. Zhang e.a. (2017) laten zien hoe bepaalde implementatiebeslissingen de training en het netwerk beïnvloeden volgens bandbreedtegebruik in vier gedistribueerde systemen.

Ben-Nun en Hoefler (2018) heeft een tutorial/enquête gepubliceerd van parallelle en gedistribueerde algoritmen voor diepgaand leren. Hier beginnen ze met een tutorial over

algemene concepten zoals begeleid leren, backpropagation en modelarchitecturen, voor parallelle en gedistribueerde trainingsgerelateerde onderwerpen (hyper-parameter en architectuur zoeken, enz.)

Langer e.a. (2020) heeft een taxonomisch overzicht gemaakt van het probleemdomein. Zij geven een grondige analyse van de verschillende ontwerpkeuzes voor training in een DDLS en leggen de onderliggende intuïties uit.

Kironte (2020) heeft een neurale netwerk module gemaakt voor Roblox. Deze wordt al gebruikt in games met toepassingen zoals zelfrijdende auto's voor politie achtervolgingen. Maar ze is extreem gelimiteerd in complexiteit omdat deze niet in parallel of gedistribueerd werkt.

A.3 Methodologie

In een eerste fase van het onderzoek wordt bekeken wat er gedaan moet worden om de communicatie tussen servers en clients gemakkelijk te laten verlopen. Hierbij worden ook de details en beperkingen van Luau geanalyseerd. In de tweede fase worden de mogelijke moderne parallellisatie en distributie methodes voor deep learning modellen onderzocht die kunnen toegepast worden met de beperkingen van Luau.

Er zullen experimenten worden uitgevoerd om te verifiëren of deze methodes toepasbaar zijn. Simulaties van de nodige computatiekracht op de client en de server worden uitgevoerd. Hiermee krijgen we een beeld van de impact dat een DDLS heeft op de eindgebruiker. We bepalen deze impact door de milliseconden(ms) dat een functie nodig heeft en hun frames per seconde(FPS). Om dit te meten gebruiken we de instudio microprofiler (Roblox, g.d.) en een benchmark plugin voor meer gedetailleerdere grafieken. (boatbomber, 2020)

A.4 Verwachte resultaten

De verwachting is dat er een ontwerp bestaat om een DDLS efficiënt te implementeren in de gelimiteerde omstandigheden van Luau. Dit systeem is compatibel met Luau en zal een gelimiteerde impact hebben op de eindgebruiker. De impact op de eindgebruiker zal afhangen van de grootte van de cluster, de hoeveelheid parameters in het neuraal netwerk en de onderliggende DDLS. Het is moeilijk om 2 verschillende implementaties van DDLS te vergelijken vanwege hun uitgebreide functielijsten en architecturale afwijkingen.

A.5 Verwachte conclusies

We verwachten te concluderen dat er een duidelijke methode is om een DDLS te implementeren en dat deze methode ook direct kan toegepast worden op de studio beta versie

van Roblox. Het zal ook duidelijk moeten zijn dat we betere resultaten kunnen halen dan de al bestaande neurale netwerk module. (Kironte, 2020) Door de versnelling van computaties en de mogelijke vergroting van complexiteit van het netwerk als gevolg van het gedistribueerd werken en parallellisatie van bewerkingen. Ook is het denkbaar een conclusie te stellen dat er veel bruikbare toepassingen zijn binnenin de game development wereld waar deze technologie een meerwaarde geeft.

B. Bijlage

B.1 Methode implementaties

Hier gaan we over de implementaties van methodes in Luau. We hebben de uitleg van elke methode gezien in hoofdstuk 2.5. Dit is niet de exacte code gebruikt in de testen omdat daar veel test specifieke code bijkomt. Deze voorbeelden kunnen gebruikt worden om andere developers op weg te helpen.

B.1.1 Gecentraliseerde synchrone planning in Luau

Zie hoofdstuk 2.5.1 voor meer uitleg over deze planning.

