H.C. ØRSTED GYMNASIET EN DEL AF TEC

SOP opgaveformulering. 2024-2025

Navn	Victor Østergaard Nielsen
Klasse	L3di

Fag	Niveau	Vejleder navn	Vejleders mailadresse
Matematik	A	Jan Strauss Hansen	Jash@tec.dk
Programmering	B	Kristian Krabbe Møller	kkm@tec.dk

Machine-learning

Hovedspørgsmål: Hvordan kan et simpelt neuralt netværk, uden unødvendige abstraktioner eller biblioteker, anvendes til genkendelse af håndskrevne tal i realtid i et tegneprogram på computeren, og hvad er matematikken bag?

Opgaveformulering:

Redegør overordnet for begrebet neuralt netværk.

Redegør for de grundlæggende matematiske principper bag neurale netværk herunder matrixregning.

Redegør for valg af programmeringssprog ift. udvikling af programmer med neuralt netværk.

Analysér hvordan et neutralt netværk kan programmeres, gerne uden unødvendige abstraktioner eller biblioteker, så det kan genkende håndskrevne tal i realtid i/fra et tegneprogram på computeren.

Undersøg hvorledes programmet kan optimeres for at opnå lavest mulig fejlrate og evt. hvorledes støj i den analyserede data kan påvirke fejlraten.

Diskuter og vurder hvorvidt neutrale netværk er den mest effektive tilgang til at genkende håndskrevne tal på en adaptiv og robust måde.

Udleverede bilag	
SOP eksamensopgaven udleveres	**29. november. 2024 kl. 14.30**
SOP eksamensopgaven skal afleveres på Netprøver	**13. december. 2024 kl. 14.30**

Studie Område Projekt

Genkendelse af håndskrevne tal med neurale netværk

Matematik A / Programmering B

Victor Østergaard Nielsen
3di - H. C. Ørsted Gymnasiet Lyngby
15/12/2024

Vejledere:
Jan Strauss Hansen (Matematik A)
Kristian Krabbe Møller (Programmering B)

Figur 1: Eksempel fra MNIST-datasættet (Delclos, 2021; LeCun m.fl., 1994)

Resumé

Denne opgave er bestående af 15,5 normalsider og 4 sider med symboltegn, total: 19,5 sider.

Denne opgave undersøger, hvordan et simpelt neuralt netværk kan implementeres uden unødvendige abstraktioner eller biblioteker for at genkende håndskrevne tal i realtid på en digital tegneflade. Formålet er at afdække de matematiske og teknologiske principper bag neurale netværk, herunder matrixregning og gradient descent algoritmen, og at vurdere deres effektivitet som løsning til denne specifikke udfordring. Analysen viser, at neurale netværk kan lære at genkende mønstre ved hjælp af træningsdata og optimeringsalgoritmer. Opgaven demonstrerer, hvordan et netværk med input-, skjulte- og outputlag kan programmeres i Python, og hvordan der kan bruges aktiveringsfunktioner som sigmoid og softmax til at beskrive ikke-linearitet samt repensentere en sandsynlighedsfordeling. Konklusionen peger på, at neurale netværk, selv i en simpel udgave, er en effektiv og nogenlunde fleksibel løsning til genkendelsen af håndskrevne tal i realtid. Samtidig erkendes det, at mere avancerede netværk som convolutional neural networks (CNNs) kan tilbyde bedre præcision, men også kræver større beregningskraft og kompleksitet.

Indhold

1 Indledning

I takt med, at kunstig intelligens får en større rolle i vores hverdag, rejser det mange fundamentale spørgsmål, heriblandt: Hvordan man kan optimere maskiners evne til at forstå og tolke menneskelige input? Denne opgave fokuserer på udviklingen af et simpelt neuralt netværk, der kan genkende håndskrevne tal i realtid gennem en digital tegneflade. Problemstillingen er spændende, fordi den kombinerer matematik og programmering for at simulere hjernens måde at lære og tilpasse sig på. Det er et emne, der både har stor relevans inden for uddannelse og teknologi. I en tid, hvor automatisering og dataforståelse er afgørende for innovation, er det vigtigt at forstå, hvordan neurale netværk kan bruges til at efterligne menneskets evne til at genkende mønstre. Specifikt undersøger denne opgave, hvordan et neuralt netværk kan implementeres uden brug af avancerede biblioteker eller unødvendige abstraktioner, og hvordan matematiske principper som matrixregning og gradient descent spiller en central rolle i træningen og optimeringen af netværket. Problemstillingen knytter sig direkte til praksis ved at tage udgangspunkt i en konkret anvendelse, nemlig genkendelse af håndskrevne tal, som kan bruges i applikationer såsom OCR-systemer og andre klassifikationsopgaver. Opgaven fokuserer dog bevidst på en simplificeret model for at holde analysen fokuseret og anvendelig. Mere avancerede teknologier såsom convolutional neural networks (CNNs) inddrages ikke, da målet er at demonstrere forståelsen af grundprincipperne bag neurale netværk og deres implementering. Læseren kan således forvente en praktisk og matematisk tilgang, der gør det muligt at forstå sammenhængen mellem teori og praksis i maskinlæring.

2 Matricer og matrixregning

2.1 Matricer

En matrice er en tabel af tal, der er arrangeret i rækker og kolonner. En matrice kan repræsenteres med et stort bogstav, f.eks. A, og elementerne i matricen kan repræsenteres som a_{ij}, hvor i er rækken og j er kolonnen. En matrice med m rækker og n kolonner kaldes en $m \times n$ matrice. En matrice med lige mange rækker og kolonner kaldes en kvadratisk matrice. En matrice med kun én række kaldes en rækkevektor, og en matrice med kun én kolonne kaldes en søjlevektor. (Lauritzen & Bökstedt, 2019)

$$\underbrace{A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}}_{m \times n \text{ Matrice}} \qquad \underbrace{B = \begin{bmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{m1} \end{bmatrix}}_{\text{Søjlevektor}} \qquad \underbrace{C = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} \end{bmatrix}}_{\text{Rækkevektor}} \tag{1}$$

2.2 Matrixregning

Matrixregning er en vigtig del af matematikken bag neurale netværk og er derfor vigtig at forstå. Der er flere forskellige operationer, der kan udføres på matricer, herunder addition, subtraktion, skalarmultiplikation og matrixmultiplikation m.m.

2.2.1 Addition og subtraktion

For at addere eller subtrahere to matricer skal de have samme dimensioner, altså de skal have samme mængde rækker og søjler. Givet dette, så er matmatkiken ikke meget andlernedes fra normale tal, det betyder at: $A + B = B + A$. Da de 2 matricer har samme dimension udføres addition og subtraktion således (Lauritzen & Bökstedt, 2019):

$$A + B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix} \tag{2}$$

Hver position i A matricen bliver altså adderet med samme position i B matricen. Samme regel gælder for subtraktion.

2.2.2 Skalarmultiplikation af matricer

Givet tallet k kan man gange k på matricen A således (Lauritzen & Bökstedt, 2019):

$$k \cdot A = k \cdot \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} k \cdot a_{11} & k \cdot a_{12} \\ k \cdot a_{21} & k \cdot a_{22} \end{bmatrix} \tag{3}$$

2.2.3 Elementvis anvendelse af en funktion på matricer

Givet en funktion $f(x)$ og en $m \times n$ matrice A, vil $f(A)$ være en $m \times n$ matrice, hvor $f(x)$ er blevet anvendt på hvert element i A (Lauritzen & Bökstedt, 2019):

$$f(A) = f\left(\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}\right) = \begin{bmatrix} f(a_{11}) & f(a_{12}) & \dots & f(a_{1n}) \\ f(a_{21}) & f(a_{22}) & \dots & f(a_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ f(a_{m1}) & f(a_{m2}) & \dots & f(a_{mn}) \end{bmatrix} \tag{4}$$

2.2.4 Matrixmultiplikation

At gange to matricer sammen er lidt mere kompliceret, det indebærer først og fremmest at de 2 matricer er af kompatibel størrelse. Hvis A er en $m \times p$ matrice og B er en $p \times r$ matrice, så er $C = A \cdot B$ en $m \times r$ matrice. Bemærk at antallet af kolonner i A matricen skal være lig antallet af rækker i B matricen. For at finde elementet c_{ij} i C matricen ganges række i i A matricen med kolonne j i B matricen. Dette gøres ved at gange elementerne i

række i i A matricen med elementerne i kolonne j i B matricen og summere dem. F.eks. hvis A er en 3×2 matrice og B er en 2×3 matrice, så er C en 3×3 matrice, og elementet c_{11} i C matricen findes således (Simonson, 2015):

$$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} \tag{5}$$

Intuitivt kan dette visualiseres ved at tegne A og B matricerne således (Simonson, 2015):

$$\begin{bmatrix} & & \\ a_{11} & a_{12} & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} & & \\ & & \end{bmatrix} \tag{6}$$

