H.C. Ørsted Gymnasiet

Opgaveformulering SOP 2021-22

Klasse:	3a1
Navn:	Marius Johan Schlichtkrull
Fødselsdato	

Fag:	Niveau:	Vejleder navn:	Vejleders mailadresse:
Matematik	A	Sara Hillerup Andersen	shan@tec.dk
Digital Design & Udvikling	A	Anders Juul Refslund Petersen	ajrp@tec.dk

Hvordan kan gradient descent algoritmen anvendes til computer-genkendelse af håndskrevne tal?

Redegør kort for teorien bag neurale netværk.

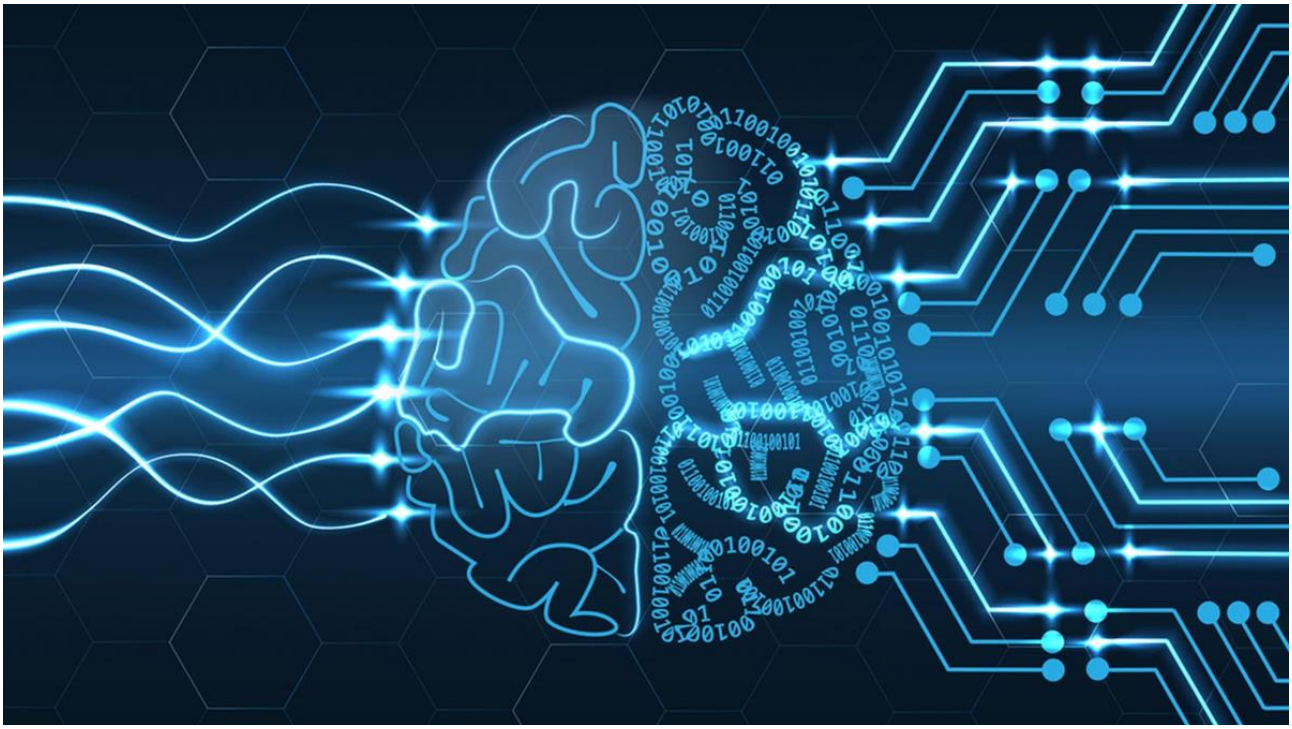
Redegør for gradient descent algoritmen og matematikken bag denne.

Design og implementer et neuralt netværk til genkendelse af håndskrevne tal.

Analyser og vurder resultaterne fra den implementerede algoritme.

Perspektiver til fremtidige anvendelser af gradient descent algoritmen, AI generelt og evt. hjernens indlæring.

Udleverede bilag	
Opgaven udleveres	**04. marts 2022 kl. 12:00**
Opgaven skal afleveres	**18. marts 2022 kl. 12:00**



Hvordan kan Gradient Descent algoritmen anvendes til computergenkendelse af håndskrevne tal

Marius Johan Schlichtkrull

H.C. Ørsted Gymnasiet

18/02/2022

Table of Contents

1. Resumé

Denne opgave handler om neurale netværk, hvordan man fandt på ideen, og hvordan dataloger beskriver dette matematisk. Opgaven vil også undersøge hvordan optimeringsalgoritmen Gradient Descent fungerer, og hvordan det neurale netværk bliver forbedret vha. Gradient Descent og backpropagation. Opgaven fremlægger også beviser for hvorfor disse metoder lige netop er anvendelige og relevante. Herefter anvendes teorien til at opstille et neuralt netværk i Python og NumPy og implementere det til praktisk anvendelse. Netværket klarer sig ikke helt som ønsket i første omgang, og der gennemgås derfor en optimeringsproces, hvor parametrene i det neurale netværk justeres. Særligt har den rigtige learning rate en stor indflydelse på netværkets performance. Til sidst perspektiveres der til store neurale netværk som har relevante anvendelses formål i mange af de største brancher, samt andre former for optimering af neurale netværk og endelig perspektiveres der til kunstig super intelligens, og Gradient Descent mulige use case deri.

2. Indledning

Lige siden Alan Turing opfandt den første computer, har dataloger, matematikere og fysikere haft tanken om at udvikle kunstig intelligens svarende til menneskers intelligens. Ved intens forskning igennem mange år blev forskere i højere og højere grad klar over, hvis man ønsker at bygge en kunstig hjerne, bliver man nødt til at efterligne, hvordan hjernen fungerer. Man fandt derfor metoder til at bygge neurale netværk af kunstige neuroner, og en gradvis optimering af disse netværk frem til i dag har ført til, at disse kan drive dele af internettet. Eksempelvis bruger Facebook i dag kunstig intelligens til at udvælge relevante reklamer til deres brugere, og Google bruger kunstig intelligens til at oversætte sprog igennem Google Translate. Disse algoritmer er alle optimeret vha. Gradient Descent, eller varianter af Gradient Descent som i korte træk fungerer ved at man minimerer en funktion, som beskriver differencen mellem det forudsagte svar og et kendt facit.

3. Teorien bag neurale netværk

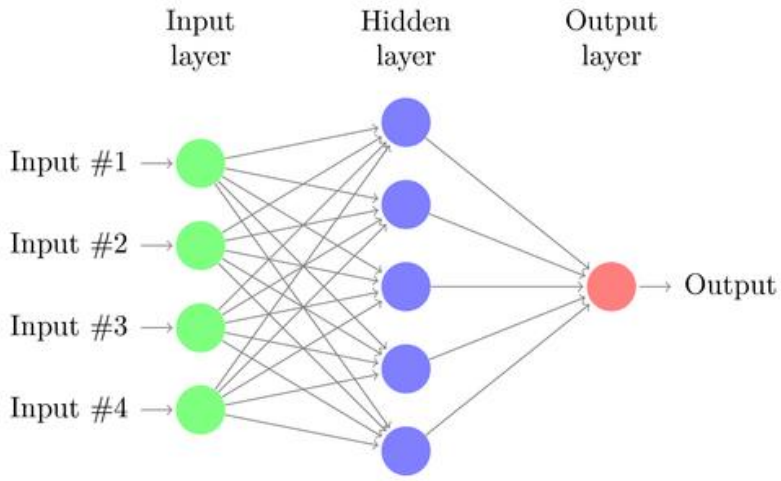
Neurale netværk er i sin grundform et forsøg på at efterligne hjernen og evnen til at tillære sig viden.

3.1 Hjernens virkemåde

Nøglen til at udvikle kunstige neurale netværk har været en dybere forståelse af, hvordan hjernen fungerer. Hovedsageligt består hjernen af mange milliarder neuroner - celler, som modtager og sender elektriske signaler videre over synapsekløfte som er en forbindelse mellem hver enkel neuron. Baseret på det input den individuelle neuron får, ændrer outputtet sig også. Dette gør hjernen i stand til at kunne se forskelle. Eksempelvis kan ens synsneuron derfor reagere kraftigere på et rødt bær end et grønt æble. Styrken af signalet afhænger af de små og gemte træk, som gør noget unikt. Hjernen regulerer sig selv ved at sende hæmmende og forstærkende molekyler over disse synapsekløfte. På den måde op- og nedjusterer den også signalet for den næste neuron.[1]

3.2 Opbygningen af neurale netværk

Denne viden er på nuværende tidspunkt grundstenene til at efterligne de mekanismer som foregår i hjernen og altså grundlaget for at skabe et neuralt netværk. Spørgsmålet er så bare lige hvordan det udføres i praksis. Lige nu skal man have skabt kunstige neuroner og kunstige synapsekløfte. Dataloger og matematikere arbejder med ideen om at bruge en graf til at repræsentere hjernen, hvor knuderne i grafen symboliserer neuronerne i hjernen, og kanterne symboliserer synapsekløften og dens styrke. På figur 3.1 ses hvordan man kan visualisere sådan en graf. [2]



Figur 3.1: Illustration af et simpelt neuralt netværk
https://texample.net/media/tikz/examples/PNG/neural-network.png

3.3 Kunstige neuroners matematik

Neuronens funktion er at modtage signaler og videresende disse signaler. Dette kan gøres ved at tage vektor-summen af alle "signalerne" som neuronen modtager. Kigger man på *figur 3.1*, kan man se, at neuronerne i "hidden-lag", altså lægger $input_1$, $input_2$, $input_3$ og $input_4$ sammen. Vi skal samtidig huske på, at vi også skal inkludere signalstyrken i vores neuron. Dette kan vi så gøre ved at benytte os af vægte. Vægtene skal symbolisere i hvor høj grad neuronen får hæmmende eller forstærkende molekyler. Har man en vægt på 10, kan vi altså se, at den forstærker signalet i højere grad sammenlignet med en vægt på 0,1.

