Het ontbinden van een matrix m.b.v. een getraind neuraal netwerk

Technische Wiskunde Bacheloreindproject

Susan Veldkamp

26 augustus 2020

TU Delft Delft University of Technology

Het ontbinden van een matrix m.b.v. een getraind neuraal netwerk

Technische Wiskunde Bacheloreindproject

door

Susan Veldkamp

ter verkrijging van de graad van Bachelor of Science
aan de Technische Universiteit Delft,
online te verdedigen op vrijdag 24 juli om 11:00 uur.

Studentnummer:	4613775	
Projectduur:	20 april 2020 – 18 juli 2020	
Bepcommissie:	dr. M. Möller	TU Delft, begeleider
	dr. I. Van Gennip,	TU Delft, contactpersoon

TU Delft

Samenvatting

Voor het rekenen met matrices is het handig om ze op een bepaalde manier te ontbinden, zoals in de LU-decompositie. Dit bespaart een computer namelijk veel rekenkracht en dus tijd. Er bestaan computerprogramma's die matrices op meerdere manieren kunnen ontbinden. Dit verslag beantwoordt echter de vraag of het ook op een andere manier kan: namelijk met behulp van het trainen van een neuraal netwerk. In dit verslag staat omschreven hoe met een neuraal netwerk LU-decomposities van 2x2 matrices gevonden kunnen worden.

Door handig gebruik te maken van de structuur van een neuraal netwerk, en een passende optimizer, leerstap, en lossfunctie te kiezen, kan een neural netwerk inderdaad snel leren om een LU-ontbinding van een matrix te vinden. Hiermee is aangetoond dat klassieke algortimes zoals LU-decomposities kunnen worden gereproduceerd met getrainde neurale netwerken.

Voorwoord

Ik vond het erg leuk om aan dit onderzoek te werken, naarmate ik er meer van begreep werd het steeds interessanter. Ik heb het gevoel dat ik nu een goed begrip van de basis van machine learning heb!

Voor alle hulp, wekelijkse Skype-gesprekken en zijn enthousiasme gedurende het hele project, wil ik mijn begeleider Matthias Möller, hartelijk bedanken.

Dit verslag was 18 juli 2020 afgerond, sindsdien zijn alleen enkele typo's verbeterd.

Inhoudsopgave

1 Inleiding

Aan de basis van dit onderzoek staat het trainen van een neuraal netwerk.

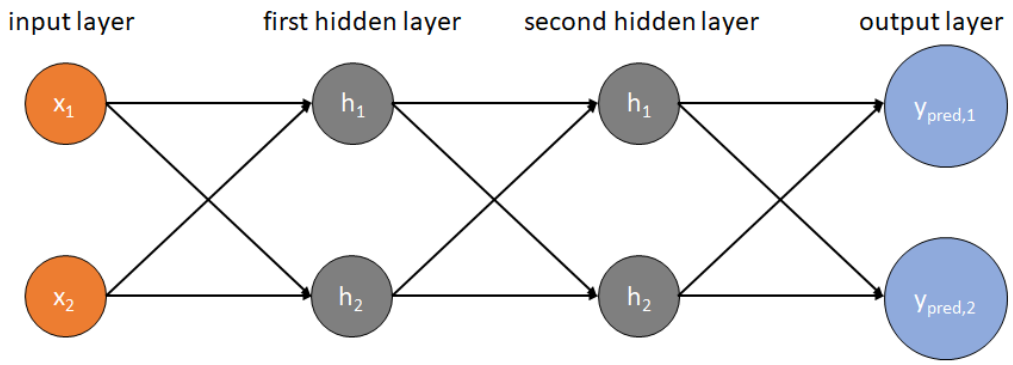
1.1 Neurale netwerken

Een neuraal netwerk is een aaneenschakeling van neuronen. Het bestaat altijd uit de volgende lagen:

- de "input layer", de laag die de input bevat. De input is een vector $\mathbf{x}$.
- één of meer "hidden layers", verborgen lagen die neuronen bevatten waarin berekeningen worden uitgevoerd op de inputvector $\mathbf{x}$
- de "output layer", de laag die de output bevat. Dit is een vector $\mathbf{y}_{pred}$ (voorspelde $\mathbf{y}$) die ook wel wordt aangeduid als *label*.

Kort gezegd neemt een neuraal netwerk dus een input op, en dit wordt doorgevoerd naar de neuronen in de verborgen lagen, die er enkele berekeningen op doen. Dan spuwt het netwerk een output uit. Het bijzondere aan neurale netwerken is dat ze getraind kunnen worden om een gewenste output te geven. Hóe het netwerk getraind wordt, staat in hoofdstuk 1.2. Voor nu is het echter belangrijk om te onthouden dat er bij elke input een "correcte/gewenste" output bestaat, $\mathbf{y}$. Ter illustratie: een veel gebruikte toepassing van neurale netwerken is een classifier, die als input een foto van een kat of hond krijgt, en als output zegt wat voor dier het is. In dit geval is de correcte output 'kat' als de input een foto van een kat is, en 'hond' in het geval dat de input een foto van een 'hond' is (of 'geen van beide' als het een foto zonder kat of hond betreft).

In dit onderzoek is gebruik gemaakt van een neuraal netwerk zoals te zien in Figuur 1, met twee verborgen lagen die elk twee neuronen bevatten. Hierbij wordt een input ter grootte van twee elementen meegegeven aan het neurale netwerk. Na de berekeningen in de verborgen lagen komt er een output uit die ook bestaat uit twee elementen.



Figuur 1: Het neurale netwerk

De berekeningen hebben de volgende vorm:

$$\sigma(\mathbf{x}^\top W^\top + \mathbf{b}^\top)$$

Hierbij is

- $\mathbf{x}$ de inputvector,
- $\mathbf{b}$ de zogenaamde *bias*,
- W een gewichtenmatrix,
- $\sigma(\cdot)$ de zogenaamde *activatiefunctie*.

Activatiefunctie

De activatiefunctie $\sigma(\cdot)$, heeft zijn naam te danken aan het feit dat deze functie bepaalt of de input die het krijgt moet worden meegenomen in het netwerk, of hij "geactiveerd"moet worden. Er wordt dus bepaald of de input relevant is voor de voorspelling die het netwerk aan het eind doet. Een mogelijke activatiefunctie is bijvoorbeeld $\sigma(x) = \max(0, x)$. Wanneer x negatief is, komt er 0 uit de activatiefunctie, en wordt de input dus als het ware als irrelevant bestempeld [1]. Als het groter is dan 0, wordt de input wel meegenomen. Daarnaast heeft een activatiefunctie ook vaak als functie om de input te normaliseren, en bijvoorbeeld naar getallen tussen de 0 en 1 te transporteren.

Bias

Met de bias $\mathbf{b}$ kan de waarde getransporteerd worden. Dit kan helpen om sommige inputs, die in het algemeen als irrelevant zouden worden gezien

door activatiefuncties, wel relevant te maken. Zoals met de activatiefunctie $\sigma(x) = \max(0, x)$, waarbij -5 naar 0 gestuurd zou worden, tenzij de bias bijvoorbeeld op $+6$ wordt gezet. Daarnaast geeft het hebben van een bias extra flexibiliteit, want met een bias kunnen er meer waarden bereikt worden met de activatiefunctie, waarden die misschien dicht bij de gewilde output liggen.

Welnu, de gewichtenmatrix W wordt vermenigvuldigd met de input $\mathbf{x}$ en neemt zo de input mee in de berekeningen. Zoals eerder vermeld maakt het neurale netwerk uit dit onderzoek gebruik van twee verborgen lagen. Deze hebben allebei hun eigen activatiefunctie σ en een eigen gewichtenmatrix W. De berekening die een neuraal netwerk van de beschreven vorm doet met de input $\mathbf{x}$, en daarmee de output $\mathbf{y}_{pred}$, wordt in totaal dus zo:

$$\mathbf{y}_{pred} = \sigma_2\left((\sigma_1(\mathbf{x}^\top W_1^\top + \mathbf{b}_1^\top))W_2^\top + \mathbf{b}_2^\top\right).$$

1.2 Het trainen van een neuraal netwerk

Wanneer er aan het begin een input $\mathbf{x}$ aan het netwerk wordt gegeven, zal $\mathbf{y}_{pred}$ nog niet gelijk zijn aan de correcte waarde $\mathbf{y}$. Aan het begin zijn de gewichtenmatrices W_1 en W_2 en de bias $\mathbf{b}_1$ en $\mathbf{b}_2$ namelijk willekeurige matrices en vectoren, dus

$$\mathbf{y} \neq \mathbf{y}_{pred} = \sigma_2\left((\sigma_1(\mathbf{x}^\top W_1^\top + \mathbf{b}_1^\top))W_2^\top + \mathbf{b}_2^\top\right).$$

Om dit te bereiken moet het netwerk 'getraind' worden. En om een neuraal netwerk te trainen zijn twee stappen nodig.