PARAMETER SERVER PROGRAMMA

Require: initieel model w, aantal workers n

1: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
2: Broadcast model w
3: Wacht voor gradiënten g^i van alle workers
4: Update model w met de gradiënten g^i
5: **end for**

PROGRAMMA VAN DE i^{th} WORKER

Require: training data D^i

1: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
2: wacht voor w
3: Maak mini-batch $x \backsim D^i$

4: Calculeer alle gradiënten g^i voor mini-batch x
5: Verstuur gradiënten g^i naar parameter server
6: **end for**

PARAMETER SERVER PROGRAMMA

```
-- Geeft een lijst van spelers
local Players = game:GetService("Players")

-- dit handelt alle wiskunde of complexe delen. Deze
   module is verkrijgbaar op aanvraag.
local ddls = require(game:GetService('ReplicatedStorage')
   .ModuleScript)

-- dit zijn de evenementen die over het netwerk worden
   gestuurd
local BroadcastModel = ReplicatedStorage:WaitForChild("
   BroadcastModel")
local SendGradients = ReplicatedStorage:WaitForChild("
   SendGradients")

-- initialiseer het Large model
local model = ddls.makeModel({10, 1024, 1024, 1024, 10})

function startAllWorkers()
-- starts alle workers die verkrijgbaar zijn
    BroadcastModel:FireAllClients(model)
    for _, player in ipairs(Players:GetPlayers()) do
        workerStatus[player.Name] = "Working"
    end
end

function checkWorkersDone()
-- kijkt ofdat alle workers resultaten hebben terug
   gestuurd
    for _, status in pairs(workerStatus) do
        if status == "Working" then
            return false
        end
    end
    return true
end

function onWorkerDone(player, gradient)
    -- deze functie word uitgevoerd als een worker een
       resultaat terug stuurt
    table.insert(gradients, gradient)
```

```
    workerStatus[player.Name] = "Available"
    if checkWorkersDone() then
        model = ddls.modelUpdateLinear(model, gradients)
        gradients = {}
        startAllPlayers()
    end
end

-- Send model to all workers
startAllWorkers()

-- triggers when a worker sends his gradients
SendGradients.OnServerEvent:Connect(onWorkerDone)
```

PROGRAMMA VAN DE WORKER

```
local ReplicatedStorage = game:GetService("
   ReplicatedStorage")

-- dit handelt alle wiskunde of complexe delen.
local ddls = require(ReplicatedStorage.ModuleScript)

-- dit zijn de evenementen die over het netwerk worden
   gestuurd
local BroadcastModel = ReplicatedStorage:WaitForChild("
   BroadcastModel")
local SendGradients = ReplicatedStorage:WaitForChild("
   SendGradients")

-- minibatch kan worden meegegeven door de parameter
   server of hier gedeclareerd worden
local minibatch = 16

function returnGradient(model)
    -- hier gaan we door het volledige gradient
       berekenings process
    -- we maken de input en verwachte outputs aan
    local inputBatch = ddls.createBatch(minibatch, #model
       [1])
    local truthBatch = ddls.createBatch(minibatch, #model
       [1])

    -- we doen inferentie om predicties te verkrijgen
    local predictionsBatch =
       ddls.modelActivationLinearBatch(model, inputBatch)
```

```
    -- de predicties worden vergeleken met het verwacht
       resultaat
    local loss = ddls.lossFunctionBatch(predictionsBatch,
        truthBatch)

    -- we calculeren gradienten met dit vershil
    local gradients =  ddls.modelGradientLinear(model,
       loss)

    -- we sturen het resultaat terug naar de parameter
       server
    SendGradients:FireServer(gradients)
end

-- als de worker het model krijgt wordt returnGradient()
   uitgevoerd
BroadcastModel.OnClientEvent:Connect(returnGradient)
```

B.1.2 Gecentraliseerde asynchrone planning in Luau

Zie hoofdstuk 2.5.3 voor meer uitleg over deze planning.