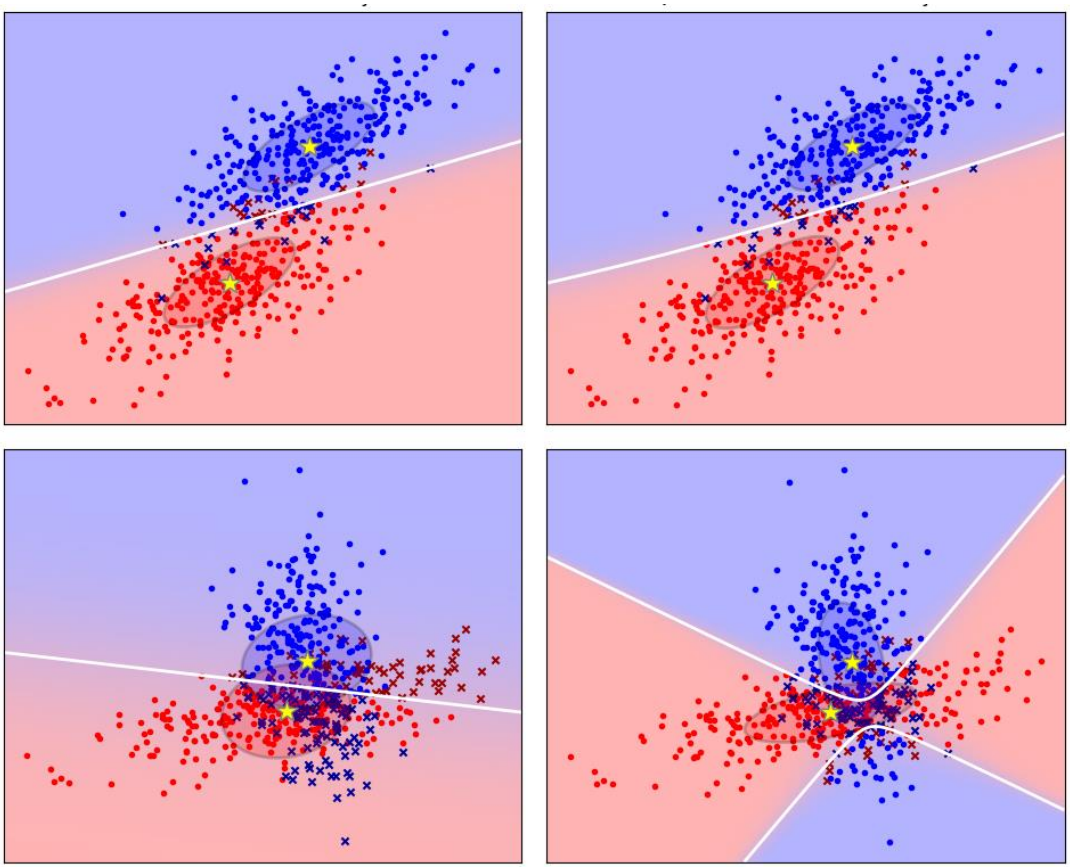
[1] Kim Scott (2016) s. 1
[2] IBM Cloud Education (2020)

Neuronen foretager en beregning af regnestykket:

$$\sum_{i=1}^{n} W_i \cdot x_i$$

Fra vektormatematikken ved vi,at prikproduktet mellem to n-dimensionelle vektorer er: $a_x \cdot b_x + a_y \cdot b_y + \cdots + a_{n-dim} \cdot b_{n-dim} = \sum_{i=1}^{n-dim} a_i \cdot b_i$

Det kan omskrives til: $\vec{W} \cdot \vec{x}$ eller en lineær funktion $f(x) = \vec{W} \cdot \vec{x}$. Det er imidlertid ønskeligt at udvikle en mere avanceret funktion, som kan beskrive mere komplekse informationer, hvilket er forsøgt illustreret på *figur 3.2*.



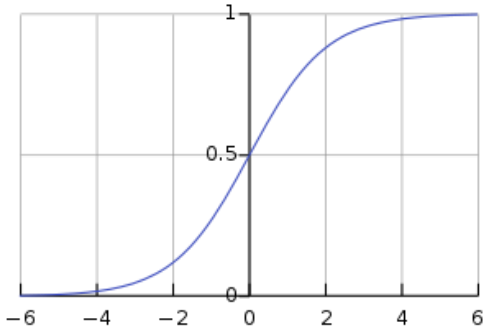
Figur 3.2: Eksempel på hvordan typer af regression kan påvirke et resultat
*https://miro.medium.com/max/1000/1*e0Jf0jSEAJOMMTeNZCapzg.png*

Figurerne illustrerer hvordan en mere kompleks funktion er bedre til at beskrive mere komplekse data. I de to nederste figurer kan vi se, hvordan informationen ikke kan opdeles ved blot en lineær funktion, mens en anden type funktion med flere parametre lige netop kan løse opgaven. Derfor introducerer man aktiveringsfunktioner samt bias. Aktiveringsfunktioner har den egenskab, at de kan gøre en lineærfunktion ikke-lineær.[3]

Eksempelvis ses sigmoid-funktion i figur 3.3, som har funktionsforskriften

[3]Torsa Talukdar (2020)

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



Figur 3.3: Visualisering af sigmoid-funktion
https://en.wikipedia.org/wiki/File:Logistic-curve.svg

Indsættes neurons outputværdi i aktiveringsfunktionen fås:

$$\sigma\left(\vec{W} \cdot \vec{x}\right) = \frac{1}{1 + e^{-(\vec{W} \cdot \vec{x})}}$$

Udtrykket udgør output-værdien og viser, at funktionen lige netop ikke er lineær.

I hjernens egne neuroner skal der en vis elektrisk strøm til at aktivere neuronen. Derfor kan vi indsætte et såkaldt bias eller threshold, som sørger for, at signalet er størrer end *b*, før den kan aktiveres.

$$\sigma\left(\vec{W} \cdot \vec{x} + bias\right) = \frac{1}{1 + e^{-(\vec{W} \cdot \vec{x} + bias)}}$$

3.4 Kunstige lag af neuroner

I beskrivelsen af et helt neuralt netværk benyttes matricer fremfor enkelte vektorer. Matricer kan ses som n-dimensionelle vektorer som indeholder m-dimensionelle vektorer.

Et eksempel på en matrix med 3 rækker og 2 kolonner:

$$m = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \\ m_{31} & m_{32} \end{bmatrix}$$

Lader vi en matrix repræsentere vægtene for et helt lag i det neurale netværk, så vil rækker og kolonner henholdsvis symbolisere antallet af neuroner i det givne lag samt antallet af inputs, hvilket i mange tilfælde vil være antallet af neuroner i forrige lag. Så repræsenterer vi lagene igennem matricer, skal vi altså kunne gange to matricer sammen. Dette gøres ved matrix-multiplikation, som er prik produktet mellem den første række i matrix m_1 og den først kolonne i m_2, hvor resultatet så vil være den første værdi i den nye matrix.

Her er et eksempel på, hvordan matrix multiplikation vil udføres for en 2x3 matrix og 3x2 matrix[4]:

[4] Bill Shillito (2013)

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \\ m_{31} & m_{32} \end{bmatrix}$$
$$= \begin{bmatrix} w_{11} \cdot m_{11} + w_{12} \cdot m_{21} + w_{13} \cdot m_{31} & w_{11} \cdot m_{12} + w_{12} \cdot m_{22} + w_{13} \cdot m_{32} \\ w_{21} \cdot m_{11} + w_{22} \cdot m_{21} + w_{23} \cdot m_{31} & w_{21} \cdot m_{12} + w_{21} \cdot m_{22} + w_{23} \cdot m_{32} \end{bmatrix}$$

Eksemplet for matrix multiplikation viser, at det er vigtigt, at de to matricer har den rette størrelse. Den ene matrix man multiplicerer med, skal have det samme antal rækker, som den anden matrix har kolonner. Havde *w*-matrixen haft 4 kolonner, havde man ikke kunnet tage prik produktet mellem den 4. kolonne med den 4. række -, (den findes jo ikke).

Det er også muligt at lægge matricer sammen. Dette kan udnyttes, når bias lægges til. Biases skal derfor være en matrix, der har 1 række og n-kolonner, som passer med matrix, og som også har m-rækker og n-kolonner.

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix}$$

Når vi så indsætter disse matricer i en funktion, svarer det til, at vi kører funktionen på hvert enkelt tal i matrixen.

$$\sigma\left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right) = \begin{bmatrix} \sigma(1) & \sigma(2) & \sigma(3) \\ \sigma(4) & \sigma(5) & \sigma(6) \\ \sigma(7) & \sigma(8) & \sigma(9) \end{bmatrix}$$

Det minder egentlig meget om skalarproduktet, idet skalaren også bliver ganget ind på hver enkel tal, her er det bare en funktion i stedet for et skalar.

Opsummering: Neurale netværk er et forsøg på at eftergøre den måde som hjernen fungerer på. Vi beskriver altså den kemiske virkelighed i matematik. Synapsekløfte simuleres ved at fastlægge vægte, og hvert neuronsignal beregnes som en vægtet sum. I beskrivelsen af neurale netværk gøres der brug af lineæralgebra, dvs. matrix regning, for at beregne hvordan hvert lag, i det neurale netværk håndterer det givne input. I bestræbelserne på at forfine resultaterne/metoderne (undgå, en lineær model), udnytter vi aktiveringsfunktioner, som kan gøre den samlede funktion mere kompleks, og derfor også beskrive mere kompleks information.

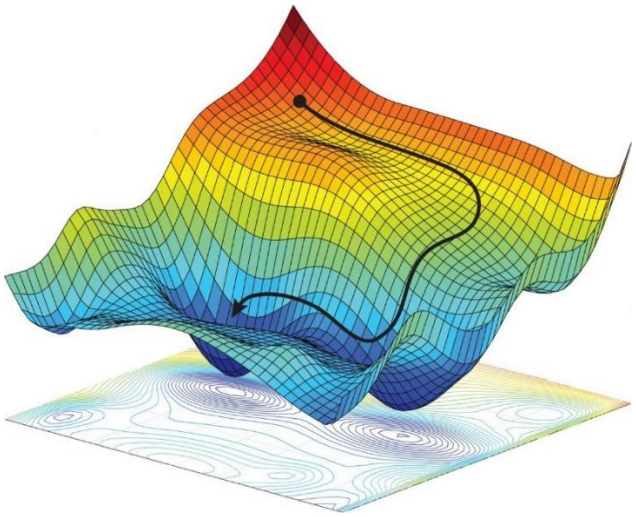
4. Teorien og matematikken bag Gradient Descent

4.1 Ideen bag loss-funktion og Gradient Descent

Det forgående udgør skitsen til et statisk neuralt netværk, som kan forsøge at afgøre om et billede er en hund eller en kat. Forskers hypotese for, hvorfor vi bliver klogere drejer sig om at molekylerne ændre i vores synapsekløfte. Overført til et neuralt netværk, skal vi altså nu fastlægge, hvordan man helt præcist kan opdatere sådan et neuralt netværk. Først og fremmest kan man benytte sig af en genetisk algoritme til at optimere det neurale netværk. Men idet genetiske algoritmer i høj grad bruger sandsynligheden som drivkraft, er denne træningsmetode yderst langsom sammenlignet med andre metoder, såsom Gradient Descent. Gradient Descent er en metode til at finde lokale minimum, som så kan udnyttes, hvis man vil minimere en funktion. Derfor kan vi opstille en Cost-funktion, som viser, hvor fejlagtig et neuralt netværk kan være. Der findes forskellige Cost-funktioner. Her tages der udgangspunkt i Mean Squared Error funktionen (MSE). Funktionen tager gennemsnittet af den kvadrerede forskel mellem det faktiske resultat og det neurale netværks resultat. Det kan matematisk beskrives ved formlen:[5]

$$MSE = \frac{1}{2n}\sum_{x}\|y(x) - a(x)\|^2$$

$y(x)$ er det faktiske resultat for det givne billede og $a(x)$ er hvad det neurale netværk gætter på. Længden af vektorforskellen kvadreres. Herefter tages gennemsnittet af alle de kvadrerede forskelle. Målet er at minimere denne værdi, sådan at netværket minimerer fejlene.[5]



Figur 4.1: Illustration af Gradient Descent
https://miro.medium.com/max/1200/0*kDWssJidqAEeZQuX.jpg

På *figur 4.1* ses hvordan resultaterne kan beskrives som et formet landskab, hvor værdierne er afbildet i et 3-dimentionelt system, som egentlig består af de forskellige Cost-værdier, når vægtene har en given værdi. Egentlig, burde dette landskab have mange flere dimensioner - dimensionerne er lig med antallet af vægte og bias, vi skal optimere på - men for at visualiserer, hvad computeren egentlig gør, plotter vi den på denne måde.

Gradient Descent anvendes således til at minimere Cost-funktionen, ved at udpege den retning hver enkelt vægt på skal ændres i.