For at finde elementet c_{11} i C matricen, ganges række 1 i A matricen med kolonne 1 i B matricen, dette er visualiseret herunder (Simonson, 2015):

$$\begin{bmatrix} & & \\ a_{11} & a_{12} & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} c_{11} & & \\ & & \end{bmatrix} \qquad c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} \tag{7}$$

Samme operation gentages for resten af positionerne i C matricen:

$$\begin{bmatrix} & & \\ a_{11} & a_{12} & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \tag{8}$$

Det ses nu visuelt at den resulterende matrice C er en 3×3 matrice når A er en 3×2 matrice og B er en 2×3 matrice. Det skal dog bemærkes at matrixmultiplikation ikke er kommutativ, altså $A \cdot B \neq B \cdot A$. Dette kan også ses visuelt ved at bytte om på A og B matricerne:

$$\begin{bmatrix} & & \\ a_{11} & a_{12} & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \qquad \begin{bmatrix} & & \\ a_{11} & a_{12} & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \tag{9}$$

Det ses at $A \cdot B$ og $B \cdot A$ ikke er ens, og derfor er matrixmultiplikation ikke kommutativ. (Lauritzen & Bökstedt, 2019) Selv med to kvadratiske matricer af samme dimension er matrixmultiplikation ikke nødvendigvis kommutativ. Dette kan betragtes i følgende eksempel:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \tag{10}$$

$$A \cdot B = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix} \qquad B \cdot A = \begin{bmatrix} 5 \cdot 1 + 6 \cdot 3 & 5 \cdot 2 + 6 \cdot 4 \\ 7 \cdot 1 + 8 \cdot 3 & 7 \cdot 2 + 8 \cdot 4 \end{bmatrix} = \begin{bmatrix} 23 & 34 \\ 31 & 46 \end{bmatrix} \tag{11}$$

Det ses at $A \cdot B \neq B \cdot A$, matrixmultiplikation er altså ikke kommutativ, hverken i den resulterende størrelse i ikke kvadratiske matricer eller i kvadratiske matricer af samme størrelse. (Lauritzen & Bökstedt, 2019)

2.2.5 Transponering af matricer

Transponering af en matrice betyder at bytte om på rækker og kolonner. Hvis A er en $m \times n$ matrice, så er transponeringen af A en $n \times m$ matrice, og elementet a_{ij} i A matricen bliver til elementet a_{ji} i A^T matricen. Dette kan visualiseres ved at tegne A matricen og A^T matricen (Lauritzen & Bökstedt, 2019):

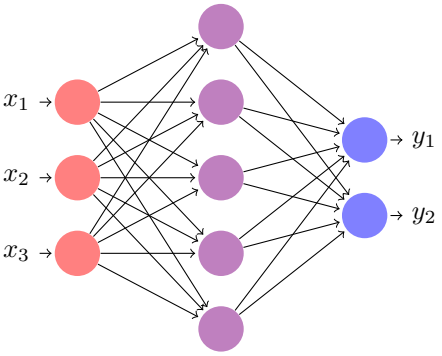
$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \qquad A^T = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \tag{12}$$

3 Neurale netværk

3.1 Introduktion

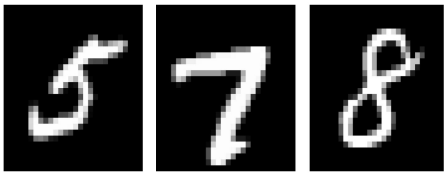
Et neuralt netværk er en matematisk model, der er inspireret af de biologiske neuroner i menneskehjernen. Et neuralt netværk består af en række lag, hvor hvert lag består af neuroner. Dette kan ses på (Figur 3). Hvert neuron i et lag er forbundet til alle neuroner i det forrige lag og det næste lag. Hver forbindelse mellem neuronerne har en vægt, der bestemmer, hvor meget signalet fra det ene neuron påvirker det næste neuron. Hvert neuron har også en bias, der bestemmer, hvor let det er for neuronet at sende et signal. Et neuralt netværk består af et inputlag, et eller flere skjulte lagog et outputlag. Dette er illustreret på (Figur 3). Denne model er altså inspireret af de mange sammenkoblede neuroner i menneskehjernen. Denne lighed er ikke tilfældig, da neurale netværk er designet til at efterligne hjernens evne til at lære. (Sanderson, 2017a)

Figur 2: Neuroner i menneskehjernen fra (St. Clair, 2021)



Figur 3: Et simpelt neuralt netværk

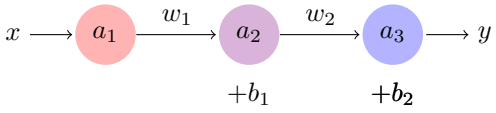
Bemærk (Figur 4). Du ved udmærket godt, hvilke tal der er på billedet, selvom du aldrig før har set netop dette 5-, 7- og 8-tal. Dette skyldes, at din hjerne er trænet til at genkende tal. Denne opgave er enormt udfordrende for et alment computerprogram, da programmet ikke har nogen intuitiv forståelse af, hvad et tal er. Den skal derfor programmeres med specifikke instruktioner for at kunne genkende tal. Denne tilgang er ikke optimal, da den kræver, at programmøren har en dyb forståelse af, hvordan tal ser ud, og hvordan de kan genkendes. Metoden er heller ikke holdbar i længden, eftersom der er uendeligt mange måder at skrive et 7-tal på. For at kunne generalisere talgenkendelse og gøre metoden mere robust overfor nye skrivemåder af tal kan neurale netværk anvendes. Antag et neuralt netværk med et tal som input og de helt rigtige vægte og biases. Denne model vil i teorien kunne genkende tal, selvom den aldrig før har set netop dette tal. Håbet er, at modellen har "lært"at generalisere træningsdataen til en mere overordnet forståelse af tal. Modellens output vil derfor være en søjlevektor med sandsynligheder for, at inputtet er et tal fra 0 til 9. Modellens prediktering vil være det tal, der har den højeste sandsynlighed ifølge modellen. (Sanderson, 2017a)



Figur 4: Eksempel fra MNIST datasættet (LeCun m.fl., 1994)

3.1.1 Feedforward

Når modellen skal prediktere, altså gå fra input til output, kaldes dette for *feedforward*. Her sendes inputtet gennem alle lagene i modellen og bliver påvirket af vægtene mellem neuronerne samt biaset i hvert neuron. Herunder ses et simpelt neuralt netværk med kun 1 neuron i hvert lag og et skjult lag, hvor a_n er aktiveringen af neuronen i lag n, og w_n er vægten mellem neuronen i lag n og lag $n+1$ (Sanderson, 2017a):



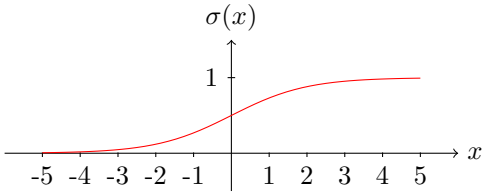
Figur 5: Et simpelt neuralt netværk

Så, hvis vi ønsker at prediktere outputtet y givet inputtet x, så kan vi gøre dette ved at følge disse trin:

$$a_1 = x \tag{13}$$
$$a_2 = \sigma(w_1 \cdot a_1 + b_1) \tag{14}$$
$$a_3 = \sigma(w_2 \cdot a_2 + b_2) \tag{15}$$
$$y = a_3 \tag{16}$$

Her er $\sigma(x)$ en aktiveringsfunktion, der tager inputtet x og returnerer et output. Denne funktion er essentiel for, at modellen kan lære mere komplekse fænomener, da den introducerer ikke-linearitet i modellen. En af de mest anvendte aktiveringsfunktioner er ReLU, der tager inputtet x og returnerer x, hvis $x > 0$, og 0 ellers (Sanderson, 2017b). Hvis outputtet skal betragtes som en sandsynlighed, er sigmoid-funktionen en god aktiveringsfunktion, da den tager inputtet x og returnerer en værdi mellem 0 og 1, som kan tolkes som en sandsynlighed. Sigmoid-funktionens definition samt et plot af funktionen er vist herunder (Nielsen, 2019b).