PARAMETER SERVER PROGRAMMA

Require: initieel model w

1: Verdeel w
2: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
3: **if** ontvangt gradiënt g^i van worker i **then**
4: Update model w met gradient g^i
5: Verstuur w naar worker i
6: **end if**
7: **end for**

PROGRAMMA VAN DE i^{th} WORKER

Require: training data D^i

1: **for** $t \leftarrow 0, 1, 2, \ldots$ **do**
2: Wacht voor model van de parameter server w
3: Maak mini-batch $x \backsim D^i$
4: Calculeer de gradiënt g^i met mini-batch x
5: Verstuur g^i naar parameter server
6: **end for**

PARAMETER SERVER PROGRAMMA

```
local Players = game:GetService("Players")
local ddls = require(game:GetService('ReplicatedStorage')
   .ModuleScript)
```

```
local BroadcastModel = ReplicatedStorage:WaitForChild("
   BroadcastModel")
local SendGradients = ReplicatedStorage:WaitForChild("
   SendGradients")

local model = ddls.makeModel({10, 1024, 1024, 1024, 10})

function updateModel(player, gradients)
    model = ddls.modelUpdateLinear(model, gradients)
    BroadcastModel:FireClient(player, model)
end

BroadcastModel:FireAllClients(model)
Players.PlayerAdded:Connect(onPlayerAdded)
SendGradients.OnServerEvent:Connect(updateModel)
```

PROGRAMMA VAN DE WORKER

```
local ReplicatedStorage = game:GetService("
   ReplicatedStorage")
local ddls = require(ReplicatedStorage.ModuleScript)
local BroadcastModel = ReplicatedStorage:WaitForChild("
   BroadcastModel")
local SendGradients = ReplicatedStorage:WaitForChild("
   SendGradients")

local minibatch = 16

function returnGradient(model)
local inputBatch = ddls.createBatch(minibatch, #model[1])
local truthBatch = ddls.createBatch(minibatch, #model[1])
local predictionsBatch = ddls.modelActivationLinearBatch(
   model, inputBatch)
local loss = ddls.lossFunctionBatch(predictionsBatch,
   truthBatch)
local gradients =  ddls.modelGradientLinear(model, loss)
SendGradients:FireServer(gradients)
end

BroadcastModel.OnClientEvent:Connect(returnGradient)
```

B.1.3 Gedecentraliseerde synchrone planning in Luau

Zie hoofdstuk 2.5.2 voor meer uitleg over deze planning.

MAIN WORKER PROGRAMMA

Require: initieel model w_0, aantal workers n

$t \leftarrow 0$
loop
  Broadcast model w_t
  Krijg models $w^i_{t+\tau}$ van alle workers
  Combineer alle modellen $w^i_{t+\tau}$ naar een nieuw model $w_{t+\tau}$
  $t \leftarrow t+\tau$
end loop

PROGRAMMA VAN DE n^{th} WORKER

Require: training data D^i

$t \leftarrow 0$
loop
  Wacht voor w_t
  for $\tau \leftarrow 1,2,\ldots\tau$ **do**
    Maak mini-batch $x \backsim D^i$
    Update model $w_{t+\tau}$ met mini-batch x
  end for
  Verstuur model $w^i_{t+\tau}$ naar parameter server
  $t \leftarrow t+\tau$
end loop