1. Het neurale netwerk definiëren met leerbare parameters

Ten eerste moet het neurale netwerk gedefinieerd zijn. Dit wordt ook wel de architectuur van het netwerk genoemd. Er moeten activatiefuncties gekozen worden en het aantal verborgen lagen en het aantal neuronen in die lagen moet bepaald worden. Hierbij zijn de 'leerbare' parameters de gewichtenmatrices W_i en de bias $\mathbf{b}_i$. Het netwerk zal namelijk leren wat die parameters moeten zijn om een output te geven die dicht zit bij de gewenste output.

2. Een dataset van inputs langsgaan

Nu het netwerk gedefinieerd is, begint het leerproces. Dit bestaat uit een for-loop waarin de onderste 4 stappen telkens achter elkaar herhaald worden.

1. **Input door het netwerk laten verwerken**
   De input $\mathbf{x}$ gaat het netwerk binnen en er komt een output

$$\mathbf{y}_{pred} = \sigma_2\left((\sigma_1(\mathbf{x}^\top W_1^\top + \mathbf{b}_1^\top))W_2^\top + \mathbf{b}_2^\top\right)$$

   uit.

2. **De loss berekenen**
   Er wordt bepaald wat de fout is tussen de output van het netwerk $\mathbf{y}_{pred}$ en de correcte waarde $\mathbf{y}$. Deze fout wordt de "loss"genoemd, en kan op verschillende manieren worden uitgerekend. Één manier is de *mean squared error*, die als volgt gedefinieerd is:

$$\text{MSE}(\mathbf{y}, \mathbf{y}_{pred}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_{pred,i})^2$$

   Hierbij zijn $\mathbf{y}$ en $\mathbf{y}_{pred}$ vectoren van lengte n.

3. **Gradiënten van loss ten opzichte van de parameters uitrekenen**
   Als de loss groot is, is de fout die het netwerk maakt dus groot. Het doel van het leerproces is om de loss zo klein mogelijk te maken. Om te bepalen welke parameters er veranderd moeten worden om de loss kleiner te maken, moet de lossfunctie naar elke parameter apart worden afgeleid. Is bijvoorbeeld de lossfunctie afgeleid naar het eerste kental van de gewichtenmatrix W_1, $w_{1,11}$, negatief, dus

$$\frac{\partial(\text{Loss})}{\partial w_{11}} < 0,$$

   dan houdt dit in dat wanneer w_{11} wordt vergroot, de loss kleiner wordt. Andersom geven *positieve* partiële afgeleiden blijk van het feit dat het *verkleinen* van die variabele de loss kleiner zou maken.

   Er moeten dus veel partiële afgeleiden uitgerekend worden. Dit kost veel computergeheugen, vooral wanneer de lossfunctie ingewikkelder wordt en de matrices groter in dimensie en aantallen. Hier is een algoritme voor bedacht dat *backpropagation* heet, dat het beste kan worden toegelicht met een voorbeeld uit het boek *Mathematics for Machine Learning* [2].

Beschouw de functie

$$f(x) = \sqrt{x^2 + \exp(x^2)}. \tag{1}$$

Het bespaart berekeningen om deze functie in delen op te slaan, als volgt:

$$\begin{aligned} a &= x^2 \\ b &= \exp(a) \\ c &= a + b \\ f &= \sqrt{c} \end{aligned}$$

Dit kost al minder operaties dan een directe implementatie van de functie (1). De afgeleiden van deze deelfuncties naar de variabelen waar ze van afhankelijk zijn, zijn gemakkelijk te bepalen:

$$\begin{aligned} \frac{\partial a}{\partial x} &= 2x \\ \frac{\partial b}{\partial a} &= \exp(a) \\ \frac{\partial c}{\partial a} &= 1 = \frac{\partial c}{\partial b} \\ \frac{\partial f}{\partial c} &= \frac{1}{2\sqrt{c}} \end{aligned}$$

Na toepassing van de kettingregel volgt:

$$\begin{aligned} \frac{\partial f}{\partial b} &= \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} \\ \frac{\partial f}{\partial a} &= \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} \\ \frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} \end{aligned}$$

Om de afgeleide $\frac{\partial f}{\partial x}$ te bepalen moet er teruggegaan worden naar $\frac{\partial f}{\partial a}$, en daarvoor moet weer teruggegaan worden naar $\frac{\partial f}{\partial b}$. Dit heet backpropagation.

Door de gradiënten te bepalen met behulp van backpropagation kan er dus met niet te veel geheugenbelasting bepaald worden aan welke parameters gesleuteld moet worden om in het vervolg een kleinere loss en dus een beter antwoord uit het neurale netwerk te krijgen.

4. **De parameters van het netwerk updaten**
   Nu de gradiënten van de lossfunctie ten opzichte van de verschillende parameters bekend zijn, kunnen ze aangepast worden. De meest voor de hand liggende manier om dit te doen is als volgt:

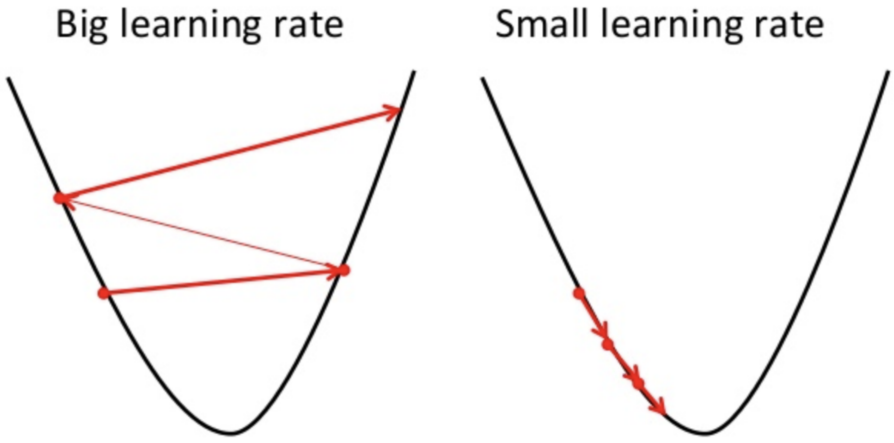
$$w^{(i+1)} = w^{(i)} - \alpha \frac{\partial(\text{Loss})}{\partial w} \tag{2}$$

   Waarbij

   - w een aan te passen parameter is
   - i aangeeft in de hoeveelste iteratie van de for-loop we zitten
   - α de *learning rate* is
   - $\frac{\partial(\text{Loss})}{\partial w}$ de gradiënt van de lossfunctie ten opzichte van w is.

Learning rate

De learning rate α bepaalt hoe groot de stap die in de goede richting gezet wordt, is. Het is belangrijk om een juiste learning rate te kiezen. Als α te klein is, duurt het namelijk misschien heel lang voordat het minimum van de lossfunctie bereikt is, maar als α te groot is, worden fouten misschien te hard verbeterd en kan er over het minimum heen gestapt worden, zoals te zien in Figuur 2. Omdat het teken voor α negatief is, zorgt een negatieve gradiënt ervoor dat de parameter w de volgende iteratie groter is, en vice versa. Zo wordt de loss kleiner.



Figuur 2: Verschil grote en kleine learning rate [3]

Nu bepaald is hoe de parameters moeten worden aangepast, worden ze veranderd. Dit kan met de hand worden geïmplementeerd, zoals in vergelijking 2, maar er kan ook een algoritme worden aangeroepen dat dit doet, een zogenaamde *optimizer*. Daarover staat meer in hoofdstuk 2.2.1.

Het moge duidelijk zijn dat, in theorie, naarmate bovenstaande stappen doorlopen worden, het de bedoeling is dat de lossfunctie uiteindelijk klein zal zijn. De output $\mathbf{y}_{pred}$ zal dan voor de set van inputs waarop het neurale netwerk in de for-loop op getraind is, dicht bij het juiste antwoord $\mathbf{y}$ zitten. Op zo'n moment is het neurale netwerk voldoende getraind, en geeft het bijna altijd het juiste antwoord.