[5] Michael Nielsen (2019) Kap. 1

Da vi har opbygget det neural netværk således, at det indeholder flere lag, som er afhængige af hinanden, kan vægtene ikke ændres på samme tid. Det er så her back propagation kommer ind i billedet. Forestiller vi os et neuralt netværk med 5 lag, hvor det 5. lag er outputtet. Idet neuronerne i lag #5 er direkte afhængige af vægtene i det 4. lag, kan vi altså fortælle vægtene i lag 4. at de skal ændre sig på sådan en måde, at lag #5 får det korrekte output. Men idet lag #4 også er afhængig af lag #3 skal vi så ændre lag #3's vægte sådan, at lag #4s neuroner giver det rigtige output og sådan at lag #5 i sidste ende giver den korrekt værdi. Hele ideen er altså, at man går tilbage igennem det neurale netværk for at ændre værdierne.[5]

At omforme denne forholdsvis enkle forklaring til matematik er mulig men vanskelig. Det følgende er en overordnet gennemgang af den underliggende matematik.

4.2 Differentialeregning

Som illustreret i figur 4.1 er målet at bevæge Cost-værdien ned i bunden af "landskabet". Hertil er differentialregning brugbart, da det i bund og grund handler om ændringer. Problemstillingen kan sammenlignes med en blind person, som står på en bakke, og hvis mål er at bevæge sig ned i bunden af bakken. Opgaven kan splittes op i to: fokuser i den rigtige retning og foretage en bevægelse derhen. For at kunne fuldføre denne opgave, er det nødvendigt at identificere hældningen på bakken, så det kan afgøres, hvad der er opad og nedad. Idet gradienten altid vil pege op, kan vi så tage et skridt i modsat retning af vektoren for at minimere funktionen i stedet for at maximere.

Dette kan beskrives matematisk, idet differentialregningen beskriver funktionernes ændring, og hvor hurtigt de ændrer sig. Funktionen $f(x) = 5 \cdot x^2$ beskriver en andengrads funktion. Her kan det være svært at afgøre, hvad hældningen er for et specifikt punkt, da den jo hele tiden ændrer sig. Metoden til at afgøre hældningen findes i tretrinsreglen som lyder:

Trin 1:
Vi definerer til at starte med hvordan Δy ser ud. Dette kan vi gøre ved at tilføje Δx til funktionen. Så vi starter i x_0 og tilføjer Δx

$$\Delta y = f(x_0 + \Delta x) - f(x_0)$$

Trin 2:
Idet man ønsker at finde hældningen, kan vi til at starte med definere hældningen helt matematisk til at være: $a = \frac{\Delta y}{\Delta x}$ hvor en ændring i *x*, medfører en lille ændring i *y*. Hvis Δy betyder forskellen mellem y_0 og y_1.
Udtrykket omformuleres og gøres lidt mere specifikt for hver enkel funktion:

$$a_s = \frac{\Delta y}{\Delta x} = \frac{f(x_0 + x) - f(x_0)}{(x + x_0) - x_0} = \frac{f(x_0 + x) - f(x_0)}{x}$$

Trin 3:
Udtrykket betyder at man finder hældningen mellem to punkter $((x_0 + x; f(x_0 + x))$ og $(x_0; f(x_0))$. Vi finder altså sekanthældningen. For at beregne tangentens hældning, kan man lade x gå mod 0, dvs. lade x blive uendelig lille, og på den måde finde hældningen for blot et enkelt punkt, da forskellen mellem de to punkter bliver uendelig lille

$$a_t = \lim_{x \to 0} \frac{f(x_0 + x) - f(x_0)}{x}$$

Herefter ønskes det at reducere udtrykket så meget som overhovedet muligt. Indsætter man andengradsligningen fra før i ligningen for hældningen af tangenten, fås:

$$
\begin{aligned}
a_s &= \frac{5 \cdot (x_0 + x)^2 - 5 \cdot x_0^2}{x} \\
&= \frac{5 \cdot (x_0^2 + x^2 + 2 \cdot x_0 \cdot x) - 5 \cdot x_0^2}{x} \\
&= 5 \cdot x_0^2 + 5 \cdot x^2 + 10 \cdot x_0 \cdot x - 5 \cdot x_0^2 \\
&= \frac{5 \cdot x^2 + 10 \cdot x_0 \cdot x}{x} \\
&= 5 \cdot x + 10 \cdot x_0
\end{aligned}
$$

Idet vi lader *x* gå mod 0, betyder det altså, den bare bliver 0, hvilket vil sige det eneste led der tilbage er $10 \cdot x_0$

$$
\begin{aligned}
a_t &= \lim_{x \to 0} 5 \cdot x + 10 \cdot x_0 \\
&= 10 \cdot x_0
\end{aligned}
$$

Princippet kan anvendes på samme måde på mere komplekse funktioner og udgør en metode til at findehældningen mellem to punkter som er uendelig tæt på hinanden.
Notationen for sådan en differentieret funktion minder lidt om $\frac{\Delta y}{\Delta x}$ idet man bruger et *d* i stedet for Δ
Helt præcist er det: $\frac{dy}{dx}$ eller $\frac{df}{dx}(x)$.

4.3 Partialdifferentialligninger

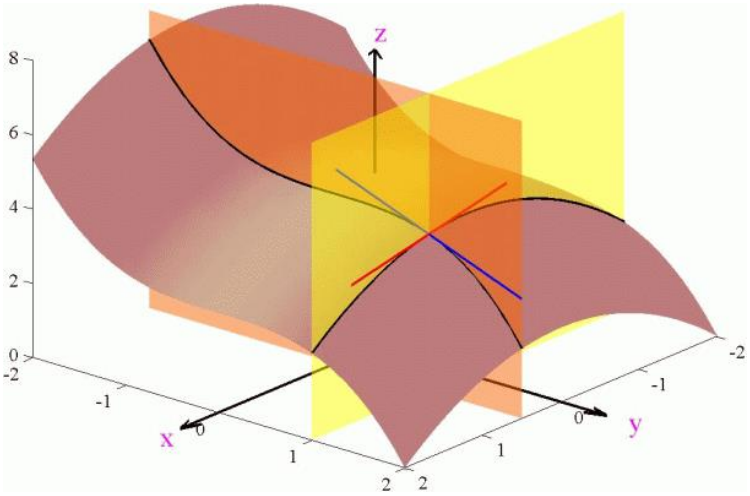
Man kan definere en Cost funktion som bruger to variable (facit og den anslåede værdi). Eksempelvis har MSE Cost følgende funktionsforskriften:

$$MSE(t, y) = \frac{1}{2n} \cdot \sum_i \|t_i - y_i\|^2$$

Man kan så spørge sig selv, hvordan man differentierer sådan en funktion. Den har nemlig mere end blot en variabel. For nu har både en ændring *t* såvel som *y*, samt en ændring i begge variabler på samme tid en indflydelse på ændringen i funktionsværdien. I stedet for at differentiere med hensyn til 2 eller flere variabler, foretages en partiel differentiering som handler om, at man differentierer en funktion med hensyn til en af de variable i funktionen. Partiel differentierer man i forhold til *y* i Cost funktion, lader man som om *t* er konstant.[6]

En god måde at forestille sig partiel differentialligninger på, er hvis man forestiller sig $f(x, y)$ som en funktion, som indeholder flere varianter af funktionen $g(y)(x)$. Dette er selvfølgelig, når man lader *y* være konstant. Dette kan eksempelvis ses i *figur 4.2*.

[6] WebMatematik (ukendt), Partielle afledede, Webmatematik.dk



Figur 4.2: Illustration af funktion med flere uafhængige variable
http://calcnet.cst.cmich.edu/faculty/angelos/m533/lectures/images/pderv1.gif

Det er tydeligt at se på *figur 4.2*, hvordan funktionen består af flere funktioner. Differentierer man funktionen med hensyn til x, kigger man kun på den røde kurve på *figur 4.2*.

Herefter partiel differentierer man funktionen $f(x,y) = 3 \cdot x^3 + 4 \cdot y \cdot x + y$ med hensyn til *x*, mens *y* lades være som en konstant. Funktionen $f(x,y)$ bliver derfor differentieret til:

$$\frac{\partial f}{\partial x}(x,y) = 9 \cdot x^2 + 4 \cdot y.$$

4.4 Gradient vektorer

For at finde det lokale minimum i en Cost funktion, er det særlig velegnet at bruge en gradient vektor. En gradient er en vektor af alle partiel differentialligninger for den givne funktion, dvs. at man har en vektor, hvis koordinater er partiel differentialligninger med hensyn til det givne koordinat. Koordinat-*x* bliver altså partiel differentieret med hensyn til *x* i funktionen, og *y* koordinatet bliver partiel differentieret med hensyn til *y* i funktionen. Det kommer til at se således ud:[7]

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

Vi betegner gradientvektorer med nabla (∇) symbolet.
Gradientvektoren har den egenskab, at den altid peger i den retning, hvor stigningen er størst. Samtidig markerer længden af vektoren, den hastighed som vektoren vokser med. Idet formålet for Gradient Descent, er at finde den hurtigste vej mod et minimum, kan gradientvektorer derfor virke uanvendelig. For at imødegå det, kan man benytte et negativt skalar, som fungerer som skridtlængde for gradienten. Denne værdi kaldes Learning Rate.

$$-\eta \cdot \nabla MSE(\vec{w})$$

Denne gradient vektor viser samtidig, i hvilke retning vi vil bevæge vores vægte, sådan at denne funktion bliver minimeret. Vi bruger gradientvektoren til at opdatere vægtene på følgende måde:

$$\vec{w}_{nye} = \vec{w} - \eta \cdot \nabla MSE(\vec{w})$$

[7] WebMatematik (ukendt), Gradienten af en funktion, Webmatematik.dk

Det samme kan vi gøre for vores bias, ved følgende formel:

$$\vec{b}_{nye} = \vec{b} - \eta \cdot \nabla MSE(\vec{b})$$

For at kunne opdatere hver enkel vægt, kan vi herefter benytte os af back propagation, som er en algoritme, designet til at kunne tilføje disse gradients til hvert lag i det neurale netværk.

4.5 Back Propagation matematik

Som beskrevet sidst under 4.1, at det ikke er muligt at opdatere vægtene på samme tid, hvilket skyldes at vægtene i de sidste lag også er afhængige af vægtene i det første lag. Har man eksempelvis et neuralt netværk med 2 lag og 1 neuron i hvert lag, kan man definere følgende funktioner:

$$L1(x) = w_1 \cdot x + b_1; \quad L2(x) = w_2 \cdot x + b_2$$

Idet laget L1 bliver indsat i næste lag, dvs. L2, indsættes L1 derfor i funktion L2, hvilket giver:

$$L2(L1(x))$$

Substituerer vi L1 og L2 med deres respektive funktionsforskrifter fås:

$$w_2 \cdot (w_1 \cdot x + b_1) + b_2$$
$$= w_2 \cdot w_1 \cdot x + w_2 \cdot b_1 + b_2$$

Vi kan her se, at w_1 også bliver påvirket af vægt w_2. Ideen er altså, at man først vil ændre w_2, hvorefter man ønsker w_1 ændret. Foretages disse beregninger, kan vi bruge kædereglen – kendt fra differentialregning.

4.6 Kædereglen

Som før vist, er et neuralt netværk i princippet en række funktioner indsat i hinanden. For at kunne differentiere dem, anvendes kædereglen. Funktionen $h(x) = f(g(x))$, fører til at $h'(x) = f'(g(x)) \cdot g'(x)$. Dette kan bevises ved at definere $G = g(x)$, sådan at $h(x) = f(G)$. Vi beviser nu kædereglen ved brug af tretrinsreglen forklaret under 4.2.