PARAMETER SERVER PROGRAMMA

```
local Players = game:GetService("Players")
local ddls = require(game:GetService('ReplicatedStorage')
   .ModuleScript)
local BroadcastModel = ReplicatedStorage:WaitForChild("
   BroadcastModel")
local SendGradients = ReplicatedStorage:WaitForChild("
   SendGradients")

local model = ddls.makeModel({10, 1024, 1024, 1024, 10})

function startAllWorkers()
    BroadcastModel:FireAllClients(model)
    for _, player in ipairs(Players:GetPlayers()) do
        workerStatus[player.Name] = "Working"
    end
end

function checkWorkersDone()
    for _, status in pairs(workerStatus) do
        if status == "Working" then
            return false
        end
```

```
    end
    return true
end

function onWorkerDone(player, gradient)
    table.insert(gradients, gradient)
    workerStatus[player.Name] = "Available"
    if checkWorkersDone() then
        model = ddls.modelUpdateLinear(model, gradients)
        gradients = {}
        startAllPlayers()
    end
end

startAllWorkers()
SendGradients.OnServerEvent:Connect(onWorkerDone)
```

PROGRAMMA VAN DE WORKER

```
local ReplicatedStorage = game:GetService("
   ReplicatedStorage")
local ddls = require(ReplicatedStorage.ModuleScript)
local BroadcastModel = ReplicatedStorage:WaitForChild("
   BroadcastModel")
local SendModel = ReplicatedStorage:WaitForChild("
   SendModel")

local epoch = 10
local minibatch = 16

function returnModel(model)
    for i=1, epoch do
        local inputBatch = ddls.createBatch(minibatch, #
           model[1])
        local truthBatch = ddls.createBatch(minibatch, #
           model[1])
        local predictionsBatch =
           ddls.modelActivationLinearBatch(model,
           inputBatch)
        local loss = ddls.lossFunctionBatch(
           predictionsBatch, truthBatch)
        local gradients =  ddls.modelGradientLinear(model
           , loss)
        model = ddls.modelUpdateLinear(model, gradients)
    end
    SendModel:FireServer(model)
```

```
end

BroadcastModel.OnClientEvent:Connect(returnModel)
```

B.1.4 Gedecentraliseerde asynchrone planning in Luau

Zie hoofdstuk 2.5.4 voor meer uitleg over deze planning.

PARAMETER SERVER PROGRAMMA

Require: initieel model w

loop
  if krijgt download aanvraag van worker i **then**
    verstuurd w naar worker i
  end if
  if krijgt een model vershil van worker i **then**
    Update model w met model verschil
  end if
end loop

PROGRAMMA VAN DE i^{th} WORKER

Require: training data D^i, parameter deling interval τ, initieel model w

for $t^i \leftarrow 0,1,2,\ldots$ **do**
  Maak mini-batch $x \backsim D^i$
  Calculeer de gradiënt g met mini-batch x
  Update model w met gradiënt g
  if $t^i \mod \tau = 0$ **then**
    Download model w van parameter server
    Calculeer model verschil
    Verstuur verschil naar parameter server
    Update model w met verschil
  end if
end for

PARAMETER SERVER PROGRAMMA

```
local Players = game:GetService("Players")
local ddls = require(game:GetService('ReplicatedStorage')
   .ModuleScript)
local BroadcastModel = ReplicatedStorage:WaitForChild("
   BroadcastModel")
local SendGradients = ReplicatedStorage:WaitForChild("
   SendGradients")

local model = ddls.makeModel({10, 1024, 1024, 1024, 10})

function onPlayerAdded(player)
```

```
    BroadcastModel:FireClient(player, model)
end

function updateModel(player, newmodel)
    model = ddls.modelCombine({model, newmodel})
    BroadcastModel:FireClient(player, model)
end

-- Start alle workers
BroadcastModel:FireAllClients(model)

-- Als een nieuwe worker connecteerd, verstuur die dan
   het model
Players.PlayerAdded:Connect(onPlayerAdded)

SendModel.OnServerEvent:Connect(updateModel)
```