Toch is het niet zo simpel als het lijkt, want er zijn tal parameters die gekozen moeten worden. Voor de toepassing van dit onderzoek, het vinden van LU-decomposities van 2×2-matrices, zijn er veel opties om de architectuur van het neurale netwerk in te richten. Zo zijn er om te beginnen al oneindig veel mogelijkheden om de begingewichten voor de matrix W_1 en W_2 te kiezen, en bestaan er veel verschillende optimizers die het allemaal misschien net iets beter of slechter doen dan de ander. Dit verslag houdt zich bezig met het vinden van een juiste samenstelling van voorgaande parameters, om LU-decomposities van 2×2-matrices te vinden.

2 LU-decompositie met een getraind 'neural network'

2.1 LU-decomposities: bestaan en uniekheid

Een inverteerbare matrix A kan op LU-wijze ontbonden worden als volgt

$$A = LU$$

met L een onderdriehoeksmatrix en U een bovendriehoeksmatrix, dan en alleen dan als de leidende hoofdminoren van A niet gelijk aan nul zijn [4]. De leidende hoofdminoren zijn de determinanten van de kleinere matrices die overblijven na het schrappen van één of meer buitenste rijen en kolommen aan de rechter- en onderkant.

Wanneer je een matrix op zodanige wijze kunt ontbinden wordt het makkelijk om stelsels zoals $Ax = b$ op te lossen. Dan geeft $LUx = b$ namelijk

$$\begin{cases} Ly = b \\ Ux = y. \end{cases}$$

Omdat L en U driehoekmatrices zijn is de oplossing snel gevonden.

LU-decomposities zijn niet uniek. Echter, als wordt aangenomen dat de elementen op de diagonaal van L of U gelijk aan 1 zijn, volgt wel dat de decompositie uniek is [4].

Bij het trainen van een neuraal netwerk om een decompositie te vinden, zullen alle matrices dus elke keer dat het neurale netwerk getraind wordt en bijvoorbeeld 5000 leeriteraties doorloopt, op een andere wijze ontbonden worden door het neurale netwerk. Zo ook de matrix

$$A = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix},$$

die dit hele onderzoek gebruikt wordt om te ontbinden.

2.2 De implementatie

Om een neuraal netwerk dat LU-decomposities vindt te implementeren, was als basis een kort programma genomen, dat geschreven was in Python en

gevonden op Stack Exchange [5]. Dat programma trainde een neuraal netwerken om oplossingen van de vergelijking ax^2+bx+c te vinden. Een aantal dingen zijn aangepast aan dit programma om het toepasbaar te maken op matrices. De aangepaste code, die als basis dient voor alle plots die in dit verslag zijn opgenomen, is te vinden in Appendix A. In Appendix B wordt de code kort doorgenomen.

Beschouw ten eerste nog even de output die een neuraal netwerk met twee verborgen lagen geeft:

$$\mathbf{y}_{pred} = \sigma_2\left((\sigma_1(\mathbf{x}^\top W_1^\top + \mathbf{b}_1^\top))W_2^\top + \mathbf{b}_2^\top\right).$$

Voor het doel van dit onderzoek worden de twee bias gelijk aan nul gesteld, $\mathbf{b}_1 = \mathbf{b}_2 = \mathbf{0}$, en de activatiefuncties $\sigma_1(\cdot)$ en $\sigma_2(\cdot)$ worden als de identiteitsfunctie genomen. Op deze manier wordt de berekening als volgt:

$$\mathbf{y}_{pred} = \mathbf{x}^\top W_1^\top W_2^\top.$$

Zo is het namelijk mogelijk om een LU-decompositie te maken. Immers, als het netwerk als input $\mathbf{x}$ krijgt, maar de juiste output $\mathbf{y} = \mathbf{x}^\top A$ is, dan zal het netwerk getraind worden om zo dicht mogelijk bij $\mathbf{y}$ te komen, en dus

$$\mathbf{x}^\top A \approx \mathbf{x}^\top W_1^\top W_2^\top.$$

Als nu afgedwongen kan worden dat $W_1^\top$ een onderdriehoeksmatrix en $W_2^\top$ een bovendriehoeksmatrix is, dan wordt er uiteindelijk bereikt dat

$$\begin{aligned} A &\approx W_1^\top W_2^\top \\ A &\approx LU. \end{aligned}$$

Daarom is de lossfunctie gedefinieerd als

$$\text{MSEloss}(\mathbf{y}, \mathbf{y}_{pred}) + W_{1,21}^2 + W_{2,12}^2.$$

Hiermee wordt het afgestraft wanneer de waarden in L en U die 0 moeten zijn, groter of kleiner dan 0 zijn.

2.2.1 Gevoeligheidsanalyse

Er zijn veel knoppen waar aan gedraaid kan worden om het programma zo mogelijk te verbeteren. Daarom is er een gevoeligheidsanalyse gedaan, waarbij telkens één variabele veranderd is en de rest hetzelfde is gehouden. Afhankelijk van hoe de loss daarmee kleiner werd, is telkens een keuze gemaakt om verder te gaan met combinaties van optimizers en andere factoren

die snel een kleine loss produceerden en niet te wisselvallig waren. Er is onder andere gekeken naar de gekozen optimizer, de learning rate bij aanvang en de initiële gewichtenmatrices, en combinaties daarvan.

Optimizers

Om de gewichten aan te passen moeten de gradiënten berekend worden en moet er daarna bepaald worden hoe de gewichten worden veranderd. Dit kan met de hand geïmplementeerd worden, maar er kan ook gebruik gemaakt worden van zogenaamde *optimizers*. Dit zijn algoritmes die, wanneer ze worden aangeroepen, de gewichten updaten. Er zijn verschillende soorten algoritmes die dit allemaal op net een andere manier doen. In dit onderzoek worden een aantal van dit soort algoritmes bekeken: de veel gebruikte optimizer Stochastic Gradient Descent (SGD) [6], een wat nieuwere en effectieve optimizer Adaptive Moment Estimator (Adam) [7] en een aantal optimizers die daar op lijken.

Stochastic Gradient Descent (SGD)
SGD is de simpelste optimizer. Hij rekent de gradiënten uit en past de parameters als volgt aan [8]:

$$w^{(i+1)} = w^{(i)} - \alpha \cdot \frac{\partial L(\mathbf{y}, \mathbf{y}_{pred})}{\partial w}.$$

Hierbij is

- $w^{(i+1)}$ de aan te passen parameter in de (i+1)'de iteratie;
- α de learning rate;
- $\frac{\partial L(\mathbf{y}, \mathbf{y}_{pred})}{\partial w}$ de afgeleide van de lossfunctie in $\mathbf{y}^{(i)}$ en $\mathbf{y}^{(i)}_{\text{pred}}$ ten opzichte van w.

Wanneer de gradiënt ten opzichte van w positief is, houdt dit in dat wanneer w groter wordt gemaakt, de lossfunctie ook groter wordt. In dit geval wordt er van w wat afgetrokken, en w wordt dus kleiner, wat als resultaat heeft dat de loss ook een stukje kleiner wordt. Wanneer de gradiënt negatief is, werkt het precies andersom.

Averaged Stochastic Gradient Descent (ASGD)
ASGD werkt hetzelfde als SGD, alleen houdt het programma ook bij wat de

parameters in elke iteratie zijn, en uiteindelijk zet hij elke parameter w als het gemiddelde van de voorgaande gewichten [9]. Dus

$$w_{final} = \bar{w} = \frac{1}{t}\sum_{i=0}^{t-1} w^{(i)}.$$

Hierbij is t het aantal iteraties of epochs. Het voordeel hiervan is dat wanneer de gewichten blijven oscilleren rond het minimum, ze uiteindelijk toch zeer dicht bij dat minimum worden gekozen [10].

Resilient Backpropagation (Rprop)
Rprop is anders van de voorgaande optimizers in het opzicht dat het niet de gehele gradiënt gebruikt om parameters aan te passen, maar alleen het teken [11]. Daarnaast houdt het niet constant dezelfde learning rate aan, maar kent aan elke parameter een aparte learning rate toe en past deze ook aan gedurende het leerproces. Een voordeel hiervan is dat wanneer één van de parameters al dicht bij het optimum zit maar andere niet, ze niet allemaal met een grote leerstap aangepast hoeven te worden, wat als gevolg kan hebben dat die eerste parameter weer verslechtert.