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{17}$$



Figur 6: Sigmoid funktionen i intervallet $x \in [-5, 5]$

Typisk har et neuralt netværk flere neuroner i hvert lag, og antallet af vægte og biases er derfor meget større. x, y, samt alle de forskellige a_n og b_n for hvert lag er søjlevektorer, og w_n i alle lag er matricer. Derfor skal der bruges matrixmultiplikation for at kunne beregne outputtet. Dette er ikke et problem, da de pågældende regneoperationer er defineret tidligere.

3.1.2 Træning af neurale netværk med Cross-Entropy

Når et neuralt netværk initialiseres, er alle vægtene og biases tilfældige indenfor et interval. Dette betyder, at modellen ikke kan genkende noget, ligesom et barn, der skal lære noget for første gang. For at træne modellen til at genkende mønstre kræves et passende datasæt med labels, der angiver, hvad hvert input repræsenterer. For at kunne forbedre modellen skal vi måle, hvor god den er til at lave forudsigelser. I stedet for blot at måle antallet af korrekte gæt anvender man en såkaldt *loss-funktion*, som kvantificerer, hvor præcist modellen forudsiger. For klassifikationsproblemer bruger man ofte *cross-entropy loss*, som evaluerer forskellen mellem modellens sandsynlighedsfordeling og de faktiske labels. Loss-funktionen tager modellens sandsynligheder for hver klasse og beregner, hvor langt de er fra de korrekte labels. (Nielsen, 2019a; Sanderson, 2017a, 2017b) Eksempelvis, hvis et datapunkt har en korrekt label i klasse 3, og modellen giver sandsynlighederne, her er y_i modellens sandsynligheder og $\hat{y}_i$ den korrekte label:

$$\mathbf{y}_i = \begin{bmatrix} 0.12 \\ 0.03 \\ 0.25 \\ 0.07 \\ 0.18 \\ 0.09 \\ 0.04 \\ 0.11 \\ 0.06 \\ 0.05 \end{bmatrix} \qquad \hat{\mathbf{y}}_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{18}$$

Cross-entropy loss beregnes ved formlen:

$$L_i = -\sum_{j=1}^{10} \hat{y}_{ij} \log(y_{ij}) \tag{19}$$

hvor $\hat{y}_{ij}$ er den korrekte label, og y_{ij} er modellens forudsigelse for klasse j. Dette sikrer, at modellen straffes hårdt for lave sandsynligheder på den korrekte klasse. Summen af loss over hele datasættet divideres med antallet af datapunkter for at få gennemsnitlig loss:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{20}$$

Cross-entropy loss er kontinuert og differentierbar, hvilket gør det muligt at optimere ved hjælp af gradient descent. Målet er at minimere loss, så modellen bliver bedre til at forudsige. Dette gør cross-entropy til en standardmetode for klassifikationsopgaver, hvor sandsynlighedsfordelinger er i spil. (Kumar, 2024; Verma, 2020)

3.2 Softmax

Softmax er en aktiveringsfunktion, der bruges i det sidste lag af et neuralt netværk, når outputtet skal repræsentere en sandsynlighedsfordeling over forskellige klasser. Softmax-funktionen tager en vektor af vilkårlige reelle tal og omdanner dem til en vektor af sandsynligheder, hvor summen af sandsynlighederne altid er lig 1. Softmax-funktionen er defineret som:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{21}$$

hvor z_i er elementet i inputvektoren, og K er antallet af klasser. (Kurbiel, 2021) Softmax-funktionen anvendes ofte i klassifikationsproblemer, hvor outputtet skal repræsentere sandsynligheden for hver klasse. Herunder ses et eksempel på softmax-funktionen anvendt på et enkelt datapunkt i:

$$\text{softmax}\left(\underbrace{\begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix}}_{\text{Sum ikke lig 1}} \right) = \begin{bmatrix} \frac{e^{2.0}}{e^{2.0}+e^{1.0}+e^{0.1}} \\ \frac{e^{1.0}}{e^{2.0}+e^{1.0}+e^{0.1}} \\ \frac{e^{0.1}}{e^{2.0}+e^{1.0}+e^{0.1}} \end{bmatrix} = \underbrace{\begin{bmatrix} 0.659 \\ 0.242 \\ 0.099 \end{bmatrix}}_{\text{Sum lig 1}} \tag{22}$$

I dette eksempel er sandsynligheden for klasse 1 (første element) 65.9%, for klasse 2 (andet element) 24.2%, og for klasse 3 (tredje element) 9.9%. Softmax-funktionen sikrer, at summen af sandsynlighederne er 1, hvilket gør det muligt at tolke outputtet som en sandsynlighedsfordeling. (Sanderson, 2017b)

3.3 Gradient descent

Gradient descent er en algoritme, der tager loss-funktionen og beregner gradienten af denne i forhold til alle modellens parametre (vægtene og biases). Gradienten er en vektor, der peger i retningen af den største stigning af loss-funktionen. For at minimere loss-funktionen bevæger vi os i den modsatte retning af gradienten. Dette gøres ved at opdatere vægtene og biases i modellen med gradienten ganget med en konstant, kaldet *learning rate*. Processen gentages, indtil loss-funktionen er tilstrækkeligt minimeret. (IBM, 1994; Nielsen, 2019b; Sanderson, 2017b) Antag, at vi organiserer modellens vægte og biases i en søjlevektor $\vec{W}$, og at loss-funktionen er $L(\vec{W})$. Gradienten af loss-funktionen er $\nabla L(\vec{W})$. Derfor beskriver søjlevektoren $-\nabla L(\vec{W})$, hvordan vi kan opdatere $\vec{W}$ for at minimere loss-funktionen og dermed forbedre modellens præstation. Algoritmen, der finder gradienten på baggrund af modellens parametre og loss-funktionen, kaldes *backpropagation*. (Nielsen, 2019b; Sanderson, 2017b) Hvordan backpropagation fungerer, vil blive gennemgået mere detaljeret i et kommende afsnit. Indtil videre antager vi blot, at den fungerer som beskrevet og returnerer den korrekte gradient for modellens parametre.

3.4 Partielle afledte

Partielle afledte beskriver, hvordan en funktion ændrer sig, når én af dens variabler ændres, mens de andre holdes konstante. Antag, vi har en funktion $f(x, y)$, der afhænger af to variabler x og y. Den partielle afledte af f med hensyn til x betegnes $\frac{\partial f}{\partial x}$ og angiver hældningen af f i x-retningen. Tilsvarende betegner $\frac{\partial f}{\partial y}$ hældningen i y-retningen. Partielle afledte er særligt nyttige i optimering af neurale netværk, hvor de spiller en central rolle i gradientbaserede algoritmer. Gradientens komponenter er netop de partielle afledte for hver variabel i modellen. Hvis vi eksempelvis ønsker at minimere en funktion $f(x, y)$, anvender vi gradienten $\nabla f(x, y)$, der består af $\frac{\partial f}{\partial x}$ og $\frac{\partial f}{\partial y}$, til effektivt at navigere mod lavere værdier af f. (Kirsanov, 2024)

3.5 Kædereglen

Kædereglen er en fundamental regel i differentialregning, der gør det muligt at differentiere sammensatte funktioner. Hvis vi har to funktioner, $f(g(x))$, hvor f afhænger af $g(x)$, og g afhænger af x, siger kædereglen, at den afledte af f med hensyn til x er produktet af den afledte af f med hensyn til g og den afledte af g med hensyn til x:

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x) \tag{23}$$

Med Leibniz notation kan kædereglen skrives som:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial x} \qquad \text{hvor } z = g(x) \tag{24}$$

Kædereglen er en central del i neurale netværk og i machine learning, især i algoritmen backpropagation, hvor det kræves at beregne gradienten af en sammensat loss-funktion i forhold til modellens parametre. (Kirsanov, 2024)