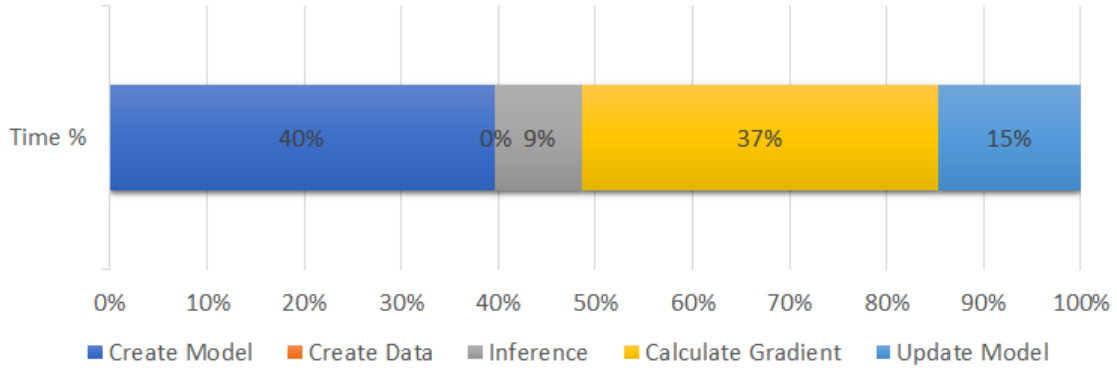
PROGRAMMA VAN DE WORKER

```
local ReplicatedStorage = game:GetService("
   ReplicatedStorage")
local ddls = require(ReplicatedStorage.ModuleScript)
local BroadcastModel = ReplicatedStorage:WaitForChild("
   BroadcastModel")
local SendModel = ReplicatedStorage:WaitForChild("
   SendModel")

local epoch = 10
local minibatch = 16

function returnModel(model)
    for i=1, epoch do
        local inputBatch = ddls.createBatch(minibatch, #
           model[1])
        local truthBatch = ddls.createBatch(minibatch, #
           model[1])
        local predictionsBatch =
           ddls.modelActivationLinearBatch(model,
           inputBatch)
        local loss = ddls.lossFunctionBatch(
           predictionsBatch, truthBatch)
        local gradients =  ddls.modelGradientLinear(model
           , loss)
        model = ddls.modelUpdateLinear(model, gradients)
    end
    SendModel:FireServer(model)
```

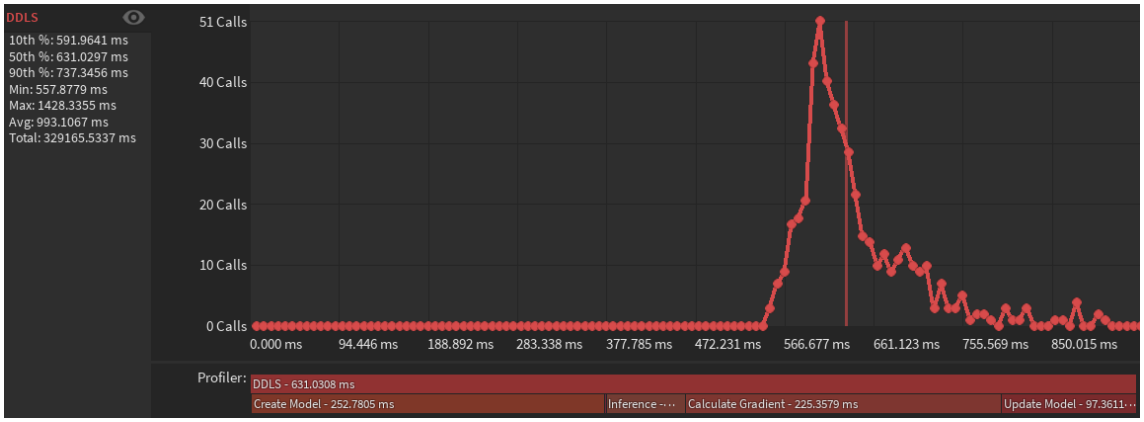
```
end

BroadcastModel.OnClientEvent:Connect(returnModel)
```

B.2 Model update details

We willen nog wat detail geven aan de grafiek in Fig.4.1. Wij geven deze grafiek hieronder nog eens.