Root Mean Square Propagation (RMSprop)
RMSprop past de learning rate aan door het te delen door de wortel van het voortschrijdende gemiddelde van de kwadrateerde gradiënten. Dat ziet er zo uit:

$$w^{(i)} = w^{(i-1)} - \frac{\alpha}{\sqrt{E[g^2]_t}}\frac{\partial L(\mathbf{y}, \mathbf{y}_{pred})}{\partial w}$$
$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)\left(\frac{\partial L(\mathbf{y}, \mathbf{y}_{pred})}{\partial w}\right)^2$$

Hierbij is $E[g^2]$ het voortschrijdende gemiddelde. Dit houdt in dat het gemiddelde wordt genomen van alle gekwadrateerde gradiënten van het gewicht w tot nu toe. Verder geldt

- α is de learning rate
- β is de voortschrijdende-gemiddelde-parameter
- $\frac{\partial L(\mathbf{y}, \mathbf{y}_{pred})}{\partial w}$ is de afgeleide van de lossfunctie naar w.

Het idee hiervan is dat, door het voortschrijdende gemiddelde te gebruiken, de learning rate telkens gedeeld wordt door een ongeveer hetzelfde getal voor

elk gewicht w.[12]

Adaptive Gradient Algorithm (Adagrad)
Adagrad is een optimizer die ook voor elke parameter een eigen learning rate heeft. Het kent een kleine learning rate toe aan parameters die bij de meeste iteraties hetzelfde zijn, en een grote learning rate aan parameters die (nog) niet veel zijn veranderd gedurende de voorgaande iteraties. Hiervoor gebruikt het de gradiënten die in het verleden zijn uitgerekend voor die parameters. Een nadeel van dit algoritme is dat het op een gegeven moment de learning rates heel klein maakt, waardoor het netwerk niet meer veel beter kan worden [8].

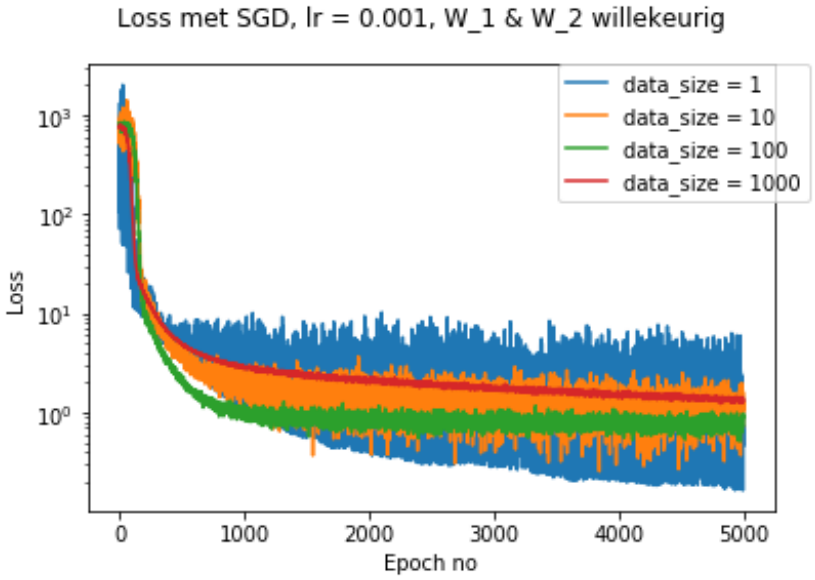
Adaptive Moment Estimation (Adam)
Adam houdt ook aparte learning rates bij voor de verschillende parameters. Daarnaast deelt het net als RMSprop de learning rate door de wortel van het voortschrijdende gemiddelde van de gradiënten (dus het eerste moment) en daarnaast maakt het ook nog gebruik van het tweede moment van de gradiënten. De bedenkers van de Adam-optimizer hebben empirisch aangetoond dat hij het heel goed doet [8].

Elke iteratie trainen op een lijst van inputs

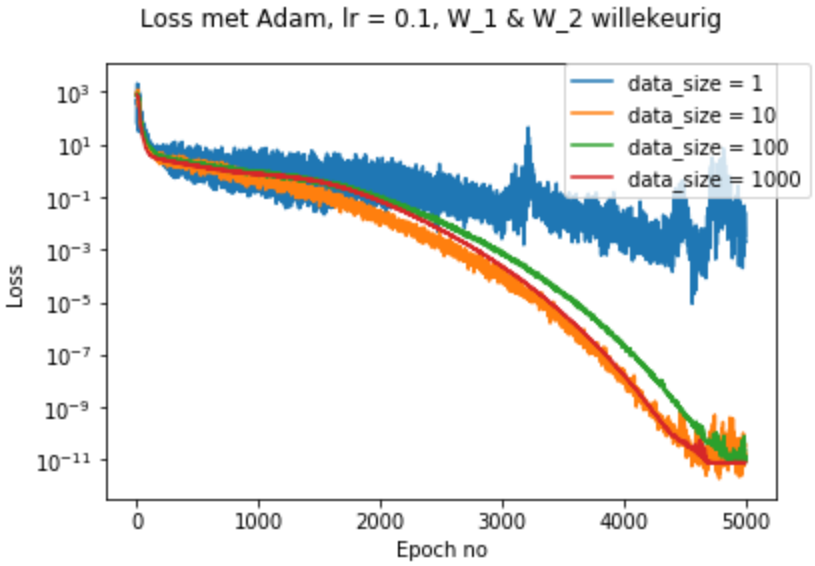
Wat wel wordt gebruikt om het neurale netwerk te trainen (en ook in Appendix B kort wordt toegelicht), maar nog niet in het hoofdverslag beschreven is, is dat het ook mogelijk is voor het neurale netwerk om elke iteratie in de for-loop niet alleen van één input en output te leren, maar van een lijst met een aantal inputs en outputs. De grootte van deze lijst wordt in de code aangeduid met `data_size`.[1] Het voordeel hiervan is dat bij het berekenen van de gradiënten van de loss er veel inputs worden meegenomen, voordat de gewichten en bias worden veranderd. Hierdoor zullen er minder snel te grote stappen worden genomen die ervoor zorgen dat de gewichten wel goed bij één bepaalde input passen, maar niet een volgende input. Het nadeel is dat het langer voor de computer duurt om te rekenen. Om het aantal plots in dit verslag enigszins in te perken is er alleen voor de optimizers SGD en Adam te zien hoe de grootte van `data_size` de loss beïnvloed, maar andere optimizers die later besproken worden reageerden soortgelijk op de aanpassing. Te zien in Figuur 3 en 4 is dat de loss veel minder gaat schommelen,

[1]Wanneer deze lijst groter is dan 1, wordt MSEloss toegepast als de L^2-norm, die werkt op matrices. De lijst van vectoren $\mathbf{y}$ en $\mathbf{y}_{pred}$ wordt dan gezien als een matrix met dimensies `data_size` bij 2.

en vooral bij de optimizer Adam wordt in de stap tussen `data_size` = 1 en 2 de loss veel minder. Conclusie: voortaan wordt er gebruik gemaakt van `data_size` = 100, want vanaf daar verbetert het niet heel erg meer, maar kost het wel een stuk meer tijd om uit te rekenen.



