

**SRP**

# **Neurale Netværk**

**Fag:** Matematik

**Emne:** Neurale Netværk

**Elev:** Mads Egelund Hoff 3.Z

**Vejleder:** Jacob Gjørtsvang Andersen

**Skole:** Himmelev Gymnasium

**Dato:** 21-03-2025

# Resume

I opgaven undersøges det, hvordan den grundlæggende matematik bag neurale netværk fungerer. Først bliver der givet en introduktion til neurale netværk, hvor der lægges særligt fokus på deres struktur og aktiveringsfunktioner. Derudover redegøres der for matematiske metoder såsom matrixoperationer, der er essentielle for at beskrive neurale netværkers struktur matematisk. Opgaven dykker også ned i koncepterne backpropagation, gradient descent og kædereolen, der alle er fundamentale i træningen af neurale netværk.

For at demonstrere, hvordan teorien anvendes i praksis, implementeres et neuralt netværk fra bunden i programmeringssproget Python til at genkende håndskrevne tal. Med en succesfuld implementering viser opgaven, at den introducerede teori er mulig at overføre til praksis. I opgaven eksperimenteres der også ved at implementere dynamiske lag, der gør det muligt at ændre antallet af skjulte lag i det neurale netværk. Dette gør det lettere at konstruere og implementere mere komplekse neurale netværk med flere skjulte lag, uden at skulle kassere den oprindelige kode, hver gang strukturen ændres.

# Indholdsfortegnelse

1.	Indledning .....	1
2.	Redegørelse .....	2
2.1	Machine learning .....	2
2.2	Neurale netværk og forward propagation .....	3
2.3	Aktiveringsfunktioner .....	4
2.4	Softmax .....	5
2.5	Matrixoperationer .....	6
2.5.1	Skalarmultiplikation .....	6
2.5.2	Matrixaddition og -subtraktion .....	7
2.5.3	Hadamard produkt .....	7
2.5.4	Matrixmultiplikation .....	8
2.5.5	Transponering .....	9
3.	Matematikken i anvendelse .....	10
3.1	Forward propagation .....	12
3.2	Backpropagation .....	14
3.3	Gradient descent .....	15
3.4	Kædereglene .....	16
3.5	Implementering af det neurale netværk .....	19
3.6	Dynamiske lag .....	21
4.	Konklusion .....	22
5.	Litteraturliste .....	23
6.	Bilag .....	25

# 1. Indledning

I takt med den teknologiske udvikling spiller kunstig intelligens og machine learning en stadig større rolle i samfundet, hvor neurale netværk er en af de mest centrale teknologier, der ligger bag denne udvikling. Neurale netværk anvendes i alt fra billedgenkendelse og naturlig sprogbehandling til medicinsk diagnostik og selvkørende biler. De består af matematiske modeller, der kan lære komplekse mønstre og træffe beslutninger på baggrund af store mængder data. Teknologien tog oprindeligt inspiration fra den menneskelige hjerne og den måde, hvorpå neuroner fungerer.

Denne opgave undersøger, hvordan neurale netværk fungerer på et grundlæggende niveau. Først gives en introduktion til neurale netværk og deres grundlæggende principper. Derefter gennemgås den matematiske teori bag teknologien. Herunder behandles begreber som vægtning, aktiveringsfunktioner og backpropagation samt matrixoperationer, der er grundlæggende for datastrukturen i neurale netværk. Til sidst implementeres et neuralt netværk fra bunden i Python, hvor der gøres brug af den introducerede teori.

## 2. Redegørelse

### 2.1 Machine learning

Machine learning er en del af kunstig intelligens, hvor man kan udvikle algoritmer, der kan træffe beslutninger, uden at man direkte har programmeret kriterierne. Det smarte ved machine learning er, at man, som navnet antyder, kan få en computer til at lære. I klassisk programmering bliver computeren givet en funktion eller nogle kriterier at arbejde ud fra, samt noget data, som computeren skal behandle. Et simpelt eksempel er en computer, der bliver sat til at regne en medarbejders månedsløn ud, hvor den får givet et timeantal og en timeløn. I machine learning fungerer denne proces omvendt. Her får computeren en masse data til at starte med, som den trænes på, og herefter skal den selv finde sammenhænge og opstille en model, som den fremover kan bruge til at give et svar ud fra nye data. Til det kan man give computeren en liste over, hvor langt forskellige personer springer i længdespring, og hvor høje personerne er. Der vil være en nogenlunde fornuftig sammenhæng mellem de to tal, som computeren vil kunne finde et nogenlunde mønster for. Hvis computeren skal gøre det endnu bedre, kunne man for hver person også give computeren input omkring deres alder, køn, vægt, mængde sport dyrket pr. uge osv. Det vil give computeren en langt bedre chance for at finde en mere detaljeret sammenhæng i den givne data og dermed en mere præcis forudsigelse af, hvor langt en tilfældig person, som den aldrig har fået data på før, springer.

Machine learning bliver brugt i alle mulige sammenhænge. Værktøjerne Chat-GPT, Google-lens og Dall-E er nogle af de mest brugte machine learning værktøjer. Men hvordan fungerer de egentlig i praksis? Fælles for dem alle er, at der bagved ligger en række kæmpemæssige neurale netværk, som er trænet på store mængder data.

Neurale netværk er ikke nogen ny ting, da ideen stammer helt tilbage fra 1943. Dengang blev der taget inspiration fra biologiske neuroner i hjernen. Ideen var dog foran sin tid, da man endnu ikke havde computere, og derfor ikke kunne lave de komplekse udregninger, der er nødvendige for at få et brugbart output. I løbet af de seneste år har udviklingen af machine learning, der i daglig tale bliver kaldt kunstig intelligens, dog taget fart på i takt med, at computere har fået større regnekraft.<sup>1</sup>

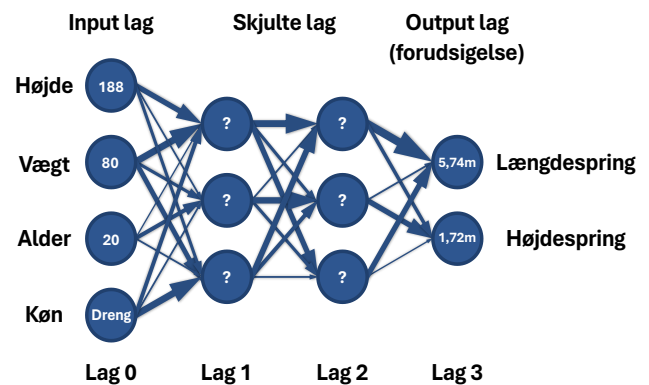
---

<sup>1</sup> IBM: "What is a neural network?", u.d. imb.com

## 2.2 Neurale netværk og forward propagation

For indledningsvist at forklare, hvad et neuralt netværk er, bliver der i dette afsnit taget udgangspunkt i et neuralt netværk, der allerede er trænet. Hvordan netværket bliver trænet og matematikken bag den proces, bliver forklaret i afsnit 3.2 Backpropagation.

Et neuralt netværk består af tre typer lag: input-laget, de skjulte lag og output-laget. De er forbundet med hinanden, som illustreret på Figur 1. Her er et netværk med strukturen 4-3-3-2, der beskriver antallet af neuroner i de forskellige lag. På figuren er eksemplet fra før brugt, hvor en computer bliver trænet til at forudsige længden af et længdespring og nu også højden af et højdespring. I input laget får hvert neuron noget data, såsom personens højde



Figur 1: Neuralt netværk, der forudsiger længden og højden på et hhv. længde- og højdespring ud fra data på en tilfældig person. Mads Hoff

og vægt. Hvert input neuron videregiver denne information til samtlige neuroner i det næste lag.

Mellem inputlaget og det første skjulte lag, er der derfor 12 forbindelser på Figur 1, der er illustreret med pile. Disse pile bliver kaldt for vægte, og de har hver især en unik værdi, som bliver ganget på det tal, der bliver givet fra neuronet i forrige lag. I det næste lag tager hvert neuron de tal den får og lægger sammen. Oveni denne sum tilføjes en bias, der er forskellig for alle neuroner. Denne bias går ind og bestemmer specifikt for det enkelte neuron, om det generelt skal være mere eller mindre aktivt ved en hhv. positiv eller negativ bias. En stor positiv bias for et neuron betyder, at det generelt er meget aktivt, mens en meget negativ værdi betyder, at neuronet generelt er meget inaktivt. Med denne introduktion er det nu muligt at opstille et generaliseret udtryk for neuronerne i netværket<sup>2</sup>:

$$z = a_1 \cdot w_1 + a_2 \cdot w_2 + \dots + a_n \cdot w_n + b$$

Her betegner  $z$  den værdi neuronet i det nuværende lag får.  $a_1$  til  $a_n$  betegner samtlige af de  $n$  aktiverede neuroner i det forrige lag. Hvad et aktiveret neuron er, bliver beskrevet i næste afsnit.  $w_1$  til  $w_n$  er de vægte, som forbinder de forrige neuroner til det specifikke nuværende neuron. Til sidst

<sup>2</sup> IBM: "What is a neural network?", u.d. imb.com

tillægges den bias, der afgør, om det specifikke neuron generelt skal være mere eller mindre aktivt. Dette udtryk kan forkortes markant med sumnotation:

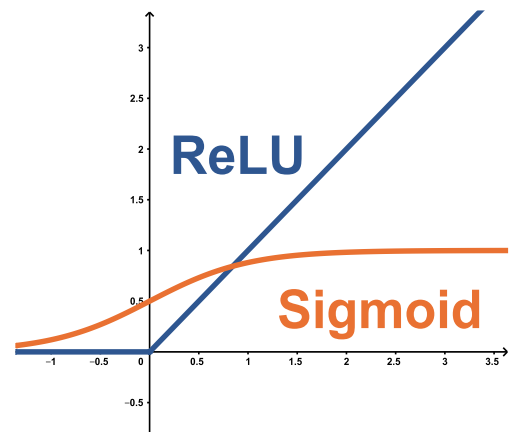
$$z = \sum_{i=1}^n a_i \cdot w_i + b \quad (1)$$

## 2.3 Aktiveringsfunktioner

Værdien af  $z$  kan dog variere meget alt afhængigt af de input den får og de vægte, som bliver ganget på. Den kan få en værdi langt under 0, men den kan også have en værdi langt over 0. Det potentielle interval, som  $z$  kan ligge i, bliver større og større desto flere neuroner, der er i netværket. Det er derfor nødvendigt at presse  $z$ -funktionen sammen, så den ligger inden for et fast interval. Det typiske for neurale netværk er, at den nye værdi for  $z$  aldrig er negativ. For hvis et negativt og dermed deaktiveret neuron bliver ganget med en negativ vægt, bliver det næste neuron lige pludselig aktiveret. Man gør derfor brug af en aktiveringsfunktion. Den klassiske aktiveringsfunktion indenfor neurale netværk har været sigmoid. Sigmoid aktiveringsfunktionen presser alle værdier ned i intervallet mellem 0 og 1, hvor  $z=0$  ligger i midten på 0,5. Funktionen sigmoid opskrives som

$$\sigma(x) = \frac{1}{1 + e^{-x}} \cdot 3$$

Umiddelbart lyder sigmoid som en fornuftig funktion, men den er ikke uden problemer. Når den skal have samtlige værdier ind i et specifikt interval, ender markant forskellige værdier med at blive nærmest ens: 5 bliver til 0,993 mens 5000 bliver til 1,000. Der går derfor en masse information tabt, da netværket vil ende med en masse neuroner meget tæt på 0 og 1, og dermed ikke fordelt ud over intervallet. Derfor er en mere brugt aktiveringsfunktion den ikke-lineære ReLU funktion, som står for rectified linear unit. Den giver 0 for alle  $x$ -værdier under 0, og returnerer  $x$ -



Figur 2: De to aktiveringsfunktioner ReLU og Sigmoid  
Mads Hoff

<sup>3</sup> Deisenroth, Marc Peter m.fl.: Mathematics For Machine Learning, 2020, s. 284

værdien, når den er større end 0. Den opskrives som  $ReLU(x) = \max(0, x)$ .<sup>4</sup> Udtrykket for et aktiveret neuron med ReLU kan skrives som:

$$a = ReLU\left(\sum_{i=1}^n w_i \cdot x_i + b\right) \text{ eller: } a = ReLU(z) \quad (2)$$

Når man udfører forward propagation i et neuralt netværk, er det netop denne udregning man laver for samtlige neuroner. Man regner dem lag for lag, indtil man når outputlaget, hvor man får netværkets output, som i det tidligere eksempel er forudsigelsen af, hvor langt og højt en given person springer. Til backpropagation skal både den aktiverede og ikke-aktiverede matrix bruges, hvorfor de to formler ikke kan sættes sammen til en enkelt formel.

## 2.4 Softmax

I eksemplet fra før, hvor et neuralt netværk forudsiger højden og længden af en persons spring, var netværkets output netop denne højde og længde. Men alt afhængigt af applikationen, hvor det neurale netværk bruges, kan det være at et specifikt tal ikke er et optimalt output. Hvis man gerne vil have netværket til at tage et valg, så giver det bedre mening at netværket giver en procentsats, til de forskellige muligheder. Et neuralt netværk er dog forfærdeligt til at regne, så summen af de forskellige muligheder vil både kunne være langt over eller under 100%, men også blot for det enkelte neuron. En bedre metode er at bruge en aktiveringsfunktion som softmax, der regner en procentfordeling ud, ved at lægge vægt på, hvor aktiverede de forskellige neuroner er. Softmax er defineret som:

$$softmax(Z_i) = \frac{e^{Z_i}}{\sum_{j=1}^N e^{Z_j}}$$

Med tre noder i et lag er den en smule lettere at forstå:  $softmax(Z_i) = \frac{e^{Z_i}}{e^{Z_1} + e^{Z_2} + e^{Z_3}}$ . Ved at opløfte værdierne i  $e^x$  bliver der lagt markant større vægt på mere aktiverede neuroner. Det har både sine fordele, men også ulemper, da to neuroner der er næsten lige aktiverede kan få markant forskellige værdier efter softmax. Hvis der kun er to værdier på 45 og 50, bliver de til hhv. 0.0067 og 0.993.

---

<sup>4</sup> Wikipedia: "Rectifier (neural networks)", u.d., wikipedia.org

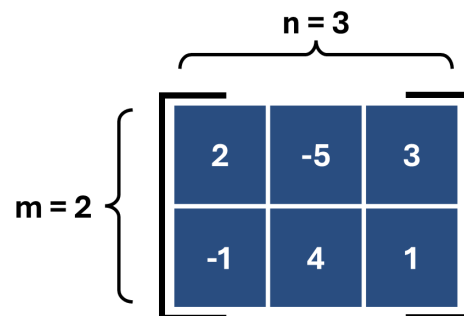


## 2.5 Matrixoperationer

Matricer er grundstenen i neurale netværk, da al data fra input til output og alt derimellem, er gemt i matricer. Derfor er forståelsen af, hvordan matricer fungerer, og hvilke operationer man kan udføre med dem, nødvendig for at kunne forstå og regne på et neuralt netværk.

En matrix er en rektangulær 2-dimensionel tabel bygget op af  $m$  antal rækker og  $n$  antal kolonner. Notationen for en matrix  $A$  med dimensionerne  $m$  rækker og  $n$  kolonner, er  $A \in \mathbb{R}^{m \times n}$ . Heri befinder sig  $a_{i,j}$  elementer, hvor

$1 \leq i \leq m$  og  $1 \leq j \leq n$  og  $i, j \in \mathbb{N}$ . På Figur 3 er et eksempel på en matrix  $A \in \mathbb{R}^{2 \times 3}$ . Her har elementet  $a_{1,2}$  værdien -5. Den simpleste type matrix er en  $1 \times n$  matrix.



Figur 3: Matrix med dimensionerne 2x3  
Mads Hoff

Den er bedre kendt som en vektor, hvor en  $1 \times 2$  matrix blot er en todimensionel vektor:  $\begin{bmatrix} x \\ y \end{bmatrix}$ . En matrix kan derfor ses som en samling af vektorer.<sup>5</sup>

For at regne på neurale netværk skal man bruge 5 forskellige operationer: addition og subtraktion, skalarmultiplikation, matrixmultiplikation, transponering og elementvis matrixmultiplikation også kaldt Hadamard-produktet. Det er værd at bemærke hvilke af disse operationer, der er kommutative og associative, men især hvilke operationer der ikke er, da det har stor betydning i opbygningen og udregningen af det neurale netværk, og den rækkefølge elementerne bliver regnet i.

### 2.5.1 Skalarmultiplikation

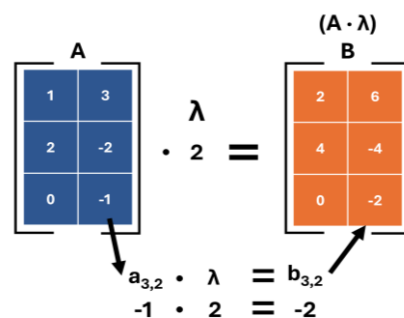
Når en matrix bliver ganget med en skalar, bliver hvert element  $a_{i,j}$  i den givne matrix ganget med en skalar  $\lambda$ , hvor  $\lambda \in \mathbb{R}$ .

Hvert element  $b_{i,j}$  i den nye matrix  $B$  bliver derfor  $a_{i,j} \cdot \lambda$ .

Multiplikation med en skalar er kommutativ da

$\lambda \cdot a_{i,j} = a_{i,j} \cdot \lambda$ . Skalar multiplikation er også associativ:

$(\lambda \cdot \phi) \cdot a_{i,j} = \lambda \cdot (\phi \cdot a_{i,j})$ .<sup>6</sup>



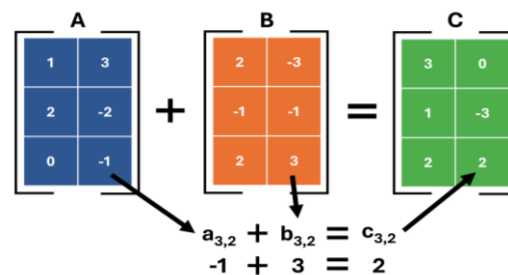
Figur 4: Multiplikation med skalar  
Mads Hoff

<sup>5</sup> Hesselholt, Lars m.fl.: "Lineær Algebra", Oktober 2016, IMFs Matematiknoter, noter.math.ku.dk s. 27

<sup>6</sup> Deisenroth, Marc Peter m.fl.: Mathematics For Machine Learning, 2020, s. 16

## 2.5.2 Matrixaddition og -subtraktion

Matrixaddition er den simpleste matrixoperation. Man skal blot have to matrixer A og B af samme dimension,  $A \in \mathbb{R}^{m \times n}$ ,  $B \in \mathbb{R}^{m \times n}$ . Herefter tages summen for hver af de parvise elementer  $a_{i,j}$  og  $b_{i,j}$  og sættes på pladsen  $c_{i,j}$  dvs.:  $a_{i,j} + b_{i,j} = c_{i,j}$ . Denne operation er illustreret på Figur 5.