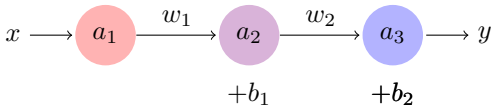
3.6 Backpropagation

Antag et simpelt neuralt netværk som det, der er vist i (Figur 5) tidligere, og loss-funktionen for dette netværk, L. L har som funktionsparameter alle modellens parametre og giver en kvantitativ vurdering af, hvor gode disse parametre er givet et datasæt. En primitiv måde at optimere modellens parametre på er at ændre hver parameter separat og undersøge, om loss-funktionen er højere eller lavere med de nye parametre. Hvis loss er lavere, ændres parameteren, hvis ikke, nulstilles parameteren, og den næste justeres. Gentages denne proces tilstrækkeligt mange gange, vil man nærme sig et minimum i loss-funktionen. Selvom denne metode er primitiv, kan den fungere for en simpel model som den, der er visualiseret i (Figur 5). Dog vil denne tilgang være ineffektiv og ekstremt langsom for en model med hundredevis eller tusindvis af parametre og er derfor ikke en praktisk metode for den gældende problemstilling. Denne vilkårlige permutationsalgoritme er den bedste metode i generelle tilfælde, da der ikke nødvendigvis findes en bedre metode. (Kirsanov, 2024) For differentiable udregninger, som dem i et neuralt netværk, findes der dog en langt bedre metode, der muliggør markant mere effektiv optimering af modellens parametre. Denne algoritmes formål er at forudsige, hvordan en ændring i modellens parametre vil påvirke loss-funktionen, uden at justere manuelt. Selvom

denne algoritme kan lyde umulig, bygger den faktisk på fundamentale matematiske principper. Algoritmen kaldes *backpropagation*. Og helt simpelt går denne algoritme ud på at udregne gradienten af loss-funktionen i forhold til alle modellens parametre, således at gradienten kan udregnes så medlelen kan effektivt optimeres. (Kirsanov, 2024; Nielsen, 2019a; Sanderson, 2017b) eksemplet herunder gennemgår hvordan de afledte udregnes i et simpelt neuralt netværk for at illustrere backpropagation anvendt.

3.6.1 Udregning af gradienten

Beskue det nedstående neurale netværk med 1 neuron i hvert lag og et skjult lag, Dette afsnit har til formål at gennemgå hvordan gradienten af loss funktionen udregnes i forhold til vægtene og biases i modellen. Dette eksempel bruger sigmoid aktiveringsfunktionen og kvadratisk loss funktion, som er en lidt anden end den tidligere beskrevne cross-entropy loss funktion. Dette er gjort da de partielle afledte er lettere at udregne i dette tilfælde og vil derfor gøre det lettere at forstå backpropagation.



Figur 7: Et simpelt neuralt netværk

For at simplificere senere udregninger, er modellen konstrueret således at kun a_2 benytter sig af en aktiveringsfunktion $\sigma(x)$. Så for at udregne y givet x kan denne udregnes således:

$$y = \sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 \tag{25}$$

Og lavet om til en funktion $f(x)$:

$$f(w_1, b_1, w_2, b_2, x) = \sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 \tag{26}$$

Og insættes i loss funktionen $L(y)$:

$$L(y_i) = (y_i - \hat{y}_i)^2 \quad \text{hvor } \hat{y}_i \text{ er det rigtige svar for } i \text{ indeks} \tag{27}$$

$$L(f(w_1, b_1, w_2, b_2, x)) = L(\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2) \tag{28}$$

Hvis vi nu har en loss funktion $L(y_i)$, der kvantificerer, hvor god modellen er til at prediktere, kan vi nu udregne gradienten af L i forhold til w_1, b_1, w_2 og b_2. Dette gøres ved at bruge kædereglen til at udregne de partielle afledte af L i forhold til w_1, b_1, w_2 og b_2. Skrivemåden y_i erstattes hermed med y da indekset ikke er vigtigt for resten af eksemplet. Først udregnes $\frac{\partial L}{\partial b_2}$
Her kan kædereglen bruges til at udregne $\frac{\partial L}{\partial b_2}$:

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial b_2} \tag{29}$$

Lad os først udregne $\frac{\partial L}{\partial y}$, som er den partielle afledte af L i forhold til y, denne er relativt simpel at udregne, da L blot kan differentieres på normal vis:

$$\frac{\partial L}{\partial y} = 2 \cdot (y - \hat{y}) = 2y - 2\hat{y} \tag{30}$$

Og $\frac{\partial y}{\partial b_2}$, som er den partielle afledte af y i forhold til b_2, denne er også simpel at udregne, da resten antages at være konstant og derfor ender vi med:

$$\frac{\partial y}{\partial b_2} = 1 \qquad L(\overbrace{\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2}^{y}) \tag{31}$$

Så $\frac{\partial L}{\partial b_2}$ kan nu udregnes:

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial b_2} = \frac{\partial L}{\cancel{\partial y}} \cdot \frac{\cancel{\partial y}}{\partial b_2} \tag{32}$$

$$= (2y - 2\hat{y}) \cdot 1 = 2y - 2\hat{y} \quad \text{hvor} \quad y = \sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 \tag{33}$$

$$= 2 \cdot \sigma(w_1 \cdot x + b_1) \cdot w_2 + 2b_2 - 2\hat{y} \tag{34}$$

Nu er $\frac{\partial L}{\partial b_2}$ udregnet! Nu kan $\frac{\partial L}{\partial w_2}$ udregnes på samme måde, først opdeles $\frac{\partial L}{\partial w_2}$ efter kædereglen:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w_2} \tag{35}$$

Eftersom $\frac{\partial L}{\partial y}$ allerede er udregnet, kan vi nu udregne $\frac{\partial y}{\partial w_2}$, som er den partielle afledte af y i forhold til w_2. Dette kan udregnes da w_2 kun afhænger af $\sigma(w_1 \cdot x + b_1)$ og da den er konstant i forhold til w_2, den partielle afledte er derfor lig $\sigma(w_1 \cdot x + b_1)$:

$$\frac{\partial y}{\partial w_2} = \sigma(w_1 \cdot x + b_1) \qquad L(\overbrace{\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2}^{y}) \tag{36}$$

Så $\frac{\partial L}{\partial w_2}$ kan nu udregnes:

$$\begin{aligned}
\frac{\partial L}{\partial w_2} &= \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w_2} = \frac{\partial L}{\cancel{\partial y}} \cdot \frac{\cancel{\partial y}}{\partial w_2} && (37)\\
&= (2y - 2\hat{y}) \cdot \sigma(w_1 \cdot x + b_1) \quad \text{hvor} \quad y = \sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 && (38)\\
&= 2 \cdot (\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 - \hat{y}) \cdot \sigma(w_1 \cdot x + b_1) && (39)
\end{aligned}$$

Nu er $\frac{\partial L}{\partial w_2}$ udregnet! Nu kan $\frac{\partial L}{\partial b_1}$ udregnes på samme måde, Dog er denne mere kompleks, da den er "dybere"i kæden, og derfor skal der bruges kædereglen flere gange. Først opdeles $\frac{\partial L}{\partial b_1}$ efter kædereglen:

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial a_2} \cdot \frac{\partial a_2}{\partial b_1} \tag{40}$$

Beskue (Figur 7) og Ligning (28) for hvordan a_2 er defineret. Så $\frac{\partial L}{\partial b_1}$ kan nu udregnes, da $\frac{\partial L}{\partial y}$ allerede er udregnet, kan vi nu udregne $\frac{\partial y}{\partial a_2}$, som er den partielle afledte af y i forhold til a_2. Her kan samme tankegang bruges idet a_2 kun afhænger af w_2:

$$\frac{\partial y}{\partial a_2} = w_2 \qquad L(\overbrace{\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2}^{y}) \tag{41}$$

Nu kan $\frac{\partial a_2}{\partial b_1}$ udregnes, som er den partielle afledte af a_2 i forhold til b_1. Her vil den partielle afledte være $\sigma'(w_1 \cdot x + b_1)$, da a_2 afhænger direkte af b_2 gennem aktiveringsfunktionen

$$\frac{\partial a_2}{\partial b_1} = \sigma'(w_1 \cdot x + b_1) \qquad L(\overbrace{\sigma(w_1 \cdot x + b_1)}^{a_2} \cdot w_2 + b_2) \tag{42}$$

Så $\frac{\partial L}{\partial b_1}$ kan nu udregnes:

$$\begin{aligned}
\frac{\partial L}{\partial b_1} &= \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial a_2} \cdot \frac{\partial a_2}{\partial b_1} = \frac{\partial L}{\cancel{\partial y}} \cdot \frac{\cancel{\partial y}}{\cancel{\partial a_2}} \cdot \frac{\cancel{\partial a_2}}{\partial b_1} && (43)\\
&= (2y - 2\hat{y}) \cdot w_2 \cdot \sigma'(w_1 \cdot x + b_1) \quad \text{hvor} \quad y = \sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 && (44)\\
&= 2 \cdot (\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 - \hat{y}) \cdot w_2 \cdot \sigma'(w_1 \cdot x + b_1) && (45)
\end{aligned}$$