Trin 1:
Funktions tilvæksten for *h* er: $\Delta y = h(x_0 + \Delta x)$

Trin 2:
Her benytter vi os af at vi har defineret *G* som $g(x)$, hvorefter vi forlænger differenskvotienten med ΔG

$$\frac{\Delta y}{\Delta x} = \frac{\Delta y}{\Delta G} \cdot \frac{\Delta G}{\Delta x}$$

Trin 3:
Grænseværdien for $\Delta x \to 0$

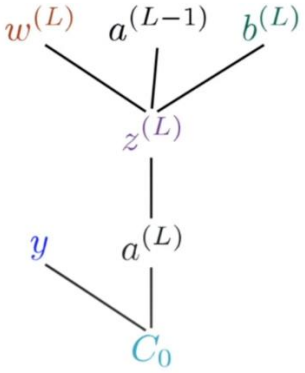
$$\lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta G} \cdot \frac{\Delta G}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta G} \cdot \lim_{\Delta x \to 0} \frac{\Delta G}{\Delta x}$$

Det gælder at grænseværdier kan opdeles i mindre grænseværdier. Dermed kan vi opdele $\lim_{\Delta x \to 0} \frac{\Delta y}{\Delta G} \cdot \frac{\Delta G}{\Delta x}$ til $\lim_{\Delta x \to 0} \frac{\Delta y}{\Delta G} \cdot \lim_{\Delta x \to 0} \frac{\Delta G}{\Delta x}$. Herefter kan vi se, at den først grænseværdi udgøres af $f'(\Delta G)$, og den anden grænseværdi udgøres af $g'(x)$. Idet $f'\big(g(x)\big)$. Idet $G = g(x)$ kan vi altså substituere dette udtryk ind på G's plads. Dette giver udtrykket:

$$f'(G) \cdot G' \implies f'\big(g(x)\big) \cdot g'(x)$$

Dermed er kædereglen bevist.

Tager vi eksemplet fra før, hvor et neuralt netværk har 2 lag med 1 neuron i hvert lag, kan opstille følgende figur, som illustrerer anvendelsen af kædereglen.



Figur 4.3: Illustration af hvordan et enkelt lag er matematisk bygget op
https://www.youtube.com/watch?v=tIeHLnjs5U8

Af *figur 4.3* ses hvordan en neuron har en vægt $w^{(L)}$, bias $b^{(L)}$ og input $a^{(L-1)}$. Herefter udføres regnestykket: $z^{(L)} = w^{(L)} \cdot a^{(L-1)} + b^{(L)}$. $z^{(L)}$ bliver herefter indsat i en aktiveringsfunktion, såsom sigmoid, dermed er $a^{(L)} = \sigma(z^{(L)})$. Til sidst beregnes Cost værdien mellem $a^{(L)}$ og y, som er facit for inputtet ved følgende funktion: $C(a, y)$. Substituerer man funktionernes forskrift, fås udtrykket:

$$C(\sigma(w^{(L)} \cdot a^{(L-1)} + b^{(L)}), y)$$

Outputneuronen kan fastlægges ved overstående matematiske formel. Da vi ønsker at differentiere Cost funktion, bliver vi derfor nødt til at benytte os af kædereglen. Vi ved at en ændring af vægten vil medføre en ændring i den endelige Cost funktion, derfor kan vi skrive følgende: $\frac{\partial C_0}{\partial w^{(L)}}$

Idet vægtene gennemgår en hel række matematiske processor, kan vi altså bruge kædereglen, til at beskrive vægtens vej til et læseligt output. Indledningsvis, så medfører en ændring i *w* samtidig en lille ændring i $z^{(L)}$. Denne ændring i $z^{(L)}$ medfører hermed også en ændring i $a^{(L)}$ som så i sidste ende medfører en ændring i C_0. Opskriver vi alt dette kommer vi frem til udtrykket:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

Denne gradient, kan vi så indsætte i formlen:

$$W_{ny} = W - \eta \cdot \frac{\partial C_0}{\partial w^{(L)}}$$

For at opdatere bias, partiel differentierer vi i forhold til bias i stedet.

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

$$b_{ny} = b - \eta \cdot \frac{\partial C_0}{\partial b^{(L)}}$$

For at udvide denne metode til flere lag, kan vi partiel differentiere med hensyn til $w^{(L-1)}$ på følgende måde:

$$\frac{\partial C_0}{\partial w^{(L-1)}} = \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} \cdot \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \cdot \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

Den eneste forskel i dette udtryk er, at man erstatter $\partial w^{(L)}$ med $\partial a^{(L)}$. Dette gøres fordi $\partial w^{(L)}$ ikke er afhængig af $\partial w^{(L-1)}$

Helt præcist viser udtrykket nedenfor, den gradient vi ønsker at optimere på.

$$\nabla C = \begin{bmatrix} \frac{\partial C_0}{\partial w^{(1)}} \\ \frac{\partial C_0}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C_0}{\partial w^{(L)}} \\ \frac{\partial C_0}{\partial b^{(L)}} \end{bmatrix}$$

5. Implementering af et neuralt netværk til genkendelse af håndskrevne tal

For at kunne genkende håndskrevne tal, vil den nuværende teori omkring neurale netværk være tilstrækkelig. Det er anvendeligt at benytte lineære neuroner, med formlen $\sum_i (w_i \cdot x_i) + b$ til at genkende tal, hvorefter Gradient Descent bruges til at optimere netværket. Programmet programmeres i Python, idet det har et veldesignet bibliotek, som kan beregne lineær algebra. Jeg vil bruge OOP, sådan at jeg let kan op- eller nedskalerer det neurale netværk.

5.1 Design og implementering af lag

Først designes og implementeres et lag som indeholder n, $m \cdot n$ vægte og n bias. Laget skal både indeholde en forward og backward funktion, derfor oprettes et lag som en klasse. Således kan man i princippet have L lag, idet man blot kan kopiere lagets klasse L gange.

Lagets constructor indeholder følgende variable:

`input_nodes, output_nodes, learning_rate.`

Input- og output_nodes bruges til at danne den tilfældigt generede vægt-matrix samt bias-matrix.

```
5   class Linear:
6
7       def __init__(self, input_nodes, output_nodes, learning_rate):
8           self.input_nodes = input_nodes
9           self.output_nodes = output_nodes
10          self.learning_rate = learning_rate
11
12          # Generer tilfældige vægte
13          self.weights = uniform(-1, 1, (input_nodes, output_nodes))
14
15          # Generer tilfældige bias
16          self.biases = uniform(-1, 1, output_nodes)
```

Som beskrevet i under 3.1 omkring teorien bag neurale netværk, så består et feedforward lag blot af en matrix multiplikation mellem vægtene og input matrixen, hvorefter man tillægger et bias. Vi kan altså tilføje følgende funktion til klassen:

```
18      def forward(self, x):
19          """ Denne funktion står for at lave hvert feedforward kald
20              Dvs. at funktionen beregner dot produktet mellem weights
21              og x og lægger bias til
22              w.x + b """
23
24          return dot(x, self.weights) + self.biases
```

Som beskrevet under 4.6, så benytter vi nu kædereglen til at finde henholdsvis vægtens og bias' indflydelse på resultatet. Vi differentierer derfor funktionen med hensyn til både vægtene, x og bias. Resultatet af differentieringen ganges herefter med den forrige gradient i henhold til kædereglen.

Vi opdaterer vægtene og bias' som beskrevet i 4.6. Vi benytter følgende formel:

$$W_{new} = W - \eta \cdot \frac{\partial z}{\partial W}$$

hvor η er vores learning rate, og $\frac{\partial z}{\partial w}$ vores gradient.

Derudover opdateres bias ved formlen $B_{new} = B - \eta \frac{\partial z}{\partial B}$. I praksis ganges blot med forrige gradient, idet $\frac{\partial z}{\partial B} = (w \cdot x + b)' = (c + b)' = 1$. Dog sørger vi også for at størrelserne passer med hinanden. Derfor ganger vi med `x.shape[0]`. Vi kan så skrive følgende kode:

```
26      def backward(self, x, prev_grad):
27          """ Denne funktion beregner så bagud vha. af kædereglen
28              Vi har derfor differentieret funktion w*x+b til w
29              Derudover opdater vi også weights og biases ved igen at
30              bruge kædereglen, men denne gang diffentiere med hensyn
31              til w og x med partial differentiering.
32              b differentieres til 1 og w differentieres til x"""
33
34          # Gennem kædereglen gang forrige gradient med weights[transpose]
35          c_grad = dot(prev_grad, self.weights.T)
36
37          # Opdater vægtende
38          self.weights -= self.learning_rate * dot(x.T, prev_grad)
39          self.biases  -= self.learning_rate * prev_grad.mean(axis=0)*x.shape[0]
40
41          return c_grad
```

5.2 Design og implementering af sigmoid funktionen

Sigmoid funktionen implementeres, da den tilføjer en ikke-linearitet til det neurale netværk. Dette gøres tilsvarende til lagets klasse, altså ved at man har en `forward` og `backward` funktion.

Vi skal nu definere aktiveringsfunktion. Dette gøres tilsvarende til lagets klasse, altså ved at lave en klasse, og definere en forward og backward funktion.

Sigmoid funktionens formel ser således ud:

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

Idet vi bruger NumPy (lineær algebra bibliotek i python), kan vi indsætte matricer i funktioner. Underliggende sker der det, at NumPy indsætter værdierne fra matricen i funktionen, hvorefter de nye tal indsættes på de gamles tals pladser. Proceduren er:

$$\sigma\left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right) = \begin{bmatrix} \sigma(1) & \sigma(2) & \sigma(3) \\ \sigma(4) & \sigma(5) & \sigma(6) \\ \sigma(7) & \sigma(8) & \sigma(9) \end{bmatrix}$$

Vi kan nu kode det som følger.

```
43  class Sigmoid:
44
45      def __init__(self): pass
46
47      def forward(self, x):
48          """ Denne funktion står for at lave hvert feedforward kald
49              Dvs. den indsætter hver værdi fra x i funktion:
50              1 / (1 + exp(-xi)) """
51
52          return 1 / (1 + exp(-x))
```

Herefter skal vi differentiere sigmoid funktionen, sådan at vi også kan backpropegate (jf. afsnit 4.5) igennem netværket.[8]

$$\sigma(x) \coloneqq \frac{1}{1 + e^{-x}}$$

Begge sider differentieres:

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left(\frac{1}{1 + e^{-x}}\right)$$

Kædereglen benyttes nu til at differentiere, ved at opdele funktionen i to mindre funktioner. Dvs. at

$$f(x) = \frac{1}{x}; g(x) = 1 + e^{-x}; \sigma(x) = f\big(g(x)\big);$$

$$f'\big(g(x)\big) \cdot g'(x)$$

[8] Rabindra Lamsal (2021)

Nu differentieres hver funktion hver for sig vha. wordmats differentieringsværktøj

$$f'(x) = -\frac{1}{x^2}$$

$$g'(x) = -e^{-x}$$

Så ifølge kædereglen, er

$$\sigma'(x) = f'\big(g(x)\big) \cdot g'(x)$$
$$\sigma'(x) = -\frac{1}{(1+e^{-x})^2} \cdot (-e^{-x}) = \frac{(e^{-x})}{(1+e^{-x})^2}$$

Deler vi brøkerne op, kan man komme frem til at $\sigma'(x) = \sigma(x) \cdot \big(1 - \sigma(x)\big)$:

$$\sigma'(x) = \frac{(e^{-x})}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \cdot \frac{(e^{-x})}{1+e^{-x}}$$
$$= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x}) - 1}{1+e^{-x}}$$
$$= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right)$$

Vi kan nu indsætte vores oprindelige ikke-differentierede sigmoid funktion:

$$\sigma'(x) = \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) = \sigma(x) \cdot \big(1 - \sigma(x)\big)$$