Figur 5: Addition af to matrixer med dimensionerne 3x2  
Mads Hoff

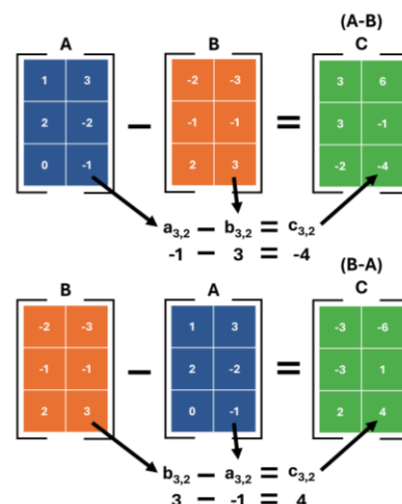
Denne operation er både kommutativ da:  $a_{i,j} + b_{i,j} = b_{i,j} + a_{i,j}$  og associativ da:  $(a_{i,j} + b_{i,j}) + c_{i,j} = a_{i,j} + (b_{i,j} + c_{i,j})$

Matrixsubtraktion foregår på samme måde, hvor + bliver til - :

$a_{i,j} - b_{i,j} = c_{i,j}$ . Denne operation er dog ikke kommutativ, da

$a_{i,j} - b_{i,j} \neq b_{i,j} - a_{i,j}$ . Dette illustreres på Figur 6. Ved

subtraktion af 3 eller flere matrixer på samme tid ses det også, at matrixsubtraktion heller ikke er associativ:<sup>7</sup>



Figur 6: Viser et eksempel på, hvorfor subtraktion af matrixer ikke er kommutativ.  
Mads Hoff

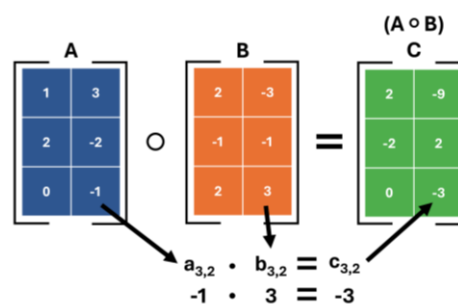
$$(a_{i,j} - b_{i,j}) - c_{i,j} \neq a_{i,j} - (b_{i,j} - c_{i,j})$$

$$\rightarrow a_{i,j} - b_{i,j} - c_{i,j} \neq a_{i,j} - b_{i,j} + c_{i,j}$$

## 2.5.3 Hadamard produkt

Udover addition og subtraktion findes der også en matrixoperation, som i stil med matrixaddition, finder produktet af de parvise elementer i to eller flere matrixer A og B ved:  $a_{i,j} \cdot b_{i,j} = c_{i,j}$ . Det hedder Hadamard produktet og kræver at:  $A \in \mathbb{R}^{m \times n}$ ,  $B \in \mathbb{R}^{m \times n}$  ligesom matrixaddition. Hadamard produktet er modsat matrixmultiplikation både kommutativt, da

$a_{i,j} \cdot b_{i,j} = b_{i,j} \cdot a_{i,j}$  og associativt da,  $(a_{i,j} \cdot b_{i,j}) \cdot c_{i,j} = a_{i,j} \cdot (b_{i,j} \cdot c_{i,j})$ .<sup>8</sup>



Figur 7: Hadamard produktet af to matrixer  
Mads Hoff

<sup>7</sup> Deisenroth, Marc Peter m.fl.: Mathematics For Machine Learning, 2020, s. 14

<sup>8</sup> Deisenroth, Marc Peter m.fl.: Mathematics For Machine Learning, 2020, s. 13

## 2.5.4 Matrixmultiplikation

Matrixmultiplikation er en del mere kompliceret, men også en del nyttigere i praksis. I matrixmultiplikation skal antallet af rækker og kolonner ikke være ens i matrix A og B, som det ellers var tilfældet i matrixaddition, -subtraktion og Hadamard produkt. I stedet skal antallet af kolonner i A være lig antallet af rækker i B:  $A \in \mathbb{R}^{m \times n}$ ,  $B \in \mathbb{R}^{p \times q}$ ,  $n = p$  for at kunne multiplicere to matricer  $A \times B$ . De to tal  $m$  og  $q$

bestemmer dimensionerne på den nye matrix, der bliver skabt. Grunden til at  $n = p$  skal være opfyldt, skyldes at  $c_{i,j}$  i matrixmultiplikation er summen af samtlige af de parvise produkter for række  $a_i$  og kolonne  $b_j$ . Det er illustreret på Figur 8 og den matematiske notation er følgende<sup>9</sup>:

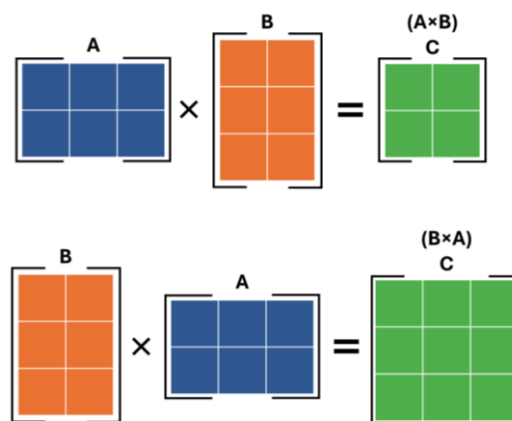
$$a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} = c_{1,1}$$

$$2 \cdot (-3) + (-2) \cdot (-1) + 3 \cdot 3 = 5$$

Figur 8: Eksempel på matrixmultiplikation af 2 matricer  
Mads Hoff

$$c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \dots + a_{i,n}b_{n,j} = \sum_{k=1}^n a_{i,k}b_{k,j} \quad (3)$$

Da det er antal rækker i den første matrix og antal kolonner i anden matrix, der bestemmer dimensionen af den nye matrix, betyder det, at matrixmultiplikation ikke er kommutativ, hvilket er illustreret på Figur 9. Operationen er dog associativ, altså:  $(A \times B) \times C = A \times (B \times C)$  i de tilfælde at  $A \in \mathbb{R}^{m \times n}$ ,  $B \in \mathbb{R}^{n \times p}$ ,  $C \in \mathbb{R}^{p \times q}$ .<sup>10</sup> De påkrævede dimensioner skyldes, at antallet af rækker og kolonner skal passe uanset associativitet eller ej, for at matrixmultiplikation overhovedet er muligt.



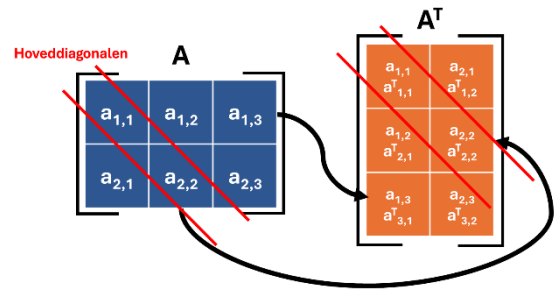
Figur 9: Matrixmultiplikation er ikke kommutativ  
Mads Hoff

<sup>9</sup> Hesselholt, Lars m.fl.: "Lineær Algebra", Oktober 2016, IMFs Matematiknoter, noter.math.ku.dk s. 29-30

<sup>10</sup> Deisenroth, Marc Peter m.fl.: Mathematics For Machine Learning, 2020, s. 14

## 2.5.5 Transponering

Nogle gange kan man ende med en eller flere matrixer, der vender den ”forkerte” vej. Matrixen har måske de rigtige dimensioner, men på de forkerte leder. Den kan for eksempel have to rækker og tre kolonner, men skulle have haft 3 rækker og to kolonner i stedet, som det ses på Figur 10. Det kan blive aktuelt, hvis man skal matrixmultiplicere to matrixer, eller når man skal



Figur 10: Transponering af en matrix  
Mads Hoff

lægge to matrixer sammen. Her kan man gøre brug af transponering, hvor  $A \in \mathbb{R}^{m \times n}$  bliver til  $A^T \in \mathbb{R}^{n \times m}$ , hvor notationen  $A^T$  betyder, at matrixen  $A$  er blevet transponeret. Kort sagt spejler man matrixen i hoveddiagonalen, som er den linje med de felter, hvor  $i = j$ . For alle elementer bliver  $a_{i,j}^T = a_{j,i}$ . Det betyder, at alle elementer på hoveddiagonalen forbliver på samme plads. Det er også værd at bemærke, at  $A^{TT} = A$ , da matrixen i dette tilfælde bliver spejlvendt to gange, hvilket blot giver den oprindelige matrix.<sup>11</sup>