Nu er $\frac{\partial L}{\partial b_1}$ udregnet! Nu kan $\frac{\partial L}{\partial w_1}$ udregnes på samme måde, På samme måde er denne mere kompleks, da den også er "dybere"i kæden, og derfor skal der bruges kædereglen flere gange. Først opdeles $\frac{\partial L}{\partial w_1}$ efter kædereglen:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1} \tag{46}$$

Så $\frac{\partial L}{\partial w_1}$ kan nu udregnes, da $\frac{\partial L}{\partial y}$ og $\frac{\partial y}{\partial a_2}$ allerede er udregnet, Nu kan $\frac{\partial a_2}{\partial w_1}$ udregnes, som er den partielle afledte af a_2 i forhold til w_1. Da $a_2 = \sigma(w_1 \cdot x + b_1)$, afhænger a_2 af w_1 gennem argumentet, og med hensyn til kædereglen, vil den partielle afledte være:

$$\frac{\partial a_2}{\partial w_1} = x \cdot \sigma'(w_1 \cdot x + b_1) \qquad L(\overbrace{\sigma(w_1 \cdot x + b_1)}^{a_2} \cdot w_2 + b_2) \tag{47}$$

Så $\frac{\partial L}{\partial w_1}$ kan nu udregnes:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1} = \frac{\partial L}{\cancel{\partial y}} \cdot \frac{\cancel{\partial y}}{\cancel{\partial a_2}} \cdot \frac{\cancel{\partial a_2}}{\partial w_1} \tag{48}$$

$$= (2y - 2\hat{y}) \cdot w_2 \cdot x \cdot \sigma'(w_1 \cdot x + b_1) \quad \text{hvor} \quad y = \sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 \tag{49}$$

$$= 2 \cdot (\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 - \hat{y}) \cdot w_2 \cdot x \cdot \sigma'(w_1 \cdot x + b_1) \tag{50}$$

Lad os tage et skridt tilbage og opsummere, hvad der er blevet udregnet. Vi har nu udregnet gradienten af loss funktionen i forhold til modellens parametre, netop w_1, b_1, w_2 og b_2. Dette er essentielt for at kunne træne modellen effektivt, da vi nu ved, hvordan loss funktionen ændrer sig, når vi ændrer vægtene og biases. Dette gør det muligt at bruge gradient descent til at minimere loss funktionen og dermed forbedre modellens præstation. Dette gøres i priksis ved at gemme aktiveringerne af de forskellige lag når modellen prediktere, og derefter bruge disse mellemregninger til at regne baglæns og dermed finde gradienten, det er her algoritmen backpropagation får sit navn. Selvom eksemplet i dette afsnit er simpelt, kan samme principper anvendes på langt mere komplekse neurale netværk med mange flere lag og neuroner. Her vil værdierne dog være matricer og vektorer, og derfor vil de partielle afledte således også være organiseret i matricer, mens dette lyder komplekst, er det i praksis blot en udvidelse af de principper, der er blevet gennemgået i dette afsnit, bare med flere indekser.

Lad os for en god ordens skyld tjekke vores svar efter ved brug af Maple CAS værktøjet:

$$L(y) := (y - \hat{y})^2 : \tag{51}$$

$$f(w_1, b_1, w_2, b_2, x) := \sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 : \tag{52}$$

$$\frac{\partial L}{\partial b_2} = \text{diff}(L(f(w_1, b_1, w_2, b_2, x)), b_2) = 2 \cdot \sigma(w_1 \cdot x + b_1) \cdot w_2 + 2 \cdot b_2 - 2 \cdot \hat{y} \tag{53}$$

$$\frac{\partial L}{\partial w_2} = \text{diff}(L(f(w_1, b_1, w_2, b_2, x)), w_2) = 2 \cdot (\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 - \hat{y}) \cdot \sigma(w_1 \cdot x + b_1) \tag{54}$$

$$\frac{\partial L}{\partial b_1} = \text{diff}(L(f(w_1, b_1, w_2, b_2, x)), b_1) = 2 \cdot (\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 - \hat{y}) \cdot w_2 \cdot \sigma'(w_1 \cdot x + b_1) \tag{55}$$

$$\frac{\partial L}{\partial w_1} = \text{diff}(L(f(w_1, b_1, w_2, b_2, x)), w_1) = 2 \cdot (\sigma(w_1 \cdot x + b_1) \cdot w_2 + b_2 - \hat{y}) \cdot w_2 \cdot x \cdot \sigma'(w_1 \cdot x + b_1) \tag{56}$$

Som det ses er vores svar korrekt, og dermed er vores udregninger korrekte. Da udledningen af de afledte for at mere kompliceret netværk ikke er meget anderledes end hvad der tidligere er gennemgået, vil de afledte fra (Keita, 2023; Kurbiel, 2021; Verma, 2020) såvel som den samme generelle tankegang blive brugt til at udlede de partielle afledte uden dybere redegørelse, da dette ligger langt over opgavens omfang/niveau.

4 Valg af programmeringssprog

Til at implementere et neuralt netværk, er det nødvændigt at vælge et programmeringssprog, der er egnet til formålet. Der findes mange forskellige programmeringssprog, der kan bruges til at implementere neurale netværk, herunder Python, R, Java, C++, og mange flere. Men da det ønskes at programmet er interaktivt og let at bruge, er Python med PyGame eller Java med Processing de mest oplagte valg. Python og mere specifikt NumPy, som er et bibliotek til Python, er enormt brugervenligt og hurtigt at udføre matriceregning i. Java er også et godt valg, da det er et meget populært programmeringssprog, med faste typer som gør det nemmere at undgå fejl. Processing er et bibliotek til Java, der gør det nemt at lave grafik og interaktive applikationer. Men grundet Python's minimale syntaks, er det valgt til udviklingen af programmet.

5 Neuralt netværk implementeret i Python

Til genkendelse af håndskrevne tal anvendes MNIST-datasættet, som er et velkendt benchmark inden for maskinlæring. Datasættet indeholder billeder af håndskrevne tal med opløsning på 28×28 pixels skal input laget have $28 \cdot 28 = 748$ inputs. til de skjulte lag blev der nogenlunde arbitrært valgt 2 lag af 16 neuroner, outputlaget er bestående af 10 neuroner hvor hver repræsenterer sandsynligheden for at inputtet er det givne tal. Der blev valgt at bruge sigmoid funktionen som aktiveringsfunktion i de skjulte lag og softmax i outputlaget. Der blev valgt at bruge loss funktioned *cross-entropy loss* da det er en standard loss funktion til klassifikationsopgaver. Og gradient decent algorimen bruges til at minimere loss funktionen.

5.1 Dataindlæsning

MNIST datasættet er meget populært og der ligger derfor allerede en masse implementeringer af indlæsning af datasættet på nettet. Der blev valgt at bruge en implementering fra (Khodabakhsh, 2019) da den er udviklet som en klasse og derfor er nem at bruge. Klassen er vist i (Kodestykke 1), og kan indlæse både trænings- og testdatasættet med metoden `load_data()`. denne process tager ca. 1000ms på en almindelig computer, og skal kun køres en gang, når programmet starter, da dataen gemmes i arbejdshukommelsen efterfølgende.