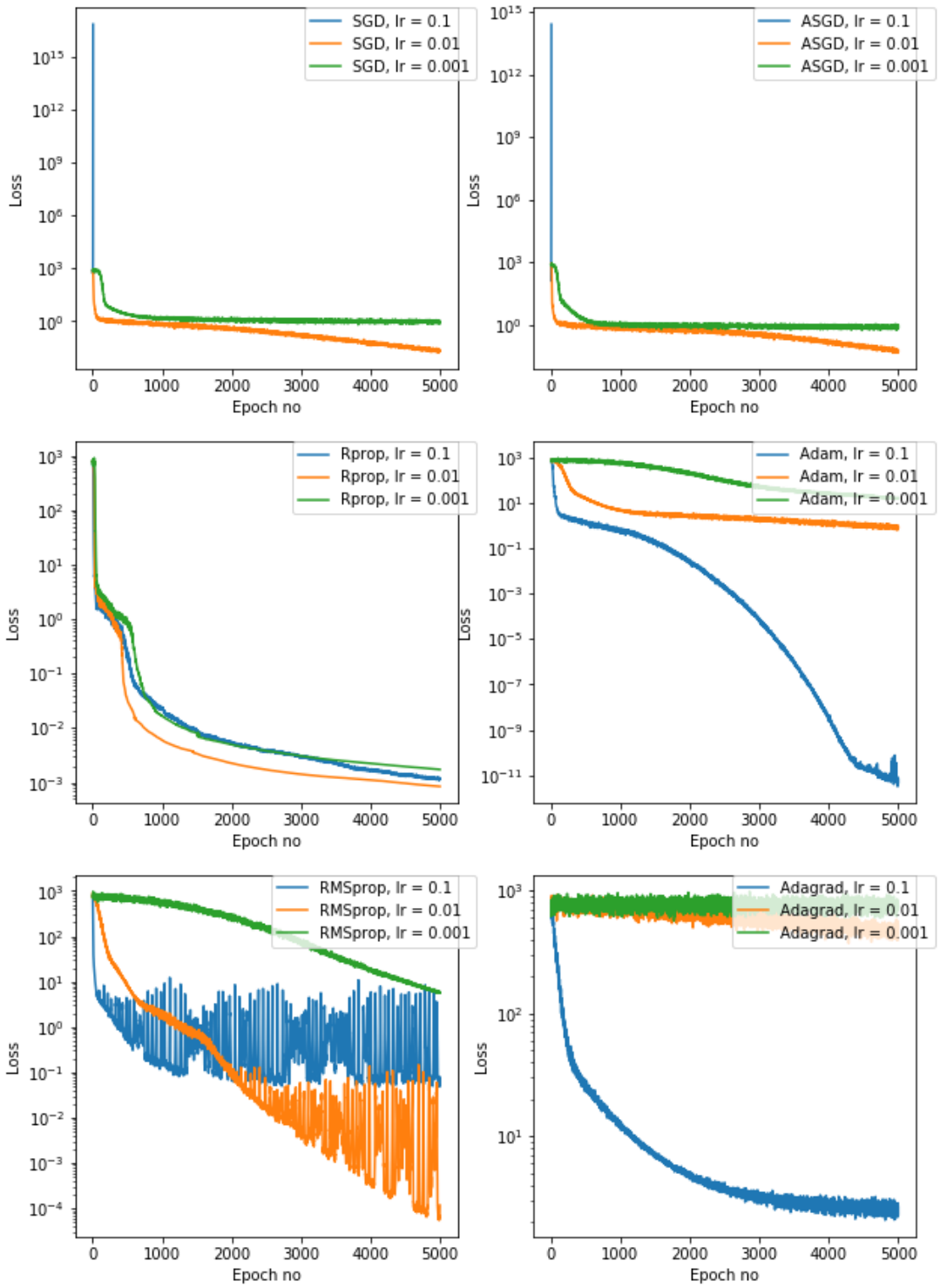
Figuur 3: Verloop van loss met optimizer SGD en verschillende `data_size`



Figuur 4: Verloop van loss met optimizer Adam en verschillende `data_size`

Learning rate

Uit Figuur 5 blijkt dat het verloop van de loss per optimizer zeer beïnvloed wordt door een andere initiële learning rate. De y-as is logaritmisch geschaald om beter te kunnen zien wat er met de loss gebeurt als hij dicht bij 0 komt. Bij SGD en ASGD worden de leerstappen gaandeweg niet aangepast, en deze optimizers doen het ook alleen bij de leerstap 0.001 en 0.01. De learning rate 0.1 is veel te groot, daarbij explodeert de loss zelfs.



Figuur 5: Loss van zes verschillende optimizers met drie verschillende initiële learning rates, `data_size` = 100, W_1 en W_2 willekeurig, met logaritmische y-as

De loss bij Rprop is niet heel erg beïnvloed door een verschil in ordegrootte van learning rates. Adam wel, en gaat keihard naar beneden bij een learning rate van 0.001.

De loss met RMSprop schommelt heel hard bij een learning rate groter dan 0.001, maar gaat wel aan het begin best snel naar beneden.

Het is opvallend dat de loss met de optimizer Adagrad niet erg goed wordt afgebouwd bij learning rates 0.01 en 0.001, maar wel hard kleiner wordt met een learning rate 0.1.

Initiële matrices

Van alle optimizers wordt de beste learning rate aan hun toegekend. Behalve bij RMSprop was er namelijk telkens één learning rate het beste. Voor RMSprop worden de learning rates 0.01 en 0.001 allebei nog even in de race gehouden.

Er zijn drie vormen die het meest voor de hand liggen om te kiezen als beginwaarde voor W_1 en W_2. De eerste vorm, die Python standaard toepast, is de matrices te vullen met willekeurige getallen met een waarde tussen -1 en 1.

De tweede mogelijkheid is om de matrices te vullen met het gemiddelde van de matrix A die ontbonden moet worden. Zo zijn de getallen in de gewichtenmatrices van dezelfde ordegrootte als A. In dat geval worden W_1 en W_2 dus

$$W_1 = W_2 = \begin{pmatrix} 25 & 25 \\ 25 & 25 \end{pmatrix}$$

De laatste vorm is om één gewichtenmatrix als de eenheidsmatrix te kiezen en de andere als $A^\top$, dus

$$W_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, W_2 = \begin{pmatrix} 10 & 30 \\ 20 & 40 \end{pmatrix}$$

Nu geldt namelijk

$$W_1^\top W_2^\top = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix} = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix} = A.$$

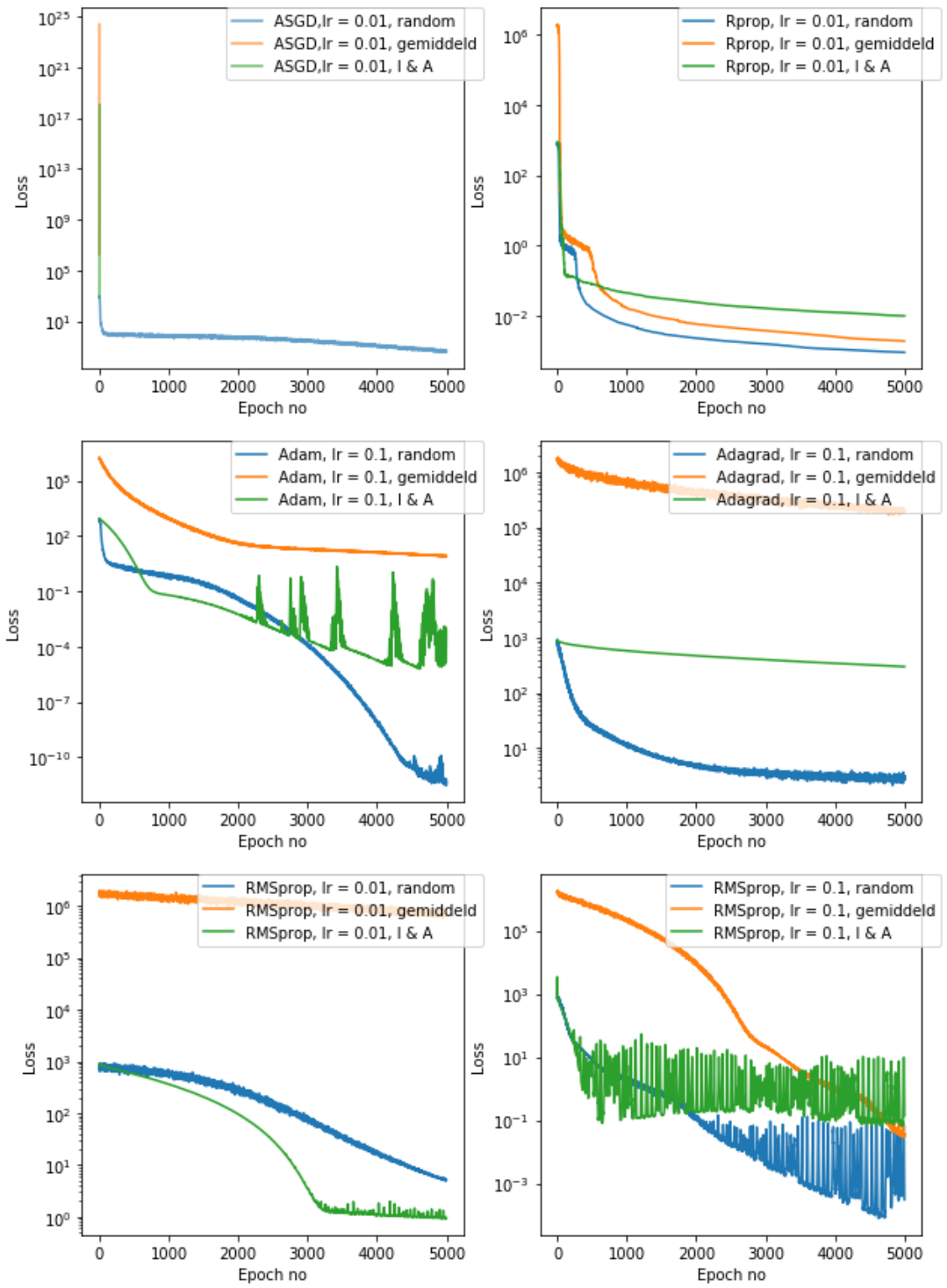
Daarnaast voldoet nu $W_1^\top$ aan het criterium een onderdriehoeksmatrix te zijn, dus de loss zal niet al te groot zijn aan het begin.