<sup>11</sup> Deisenroth, Marc Peter m.fl.: Mathematics For Machine Learning, 2020, s. 15

### 3. Matematikken i anvendelse

En ting er at lære teorien bag et emne, en anden ting er at anvende teorien i praksis. Det første skridt i denne proces er at finde ud af, hvad det neurale netværk skal trænes til. Da ideen om neurale netværk blev introduceret, blev der brugt et eksempel med sammenhængen mellem en persons højde, alder, køn osv. og den højde og længde, man kunne forvente, at personen ville springe i et hhv. højde- og længdespring. Nu sættes denne forsimplede idé til side, og i stedet bruges et lidt mere komplekst eksempel, der samtidig er mere praktisk anvendeligt. Det neurale netværk skal nu lære at genkende håndskrevne tal. Fordelen er, at der findes det frit tilgængelige open-license MNIST datasæt bestående af håndskrevne tal fra 0-9. Det er delt op i 60.000 elementer til træning og 10.000 til validering. Valideringsdelen er vigtig, for at sikre at det neurale netværk ikke bliver overtilpasset til det data det bliver trænet på. Datasættet er dog ikke 60.000 separate billeder, men i stedet én lang fil på 47.040.016 bytes. De første 16 bytes giver information om dataformatet, hvormed den resterende data kan deles op i 60.000 dele, hvor hver del er et billede på 28x28 pixels. Det betyder, at hvert billede sammenlagt har 784 pixels, hvilket gør det oplagt, at input-laget i det neurale netværk, som konstrueres, har det samme antal neuroner. Laget laves som matrixen  $Z_1$  med dimensionerne (1,784). Dette er ikke en vektor, da den vender den "forkerte" vej.



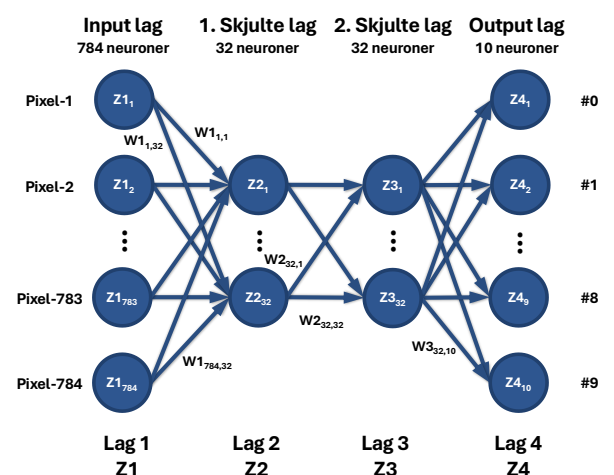
Figur 11: Den binære data af det ottende element i MNIST datasættet lavet om til et billede.

Den data som  $Z_1$  får, er den korresponderende pixel's værdi. Normalt i et billede vil der være 3 bytes for hver pixel: 1 til hhv. rød, grøn og blå. Men når man regner på det, finder man frem til, at der kun er  $\frac{47040016-16}{28 \cdot 28 \cdot 60000} = 1$  byte per pixel. Det skyldes, at hver pixel ikke har en rød, grøn og blå værdi, men i stedet blot en enkel byte mellem 0 og 255, der fortæller hvor mørk eller lys den er. Intervallet 0 og 255 skyldes, at en byte består af 8 bits, hvorfor man har med et base 2 tal at gøre, hvor den laveste værdi er  $2^0 - 1 = 0$  og højeste værdi er  $2^8 - 1 = 255$ . Denne forskel er dog al for stor for det neurale netværk. Derfor skal alle input-værdier normaliseres, så de ligger mellem 0 og 1. Da inputlaget er i en matrix, kan man blot gange med skalar  $\frac{1}{255}$ , som blev introduceret i afsnit "2.5.1 Skalarmultiplikation". Denne nye matrix kaldes  $A_1$ , som står for det aktiverede lag 1.

Det er også nu, hvor det skal besluttes, hvor mange skjulte lag, man gerne vil have i det neurale netværk, og hvor store de skal være. Det er en balancegang, for som tommelfingerregel betyder flere og større skjulte lag et bedre netværk, når det er trænet. På samme tid betyder flere og større skjulte lag dog også, at det både bliver mere ressourcetungt for en computer at regne, og samtidigt bliver det også svære at overskue. Derfor vælges der nu 2 skjulte lag, der begge har 32 neuroner, til netværket, der skal genkende de håndskrevne tal og dermed dimensionerne (1,32). For at sætte det i kontrast har Facebooks 2 år gamle sprogmodel Llama2 ifølge dens dokumentation<sup>12</sup> 32 lag med 4096 neuroner i hvert lag, blot til dens forbehandling af den indtastede data. Selvom man nu fastlåser sig på antallet og størrelsen af de skjulte lag, så bliver det vist, hvordan det er muligt at have dynamiske lag i afsnit ”3.6 Dynamiske lag”.

Til output-laget er der ikke frihed til at bestemme antallet af neuroner, da det essentielt er bestemt på forhånd, ligesom det også var tilfældet med input-laget. Da netværket skal gætte på 1 ud af 10 tal, skal der være netop ét neuron for hvert tal, hvilket giver 10 output-neuroner i alt.

Med de valgte tal får det neurale netværk strukturen 784-32-32-10, hvilket er illustreret på Figur 12. Det betyder at der sammenlagt er  $784 \cdot 32 + 32 \cdot 32 + 32 \cdot 10 = 26.432$  vægte og  $32 + 32 + 10 = 74$  bias-parametre. Når man har fastlagt strukturen, er det muligt at initialisere både vægtene og bias. Den først vægtmatrix  $W1$  forbinder  $Z1$  og  $Z2$  med matrixmultiplikation ved  $Z2 = Z1 \times W1$ . Hvorfor matrixmultiplikation kan bruges, bliver forklaret i næste afsnit, men for nu skal dimensionerne af operationen kun bruges.  $Z1$  har dimensionerne (1,784), og da  $Z2$  er bestemt til at skulle have dimensionerne (1,32), kræver det, at  $W1$  har dimensionerne (784,32), for at matrixmultiplikation giver de rigtige dimensioner til  $Z2$ . For  $W2$ , der forbinder  $Z2$  og  $Z3$ , er det samme princip, og man får dermed dimensionerne (32,32), da  $Z2$  har dimensionerne (1,32), og  $Z3$  skal ende med de samme dimensioner. Det sidste vægtlag,  $W3$ , forbinder  $Z3$  og  $Z4$ , der hver især har dimensionerne (1,32) og (1,10), og dermed får  $W3$  dimensionerne (32,10).



Figur 12: På figuren er det neurale netværk, der skal trænes til at genkende håndskrevne tal illustreret. Det har strukturen 784-32-32-10. Der er også eksempler på de vægte, der forbinder de forskellige lag.  
Mads Hoff

<sup>12</sup> Facebook: "Llama documentation", u.d., huggingface.co.

De enkelte vægte i alle vægtlag bliver tildelt med en normalfordeling med middelværdi 0.

Spredningen følger den såkaldte Kaiming He-initialisering<sup>13</sup>, der tager højde for antallet af neuroner

i det forrige lag:  $\sqrt{\frac{2}{n_{\text{neuroner}}}}$ . Som nævnt i afsnit ”2.3 Aktiveringsfunktioner” betyder flere neuroner i

det forrige lag et større potentielt interval for summen af det enkelte neuron i det nuværende lag.

Med aktiveringsfunktionen sigmoid bliver der taget højde for det, men da ReLU fungerer bedre i praksis, skal der tages højde for dette problem på en anden måde. Det er derfor, man benytter sig af Kaiming He-initialisering, da det gør størrelsen af vægtene mindre, jo flere neuroner der er i forrige lag, for at kompensere for det større interval, som summen kan ende i.

Der hører også en bias-matrix til alle neuronlagene undtagen input-laget. Bias-matricerne B1, B2 og B3 har hhv. dimensionerne (1,32), (1,32), (1,10), som er den samme som deres tilhørende neuronlag. Da bias-matricerne fortæller om de enkelte neuroner generelt er mere eller mindre aktive, giver det ikke mening at tildele dem værdier til start, og de bliver derfor initialiseret som nulmatricer.