Dette omsættes til følgende kode. Dog skal vi også gange med forrige gradient, ligesom vi også gjorde ved `lagets backward` funktion (5.1).

```
55      def backward(self, x, prev_grad):
56          """ Denne funktion beregnes hver gang der er backprop igennem netværket
57              Vi har så differentieret funktionen 1 / (1 + exp(-xi)) og får
58              y' = y(1-y) """
59
60          y = self.forward(x)
61          return y * (1-y) * prev_grad
```

5.3 Design og implementering af Cost function

For så at kunne optimere netværket, bliver vi nødt til at have en funktion, som kan beregne Cost-værdien - dvs. den værdi, der angiver i hvor høj grad netværket var korrekt i dens forudsigelse. Jeg har før beskrevet MSE Cost funktionen, men i det vi her udfører en klassifikationsopgave, ønsker vi at straffe forkerte forudsigelser hårdere end i regressions Cost-funktion[9]. Ideen er egentlig, at siden regression er på en kontinuær talfølge, skal tal, der er tæt på den sande værdi, ikke straffes ligeså hårdt som tal, der er langt fra

[9] Saumya Awasthi (2020)

den sande værdi, mens i klassifikationsproblemer er en forkert forudsigelse ligeså forkert, uanset om den gættede på et tal, der er tæt på eller langt fra det sande svar. Derfor kan vi bruge CrossEntropyCost-funktionen, som er defineret på følgende måde:

$$CrossEntropyCost(y,t) = -\sum_i t_i \cdot \ln(y_i)$$

idet den sande værdi er *t* og *y* er netværkets anslåede værdi. Vores mål er at beregne en sandsynlighedsvektor og vi skal derfor normalisere vores værdier. Dette gøres vha. Softmax funktionen eller følgende funktionsforskrift:

$$a(\vec{z})_i = \frac{e^{z_i}}{\sum_j e^j}$$

Der findes allerede veldokumenteret eksempel af CrossEntropyCost på internettet[10] Nedstående er et eksempel på sådan en.

```
3   def softmax_crossentropy_with_logits(logits,reference_answers):
4       # Compute crossentropy from logits[batch,n_classes] and ids of correct answers
5       logits_for_answers = logits[np.arange(len(logits)),reference_answers]
6
7       xentropy = - logits_for_answers + np.log(np.sum(np.exp(logits),axis=-1))
8
9       return xentropy
10
11  def grad_softmax_crossentropy_with_logits(logits,reference_answers):
12      # Compute crossentropy gradient from logits[batch,n_classes] and ids of correct answers
13      ones_for_answers = np.zeros_like(logits)
14      ones_for_answers[np.arange(len(logits)),reference_answers] = 1
15
16      softmax = np.exp(logits) / np.exp(logits).sum(axis=-1,keepdims=True)
17
18      return (- ones_for_answers + softmax) / logits.shape[0]
```

Anvendt i ifh. til mit formål, har jeg videreudviklet overstående funktioner. Ligesom i de andre klasser, får CrossEntropyCost klassen også en forward funktion, som blot beregner Cost værdien for den givne batch efter, at den har beregnet softmax værdien for vores logits, hvilket er det neurale netværks anslåede svar.

```
141     def forward(self, logits, targets):
142         # Omdan onehot til normalt tal
143         # [0,0,1, ... , 0] -> 2
144         targets = argmax(targets, axis=-1)
145
146         # Find alle korrekte værdier
147         a_correct = logits[arange(len(logits)), targets]
148
149         # Beregn cost værdien
150         return log(sum(exp(logits), axis=-1)) - a_correct
```

[10]Aayush Agrawal (2018)

`Backward funktionen` beregner den differentierede version af `forward funktionen`. Løsningen til denne differentialligning, er yderst kompleks, og vil ikke blive uddybet her.

```
152     def backward(self, logits, targets):
153         # Omdan onehot til normalt tal
154         # [0,0,1, ... , 0] -> 2
155         targets = argmax(targets, axis=-1)
156 
157         # Sæt korrekte værdier til 1
158         ones_for_answers = zeros_like(logits)
159         ones_for_answers[arange(len(logits)), targets] = 1
160 
161         # beregn Softmax
162         softmax = exp(logits) / exp(logits).sum(axis=-1, keepdims=True)
163 
164         return (softmax - ones_for_answers) / logits.shape[0]
```

Der er nedenfor i bilag 10.1 redegjort for hvordan datasættet hentes og præpareres. Derudover er der også redegjort i bilag 10.2 for sammensætningen af komponenterne til et neuralt netværk, som kan tillære sig viden omkring håndskrevne tal.

---

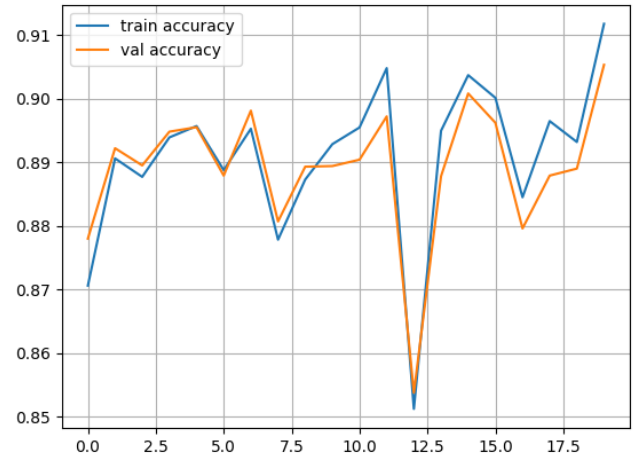
6. Analyse og forbedring af det neurale netværk

6.1 Performance

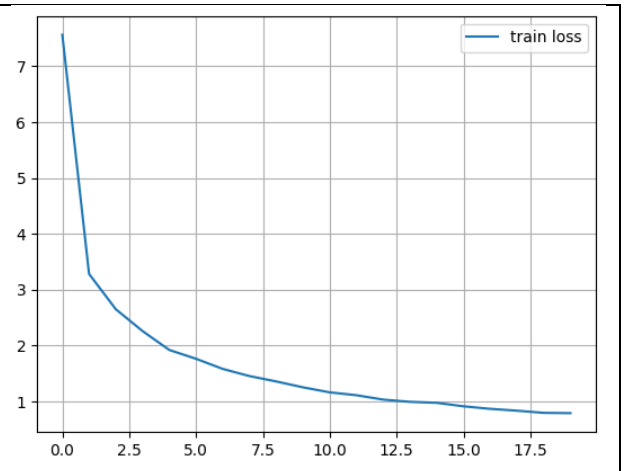
For at teste det neurale netværks nuværende performance, træner/øver man først netværket på et træningssæt, hvorefter man tester det neurale netværk på et ukendt valideringssæt for at vurdere den sande nøjagtighed i praktiske anvendelser. Ligesom i skolen, øver elever sig først på kendte eksamensopgaver, og til sidst tester man elevens viden til en eksamen. Pointen er, at man opdaterer det neurale netværks vægte og bias ved hjælp af træningssættet således at algoritmen bliver klogere, mens vægte og bias forbliver statiske under testen med valideringssættet.

I første omgang træner vi det neurale netværk i 20 iterationer (epochs), med ét hidden-lag med 512 neuroner og en learning rate på 0,05.

Her opnås en nøjagtighed på ca. 91% i både træningssættet og valideringssættet.



Figur 6.1: Graf over validerings og trænings nøjagtighed



Figur 6.2: Graf over udviklingen af Cost værdien

På *figur 6.1* og *figur 6.2* ses den udvikling modellen gennemgår i dens træningsforløb. Umiddelbart er der et forholdsvis stort fald (5%) i nøjagtigheden. Dette skyldes sandsynligvis de relativt få lag, da hver input kun bliver vægtet 2 gange.

For at optimere netværket, kan vi justere antallet af iterationer, størrelsen af det neurale netværk, og til sidst learning raten.

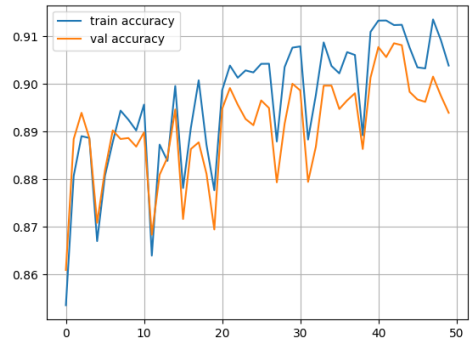
6.2 Optimering af netværket

De bedste neurale netværk, er i stand til at kunne genkende håndskrevne tal, med en succesrate på 99,5%[11], hvilket er markant meget mere end de 91% i det udførte forsøg. Det forekommer sandsynligt at en justering af de forskellige parametre vil maximere nøjagtigheden for det neural netværk i valideringssættet. Det undersøges ved at justere på antallet af iterationer, netværkets størrelse, aktiveringsfunktioner og learning raten.

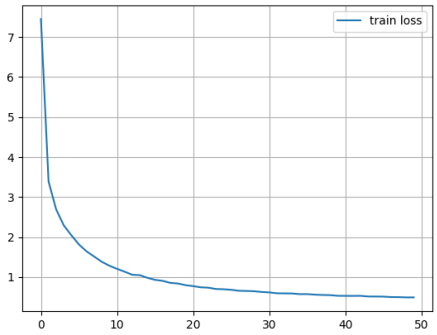
6.2.1 Iterationer

Grunden til det neurale netværks relativ dårlige nøjagtighed skyldes måske det neurale netværk mangler tid til at kunne lære alle billeder. Derfor ændres antal iterationer fra 20 til 50.

[11] https://tdody.github.io/MNIST/



Figur 6.3: Graf af det neural netværks nøjagtighed ved 50 epochs



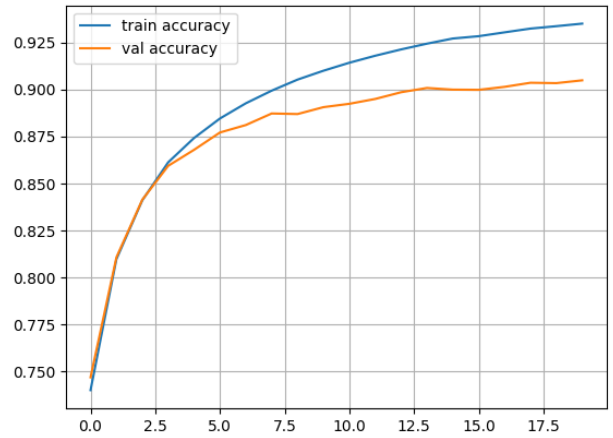
Figur 6.4: Graf og Cost funktion ved 50 epochs

Her fås igen en maximal nøjagtighedsprocent på 91% på valideringsættet, og en anelse højere trænings nøjagtighed.

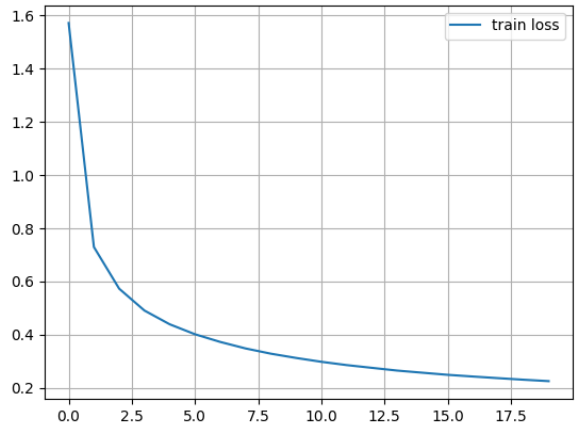
Programmets manglende indlæringsevne skyldes altså ikke, tiden det tager at træne netværket.