De resultaten zijn te vinden in Figuur 6. Merk op dat SGD uit de reeks is gehaald, deze is namelijk bijna hetzelfde als ASGD. Rprop staat er twee keer in.

De blauwe lijn komt bijna altijd het laagste uit. Zo blijkt dus dat Python goede standaardinstellingen heeft. Daarnaast volgt hieruit dat de intuïtie niet klopt dat het helpt om te beginnen met W_1 en W_2 van dezelfde ordegrootte als A, en ook niet om te beginnen met twee gewichtenmatrices die al beginnen met een kleine loss. Wel moet gezegd worden dat laatstgenoemde beter werkt dan de matrices met gemiddelden. Daarnaast is er een optimizer waarbij het beter werkt om te beginnen met I en $A^\top$ als initiële gewichtenmatrices: RMSprop met een learning rate van 0.01.

Merk verder op dat de optimizer ASGD niet goed functioneert bij de gemiddeldenmatrices en $W_1 = I$, $W_2 = A^\top$, na enkele epochs schiet de loss de hoogte in. Verder kan Rprop gek genoeg wel goed omgaan met de eerste twee opties, maar minder met $W_1 = I$, $W_2 = A^\top$. De loss bij Adam schudt ook flink heen en weer wanneer er gekozen wordt voor die optie. Ook blijkt dat de loss van RMSprop hevig fluctueert bij de Python default-optie.

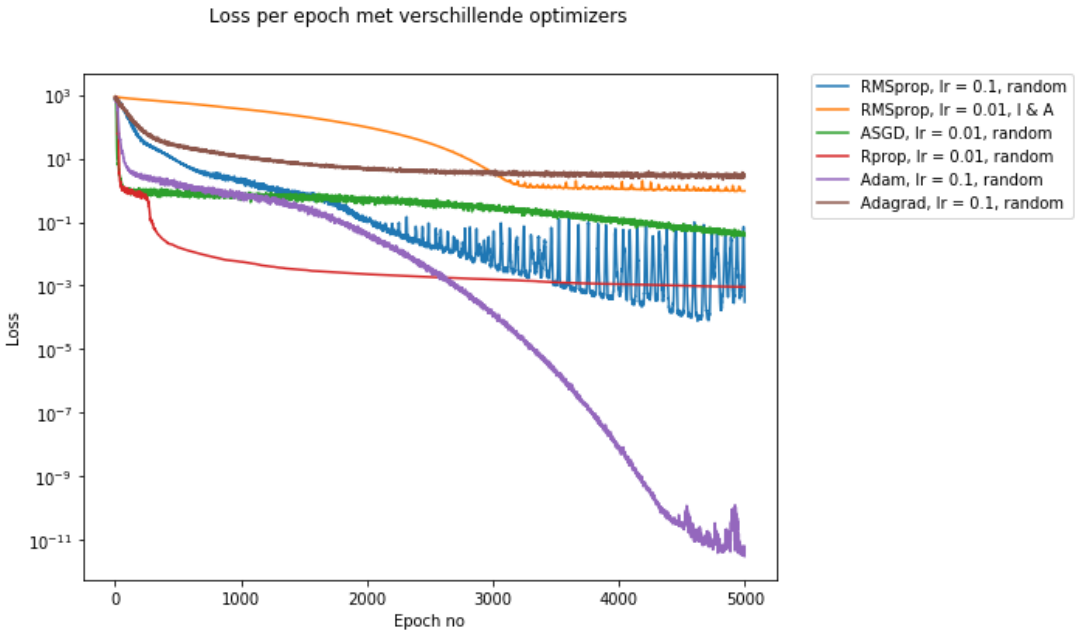
Al met al is het dus het beste om W_1 en W_2 met willekeurige getallen tussen de -1 en 1 te laten beginnen, behalve bij RMSprop, en dat is ook wat er wordt meegenomen naar de laatste proef, waarbij de optimizers tegen elkaar worden afgezet en de beste wordt gekozen.



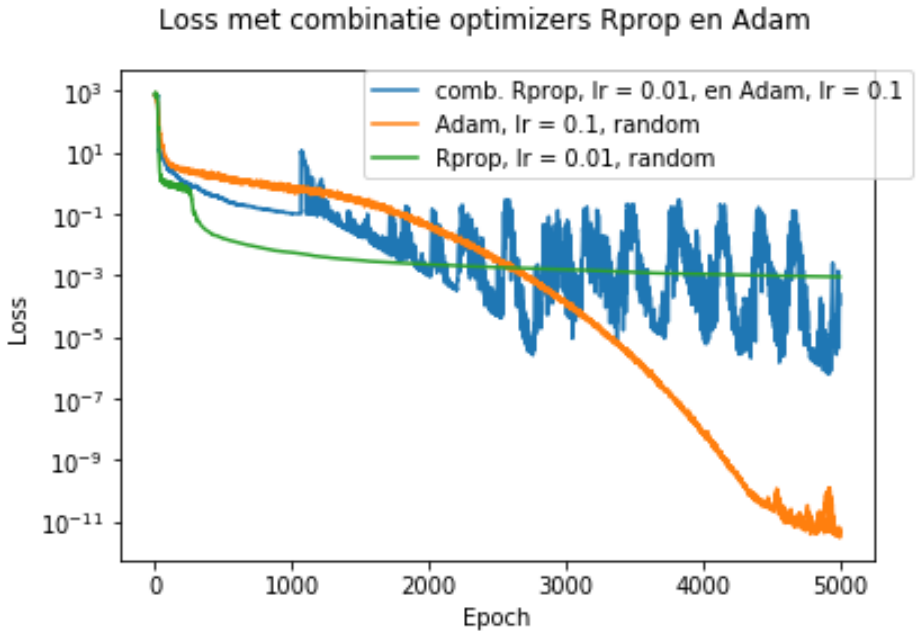
Figuur 6: Verloop van loss bij zes optimizers en drie soorten W_1 en W_2, `data_size` = 100, logaritmische y-as

De beste keuze

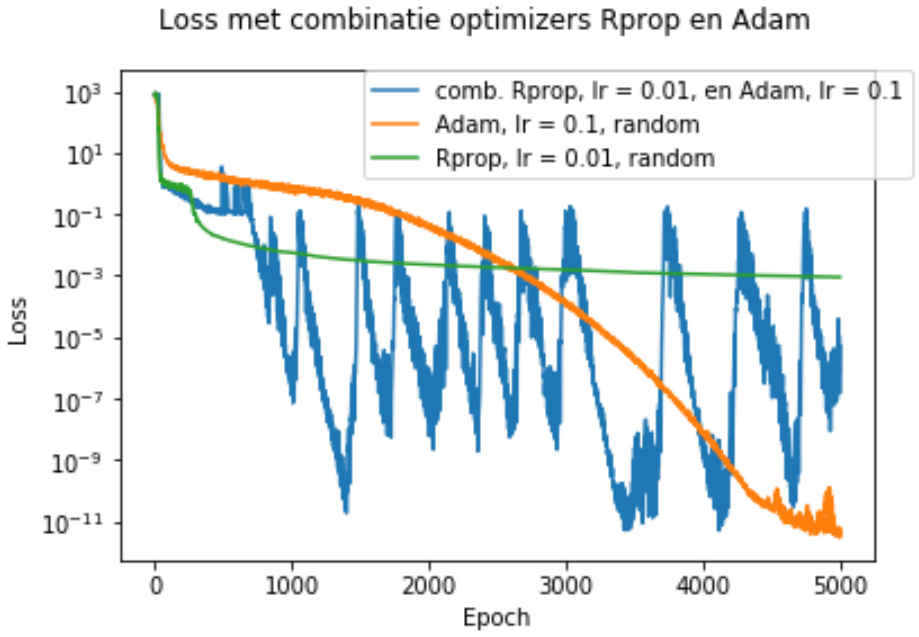
In Figuur 7 vallen twee optimizers erg op: Ten eerste Adam, die een ontzettend lage loss bereikt; en ten tweede Rprop, die al veel sneller dan de andere optimizers 10^{-1} bereikt. Dit geeft aanleiding tot nog een laatste idee: namelijk het combineren van deze twee optimizers. Eerst Rprop totdat de loss lager dan 10^{-2} is, en daarna Adam. Dit is te zien in Figuur 8 en 9. Er onstaat een vreemd tafereel: ten eerste oscilleert de lossfunctie heel erg, ten tweede eindigt de loss niet bepaald laag, en ten derde verschilt het per run van het programma. Daarnaast staat het lossverloop van hoe de optimizers het alleen doen er ook nog in, ter vergelijking. De beste optie is dus om het gewoon bij alleen Adam te laten.