### 3.1 Forward propagation

Nu hvor de basale matrixoperationer er blevet introduceret, og netværket er stillet op, kan funktionerne for hvert lag af neuroner skrives op. Funktionerne er nødvendige til backpropagation, der beskrives i de følgende afsnit. Til det skal de følgende to formler fra tidligere afsnit bruges gentagne gange: formel (1):  $z = \sum_{i=1}^n w_i \cdot a_i + b$  og formel (2):  $a = \text{ReLU}(z)$ . Som tidligere nævnt skal både de aktiverede og ikke-aktiverede matricer bruges til backpropagation, så de to formler kan derfor ikke samles i én. For det første skjulte lag, Z2, kan det  $j$ 'te element af Z2 findes med formel (1), hvor  $n$  er antallet af neuroner i forrige lag:

$$Z2_{1,j} = b_{2,j} + \sum_{i=1}^{n=784} A1_{1,i} \cdot w_{i,j}$$

Hvis man tager et kig på formel (3) fra afsnit ”2.5.4 Matrixmultiplikation” så den sådan ud:

<sup>13</sup> GeeksforGeeks: “Kaiming Initialization in Deep Learning”, 27.12.2023, [geeksforgeeks.org](https://www.geeksforgeeks.org/kaiming-initialization-in-deep-learning/)

$$c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \dots + a_{i,n}b_{n,j} = \sum_{k=1}^n a_{i,k}b_{k,j}$$

Det betyder, at man derfor blot kan bruge matrixmultiplikation mellem  $Z1$  og  $W1$  og samtidig lægge bias-matrixen til for at finde  $Z2$ :  $Z2 = A1 \times W1 + B1$ . Herefter kan den aktiverede matrix  $A2$  findes med formel (2):  $A2 = ReLU(Z2)$ . Selvom det måske ikke ser helt naturligt ud at sætte en hel matrix ind i en funktion, er det både den notationsmæssige måde at gøre det på<sup>14</sup>, og den måde det regnes og skrives i bl.a. programmeringssproget Python. Funktionen tager blot et element fra matrixen ad gangen og returnerer en ny værdi, som bliver sat ind på samme plads i den nye matrix.

Man kan gentage udregningen for det andet skjulte lag, som her bliver:  $Z3 = A2 \times W2 + B2$ . Neuronerne fra forrige lag,  $Z2$ , er dem, som lige er beregnet, og vægtene er skiftet til dem, der forbinder  $Z2$  og  $Z3$ . På samme måde er bias-matrixen også ændret til den, der passer til det andet skjulte lag. Den aktiverede funktion bliver  $A3 = ReLU(Z3)$ .

For outputlaget  $Z4$  er processen det samme, selvom det ikke er et skjult lag. Det skyldes, at det stadig er forbundet med vægte på samme måde som de forrige lag, hvorfor processen er den samme:  $Z4 = A3 \times W3 + B3$ .

Aktiveringen af output-laget,  $A4$ , adskiller sig en smule fra de øvrige lag. Som beskrevet i afsnit 2.4 om Softmax funktionen bruges dette output til at repræsentere en form for "sandsynlighed". Når sandsynlighed er sat i citationstegn, er det fordi netværket ikke giver en reel procentvis sandsynlighed for hvert tal, men i stedet en relativ værdi, der kan tolkes og bruges som en sandsynlighedsfordeling. Men på samme måde som med ReLU kan man også med softmax indsætte matrixen i funktionen for at få  $A4$ :  $A4 = Softmax(Z4)$ . Denne funktion giver netværkets output og er dermed den sidste funktion i forward propagation. Det betyder at samtlige udregninger i forward propagation er på plads, og der mangler dermed kun backpropagation for at kunne træne netværket.

---

<sup>14</sup> Deisenroth, Marc Peter m.fl.: Mathematics For Machine Learning, 2020, s. 284



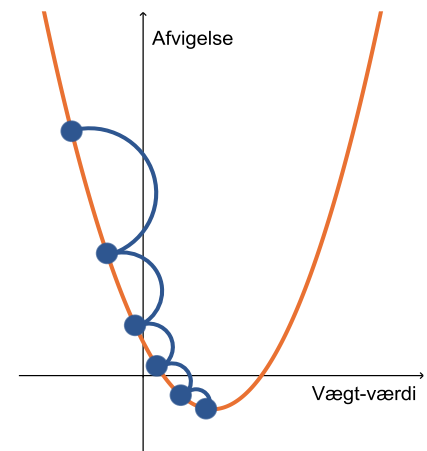
## 3.2 Backpropagation

Når man træner et neuralt netværk, er det fordi man ønsker at forbedre netværket, så det bliver bedre til at udføre den opgave, det er lavet til. Måden et neuralt netværk bliver forbedret på, er ved at korrigere alle dets forskellige vægte og bias. Det bliver gjort ved at gå baglæns gennem netværket, hvoraf navnet "backpropagation" kommer. Hvis man blot ser på neuronerne i de skjulte lag, er det umuligt at sige, hvilke neuroner der er overaktiverede, og hvilke der er underaktiverede. Det er dog muligt at se i outputlaget. Hvis netværket har fået et billede af et syvtal, skal neuronet, der repræsenterer 7, have værdien 1, mens resten af neuronerne skal have værdien 0. De ideelle værdier kan opstilles i matrixen  $T$ , der står for target. Afvigelsen for hvert neuron kan nu findes ved  $dZ^4 = T - Z^4$ . Navnet bliver der vendt tilbage til i afsnit "3.4 Kæderegl". Resultatet er en matrix, der fortæller, hvor meget hvert enkelt neuron skal enten op eller ned i værdi.

Der er tre måder, hvorpå værdierne af neuronerne kan ændres, og træningen er mest effektiv, hvis man gør brug af dem alle. Den første er at ændre bias-værdien for hvert neuron jævnt  $dZ^4$ . Den anden måde er at ændre de vægte, der forbinder det nuværende lag med det forrige lag. Hvis neuronet skal være mere inaktivt, gøres vægten mindre, og hvis neuronet skal være mere aktivt, gøres vægten større. Da vi gør brug af en aktiveringsfunktion, har vi sikret os at det forrige neuron aldrig er negativt, hvilket er vigtigt, da en større vægt ellers vil have den modsatte effekt. Den tredje måde er at ændre aktiveringen af neuronerne i det forrige lag, hvilket sender os videre til netop det forrige lag. Her gentages de tre måder at ændre aktiveringen af neuronerne på. Man fortsætter med at gå baglæns, indtil man når input-laget som ikke kan ændres.

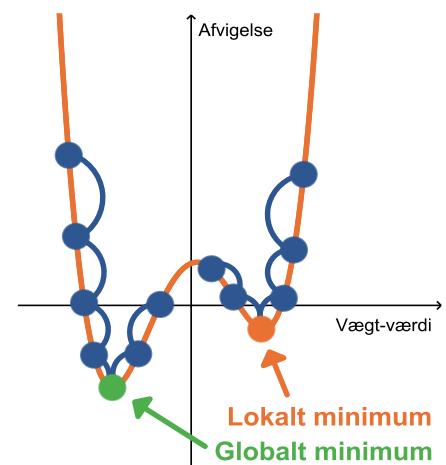
### 3.3 Gradient descent

Når man træner det neurale netværk, ændrer man vægtene og bias, for at få en lavere afvigelse. Hver enkelte vægt (og bias) kan tænkes som en funktion, der beskriver sammenhængen mellem den specifikke vægt og den endelige afvigelse (se Figur 13). Her vil man gerne finde den værdi for vægten, der giver den lavest mulige afvigelse, hvilket vil være det globale minimum. Normalt vil man kunne regne denne værdi ud, men det er ikke muligt i neurale netværker. Det skyldes at hver gang en vægt eller bias bliver ændret, så ændrer forholdet sig mellem samtlige af de andre vægte og afvigelsen. Samtidig afhænger funktionernes kompleksitet af antallet af lag og neuroner i det neurale netværk, hvilket er endnu en grund til at denne metode ikke virker.



Figur 13: Konceptet gradient descent er illustreret, hvor man gradvist bevæger sig mod et lokalt minimum.  
Mads Hoff

Man bliver derfor nødt til at gøre brug af gradient descent i stedet. Konceptet går ud på at man tager et lille skridt ad gangen ned ad grafen som det ses på Figur 13. Når vægtene oprindeligt bliver initialiseret, er det med tilfældige værdier, hvilket betyder man starter et tilfældigt sted på grafen. Herefter kan man finde hældningen af grafen i det givne punkt. Når hældningen er stor, tager man et tilsvarende stort skridt. I takt med at hældningen bliver mindre, tager man mindre og mindre skridt, indtil skridtene er så små at man praktisk talt ikke bevæger sig.<sup>15</sup> En ting der er værd at bemærke er, at man ikke er garanteret at ramme det



Figur 14: Gradient descent kan risikere at ende i et lokalt minimum.  
Mads Hoff

globale minimum. I virkeligheden er funktionen langt mere kompleks og på Figur 14 kan det ses at afhængigt af den værdi man starter med, ender man enten i et lokalt minimum eller det globale. Det interessante er at en række forskere fra Courant instituttet har fundet frem til at det ikke er noget problem, da de forskellige lokale minima er af omtrent samme kvalitet<sup>16</sup>. Det betyder at hvilket lokalt minimum man lander i, ikke spiller nogen betydelig rolle.

<sup>15</sup> Sanderson, Grant: "Gradient descent, how neural networks learn", 16.10.2017, 3blue1brown, 3blue1brown.com

<sup>16</sup> Courant Institute of Mathematical Sciences: "The Loss Surfaces of Multilayer Networks", 21.01.2015, arxiv.org

### 3.4 Kædereglen

I forrige afsnit blev det beskrevet, hvordan sammenhængen mellem vægtene og den endelige afvigelse kan tænkes som en funktion. Til gradient descent handler det derfor om at finde ud af, hvor meget den samlede afvigelse ændrer sig, når vægten justeres. Det kan man finde frem til med differentiering. For vægtlag 1 skrives det op som  $\frac{\partial \text{afvigelse}}{\partial W_1}$ . Men i afsnit "3.1 Forward propagation" hvor samtlige funktioner for det neurale netværk blev skrevet op, var der ikke nogen funktion, der både indeholdt den totale afvigelse og vægtlag 1. Det var derimod en lang kæde af udregninger, der startede med  $Z_2 = A_1 \times W_1 + B_1$ , herefter  $A_2 = \text{ReLU}(Z_2)$ , efterfulgt af  $Z_3 = A_2 \times W_2 + B_2$ . Det fortsætter indtil man når  $dZ_4 = T - A_4$ . Navnet for afvigelsen antyder at den allerede er blevet differentieret, hvilket også er sandt. Ved at gøre brug af den såkaldte cross-entropy<sup>17</sup> kan man undgå at skulle differentiere softmax og i stedet bruge afvigelsesmatrixen direkte.