6.2.2 Netværkets størrelse

Vi har indtil videre kun haft et hidden-lag med i alt 512 neuroner, det er måske muligt at forbedre det neurale netværk ved at tilføje flere neuroner og lag, sådan at mere information kan blive lagret. Først tilføjes endnu et lag med 256 neuroner. Det neurale netværk har følgende format: $784 \rightarrow 512 \rightarrow 256 \rightarrow 10$.



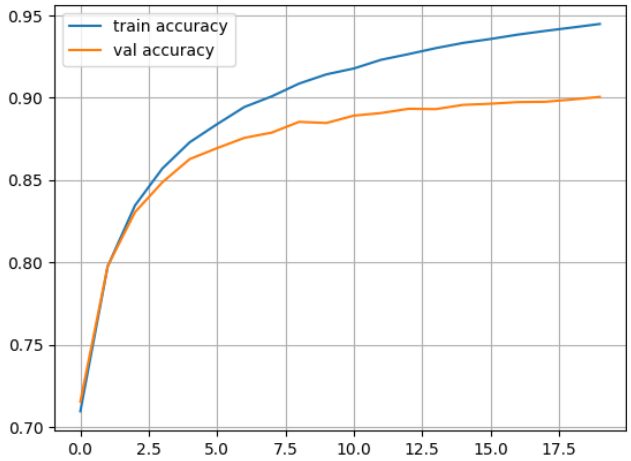
Figur 6.5: Graf af accuracies med yderligere 256 neuroner



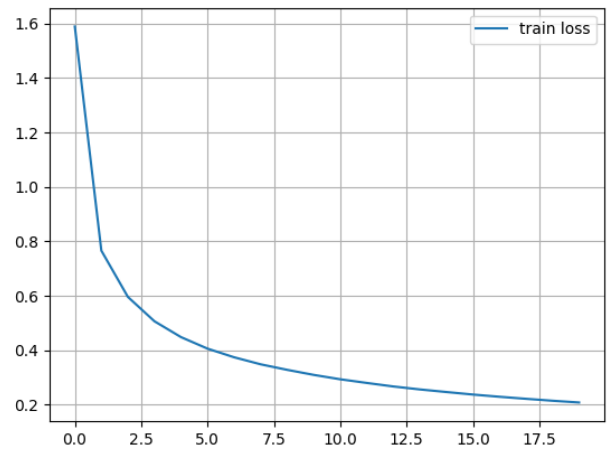
Figur 6.6: Graf og Cost funktion med yderligere 256 neuroner

Resultatet viste at træningssættes nøjagtighed blev forbedret med nogle flere procentpoint, hvilket gav anledning til at forøge netværkets størrelse yderligere. Vi kan nu prøve at øge lagene fra 4 til 5, med endnu et lag med 128 neuroner. Det neurale netværk har følgende format:

$784 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 10$



Figur 6.7: Graf af accuracies med yderligere 128 neuroner



Figur 6.8: Graf og Cost funktion med yderligere 128 neuroner

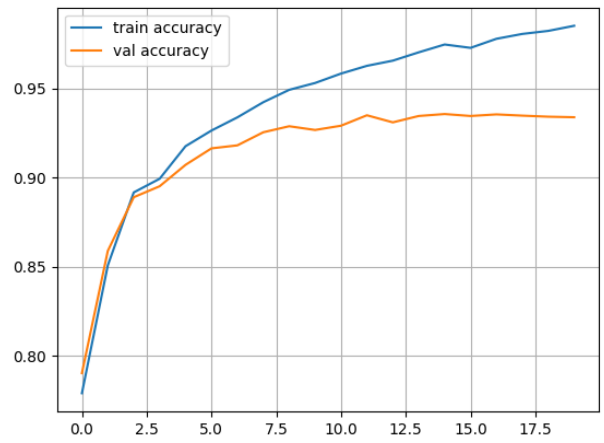
Her forbedres træningssættets præcision med lidt flere procentpoint. Desværre forbedres valideringssættets nøjagtighed ikke yderligere.

Der kan derudover også læses om hvordan jeg forbedre vores aktiveringsfunktioner under bilag 10.3

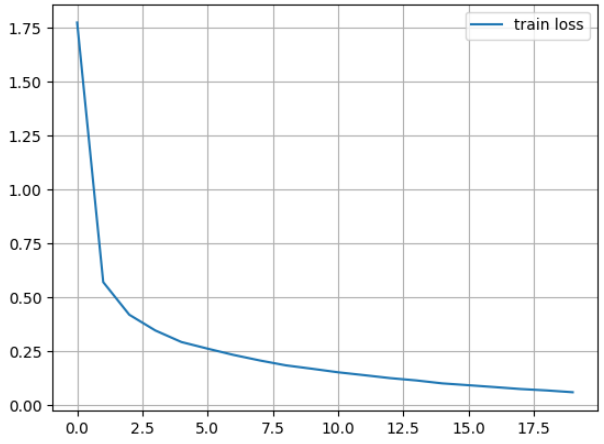
6.2.3 Learning Rate

Man kan så være fristet til at tro, det måske har noget at gøre med learning raten. En mulig grund, kan da også være at Gradient Descent bliver fanget i et småt lokalt minimum, hvilket gør at netværket ikke forbedres yderligere. Tager man længere skridt, kan man måske også komme længere ned (jf. Figur 4.1).

Learning raten sættes op til 0,2. Learning raten bliver altså firdoblet fra 0,05 til 0,2

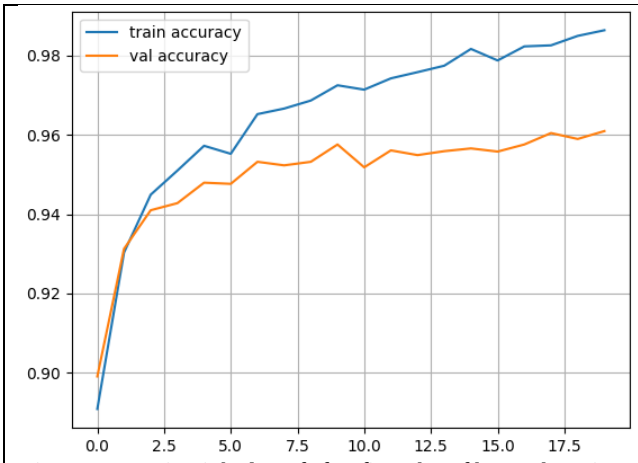


Figur 6.9: Nøjagtighedsgraf efter firdobling af learning rate

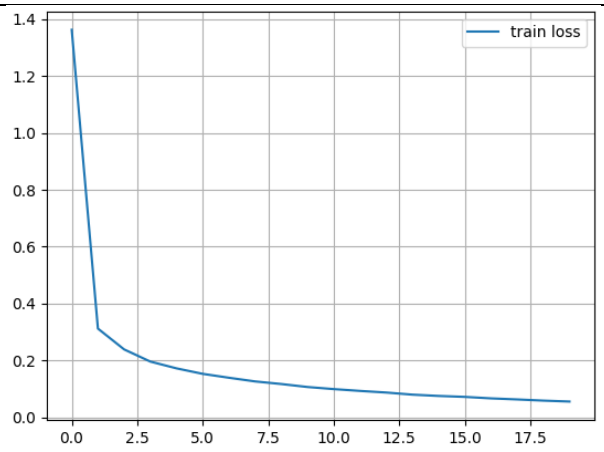


Figur 6.10: Costgraf efter firdobling af learning rate

Efter at have firdoblet den op til 0,2, ses hvordan valideringssættets nøjagtighed stiger fra 91% til 94%. Derudover bliver træningssættets nøjagtighed 98.5%. Pga. den anvendte computers kapacitet, er det ikke muligt at hæve learning raten yderligere (computeren meldte Overflow Errors). Dog kan man fjerne 512-laget og 256-laget og opnår højere learning rate. De to lag fjernes og learning raten sættes til 1.



Figur 6.11: Nøjagtighedsgraf efter fjernelse af lag og learning rate = 1



Figur 6.12: Costgraf efter fjernelse af lag og learning rate = 1

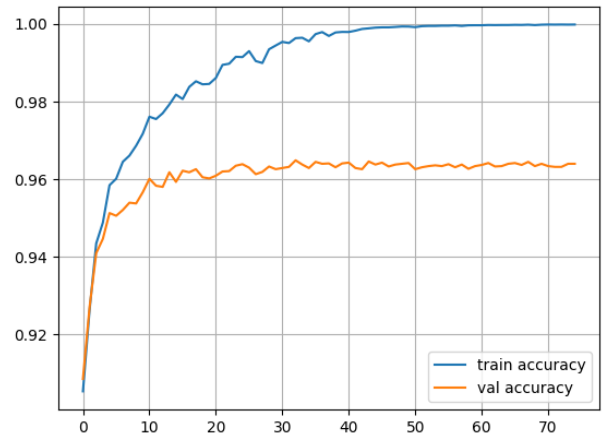
På trods af at man fjerner de 2 største lag, bliver valideringssættets akkurathed hævet til 96% og ca 98,5% for træningssættet.

6.2.4 Endelige ændringer

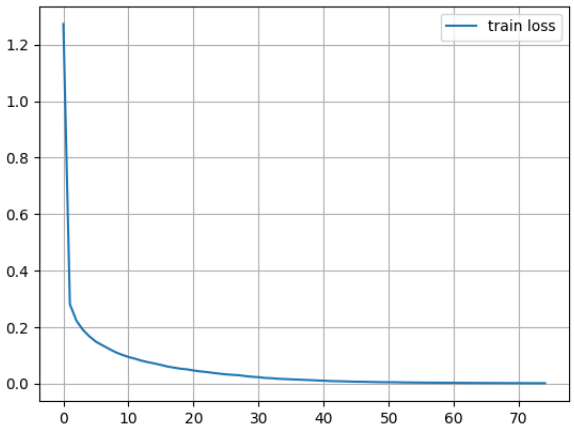
Ved brug af de forrige observationer, kan vi skrive parametrene for det endelige netværk, som skønnes at forbedre det neurale netværks nøjagtighed.

Iterationer: 75; Learning rate: 1,0; ét Hidden lag med 128 neuroner.

Jeg forhøjer iterationerne da det kunne ses på forrige test, at hældningen på begge akkurathedsgrafer ikke kommer tæt nok på 0, hvilket vil sige, at den muligvis har mere potentiale i sig.



Figur 6.13:
Nøjagtighedsgraf efter endelige ændringer



Figur 6.14:
Cost funktion efter endelige ændringer

Vi kan her se at vi ender med en træningsakkurathed på næsten 99,98% og en valideringsakkurathed på 96,48%. Her kan vi også se, at hældningen kommer meget tæt på 0 for begge akkurathedsgrafer, hvilket igen vil sige, at algoritmen ikke behøver mere tid at træne i.

7. Fremtidige anvendelser af Gradient Descent og andre optimeringsalgoritmer

7.1 Gradient Descent i nuværende netværk

På trods af at Gradient Descent blev opfundet i 1847[12] og Back Propagation algoritmen i 1962[13], er disse algoritmer samt mange flere mere specifikke varianter stadig i brug i dag til selv de største neurale netværk, som bruges så forskellige brancer som bankverdenen, medicinalindustrien og i landbruget. Eksempelvis i forbindelse med mammografi scanning, som giver hurtigere og mere korrekt fortolkning af billederne. [14]

7.2 Gradient Descents begrænsninger

På trods af den store udbredelse er Gradient Descent langt fra perfekt.