5.2 Tegnefladen

Da opgaveformuleringen kræver et "tegneprogram", hvori brugeren selv kan indtaste tal, vil netop dette udvikles. Her tages Python med biblioteket PyGame i brug, da det tilbyder en række funktioner til at tegne til et vindue på computeren. Først skal et vindue laves:

```
8  pygame.init()
9  WIDTH, HEIGHT = 800, 600
10 screen = pygame.display.set_mode((WIDTH, HEIGHT))
11 pygame.display.set_caption("Drawing Pad")
```

Nu er vinduet tegnet. Det blev valgt at lave tegnefladen som en klasse. Klassen skal holde styr på opløsningen såvel som alle pixelsene. Derfor kan vi definere konstruktøren således:

```
26 class DrawingPad:
27     def __init__(self):
28         self.dx = 28
29         self.dy = 28
30         self.last_pixel_x = None
31         self.last_pixel_y = None
32         self.pixel_size = min(WIDTH // self.dx, HEIGHT // self.dy)
33         self.pixel_values = [
34             [0 for _ in range(self.dy)] for _ in range(self.dx)]
```

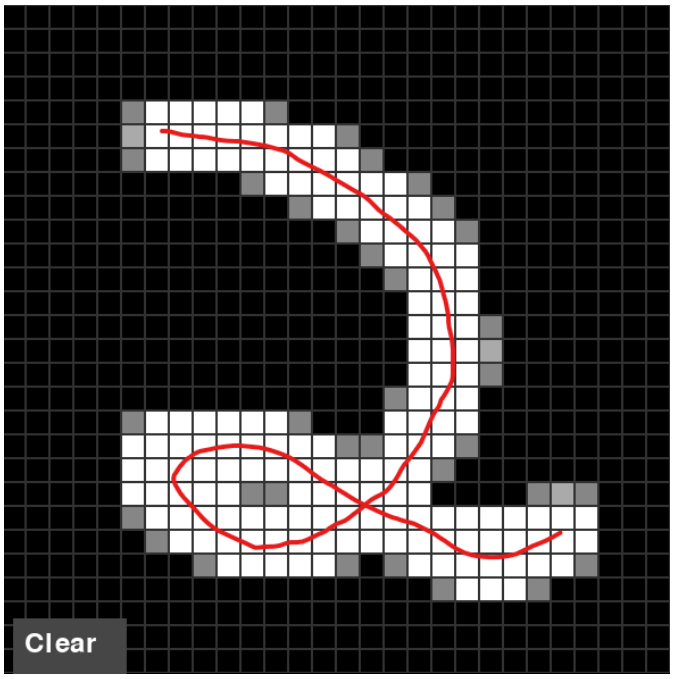
Her er `self.dx` og `self.dy` opløsningen af tegnefladen. `self.pixel_size` er størrelsen af pixelsene. Da pixelsene altid bør være kvadratiske, tages `min()` af både den lodrette og vandrette dimension. `self.pixel_values` er værdierne af alle de forskellige pixels i et 2D-array. Alle værdierne starter som 0. `self.last_pixel_x` og `self.last_pixel_y` er interne variable til at holde styr på tegnelogikken. Dette forklares senere. Nu kan tegnefladen tegnes til vinduet:

```
62     def draw(self):
63         for i in range(self.dx):
64             for j in range(self.dy):
65                 color_value = self.pixel_values[i][j]
66                 rect = pygame.Rect(
67                     i * self.pixel_size,
68                     j * self.pixel_size,
69                     self.pixel_size,
70                     self.pixel_size,
71                 )
72                 pygame.draw.rect(
73                     screen, (color_value, color_value, color_value), rect)
74                 pygame.draw.rect(screen, (50, 50, 50), rect, 1)
```

Denne kode tegner et gitter af rektangler på skærmen ved at iterere gennem `self.dx` og `self.dy` og bruger `pygame.draw.rect` til at tegne hvert rektangel med en farve baseret på `self.pixel_values`, såvel som en grå 1-pixel-tyk border. For at opdatere tegnefladen, når brugeren tegner med musen, blev metoden `handle_event()` udviklet:

```
36      def handle_event(self, event):
37          if event.type == pygame.MOUSEBUTTONDOWN:
38              x, y = event.pos
39              grid_x = x // self.pixel_size
40              grid_y = y // self.pixel_size
41              if (self.last_pixel_x == grid_x and self.last_pixel_y == grid_y):
42                  return
43              if 0 <= grid_x < self.dx and 0 <= grid_y < self.dy:
44                  self.last_pixel_x = grid_x
45                  self.last_pixel_y = grid_y
46                  for i in range(-1, 2):
47                      for j in range(-1, 2):
48                          new_x = grid_x + i
49                          new_y = grid_y + j
50                          if 0 <= new_x < self.dx and 0 <= new_y < self.dy:
51                              distance = (
52                                  (new_x - grid_x) ** 2 + (new_y - grid_y) ** 2
53                              ) ** 0.5
54                              newPixelValue = min(
55                                  255,
56                                  self.pixel_values[new_x][new_y]
57                                  + int(255 * (1 - distance / 3)),
58                              )
59                              if newPixelValue > self.pixel_values[new_x][new_y]:
60                                  self.pixel_values[new_x][new_y] = newPixelValue
```

`handle_event()` tjekker først, om hændelsen er `pygame.MOUSEBUTTONDOWN`. Hvis ikke, afsluttes funktionen. Musens klik-koordinater konverteres til gitterkoordinater på tegnefladen og sammenlignes med det sidste klik for at sikre, at den kun opdaterer én gang per pixel. Hvis gitterkoordinaterne er gyldige, opdaterer funktionen den seneste klikposition (`self.last_pixel_x` og `self.last_pixel_y`) og justerer pixelværdierne i en 3x3-rude omkring klikket. For hver nabocelle beregnes afstanden til musen. En ny pixelværdi (maks. 255) beregnes og opdateres, hvis den nye værdi er højere end den nuværende. Herunder på (Figur 8) ses et eksempel på dette, hvor den røde streg er den faktiske sti, som musen tog. Yderligere kan hele tegnefladeklassen beskues i (Kodestykke 4) på linje 26-87:



Figur 8: Eksempel på tegnefladen og den faktiske sti, som musen tog

Som ses på billedet muliggør afstandsmetoden at tegne en mere realistisk sti hvor der opstår et naturligt gradient i intensiteten. denne metode er valgt da den imiterer MNIST-datasættets udseende bedre end en simpel binær metode, og siden modellen er trænet på MNIST-datasættet, vil det give bedre resultater at komme så tæt på datasættet som muligt.

5.3 Prediktering

Hver frame opdateres modellens prediktion af det tegnede tal. Til dette bruges metoden `predict()`, den bruger modellens parametre som er gemt i `model.npz` filen, herunder ses indlæsningen af disse parametre samt selve `predict()` metoden:

```
100 def main():
101     model = None
102     W1, b1, W2, b2, W3, b3 = None, None, None, None, None, None
103     try:
104         model = np.load("model.npz")
105         W1, b1 = model['W1'], model['b1']
106         W2, b2 = model['W2'], model['b2']
107         W3, b3 = model['W3'], model['b3']
108     except FileNotFoundError:
109         print("Model file not found. Please run train.py to train the model.")
110         sys.exit(1)
```

```
89 def predict(image, W1, b1, W2, b2, W3, b3):
90     x = np.array(image).reshape(-1) / 255.0
91     Z1 = x @ W1 + b1
92     A1 = sigmoid_activation(Z1)
93     Z2 = A1 @ W2 + b2
94     A2 = sigmoid_activation(Z2)
95     Z3 = A2 @ W3 + b3
96     A3 = softmax(Z3)
97     return np.argmax(A3), A3
```

`.npz` formatet er blot et format til at gemme flere NumPy arrays i en fil. i `main()` metoden indlæses disse parametre og gemmes i variablerne `w1`, `b1`, `w2`, `b2`, `w3` og `b3`. Disse bruges i `predict()` metoden sammen med et input, som er en 2D-array af pixelværdierne, som bliver lavet om til `x` med `.reshape(-1)`, hvor hver række er sat sammen til en lang række. Herunder ses selve `predict()` metoden bare skrevet som matematik:

$$y = \text{softmax}\left(W_3 \cdot \sigma\left(W_2 \cdot \sigma\left(W_1 \cdot X + B_1\right) + B_2\right) + B_3\right) \tag{57}$$

Denne ligning repræsenterer outputtet y fra `predict()`. Modellen udfører følgende beregninger: Inputtet X multipliceres med vægtmatricen W_1 og lægges sammen med biasvektoren B_1. Resultatet `Z1` gennemgår sigmoid aktiveringsfunktionen σ. Denne proces gentages for det andet lag med vægte W_2 og bias B_2. I det tredje lag multipliceres de transformerede data med W_3, lægges sammen med B_3, og sendes gennem softmax-funktionen. Til sidst anvendes en cross-entropy loss funktionen L på outputtet. Operatoren $\cdot$ betegner matrixmultiplikation, hvilket er passende i denne sammenhæng, dog er samme operator i Python-notation lig `@`. i koden er `Z` blot en betegnelse for værdierne i de forskellige lag før de sendes gennem aktiveringsfunktionen og bliver til `A`. Så for at prediktere modellen med værdierne fra tegnefladen kan man gøre således:

```
133         image = drawing_pad.get_image()
134         predicted_digit, probabilities = predict(image, W1, b1, W2, b2, W3, b3)
```