Selvom afvigelsen og  $W_1$  ikke befinder sig i den samme funktion, så er de indirekte forbundet gennem en række sammensatte funktioner. Det betyder at man kan gøre brug af kædereglen, for at finde de differentierende funktioner for alle lagene. Den skrives op som  $f(g(x))' = f'(g(x)) \cdot g'(x)$ , hvor man først differentierer den ydre funktion og indsætter den indre funktion, hvorefter man ganger med den indre funktions afledte. Uanset hvor dybt antallet af indre funktioner er, kan man blot fortsætte ved at gentage kædereglen:

$$f(g(h(x)))' = f'(g(h(x))) \cdot g'(h(x)) = f'(g(h(x))) \cdot g'(h(x)) \cdot h'(x)$$

Da netværket netop består af en lang række indre funktioner, kan  $\frac{\partial \text{afvigelse}}{\partial W_1}$  deles op led for led og skrives som:

$$\frac{\partial \text{afvigelse}}{\partial W_1} = \frac{\partial \text{afvigelse}}{\partial A_4} \cdot \frac{\partial A_4}{\partial Z_4} \cdot \frac{\partial Z_4}{\partial A_3} \cdot \frac{\partial A_3}{\partial Z_3} \cdot \frac{\partial Z_3}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial W_1}$$

Nu er det muligt led for led at finde den afledte funktion og dermed hældningen, der skal bruges til gradient descent. Ved at starte fra venstre mod højre kan man tage et element ad gangen, men man skal huske at gange med det forrige element. Da der er tale om matricer, sker det med hadamard produktet. Da  $dZ_4$  allerede er kendt kan de to første led dog springes over.<sup>18</sup>

<sup>17</sup> Starmer, Josh: "Cross Entropy Derivatives and Backpropagation", Statquest, 01.03.2021, youtube.com

<sup>18</sup> Wikipedia: "Backpropagation", u.d., wikipedia.org

En sidste ting at huske på er at ReLU ikke er lineær, hvilket betyder at den ikke kan differentieres i  $x=0$ . Man kan dog komme rundt om dette problem ved blot at sætte den differentierede ReLU til at have hældningen 0, når  $x \leq 0$  og hældningen 1 når  $0 < x$ . For simplicitetens skyld skrives det blot som Relu'. Først findes alle noderne:

$$\begin{aligned} dA3 &= \frac{\partial \text{afvigelse}}{\partial A3} = \frac{\partial Z4}{\partial A3} \cdot dZ4 = \frac{\partial}{\partial A3} \cdot Z4 \cdot dZ4 = (A3 \times W3 + B3)' \cdot dZ4 \\ &= (1 \times W3 + 0) \cdot dZ4 = (1 \cdot dZ4) \times W3 = dZ4 \times W3 \end{aligned}$$

$$dZ3 = \frac{\partial \text{afvigelse}}{\partial Z3} = \frac{\partial A3}{\partial Z3} \cdot dA3 = \frac{\partial}{\partial Z3} \cdot A3 \cdot dA3 = \text{ReLU}(Z3)' = \text{ReLU}(Z3)' \cdot dA3$$

$$\begin{aligned} dA2 &= \frac{\partial \text{afvigelse}}{\partial A2} = \frac{\partial Z3}{\partial A2} \cdot dZ3 = \frac{\partial}{\partial A2} \cdot Z3 \cdot dZ3 = (A2 \times W2 + B2)' \cdot dZ3 \\ &= (1 \times W2 + 0) \cdot dZ3 = (1 \cdot dZ3) \times W2 = dZ3 \times W2 \end{aligned}$$

$$dZ2 = \frac{\partial \text{afvigelse}}{\partial Z2} = \frac{\partial A2}{\partial Z2} \cdot dA2 = \frac{\partial}{\partial Z2} \cdot A2 \cdot dA2 = \text{ReLU}(Z2)' \cdot dA2 = \text{ReLU}(Z2)' \cdot dA2$$

Herefter kommer vægtene:

$$\begin{aligned} dW3 &= \frac{\partial \text{afvigelse}}{\partial W3} = \frac{\partial Z4}{\partial W3} \cdot dZ4 = \frac{\partial}{\partial W3} \cdot Z4 \cdot dZ4 = (A3 \times W3 + B3)' \cdot dZ4 \\ &= (A3 \times 1 + 0) \cdot dZ4 = A3 \times (dZ4 \cdot 1) = A3 \times dZ4 \end{aligned}$$

$$\begin{aligned} dW2 &= \frac{\partial \text{afvigelse}}{\partial W2} = \frac{\partial Z3}{\partial W2} \cdot dZ3 = \frac{\partial}{\partial W2} \cdot Z3 \cdot dZ3 = (A2 \times W2 + B2)' \cdot dZ3 \\ &= (A2 \times 1 + 0) \cdot dZ3 = A2 \times (1 \cdot dZ3) = A2 \times dZ3 \end{aligned}$$

$$\begin{aligned}
 dW_1 &= \frac{\partial \text{afvigelse}}{\partial W_1} = \frac{\partial Z_2}{\partial W_1} \cdot dZ_2 = \frac{\partial}{\partial W_1} \cdot Z_2 \cdot dZ_2 = (A_1 \times W_1 + B_1)' \cdot dZ_2 \\
 &= (A_1 \times 1 + 0) \cdot dZ_2 = A_1 \times (1 \cdot dZ_2) = A_1 \times dZ_2
 \end{aligned}$$

Til sidst kommer bias:

$$\begin{aligned}
 dB_3 &= \frac{\partial \text{afvigelse}}{\partial B_3} = \frac{\partial Z_4}{\partial B_3} \cdot dZ_4 = \frac{\partial}{\partial B_3} \cdot Z_4 \cdot dZ_4 = (A_3 \times W_3 + B_3)' \cdot dZ_4 = (0 + 1) \cdot dZ_4 \\
 &= dZ_4
 \end{aligned}$$

$$\begin{aligned}
 dB_2 &= \frac{\partial \text{afvigelse}}{\partial B_2} = \frac{\partial Z_3}{\partial B_2} \cdot dZ_3 = \frac{\partial}{\partial B_2} \cdot Z_4 \cdot dZ_3 = (A_2 \times W_2 + B_2)' \cdot dZ_3 = (0 + 1) \cdot dZ_3 \\
 &= dZ_3
 \end{aligned}$$

$$\begin{aligned}
 dB_1 &= \frac{\partial \text{afvigelse}}{\partial B_1} = \frac{\partial Z_2}{\partial B_1} \cdot dZ_2 = \frac{\partial}{\partial B_1} \cdot Z_2 \cdot dZ_2 = (A_1 \times W_1 + B_1)' \cdot dZ_2 = (0 + 1) \cdot dZ_2 \\
 &= dZ_2
 \end{aligned}$$

Nu er alle ændringerne for både vægte og bias fundet. De nye vægte bliver regnet ved

$W_i = W_i - dW_i$ . For bias er det  $B_i = B_i - dB_i$ .

### 3.5 Implementering af det neurale netværk

Når neurale netværk bliver implementeret i kode, er det ofte med biblioteker som Tensorflow eller Pytorch. De har alle de basale ting klar på forhånd, såsom aktiveringsfunktioner, forward propagation og backpropagation. Men der er ingen grund til at introducere en masse grundlæggende teori og matematik, hvis ikke det bliver brugt. Derfor valgte jeg at implementere det neurale netværk, der skulle kunne genkende håndskrevne tal, helt fra bunden i kodesproget Python. En forsimplet version af den fulde kode, er kopieret ind i bilagssektionen. Der ligger også et link i samme sektion til min GitHub, hvor både koden og datasættene til at træne netværket ligger. Der vil også ligge en ReadMe-fil med instrukser, til hvordan du kan køre koden, og selv prøve at træne det neurale netværk, og lege rundt med parametrene, og se hvilken indflydelse det har på nøjagtigheden.

I afsnit "3.1 Forward propagation" blev alle funktionerne for de forskellige lag i forward propagation opskrevet. Det er de samme som skal bruges i Python. Funktionen for  $Z_2$  var  $Z_2 = A_1 \times W_1 + B_1$ . I Python skrives matrixmultiplikation med operatoren @, hvilket gør funktionen til  $z_2 = a_1 @ w_1 + b_1$  som ses på Figur 15. Aktiveringsfunktionerne skrives op på samme måde som i afsnit "3.1 Forward propagation", hvor  $a_2 = \text{ReLU}(z_2)$ .

```
def forward_propagation(a1,w1,w2,w3,b1,b2,b3):
    z2 = w1@a1+b1
    a2 = ReLU(z2)

    z3 = w2@a2+b2
    a3 = ReLU(z3)

    z4 = w3@a3+b3
    a4 = softmax(z4)

    return(a1,z2,a2,z3,a3,z4,a4)
```

Figur 15: Forward propagation i Python. Matrixmultiplikation skrives med @ mellem de to matricer.