Først og fremmest har Gradient Descent den ulempe, at det kan være en og meget vanskeligt at optimere disse algoritmer, da det ofte kan tage flere dage at træne sådan et netværk, og da parametrene skal være helt i orden, bliver man nødt til at bruge store ressourcer i disse projekter. Eksempelvis blev et af historiens største neurale netværk GPT-3 trænet i 2020, og denne øvelse alene er estimeret til at have kostet omkring $12 mio. at træne.[15] Dette er derfor omkostningsfuldt at træne.

En anden tydelig begrænsning for Gradient Descent er desværre, at det neurale netværk skal have input og et facit. Den kan derfor ikke lære udelukkende af inputs. Dette medfører derfor at neurale netværk, som bliver optimeret af Gradient Descent, ikke har mulighed for at blive mere nøjagtig i forhold til menneskelig intelligens, da den er afhængig af et menneskeskabt datasæt. Det er derfor teoretisk umuligt at kunne få et neuralt netværk til at designe nye fusions reaktorer eller superledende molekyler, udelukkende ved hjælp af dens egen intellekt. På trods af denne begrænsning, har forskere fundet ud af at optimere neurale netværk med en blanding af Gradient Descent samt andre former for optimering såsom reinforcement learning.

7.3 Reinforcement learning

Reinforcement learning er en anden type af optimering for neurale netværk. Her bliver en robot sat i et miljø, hvor datasæt erstattes af faste belønningsmekanismer. Robotten formår derved at optimere sig selv. Det minder altså i langt højere grad om vores egen natur, idet hjernen udskiller dopamin og serotonin til at styre ønskelige handlinger. Eksempelvis udløses dopamin som belønning, når mennesket foretager handlinger, der er gavnlige for den overlevelse (nedlægger et bytte, indtager føde eller når vi reproducer os selv).

Idet parametrene er så løse for denne slags optimeringsalgoritmer, giver miljøet det neurale netværk nogle ganske særlige muligheder for at være kreativ. Den kan eksempelvis lære, hvordan man vinder i skak, uden at have fået nogen rigtig undervisning i spillet. Denne metode blev bl.a. brugt til udviklingen af AlphaGo, som var en banebrydende AI, som slog verdensmesteren i spillet Go, som antages at være et af de mest komplekse spil i verdenen.

7.4 AlphaGo / AlphaZero

Helt præcist blev AlphaGo bygget på nogle neurale netværk, som alle havde hver deres funktion. Eksempelvis havde de et ”policy-network” som stod for at vælge det næste move samt ”value-network” som forsøgte at forudsige den nuværende vinder. Den blev så først trænet mod amatør Go-spillere, for at få en basisforståelse for spillet. Herefter valgte udviklerne at træne netværket mod sig selv flere tusinde gange,

[12] https://en.wikipedia.org/wiki/Gradient_descent
[13] https://en.wikipedia.org/wiki/Backpropagation#History
[14] Herlev Hospital (2022)
[15] Kyle Wiggers (2020)

hvilket gjorde, at den lærte af sine egne fejl, og blev eksponentielt bedre. Alt dette kulminerede i, at den foretog træk som hidtil aldrig havde været set.[16]

Teamet bag AlphaGo har siden videreudviklet algoritmen, og er i dag i stand til at spille både Go, Shogi, Skak samt en række Atari spil, helt uden at få fortalt reglerne i spillet.

7.5 Kunstig super intelligens

Det kan være svært at spå om, hvordan, en kunstig superintelligens vil se ud, men der er i min optik tegn på at man skal lave en form for unik blanding mellem Gradient Descent og reinforcementlearning, idet de begge har fordele og ulemper. Gradient Descent har som nævnt den svaghed, at den ikke kan være mere kreativ end datasættet selv, men samtidig er den relativ hurtig hurtig at træne. Omvendt kan reinforcement learning introducere en unik form for kreativitet, som ikke engang den menneskelige hjerne er i stand til at komme frem til. Kan man derfor opstille et læringsmiljø for en robot, som gør den i stand til, at kunne lære kompliceret fysik og ingeniørvidenskab, vil man muligvis kunne få den til at simulere en virkende fusionsreaktor. Ulempen er her, at det er svært at opstille et korrekt miljø for en reinforcement learnings intelligens.

Derudover virker den her blanding også mere naturlig. Vi mennesker bliver født med en form for intelligens til at starte med, idet vi f.eks. kan trække vejret uden at få det fortalt. Men igennem livet tilpasser vi så vores neuroner, sådan at vi bedre kan forstå verdenen vi bliver født i.

8. Konklusion

Denne opgaves formål har været at undersøge, hvordan Gradient Descent algoritmen anvendes til genkendelse af håndskrevne tal. Undervejs fandt jeg frem til at Gradient Descent algoritmen bruges til at optimere et neuralt netværk, ved at justere på vægte og bias små skridt ad gangen. Det er ikke nok bare at programmere det neurale netværk, idet det er nødvendigt at justere på de forskellige parametre i træningsprocessen.
Jeg kom frem til, at en forhøjelse af learning raten samt antallet af iterationer var nok til at forbedre det neurale netværk så meget, at det opnåede en nøjagtighed på 96.5% i et valideringssæt. Andre aspekter såsom computerens kapacitet begrænser hvor stort et neuralt netværk, man kan lave.

Generelt set har Gradient Descent været en stor succes, idet den er med til at drive mange af de største neurale netværk vi bruger til dagligt. Dog har det den begrænsning, at et neuralt netværk optimeret af Gradient Descent aldrig vil kunne blive klogere end hvad datasættet tillader. Derfor kan Gradient Descent ikke stå alene i udviklingen af en kunstig super intelligens. Nogle, heriblandt jeg, tror dog at Gradient Descent kan være med til at give et fundament for en potentiel super intelligens.

[16] Google DeepMind (ukendt)

9. Kilder

Aayush Agrawal (2018), Building Neural Network from scratch, aayushmnit.github.io/
Hentet 08. Marts 2022
Fra https://aayushmnit.github.io/posts/2018/06/Building_neural_network_from_scratch/

Bill Shillito (2013), How to organize, add and multiply matrices - Bill Shillito, 4:40 min, YouTube.com
Hentet 07. Marts 2022
Fra https://www.youtube.com/watch?v=kqWCwwyeE6k

Błażej Osiński and Konrad Budek (2018), What is reinforcement learning? The complete guide, deepsense.ai
Hentet 16. Marts 2022
Fra https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/

Farhad Malik (2019), Neural Networks Bias and Weights, medium.com
Hentet 05. Marts 2022
Fra https://medium.com/fintechexplained/neural-networks-bias-and-weights-10b53e6285da

Google DeepMind (ukendt), AlphaGo - the story so far, deepmind.com
Hentet 16. Marts 2022
Fra https://deepmind.com/research/case-studies/alphago-the-story-so-far

Grant Sanderson (2017), But what is a neural network, 19:13 min, YouTube.com
Hentet 04. Marts 2022
Fra https://www.youtube.com/watch?v=aircAruvnKk

Grant Sanderson (2017), Gradient Descent, how neural networks learn, 21:00 min, YouTube.com
Hentet 04. Marts 2022
Fra https://www.youtube.com/watch?v=IHZwWFHWa-w

Grant Sanderson (2017), What is backpropagation really doing, 13:53 min, YouTube.com
Hentet 04. Marts 2022
Fra https://www.youtube.com/watch?v=Ilg3gGewQ5U

Grant Sanderson (2017), Backpropagation calculus, 10:17 min, YouTube.com
Hentet 04. Marts 2022
Fra https://www.youtube.com/watch?v=tIeHLnjs5U8

Herlev Hospital (2022), Lægerne får hjælp af kunstig intelligens, herlevhospital.dk
Hentet 14. Marts 2022
Fra https://www.herlevhospital.dk/presse-og-nyt/pressemeddelelser-og-nyheder/nyheder/Sider/L%C3%A6%C2%ADger%C2%ADne-f%C3%A5r-hj%C3%A6lp-af-kun%C2%ADstig-in%C2%ADtel%C2%ADli%C2%ADgens.aspx

IBM Cloud Education (2020), Neural Networks
Hentet 04. Marts 2022
Fra *https://www.ibm.com/cloud/learn/neural-networks*

Kim Scott (2016), artificial neural networks, MIT,
Hentet 04. marts 2022
Fra https://www.mit.edu/~kimscott/slides/ArtificialNeuralNetworks_LEAD2011.pdf

KhanAcademy (ukendt), Gradient Descent, khanacademy.com
Hentet 09. Marts 2022

Fra https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/optimizing-multivariable-functions/a/what-is-gradient-descent

Kyle Wiggers (2020), OpenAI launches an API to commercialize its research, venturebeat.com
Hentet 16. Marts 2022
Fra https://venturebeat.com/2020/06/11/openai-launches-an-api-to-commercialize-its-research/

Mehreen Saeed (2021), A Gentle Introduction To Partial Derivatives and Gradient Vectors, machinelearingmastery.com
Hentet 12. Marts 2022
Fra https://machinelearningmastery.com/a-gentle-introduction-to-partial-derivatives-and-gradient-vectors/

Michael Nielsen (2019), Using neural nets to recognize handwritten digits,
Hentet 04. Marts 2022
Fra http://neuralnetworksanddeeplearning.com/chap1.html

Michael Nielsen (2019), How the backpropagation algorithm works,
Hentet 05. Marts 2022
Fra http://neuralnetworksanddeeplearning.com/chap2.html

Rabindra Lamsal (2021), Derivative of Sigmoid Function, theneuralblog.com
Hentet 8. Marts 2022
Fra https://theneuralblog.com/derivative-sigmoid-function/

Stefania Cristina (2021), The Chain Rule of Calculus for Univariate and Multivariate Functions, machinelearingmastery.com
Hentet 12. Marts 2022
Fra https://machinelearningmastery.com/the-chain-rule-of-calculus-for-univariate-and-multivariate-functions/

Saumya Awasthi (2020), Understanding Cross Entropy Loss, datamahadev.com
Hentet 12. Marts 2022
Fra https://datamahadev.com/understanding-cross-entropy-loss/

The365team (2021), What is cross entropy loss, 365datascience.com
Hentet 12. Marts 2022
Fra https://365datascience.com/tutorials/machine-learning-tutorials/cross-entropy-loss/

Torsa Talukdar (2020), Why do we need activation functions in neural networks, Medium.com
Hentet 06. Marts 2022
Fra https://medium.com/@torsatalukdar11/why-do-we-need-activation-functions-in-neural-network-c72c340c78fa

WebMatematik (ukendt), Partielle afledede, Webmatematik.dk
Hentet 10. Marts 2022
Fra https://www.webmatematik.dk/lektioner/matematik-a/funktioner-af-to-variable/partielle-afledede

WebMatematik (ukendt), Gradienten af en funktion, Webmatematik.dk
Hentet 11. Marts 2022
https://www.webmatematik.dk/lektioner/matematik-a/funktioner-af-to-variable/gradient

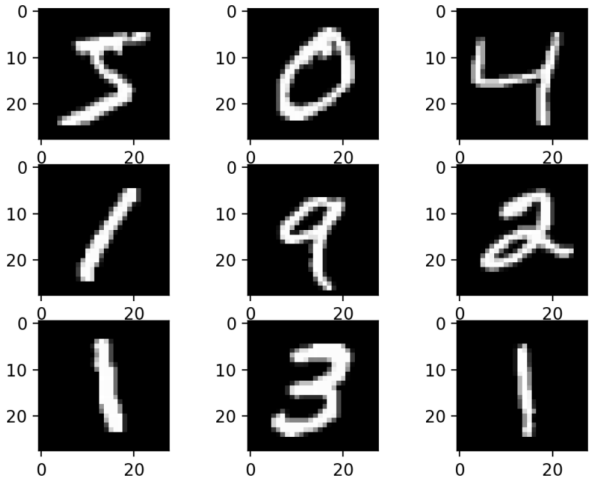
Wikipedia (ukendt), Gradient Descent, wikipedia.com
Hentet 16. Marts 2022
Fra https://en.wikipedia.org/wiki/Gradient_descent

Wikipedia (ukendt), Backpropagation, wikipedia.com
Hentet 16. Marts 2022
Fra https://en.wikipedia.org/wiki/Backpropagation#History

10.Bilag

10.1 Håndtering af data(sættet)

Idet vi ønsker at kategoriserer håndskrevne tal, kan bruge datasættet MNIST. MNIST datasættet består af mange tusinde håndskrevne tal, som er blevet pixeleret i størrelsen 28x28. Datasættet indeholder alle tal fra 0 til 9 og har allerede fået angivet facit, hvilket gør det let at implementere i en algoritme.