5.4 De afledte

Som tidligere beskrevet er selve udregningen af de partielle afledte over opgavens omfang og vil derfor ikke blive udregnet fra bunden. Der vil tages udgangspunkt i de afledte fra (Keita, 2023; Kurbiel, 2021; Verma, 2020). De afledte nævnt i disse kilder antages at være korrekte, dog vil de senere blive testet med *finite difference* metoden for at sikre deres korrekthed. (Verma, 2020) påstår følgende afledte for de forskellige lag i et netværk med 1 skjult lag med Cross-Entropy loss og Softmax aktiveringsfunktion i outputlaget:

$$dZ_2 = \frac{\partial L}{\partial Z_2} = A_2 - Y \tag{58}$$

$$dW_2 = \frac{\partial L}{\partial W_2} = \frac{1}{m}(dZ_2 \cdot A_1^T) \tag{59}$$

$$dB_2 = \frac{\partial L}{\partial B_2} = \frac{1}{m}\sum dZ_2 \tag{60}$$

$$dZ_1 = \frac{\partial L}{\partial Z_1} = W_2^T \cdot \sigma'(Z_1) \tag{61}$$

$$dW_1 = \frac{\partial L}{\partial W_1} = \frac{1}{m}(dZ_1 \cdot X^T) \tag{62}$$

$$dB_1 = \frac{\partial L}{\partial B_1} = \frac{1}{m}\sum dZ_1 \tag{63}$$

Hvor m er antallet af træningsdata, A_1 og A_2 er aktiveringerne i de forskellige lag, Y er de rigtige svar, og σ' er den afledte af sigmoid funktionen, bemærk at Z er værdierne i de forskellige lag før de sendes gennem aktiveringsfunktionen og hermed bliver til A. Kilden (Kurbiel, 2021) er enige med denne differentiering af Cross-entry loss og Softmax aktiveringsfunktionen. Da vores model er bestående af 2 skjulte lag, kan disse afledte udvides til at gælde for 2 skjulte lag, da hver lag kun er afhængig af det forrige, er det blot en udvidelse af de samme principper som tidligere beskrevet ba begge de skjulte lag er aktiveret med sigmoid funktionen. Derfor kan de afledte for hele modellen skrives som:

$$dZ_3 = \frac{\partial L}{\partial Z_3} = A_3 - Y \tag{64}$$

$$dW_3 = \frac{\partial L}{\partial W_3} = \frac{1}{m}(dZ_3 \cdot A_2^T) \tag{65}$$

$$dB_3 = \frac{\partial L}{\partial B_3} = \frac{1}{m}\sum dZ_3 \tag{66}$$

$$dZ_2 = \frac{\partial L}{\partial Z_2} = W_3^T \cdot \sigma'(Z_2) \tag{67}$$

$$dW_2 = \frac{\partial L}{\partial W_2} = \frac{1}{m}(dZ_2 \cdot A_1^T) \tag{68}$$

$$dB_2 = \frac{\partial L}{\partial B_2} = \frac{1}{m}\sum dZ_2 \tag{69}$$

$$dZ_1 = \frac{\partial L}{\partial Z_1} = W_2^T \cdot \sigma'(Z_1) \tag{70}$$

$$dW_1 = \frac{\partial L}{\partial W_1} = \frac{1}{m}(dZ_1 \cdot X^T) \tag{71}$$

$$dB_1 = \frac{\partial L}{\partial B_1} = \frac{1}{m}\sum dZ_1 \tag{72}$$

Bemærk her hvordan dZ_2, dW_2 og dB_2 på samme måde som dZ_1, dW_1 og dB_1 da de på lige vis kun er afhængige af det forrige lags værdier. Disse afledte kan nu implementeres i Python således:

```
43      def update(self, X, Y, pred, rate=1):
44          m = X.shape[0]
45          A1, A2, A3 = pred['A1'], pred['A2'], pred['A3']
46          dZ3 = A3 - Y
47          dW3 = (A2.T @ dZ3) / m
```

```
48          db3 = np.sum(dZ3, axis=0) / m
49          dZ2 = (dZ3 @ self.W3.T) * sigmoid_derivative_from_sigmoid_output(A2)
50          dW2 = (A1.T @ dZ2) / m
51          db2 = np.sum(dZ2, axis=0) / m
52          dZ1 = (dZ2 @ self.W2.T) * sigmoid_derivative_from_sigmoid_output(A1)
53          dW1 = (X.T @ dZ1) / m
54          db1 = np.sum(dZ1, axis=0) / m
55          self.W1 -= rate * dW1
56          self.b1 -= rate * db1
57          self.W2 -= rate * dW2
58          self.b2 -= rate * db2
59          self.W3 -= rate * dW3
60          self.b3 -= rate * db3
61          return {'W1': dW1, 'b1': db1, 'W2': dW2, 'b2': db2, 'W3': dW3, 'b3': db3}
```

Bemærk på linje 54-59 hvordan indlæringsraten ganges med gradienten og subtraheres fra vægtene og biases. Dette er selve gradient descent algoritmen, som bruges til at minimere loss funktionen. Bemærk også at de afledte bruger den afledte af sigmoid funktionen, hvor i koden bruges `sigmoid_derivative_from_sigmoid_output(y)` frem for `sigmoid_derivative(x)` til at udregne denne. Beskue (Kodestykke 2) i bilag for at se implementeringen af denne funktion, Men grunden til dette er at sigmoid funktionen allerede er udregnet i `predict()` metoden tidligere, og derfor kan den bruges til at udregne hældningen meget hurtigere end hvis den skulle udregnes igen. Det kan også tydligt ses at `sigmoid_derivative_from_sigmoid_output(y)` er hurtigere end `sigmoid_derivative(x)` i (Kodestykke 2) i bilag på linje 6-11. Selvom det er en lille besparelse enkeltvis, vil denne funktion blive kaldt enormt mange gange, og derfor er det vigtigt at den er så hurtig som muligt, så man undgår at udregne ting flere gange, når det ikke er nødvendigt. (Smith, 2024)

5.5 Træning

På baggrund af de implementerede metoder kan modellen nu trænes. Dette gøres i `train()` metoden, som ses herunder:

```
63      def train(self, X, Y, epochs):
64          idx = np.random.permutation(X.shape[0])
65          X, Y = X[idx], Y[idx]
66
67          X_batches = np.array_split(X, X.shape[0] // 1000)
68          Y_batches = np.array_split(Y, Y.shape[0] // 1000)
69          for epoch in range(1, epochs + 1):
70              for X_batch, Y_batch in zip(X_batches, Y_batches):
71                  pred = self.predict(X_batch)
72                  self.update(X_batch, Y_batch, pred, rate=1)
73              if epoch % 10 == 0:
74                  loss = -np.mean(np.sum(Y_batch * np.log(pred['A3'] + 1e-8), axis=1))
75                  accuracy = np.mean(np.argmax(pred['A3'], axis=1) == np.argmax(Y_batch, axis=1))
76                  test_pred = self.predict(x_test)
77                  test_accuracy = np.mean(np.argmax(test_pred['A3'], axis=1) == y_test)
78                  print(f"Epoch {epoch} Loss: {loss:.4f}, Accuracy: {accuracy * 100:.2f}%, Test
                        ↪  Accuracy: {test_accuracy * 100:.2f}%")
79              if epoch % 100 == 0:
80                  self.export_to_file("model.npz")
```

`train()` er konstrueret således, at den træner modellen over et specificeret antal epochs. For hver epoch bliver træningsdataene blandet med en tilfældig permutation af indeks, der genereres med `np.random.permutation(X.shape[0])`. Herefter omorganiseres `X` og `Y` baseret på denne rækkefølge for at sikre, at træningsforløbet ikke bliver påvirket af dataenes oprindelige rækkefølge. Efter blandingen opdeles de blandede data i mindre batches af 1.000 datapunkter hver. Dette muliggør brugen af mini-batch gradient descent, som er en mere effektiv, men støjende tilgang til

minimering af loss-funktionen, da den bruger en tilnærmelse af gradienten. Dette har som biprodukt, at den kan undslippe visse lokale minima. For hvert batch kaldes først `predict()`-metoden, som beregner modellens prediktioner, og derefter `update()`-metoden, som opdaterer modellens parametre med backpropagation-algoritmen. Hver 10. epoch evalueres og logges modellen ydeevne, både på træningsdataene og på et separat testdatasæt, som består af 10.000 datapunkter, der ikke er blevet brugt til træning. Dette gøres for at overvåge modellens præstation på nye, usete data, såvel som for at undgå overfitting. For at gemme modellen bliver den løbende gemt i en fil kaldet `model.npz` hver 100. epoch ved hjælp af `export_to_file()`-metoden, som kalder `np.savez()`-metoden fra NumPy. Dette giver mulighed for at stoppe træningen, når et tilfredsstillende resultat er opnået. Samlet set implementerer `train()` en standard træningsproces for et neuralt netværk, som inkluderer datablanding, mini-batch-træning, løbende evaluering og lagring af modellen.