Figuur 7: Verloop van loss bij zes optimizers met de beste learning rates en initiële gewichtenmatrices, `data_size = 100`, logaritmische y-as



Figuur 8: Combinatie van twee optimizers: Rprop aan het begin, maar vanaf loss $< 10^{-1}$ Adam



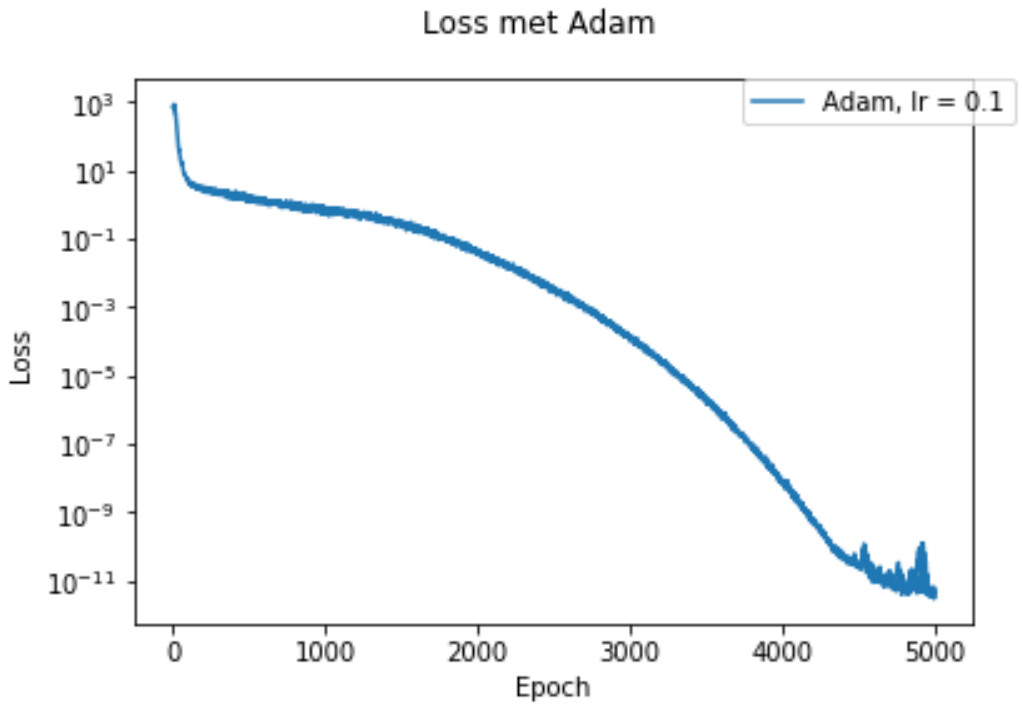
Figuur 9: Combinatie van twee optimizers: Rprop aan het begin, maar vanaf loss $< 10^{-1}$ Adam

3 Toepassing

Als alle gevonden resultaten gecombineerd worden, is het neurale netwerk met de beste architectuur een die de volgende elementen bevat:

- Een `data_size` van 100
- Een learning rate van 0.1
- Initiële matrices W_1 en W_2 die willekeurige getallen bevatten waarvoor geldt $|w| \leq 1$.
- De optimizer Adam.

Zie Figuur 10 om te zien hoe de loss van een ordegrootte van 10^3 naar 10^{-11} loopt.



Figuur 10: Verloop van loss met Adam, lr = 0.1, `data_size`=100, W_1 en W_2 willekeurig, logaritmische y-as

Uiteindelijk levert dit als LU-ontbinding het volgende:

$$\begin{pmatrix} -2.6174 & -1.6566 \cdot 10^{-7} \\ -7.8523 & 3.9911 \end{pmatrix} \begin{pmatrix} -3.8205 & -7.6411 \\ 1.2006 \cdot 10^{-7} & -5.0111 \end{pmatrix} = \begin{pmatrix} 10.0000 & 20.0000 \\ 30.0000 & 40.0000 \end{pmatrix}$$

Dit is een zeer redelijke LU-decompositie. De loss is slechts

$$4.38 \cdot 10^{-12}.$$

Helaas werkt het niet voor alle matrices zo goed. Als de matrix

$$\begin{pmatrix} 2 & 10 \\ 8 & 13 \end{pmatrix}$$

wordt genomen, vindt het programma namelijk de volgende decompositie:

$$\begin{pmatrix} 1.6313 & -7.4378 \cdot 10^{-4} \\ 6.4724 & 5.1555 \end{pmatrix} \cdot \begin{pmatrix} 1.2340 & 6.1257 \\ 1.3585 \cdot 10^{-3} & -5.1688 \end{pmatrix} = \begin{pmatrix} 2.0130 & 9.9969 \\ 7.9938 & 13.0008 \end{pmatrix}$$

En de loss is $1.40 \cdot 10^{-5}$.

Met de matrix

$$\begin{pmatrix} 100 & 3 \\ 60 & 0.4 \end{pmatrix}$$

doet het netwerk het weer wel goed:

$$\begin{pmatrix} -10.947 & -9.7006 \cdot 10^{-9} \\ -6.5683 & 5.3355 \end{pmatrix} \begin{pmatrix} -9.1348 & -0.2740 \\ 2.1213 \cdot 10^{-7} & -0.2624 \end{pmatrix} = \begin{pmatrix} 100.0000 & 3.0000 \\ 60.0000 & 0.4000 \end{pmatrix}$$

Loss $= 3.68 \cdot 10^{-11}$.

4 Conclusie

Uit dit onderzoek volgt dat het zeker mogelijk is om met behulp van een getraind neuraal netwerk matrices te ontbinden in een LU-decompositie. Sterker nog, door de structuur van neurale netwerken zijn ze heel erg geschikt om allerhande algebraïsche structuren te produceren voor matrices. Dit biedt perspectief: door de architectuur van het netwerk aan te passen, kunnen er nog veel meer, interessantere matrixontbindingen gemaakt worden op deze manier. Ontbindingen zoals Cholesky-decomposities, LDU, PDP^{-1}, met of zonder orthogonale kolommen, liggen nu binnen handbereik.

Wel is gebleken dat het gevonden neurale netwerk, dat LU-decomposities maakt, niet perfect voor alle matrices werkt. Er is veel aan het netwerk gesleuteld voordat het een goede LU-decompositie van de matrix

$$\begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$$

kon opleveren, en het was een kwestie van geduld en een gevoeligheidsanalyse om veel verschillende parameters gelijk te houden en telkens eentje aan te passen totdat er langzamerhand een steeds betere uitkomst uitkwam, want van te voren was het moeilijk om te voorspellen wat de beste samenstelling van parameters was.

5 Discussie

Dit onderzoek heeft laten zien dat een getraind neuraal netwerk erg geschikt is om matrixdecomposities te vinden, maar toch zijn er een aantal punten die nog wat aandacht verdienen.

Zo is een gevoeligheidsanalyse een nuttig apparaat, maar eigenlijk zijn er veel te veel factoren die kunnen worden aangepast, dat het haast onmogelijk wordt om alle combinaties uit te proberen. Er zijn namelijk nog meer optimizers, en binnen die optimizers zijn er nog meer parameters om aan te passen. Het zou zo maar kunnen dat er nog een betere combinatie bestaat, maar het is moeilijk om de beste te vinden met de gevoeligheidsanalyse.

Verder zijn er nog een aantal zaken die het programma misschien zouden verbeteren zodat het ook werkt voor andere, grotere matrices. Er bijvoorbeeld zou nog een regel code bij kunnen die ervoor zorgt dat het netwerk stopt met inputs aannemen wanneer de loss op een bepaald minimum is, zodat de gewichten die bepaald zijn wanneer de lossfunctie laag is, niet weer veranderen.