Figuur B.1: Het percentage tijd dat elk onderdeel nodig heeft om een volledige model update uit te voeren op een mini-batch van 1 en 1 epoch.



Figuur B.2: De tijd dat elk onderdeel nodig heeft om een volledige model update uit te voeren van het large model op een mini-batch van 1 en 1 epoch.

We kunnen meer details lezen in de tweede grafiek over de hoeveelheid functies dat geroepen worden en de tijd dat elk onderdeel nodig had om uitgevoerd te worden, deze waarden worden geconverteerd naar percentages voor de bovenste grafiek. Geef aandacht aan de totale tijd en de tijd die echt wordt gebruikt om het proces te meten. We kunnen al vaak verder doen met andere functies of processen zelfs als nog niet alle calls verwerkt zijn. Hiermee is het 50th percentiel een goede meeting. voor meer details hierover zie boatbomber (2020).

Bibliografie

1waffle1. (2018, augustus 3). *Text compression*. https://devforum.roblox.com/t/text-compression/163637

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., … Zhang, X. (2016). TensorFlow: A system for large-scale machine learning. *CoRR*, *abs/1605.08695*. http://arxiv.org/abs/1605.08695

Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., Elsen, E., Engel, J. H., Fan, L., Fougner, C., Han, T., Hannun, A. Y., Jun, B., LeGresley, P., Lin, L., … Zhu, Z. (2015). Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *CoRR*, *abs/1512.02595*. http://arxiv.org/abs/1512.02595

Ben-Nun, T. & Hoefler, T. (2018). Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR*, *abs/1802.09941*. http://arxiv.org/abs/1802.09941

boatbomber. (2020, oktober 20). *Benchmarker Plugin - Compare function speeds with graphs, percentiles, and more!* https://devforum.roblox.com/t/benchmarker-plugin-compare-function-speeds-with-graphs-percentiles-and-more/829912

Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J. & Zieba, K. (2016). End to End Learning for Self-Driving Cars. *CoRR*, *abs/1604.07316*. http://arxiv.org/abs/1604.07316

Chen, J., Monga, R., Bengio, S. & Józefowicz, R. (2016). Revisiting Distributed Synchronous SGD. *CoRR*, *abs/1604.00981*. http://arxiv.org/abs/1604.00981

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C. & Zhang, Z. (2015). MXNet: A Flexible and Efficient Machine Learning Library for

Heterogeneous Distributed Systems. *CoRR*, *abs/1512.01274*. http://arxiv.org/abs/1512.01274

Chilimbi, T., Suzue, Y., Apacible, J. & Kalyanaraman, K. (2014). Project Adam: Building an Efficient and Scalable Deep Learning Training System. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 571–582. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

Cirean, D. C., Giusti, A., Gambardella, L. M. & Schmidhuber, J. (2013). Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks. In K. Mori, I. Sakuma, Y. Sato, C. Barillot & N. Navab (Red.), *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013* (pp. 411–418). Springer Berlin Heidelberg.

Clement, J. (2021, april 29). *Roblox Corporation - statistics and facts*. https://www.statista.com/topics/7594/roblox-corporation/

Dai, J. J., Wang, Y., Qiu, X., Ding, D., Zhang, Y., Wang, Y., Jia, X., Zhang, C. L., Wan, Y., Li, Z., Wang, J., Huang, S., Wu, Z., Wang, Y., Yang, Y., She, B., Shi, D., Lu, Q., Huang, K. & Song, G. (2018). BigDL: A Distributed Deep Learning Framework for Big Data. *CoRR*, *abs/1804.05839*. http://arxiv.org/abs/1804.05839

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. & Ng, A. (2012). Large Scale Distributed Deep Networks. In F. Pereira, C. J. C. Burges, L. Bottou & K. Q. Weinberger (Red.), *Advances in Neural Information Processing Systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

EthicalRobot. (2021, maart 11). *Parallel Lua Beta*. https://devforum.roblox.com/t/parallel-lua-beta/1098901

Ho, Q., Cipar, J., Cui, H., Kim, J. K., Lee, S., Gibbons, P. B., Gibson, G. A., Ganger, G. R. & Xing, E. P. (2013). More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, 1223–1231.