I afsnit "3.4 Kædereglene" blev samtlige funktioner til backpropagation skrevet op. Funktionen for  $dZ_3$  kunne skrives som  $\text{ReLU}(Z_3)' \times dA_3$  hvor  $dA_3$  kunne skrives som  $W_3 \times dZ_4$ . Da funktionen  $dA_3$  giver det bedre mening at implementere  $dZ_3$  som en enkelt funktion, hvor  $dZ_3 = \text{ReLU}(Z_3)' \cdot dZ_4 \times W_3$ . I Python er der byttet rundt på dimensionerne af både neuronlagene og vægtlagene, da det gjorde det lettere at implementere. Derfor matrixmultiplikationen i omvendt rækkefølge på Figur 16, selvom matrixmultiplikation som nævnt i afsnit "2.5.4 Matrixmultiplikation" ikke er kommutativ.

```
def backpropagation(targetlist,a1,z2,a2,z3,a3,z4,a4,w1,w2,w3):
    dz4 = targetlist - a4

    db3 = dz4
    dw3 = dz4@a3.T
    dz3 = w3.T@dz4*ReLU_dif(z3)

    db2 = dz3
    dw2 = dz3@a2.T
    dz2 = w2.T@dz3*ReLU_dif(z2)

    db1 = dz2
    dw1 = dz2@a1.T
    return(db1,db2,db3,dw1,dw2,dw3)
```

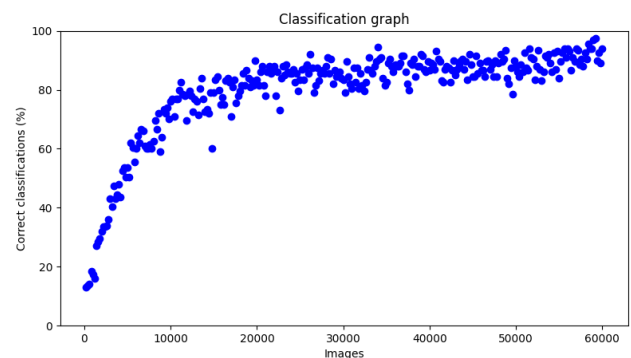
Figur 16: Backpropagation i Python. Mads Hoff

Nu kommer der tre nye begreber på banen: learning rate, batch størrelse og epoch. De er vigtige i træningen af det neurale netværk, men hænger ikke direkte sammen med resten af den teoretiske matematik. Learning rate er en konstant, der bestemmer, hvor store skridt, der bliver taget i steppet med gradient descent. Den er tricky at få rigtig, da en for stor værdi, betyder at man tager alt for store skridt, hver gang man justerer vægtene. Omvendt betyder en for lille learning rate, at man tager museskridt, der er så små, at netværket slet ikke forbedrer sig. Jeg fandt frem til at det optimale interval lå mellem 0,05 og 0,3. Alt andet uden for dette interval gjorde at netværket ikke så nogen fremgang i træningen.

Batch størrelse refererer til, hvor mange billeder man går igennem, før man ændrer vægtene. Det er igen en fin balancegang. Hvis man ændrer vægtene og bias efter at have været for få billeder igennem, er den mængde vægtene og bias bliver ændret alt for tilfældig. Hvis man omvendt venter for lang tid imellem at opdatere vægtene, tager det meget lang tid for netværket at forbedre. Generelt virkede en batchstørrelse på alt mellem 50-2000 fornuftigt nok.

Antallet af epochs fortæller, hvor mange gange man kører træningsdatasættet igennem det neurale netværk. Det er igen en fin balance. Jo flere gange man kører datasættet igennem desto mere præcist bliver netværket. På Figur 17 ses det neurale netværks forbedring over tid. Jo flere billeder det har set, desto færre fejl laver det.

Allerede efter de blot 5.000 første billeder, gætter netværket korrekt omkring 50% af tiden. Men selv efter samtlige 60.000 billeder, har netværket

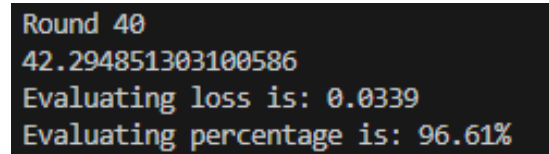


Figur 17: Klassifikationsgrafen viser, hvor mange billeder det neurale netværk gætter korrekt. Jo flere billeder det har set, desto færre fejl laver det.

Mads Hoff

stadig kun en klassificeringsprocent på 89%. Derfor giver det mening at køre hele datasættet igennem igen flere gange. Det sker dog på bekostning af at det tager længere tid og kræver flere ressourcer. Samtidig er der også en øvre grænse for, hvor mange gange man kan køre datasættet igennem. Det er det tidspunkt man når overfitting. Det betyder at det neurale netværk er begyndt at huske de specifikke billeder i træningsdatasættet, frem for at lære mønstre. Netværket bliver hele tiden bedre og bedre til at klassificere træningsdatasættet, men når man prøver netværket af på testdatasættet, bliver det dårligere. Når man når det punkt at netværket bliver dårligere på testdatasættet har man nået overfit grænsen og bør stoppe træningen. Det er derfor det er vigtigt at have et opdelt datasæt, hvor der er en lille del reserveret til testning af det neurale netværk.

Jeg prøvede at optimere netværket til at få så høj klassificeringsprocent som mulig ved at justere batch størrelse og learning rate. Det bedste jeg kom frem til med strukturen 784-32-32-10, var med batchstørrelsen 200 og learning-rate på 0,05. Her ramte netværket overfit-grænsen omkring 40 epochs, hvor det havde en klassificeringsprocent på 96,61%.



```
Round 40
42.294851303100586
Evaluating loss is: 0.0339
Evaluating percentage is: 96.61%
```

Figur 18: Klassificeringsprocent af et neuralt netværk med strukturen 784-32-32-10 efter 40 epochs med batchstørrelse 200 og learning rate på 0,05  
Mads Hoff

### 3.6 Dynamiske lag

Som set på Figur 16 på side 19 er funktionerne for de forskellige lag næsten ens. Ændringen af vægten i et lag er resultatet af ændringen af neuronet i forrige lag matrixmultipliseret med det aktiverede neuron i det nuværende lag. Det kan der opstilles en generaliseret formel for:

$$dW_i = dZ_{i+1} \times A_i$$

For neuronerne er formelen:

$$dZ_i = Relu'(Z_i) \cdot dZ_{i+1} \times W_i$$

Med disse formler behøver man ikke længere finde hver enkelte differentierede formel til gradient descent. Det gør det muligt at have et dynamisk antal lag, som er bestemt ud fra én enkelt numerisk variabel. Det er ikke nogen let ting at implementere, men det lykkedes. Denne version af koden er markant længere og er uploadet på GitHub. Det gør det dog muligt at lave nogle neurale netværk med både utroligt mange skjulte lag, men også med 0. Med 0 skjulte lag bliver output neuronerne forbundet direkte til input-neuronerne med et lag af vægte. Overraskende kan det neurale netværk trænes til at komme helt op på 90,69% nøjagtighed med denne konfiguration. Det skyldes at datasættet ikke er ret komplekst, da det kun består af 784 pixels. For større billeder eller andre mere komplekse applikationer, vil et neuralt netværk med 0 skjulte lag kun kunne gøre det marginalt bedre end at gætte tilfældigt, da det mangler kompleksiteten til at lære at genkende mønstre.

Med dynamiske lag blev den optimale struktur fastsat til at have 3 skjulte lag med 64 neuroner hver. Det giver strukturen 784-64-64-64-10. Her kom klassificeringsprocenten op på 97,41%.



## 4. Konklusion

For at forstå moderne machine learning, er det nødvendigt først at forstå teorien bag neurale netværk og deres opbygning. For at kunne konstruere et neuralt netværk er det vigtigt med en forståelse for, hvordan den bagvedliggende matematik såsom matrixoperationer, gradient descent og kædereglen spiller sammen. Her er kædereglen ekstra vigtig for at kunne finde frem til forholdet mellem de enkelte vægte og den totale afvigelse som bruges i træningsprocessen af det neurale netværk kaldet backpropagation.

Opgaven kom også frem til at det er muligt at implementere den teoretiske matematik i praksis. Et neuralt netværk blev konstrueret og trænet til at genkende håndskrevne tal. Med strukturen 784-32-32-10, der refererer til antallet af neuroner i de forskellige lag, endte netværket med en klassificeringsprocent på 96,61%. Det skete ved at finjustere batchstørrelsen og learning rate, hvorefter netværket blev trænet indtil det ramte overfitting-stadiet og træningen blev stoppet. Ved at implementere dynamiske lag, blev der fundet en endnu mere optimal struktur for netværket med 3 skjulte lag med 64 neuroner i hvert. Det resulterede i en klassificeringsprocent på 97,41%.