Figur 3.1: Eksempler på MNIST datasæt billeder
https://machinelearningmastery.com/wp-content/uploads/2019/02/Plot-of-a-Subset-of-Images-from-the-MNIST-Dataset-1024x768.png

Vi henter datasættet igennem Tensorflow Keras, som er et python bibliotek, ved blot at importere biblioteket i vores fil, samt kører en enkel kommando som henter dataen.

```
3   from tensorflow.keras.datasets import mnist
4   (X_train, y_train), (X_test, y_test) = mnist.load_data()
```

For at kunne bruge dette datasæt i vores egen algoritme, bliver vi nødt til at lave en form for preprocessing

Vi ønsker at have vores output som en one-hot array. En one hot array er en liste af nuller med undtagelse af et enkelt index som har værdien 1. Eksempelvis bliver 5 one hot enkodet ved at sætte det 5'te index i listen til 1. $onehot(5) = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$. Dette kan kodes vha. Numpys eye funktion.

```
16      # One hot encoding
17      y_train = eye(10)[y_train]
18      y_test  = eye(10)[y_test]
```

Derudover ønsker vi at normaliserer billedernes pixelværdi mellem 0 og 1. Derfor kan vi dele hver enkel pixel farve med 255, for at få værdien mellem 0 og 1.

```
20      # Sæt hver pixels værdi mellem 0 og 1
21      X_train = X_train.astype(float) / 255.
22      X_test = X_test.astype(float) / 255.
```

Og til sidste ønsker vi at gøre billederne "flade". At gøre billederne flade vil sige at man tager billede matricen som har formen 28 x 28 og gøre det til en matrix med formen 1 x 784. Vi gør dette sådan at vi kan

indsætte det direkte i det neural netværks første lag som har 784 neuroner. Vi bruger numpy funktionen reshape, som kan omforme en matrix til en given dimension

```
29      # Flatten billederne (28x28 -> 784)
30      X_train = X_train.reshape([X_train.shape[0], -1])
31      X_val = X_val.reshape([X_val.shape[0], -1])
32      X_test = X_test.reshape([X_test.shape[0], -1])
```

Vi gemmer nu disse billeder til senere brug, sådan at vi kan træne det neurale netværk.

10.2 Hvordan vi sammensætter det

For så at træne netværket opsætter vi en trænings klasse som vil komme til at indeholde funktionerne: *feedforward*, *backprop*, *train_step* og *predict*.

Først skal vi dog initialiserer klassen. Når vi initialiserer klassen, definerer vi samtidig hele det neurale netværk samt en Cost funktion.

```
11  def __init__(self, input_nodes:int, hidden_nodes:List[int], output_nodes:int, lr:float):
12      # Definer det neurale netværk
13      input_layer  = Linear(input_nodes, hidden_nodes[0], lr)
14      hidden_layers = [Relu()]
15      for i in range(len(hidden_nodes)-1):
16          hidden_layers.append(Linear(hidden_nodes[i], hidden_nodes[i+1], lr))
17          hidden_layers.append(Sigmoid())
18      hidden_layers[-1] = Relu()
19      output_layer = Linear(hidden_nodes[-1], output_nodes, lr)
20
21      # Sammensæt alle lagene i en samlet liste
22      self.network = [input_layer] + hidden_layers + [output_layer]
23
24      # Initialiser cost funktionen
25      self.cost_fn = Cost()
```

Feedforward funktionen indsætter billederne i det neural netværk og giver et output, uanset om det er korrekt eller ej. Funktionen returnere herefter et memory som er en liste af lags output. Det er altså i denne funktion hvor vi bruger lagene og aktiveringsfunktionerns forward funktion.

```
27  def feedforward(self, inputs) -> List[array]:
28      """ Indsæt et batch med mange billeder, som herefter bere-
29          gnes. Hvert neuron lags outputs gemmes i en liste som
30          bruges når der skal optimeres. Derfor returer `memory` """
31
32      # Definer en variabel til hukommelse af hvert lags output
33      memory = [inputs]
34
35      # Indsæt forrige lags output i næste lag, indtil man når enden
36      for layer in self.network:
37          inputs = layer.forward(inputs)
38          memory.append(inputs)
39
40      return memory
```

Backprop funktionen, er så funktion hvor vi bruger backward funktion for hvert lag og aktiveringsfunktion. Helt præcist tilføjer vi gradienterne til hvert lag samtidig med at vi opdaterer vægtene og bias. Funktionen returnerer til sidst en gennemsnitlig Cost værdi for alle indsatte billeder

```
42      def backprop(self, logits, targets, memory):
43          """ Beregn bagud igennem netværket sådan at den sidste gradients
44              Vi starter med at beregne gradienten for cost funktionen
45              hvorefter vi bevæger os bagud. Herefter beregnes cost værdien
46              og returneres"""
47
48          # Beregn den første gradient
49          prev_grad = self.cost_fn.grad(logits, targets)
50
51          # Beregn bagud igennem netværket
52          for layer_idx in range(len(self.network))[::-1]:
53              layer = self.network[layer_idx]
54              prev_grad = layer.backward(
55                  memory[layer_idx],
56                  prev_grad
57              )
58
59          # Beregn cost og returner
60          cost = self.cost_fn.cost(logits, targets)
61          return cost
```

I funktion train_step, bruger vi de to forrige funktioner til at træne det neurale netværk på hele datasættet. Samt tester vi netværkets performance på billeder den ikke er blevet trænet i at genkende.

```
63      def train_step(self, inputs, targets, val_inputs, val_targets, batch_size):
64          """ Optimer 1 gang netværket på de angivne inputs og targets
65              Derudover test også netværket på val_inputs og val_targets """
66
67          # Træn netværket på alle batches
68          costs = []
69          for x_batch, y_batch in get_batch(inputs, targets, batch_size):
70              neuralnet_memories = self.feedforward(x_batch)
71
72              # Beregn cost og optimer vægte og bias
73              logits = neuralnet_memories[-1]
74
75              cost = self.backprop(logits, y_batch, neuralnet_memories)
76              costs.append(cost)
77
78          # Test det neural netværks performance
79          train_predictions = self.predict(inputs)
80          train_acc = mean(train_predictions == targets.argmax(-1))
81
82          val_predictions = self.predict(val_inputs)
83          val_acc = mean(val_predictions == val_targets.argmax(-1))
84
85          return mean(costs), train_acc, val_acc
```

Til sidst definerer vi så en predict funktion, som har til opgave blot at fortælle hvad det neurale netværk tror billedet er. Den returnerer altså et tal, den finder igennem vektor med sandsynligheder for at den givne index er det endelige resultat.

```
87      def predict(self, inputs):
88          """ Kommer med et gæt på hvilket tal billedet er ved at
89              indsætte billedet i det neuralnetværk hvorefter vi
90              tager indexet for den neuron der scorer højst """
91
92          # Input billederne i feedforward netværket
93          logits = self.feedforward(inputs)[-1]
94
95          # Find herefter den højst scorende index og returner
96          return logits.argmax(axis=-1)
97
```

10.3 Forbedring af Aktiveringsfunktioner

Hvert

Lige nu har vi en struktur, hvor hvert gemt lag, altid ender med at blive aktiveret af en sigmoid funktion. Måske det er muligt at forbedre valideringssættets akkurathed ved at tilføje en binær aktiveringsfunktion.

Sådan en fungerer ved at, hvis en værdi er 0 eller under, bliver resultatet igen 0, og ellers bliver resultatet x

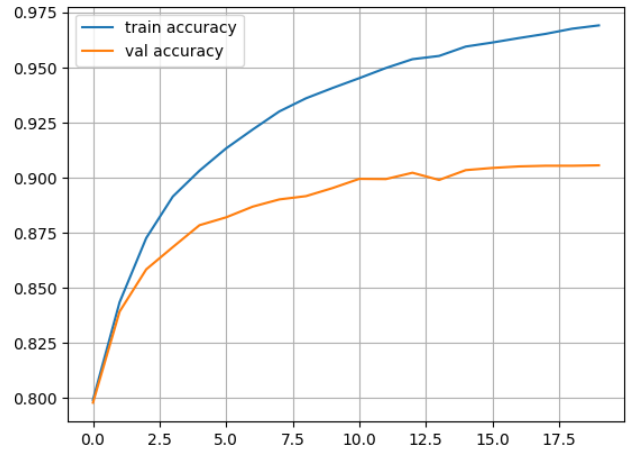
$$relu(x) = \begin{Bmatrix} 0 & x \leq 0 \\ x & 1 \leq x \end{Bmatrix}$$

Denne funktion har den fordel, at vi helt ignorerer negative signaler. Vi kan så lave følgende klasse:

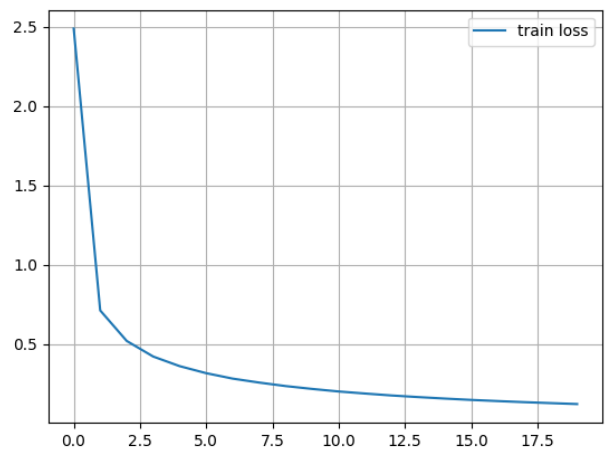
```
64  class Relu:
65
66      def __init__(self): pass
67
68      def forward(self, x):
69          """ I denne funktion ignorer man alle negative signaler.
70              Hvis tallet er under 0, returnes bare 0, eller
71              returnes tallet selv.
72               """
73
74          return maximum(0,x)
75
76      def backward(self, x, prev_grad):
77          """ Differentieres funktionen ovenfor bliver værdien
78              enten 0 eller 1, idet en konstant differentieret
79              er 1
80              """
81
82          relu_grad = x > 0
83          return relu_grad * prev_grad
```

Når man ignorer de negative signaler, giver det den fordel, at negative signaler ikke trækker ned i den vægtede sum, hvilket kan give det neurale netværk bedre mulighed for at lære, idet neuroner påvirker hinanden mindre.

Vi kan så træne netværket med de samme parametre som da vi optimerede på netværkets størrelse. Altså ser vores lag således: $784 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 10$



Figur 6.9: Akkurarethedsgraf efter tilføjelse af ReLu aktiveringsfunktion



Figur 6.10: Costgraf efter tilføjelse af ReLu aktiveringsfunktion

Det aflæses på grafen hvordan træningssættets akkurathed igen forbedres, og kan nu genkende ca. 97% træningssættet. Men desværre optimeres valideringssættets akkurathed ikke.