Daarnaast kan er nog gekeken worden naar een andere lossfunctie, die bijvoorbeeld de Frobeniusnorm gebruikt om het verschil tussen $\mathbf{y}$ en $\mathbf{y}_p red$ te bepalen, in plaats van MSEloss. Ook focust het netwerk nu heel erg op het ontzettend klein krijgen van de waarden die 0 moeten zijn, terwijl die nooit helemaal 0 kunnen worden. Een idee is om te eisen van het netwerk dat hij niet nullen neerzet op de juiste plekken in de L en U, maar bijvoorbeeld enen, zodat het netwerk minder nadruk legt op die waarden zo klein mogelijk maken.

Iets anders dat ook nog kan is de gewichten tussendoor telkens L- en U-vormig te maken. Dit was in dit project helaas niet gelukt maar het zou misschien helpen.

Tot slot is er nog de aanrader om verder te gaan met dit neurale netwerk en om naast grotere matrices aan te pakken, nieuwe decomposities te maken, zoals onder andere Cholesky-decomposities (die als het ware de unieke variant van LU-decomposities zijn).

Referenties

[1] MissingLink.ai. *7 Types of Neural Network Activation Functions: How to Choose?*
Gevonden op: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/

[2] Deisenroth, M.P., Faisal, A.A. en Ong, C.S. (maart 2020) *Mathematics in Machine Learning*, p. 162-163
Gevonden op: https://mml-book.github.io/book/mml-book.pdf

[3] Gandhi, R. (juni 2018). *A Look at Gradient Descent and RMSprop Optimizers*
Gevonden op: https://towardsdatascience.com/a-look-at-gradient-descent-textand-rmsprop-optimizers-f77d483ef08b

[4] Horn, R.A. en Johnson, C.R. *Matrix Analysis*. p. 162 Corollary 3.5.5
Gevonden op: https://books.google.nl/books?id=PlYQN0ypTwEC&printsec=frontcover&dq=isbn:9780521386326&hl=nl&sa=X&ved=2ahUKEwia_Ne0qdDqAhXK2KQKHaRPB-gQ6wEwAHoECAAQAQ#v=onepage&q=LU&f=false

[5] Rothkamm, M. (april 2019). *How to realize a polynomial regression in Pytorch / Python*
Gevonden op: https://stackoverflow.com/questions/55920015/how-to-realize-a-polynomial-regression-in-pytorch-python

[6] Algorithmia (mei 2018) *Introduction to Optimizers*
Gevonden op: https://algorithmia.com/blog/introduction-to-optimizers

[7] Brownlee, J. (juli 2017) *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*
Gevonden op: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[8] Ruder, S. (januari 2016) *An overview of gradient descent optimization algorithms*
Gevonden op: https://ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent

[9] Polyak, T. en Juditsky, B. (juli 1992) *Acceleration of stochastic approximation by averagering*
Gevonden op: http://www.meyn.ece.ufl.edu/archive/spm_files/Courses/ECE555-2011/555media/poljud92.pdf

[10] Nachum, O. (november 2015) *How does Averaged Stochastic Gradient Decent (ASGD) work?* [forumreactie]
Gevonden op: https://www.quora.com/How-does-Averaged-Stochastic-Gradient-Decent-ASGD-work

[11] Hartmann, F. (april 2018) *Rprop*
Gevonden op: https://florian.github.io/rprop/

[12] Bushaev, V. (september 2018) *Understanding RMSprop — faster neural network learning*
Gevonden op: https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a

[13] Trsvchn (februari 2020) *What does model.eval() do in pytorch?* (forumreactie)
Gevonden op: https://stackoverflow.com/questions/60018578/what-does-model-eval-do-in-pytorch

A Pythoncode

```
import torch
from torch import Tensor
from torch.nn import Linear, MSELoss, functional as F
from torch.optim import SGD, Adam, RMSprop
from torch.autograd import Variable
import numpy as np

A_data = [[10., 20.], [30., 40.]]
A = np.array(A_data)

n = 2 #afmetingen

# define our data generation function
def data_generator(data_size):
    inputs = []
    labels = []

    # loop data_size times to generate the data
    for i in range(data_size):
        x = np.random.rand(1,n)
        y = np.matmul(x,A)
        inputs.append([x])
        labels.append([y])
    return inputs, labels

# define the model
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = Linear(n,n, bias = False)
        self.fc2 = Linear(n,n, bias = False)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        return x

model = Net()
# define the loss function
```

```
40 critereon = MSELoss()
41 # define the optimizer
42 optimizer = SGD(model.parameters(), lr=0.001)
43
44 # define the number of epochs and the data set size
45 nb_epochs = 5000
46 data_size = 100
47
48 SGDlist= []
49
50 # create our training loop
51 for epoch in range(nb_epochs):
52     X, y = data_generator(data_size)
53     X = Variable(Tensor(X))
54     y = Variable(Tensor(y))
55
56     epoch_loss = 0;
57
58     y_pred = model(X)
59
60     loss = critereon(y_pred, y) + (model.fc2.weight[0,1])**2 +
       ↪  (model.fc1.weight[1,0])**2
61     epoch_loss = loss.data
62
63     optimizer.zero_grad()
64
65     loss.backward()
66
67     optimizer.step()
68     SGDlist.append(float(format(epoch_loss)))
69
70
71 # test the model
72 model.eval()
73 [test_data, test_labels] = data_generator(1)
74 prediction = model(Variable(Tensor(test_data)))
75 print ("W1^T .W2^T  =",np.matmul(model.fc1.weight.t().data,
   ↪  model.fc2.weight.t().data))
76 print("L =", model.fc1.weight.t())
77 print("U =", model.fc2.weight.t())
78
```

```
79
80 plt.plot(SGDlist, label= "SGD", alpha = 1)
81 #plt.yscale('log')
82 plt.ylabel('Loss')
83 plt.xlabel('Epoch no')
84 plt.suptitle('Loss per epoch')
85 plt.legend(bbox_to_anchor=(1.05, 1), loc='best',
   ↪  borderaxespad=0.)
86 plt.show()
```

B Toelichting code

Ten eerste worden er een aantal dingen gedefinieerd: de te ontbinden matrix A natuurlijk, maar ook de functie `data_generator(data_size)` die twee lijsten maakt: een met inputs en een met labels. Hierbij is input $\mathbf{x}^\top$ een willekeurige rijvector van 2 lang, en label $\mathbf{y} = \mathbf{x}^\top A$. De rijen zijn zo lang als wordt aangegeven door `data_size`.

Dan wordt het neurale netwerk gedefinieerd: het krijgt twee lineaire lagen die een vector van grootte 2 aannemen en ook weer een vector van grootte 2 doorgeven, precies zoals in Figuur 1.

De functie `forward` is wat de input door het netwerk stuwt: het neemt de input, roept de gewichtenmatrix en bias op en vermenigvuldigt het daarmee, en laat de gekozen activatiefunctie erover heen gaan, en doet dit nog een keer. De eerste laag heeft als activatiefunctie de identiteit, en de tweede laag heeft de functie `functional.ReLU`, wat gelijk is aan $f(x) = \max(0, x)$.

Het aantal leeriteraties, dat ook wel *epochs* genoemd wordt, wordt gedefinieerd, net als de `data_size`.

Dan wordt het neural netwerk in het leven geroepen met `model = Net()`. De optimizer wordt gedefinieerd, en het losscriterion.

Nu begint het leerproces. We komen in een for-loop die zo vaak herhaald wordt als het aantal epochs dat gedefinieerd is. In elke epoch worden twee lijsten, één met inputs en één met labels gemaakt, ter grote van `data_size`. Dan wordt de loss op 0 gezet, en de hele lijst met inputs gaat door het neurale netwerk heen om een lijst met `y_pred` te creëren. Dan wordt de loss gedefinieerd: het bestaat uit de eerder besproken MSEloss en de waarden in de gewichtenmatrices W_1 en W_2 die 0 moeten worden. Op deze manier wordt het afgestraft als ze groot zijn, want als ze klein zijn is de loss ook kleiner.

Dan wordt de optimizer aangeroepen: eerst worden alle gradiënten op 0 gezet, dan wordt met behulp van backpropagation de gradiënten uitgerekend, en als laatste worden de parameters aangepast door de optimizer.

Op het eind wordt `model.eval()` aangeroepen. Dit is een soort schakelaar voor bepaalde delen van het lerende netwerk: bij gebruik van de optimizer ASGD zorgt dit er bijvoorbeeld voor dat de uiteindelijke gewichten het gemiddelde zijn van alle voorgaande gewichten [13].

Er wordt nog één keer een input ingevoerd en een output ontvangen om te bekijken wat de voorspelling van het netwerk is. Dan volgen nog enkele printstatements die laten zien wat de uiteindelijke gewichten zijn.