Hoefler, T. & Snir, M. (2011). Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. *Proceedings of the International Conference on Supercomputing*, 75–84. https://doi.org/10.1145/1995896.1995909

Huang, G., Liu, Z. & Weinberger, K. Q. (2016). Densely Connected Convolutional Networks. *CoRR*, *abs/1608.06993*. http://arxiv.org/abs/1608.06993

Iandola, F. N., Ashraf, K., Moskewicz, M. W. & Keutzer, K. (2015). FireCaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR*, *abs/1511.00175*. http://arxiv.org/abs/1511.00175

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M. & Tang, P. T. P. (2016). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR*, *abs/1609.04836*. http://arxiv.org/abs/1609.04836

Kingma, D. & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*.

Kironte. (2020, november 13). *Neural Network Library 2.0*. https://devforum.roblox.com/t/neural-network-library-2-0/869557

Langer, M. (2018). *Distributed Deep Learning in Bandwidth-Constrained Environments* (proefschrift).

Langer, M., Hall, A., He, Z. & Rahayu, W. (2018). MPCAăSGDA Method for Distributed Training of Deep Learning Models on Spark. *IEEE Transactions on Parallel and Distributed Systems*, *29*(11), 2540–2556. https://doi.org/10.1109/TPDS.2018.2833074

Langer, M., He, Z., Rahayu, W. & Xue, Y. (2020). Distributed Training of Deep Learning Models: A Taxonomic Perspective. *CoRR*, *abs/2007.03970*. https://arxiv.org/abs/2007.03970

LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

Li, M., Andersen, D. G., Smola, A. J. & Yu, K. (2014). Communication Efficient Distributed Machine Learning with the Parameter Server. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence & K. Q. Weinberger (Red.), *Advances in Neural Information Processing Systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2014/file/1ff1de774005f8da13f42943881c655f-Paper.pdf

Moritz, P., Nishihara, R., Stoica, I. & Jordan, M. I. (2016). SparkNet: Training Deep Networks in Spark.

Roblox. (g.d.). *MicroProfiler*. https://developer.roblox.com/en-us/articles/MicroProfiler

Sutskever, I., Martens, J., Dahl, G. & Hinton, G. (2013). On the Importance of Initialization and Momentum in Deep Learning. *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, III1139III–1147.

Tandon, R., Lei, Q., Dimakis, A. G. & Karampatziakis, N. (2017). Gradient Coding: Avoiding Stragglers in Distributed Learning. In D. Precup & Y. W. Teh (Red.), *Proceedings of the 34th International Conference on Machine Learning* (pp. 3368–3376). PMLR. http://proceedings.mlr.press/v70/tandon17a.html

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., … Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, *575*(7782), 350–354. https://doi.org/10.1038/s41586-019-1724-z

Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A. & Yu, Y. (2015). Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE*, *1*(2), 49–67. https://doi.org/10.1109/TBDATA.2015.2472014

Ye, A. (2021). *Three model compression methods you need to know in 2021*. https://towardsdatascience.com/three-model-compression-methods-you-need-to-know-in-2021-1adee49cc35a

Zhang, K., Alqahtani, S. & Demirbas, M. (2017). A Comparison of Distributed Machine Learning Platforms. *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, 1–9. https://doi.org/10.1109/ICCCN.2017.8038464

Zhang, S., Choromanska, A. & LeCun, Y. (2015). Deep Learning with Elastic Averaging SGD. *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, 685–693.

Zhang, S. (2016). Distributed stochastic optimization for deep learning (thesis).