## 5. Litteraturliste

1. Deisenroth, Marc Peter (m.fl.): Mathematics For Machine Learning, Cambridge University Press, 2020
2. Facebook: "Llama2", u.d., [https://huggingface.co/docs/transformers/model\\_doc/llama2](https://huggingface.co/docs/transformers/model_doc/llama2) (sidst besøgt d. 19.03.2025)
3. GeeksforGeeks: "Kaiming Initialization in Deep Learning", 27.12.2023, <https://www.geeksforgeeks.org/kaiming-initialization-in-deep-learning/> (sidst besøgt d. 19-03-2025)
4. Hesselholt, Lars (m.fl.): "Lineær Algebra", IMFs Matematiknoter, Oktober 2016, <https://noter.math.ku.dk/linalg16.pdf> (sidst besøgt 15.03.2025)
5. IBM: "What is a neural network?", u.d., <https://www.ibm.com/think/topics/neural-networks> (sidst besøgt 17.03.2025)
6. LeCun, Yann m.fl.: "The Loss Surfaces of Multilayer Networks", Courant Institute of Mathematical Sciences, 21.01.2015 <https://arxiv.org/pdf/1412.0233> (sidst besøgt d. 20.03.2025)
7. Sanderson, Grant: "Gradient descent, how neural networks learn", 3blue1brown, 16.10.2017, <https://www.3blue1brown.com/lessons/gradient-descent> (sidst besøgt d. 20.03.2025)
8. Starmer, Josh: "Cross Entropy Derivatives and Backpropagation", Statquest, 01.03.2021, <https://www.youtube.com/watch?v=xBEh66V9gZo> (sidst besøgt d. 21.03.2025)
9. Wikipedia: "Rectifier (neural networks)", u.d., [https://en.wikipedia.org/wiki/Rectifier\\_\(neural\\_networks\)](https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) (sidst besøgt d. 17.03.2025)

10. Wikipedia: “Backpropagatio”, u.d., <https://en.wikipedia.org/wiki/Backpropagation> (sidst besøgt d. 21.03.2025)

## 6. Bilag

1. MNIST dataset. <https://www.kaggle.com/datasets/hojjatk/mnist-dataset>
2. Kode. Nedenunder er forsimplet version af koden med statiske lag i det neurale netværk kopieret ind. Den komplette kode, hvor der er implementeret dynamiske lag, ligger på min GitHub: <https://github.com/HoffLund/Neural-network>. Her kan man også tilgå datasættene til træning, hvis man vil prøve at træne det fulde netværk selv.

```
import numpy as np                # used for the the matrix datastructure
import matplotlib.pyplot as plt   # used for plotting the graphs
import time                       # used for tracking the training time

image_size = 28                  # ⚠ Shouldn't be changed - Read note:          Should only
                                # be changed if the dataset is changed to another one with other image dimensions
output_size = 10                 # ⚠ Shouldn't be changed - Read note:          Shouldn't be
                                # changed for this dataset. One output node for each number (0-9). If another dataset
                                # is used it should match the number of categories and each category (eg. shirt)
                                # should be assigned a category number (eg shirt=2)

learning_rate = 0.05             # ✓ Can be toggled
batchsize = 200                  # ✓ Can be toggled
epochs = 100                     # ✓ Can be toggled
evaluate = True                  # ✓ Can be toggled (either True or False)
show_error = False               # ⦿ This feature is pretty glitchy, but can be toggled
                                # to true.

def ReLU(x):
    return(np.maximum(0,x))

def ReLU_dif(x):
    return(np.where(x > 0, 1,0))

def softmax(z):
    exp_z = np.exp(z - np.max(z))
    return exp_z / np.sum(exp_z)

def load_images_ubyte(type):
    image_buf = np.fromfile(file=type+'_images/'+type+'-images.idx3-
ubyte', dtype=np.ubyte)[16:]
    label_buf = np.fromfile(file=type+'_images/'+type+'-labels.idx1-
ubyte', dtype=np.ubyte)[8:]
```

```

num_images = len(label_buf)

imagedata = image_buf.reshape(num_images, image_size, image_size, 1)
rawdata = image_buf.reshape(num_images, image_size*image_size, 1)
rawdata = np.divide(rawdata, 255)

return(rawdata, imagedata, label_buf, num_images)

def show_image(image_matrix):
    image = np.asarray(image_matrix).squeeze()
    plt.imshow(image)
    plt.show()

def evaluate(a1, w1, w2, w3, b1, b2, b3):
    evaluate_start = time.time()
    rawdata, imagedata, labelbuf, num_images = load_images_ubyte("test")
    loss = 0
    for i in range(num_images):
        a1 = rawdata[i]
        a1, z2, a2, z3, a3, z4, a4 = forward_propagation(a1, w1, w2, w3, b1, b2, b3)
        target_number = labelbuf[i]
        guess = np.argmax(a4)

        if guess != target_number:
            loss += 1

    print("Evaluating loss is:", loss/num_images)
    print("Evaluating percentage is:", str(round((1-loss/num_images)*100, 3))+"%")
    evaluate_end = time.time()
    print("Evaluate time:", evaluate_end-evaluate_start)
    return

def initialize_layers():
    w1 = np.random.randn(32, 784) * np.sqrt(2 / 784)
    w2 = np.random.randn(32, 32) * np.sqrt(2 / 32)
    w3 = np.random.randn(10, 32) * np.sqrt(2 / 32)
    b1 = np.zeros((32, 1)) # bias
    b2 = np.zeros((32, 1))
    b3 = np.zeros((10, 1))
    return(w1, w2, w3, b1, b2, b3)

def forward_propagation(a1, w1, w2, w3, b1, b2, b3):
    z2 = w1@a1+b1

```

```

a2 = ReLU(z2)

z3 = w2@a2+b2
a3 = ReLU(z3)

z4 = w3@a3+b3
a4 = softmax(z4)

return(a1, z2, a2, z3, a3, z4, a4)

def apply_backpropagation(w1,w2,w3,b1,b2,b3,db1,db2,db3,dw1,dw2,dw3):
    w1 += dw1*(learning_rate/batchsize)
    w2 += dw2*(learning_rate/batchsize)
    w3 += dw3*(learning_rate/batchsize)
    b1 += db1*(learning_rate/batchsize)
    b2 += db2*(learning_rate/batchsize)
    b3 += db3*(learning_rate/batchsize)

    return(w1,w2,w3,b1,b2,b3)

def backpropagation(targetlist,a1,z2,a2,z3,a3,z4,a4,w1,w2,w3):
    dz4 = targetlist - a4

    db3 = dz4
    dw3 = np.dot(dz4,a3.transpose())
    dz3 = np.dot(w3.transpose(),dz4)*ReLU_dif(z3)

    db2 = dz3
    dw2 = np.dot(dz3,a2.transpose())
    dz2 = np.dot(w2.transpose(),dz3)*ReLU_dif(z2)

    db1 = dz2
    dw1 = np.dot(dz2,a1.transpose())
    return(db1,db2,db3,dw1,dw2,dw3)

def create_loss_plot():
    global fig, ax, x_data, y_data, line

    plt.ion()
    fig, ax = plt.subplots()
    x_data, y_data = [], []
    (line,) = ax.plot([], [], "bo")

    ax.set_ylim(0,1)

```

```

ax.set_xlabel("Iterations")
ax.set_ylabel("Loss")
ax.set_title("Loss graph")

def add_point(x, y):
    x_data.append(x)
    y_data.append(y)
    line.set_xdata(x_data)
    line.set_ydata(y_data)
    ax.relim()
    ax.autoscale_view()
    fig.canvas.draw()
    fig.canvas.flush_events()

w1,w2,w3,b1,b2,b3 = initialize_layers()

rawdata, image_array, labels, num_images = load_images_ubyte("train")
create_loss_plot()

loss = 0

for epoch in range(epochs):
    print("Round", epoch)
    starttime = time.time()
    for x in range(num_images//batchsize):
        db1,db2,db3,dw1,dw2,dw3 = [0,0,0,0,0,0]
        for i in range(batchsize):
            a1 = rawdata[x*batchsize+i]
            a1,z2,a2,z3,a3,z4,a4 = forward_propagation(a1,w1,w2,w3,b1,b2,b3)

            target_number = labels[x*batchsize+i]
            guess = np.argmax(a4)
            if guess != target_number:
                loss +=1
                if show_error == True:
                    print("Guessed:", guess)
                    print("Actual number was:", target_number)
                    print()
                    show_image(a1.reshape(28,28))

        targetarray = np.resize(0, (10,1))
        targetarray[target_number] = 1

```

```
        db1_,db2_,db3_,dw1_,dw2_,dw3_ =  
list(backpropagation(targetarray,a1,z2,a2,z3,a3,z4,a4,w1,w2,w3))  
        db1,db2,db3,dw1,dw2,dw3 =  
db1+db1_,db2+db2_,db3+db3_,dw1+dw1_,dw2+dw2_,dw3+dw3_  
  
        w1,w2,w3,b1,b2,b3 =  
apply_backpropagation(w1,w2,w3,b1,b2,b3,db1,db2,db3,dw1,dw2,dw3)  
  
        add_point((x+1)*(batchsize),loss/batchsize)  
        loss = 0  
  
endtime = time.time()  
print(endtime-startime)  
evaluate(a1,w1,w2,w3,b1,b2,b3)
```