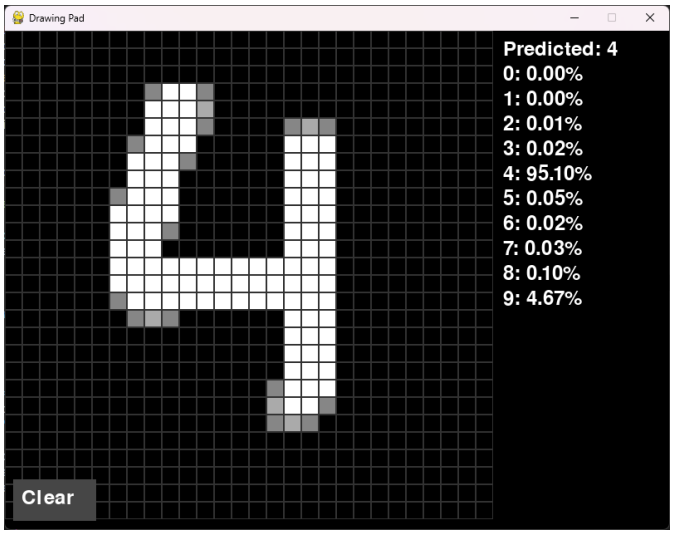
Når denne metode kaldes ses følgende i konsollen:

```
C:/.../sop> python train.py
Loading MNIST data...
MNIST data loaded.
Epoch 10 Loss: 0.3930, Accuracy: 88.67%, Test Accuracy: 88.72%
Epoch 20 Loss: 0.2989, Accuracy: 91.32%, Test Accuracy: 91.01%
Epoch 30 Loss: 0.2545, Accuracy: 92.58%, Test Accuracy: 92.12%
Epoch 40 Loss: 0.2264, Accuracy: 93.43%, Test Accuracy: 92.66%
Epoch 50 Loss: 0.2062, Accuracy: 94.03%, Test Accuracy: 93.15%
Epoch 60 Loss: 0.1910, Accuracy: 94.46%, Test Accuracy: 93.53%
Epoch 70 Loss: 0.1792, Accuracy: 94.84%, Test Accuracy: 93.85%
Epoch 80 Loss: 0.1694, Accuracy: 95.09%, Test Accuracy: 94.07%
Epoch 90 Loss: 0.1610, Accuracy: 95.33%, Test Accuracy: 94.25%
Epoch 100 Loss: 0.1536, Accuracy: 95.54%, Test Accuracy: 94.31%
```

Med denne model arkitektur på 2 skjulte lag med 16 neuroner i hvert er en testnøjagtighed på 94.31% efter 100 epochs yderst fyldestgørende til denne opgaves brugstilfælde. Dette resultat er opnået med en træningstid på ca. 40 sekunder på en almindelig computer. Yderligere epochs forbedre ikke medellen markant og derfor er det ikke nødvendigt at træne modellen længere. Hvordan modellen kan forbedres yderligere vil blive diskuteret i et senere afsnit.

5.6 Genkendelse af brugerinput

Når modellen er trænet, kan den bruges til at genkende brugerinput. De vigtige dele af denne rutine er allerede præsenteret tidligere, se (Kodestykke 4) for hele rutinen. Når brugeren tegner et tal på tegnefladen, opdateres modellens prediktioner i realtid. Denne metode er effektiv nok til at tidstagning på metoden ikke er relevant, på en almindelig computer er denne metode hurtig nok til at kunne køre hver frame uden problemer. Herunder ses programmet i aktion:



Figur 9: Tegnefladen i aktion

På (Figur 9) ses tegnefladen i aktion, hvor brugeren har tegnet et 4-tal som modellen aldrig har set før. Modellen har dog ingen problemer med at genkende tallet som et 4-tal med overvejende sikkerhed. Dette er et godt tegn på at modellen er generaliseret og ikke overtilpasset til træningsdataene.

6 Vurdering af neurale netværk til genkendelse af håndskrevne tal

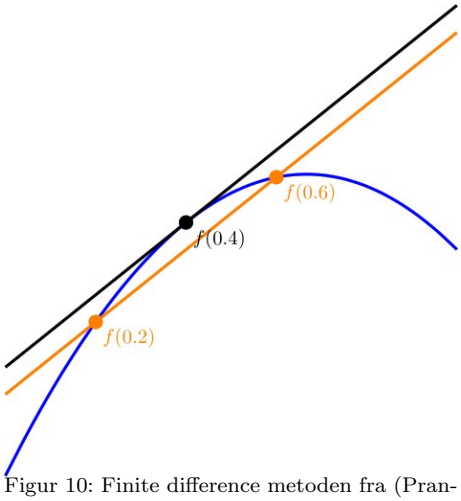
Som ses i træningsprocessen, er modellen trænet til en testnøjagtighed på 94.31% efter 100 epochs. I virkeligheden, med udgangspunkt i brugerens input, vil denne nøjagtighed være lavere. Grunden til dette er, at MNIST-datasættet er centreret med *center of mass*-algoritmen, som er en algoritme, der finder centrum af "massen"i et billede og translaterer billedet, således at centrum af massen er i midten af billedet. Derudover er datasættet også skaleret, således at der hverken er meget små eller meget store tal i datasættet størrelsesmæssigt. Dette er ikke tilfældet med brugerinput, som kan være skaleret og translateret på mange forskellige måder, som modellen ikke har lært at håndtere. Derfor vil modellen have en lavere nøjagtighed på brugerinput end på testdatasættet. Der er to måder uden om denne begrænsning: Den ene er at bruge data augmentation, som er en teknik, hvor man tilføjer støj som rotering, skalering og tilfældige pixels til træningsdatasættet for at gøre modellen mere robust overfor forskellige inputs. Den anden måde er at udføre samme preprocesseringsalgoritme, som datasættet har gennemgået, på brugerinputtet. Dette vil fjerne variansen i brugerinputtet og øge nøjagtigheden.

To vigtige ting, man altid skal være opmærksom på, når man opstiller og træner neurale netværk, er *overfitting* og *underfitting*. Overfitting opstår, når modellen er for kompleks i forhold til træningsdataene, og derfor lærer modellen træningsdataene udenad og ikke kan generalisere denne viden til nye data. Underfitting opstår, når modellen er for simpel i forhold til træningsdataene og derfor ikke kan lære træningsdataene godt nok. Dette kan ses i træningsprocessen ved at sammenligne træningsnøjagtigheden med testnøjagtigheden. Hvis træningsnøjagtigheden er meget højere end testnøjagtigheden, er modellen overtilpasset, og hvis både træningsnøjagtigheden og testnøjagtigheden er lav, er modellen undertilpasset (eller datasættet er ikke generaliserbart). I dette tilfælde er træningsnøjagtigheden og testnøjagtigheden meget tæt på hinanden, hvilket er et godt tegn på, at modellen er generaliseret og ikke overtilpasset. Dette kan også verificeres ved at afprøve modellen på egne tegninger, som tidligere beskrevet. Det blev nævnt, at valget af 2 lag af 16 neuroner var nogenlunde arbitrært, og derfor kan det være interessant at undersøge, hvordan modellen præsterer med flere eller færre neuroner i de skjulte lag, såvel som flere eller færre lag. Dette er en kunst i sig selv og kræver en masse eksperimentering og erfaring for at finde den optimale modelarkitektur. Man kan dog overbevise sig selv om, at det første skjulte lag finder de mindre detaljer i billedet, såsom hjørner og kurver, mens det andet skjulte lag finder de større detaljer, som cirkler og linjer. Dette kulminerer i outputlaget, som tager disse detaljer og bruger dem til at genkende tallet. (Sanderson, 2017b) Dette er blot et gæt på, hvordan denne maskine tænker. I virkeligheden er det relativt komplekst at undersøge, hvordan modellen reelt "tænker,"da ingen har fortalt den, hvordan den skal genkende tal, modellens adfærd er fremspirende som funktion af datasættet.

6.1 Validering af de afledte ved at teste gradienten med finite difference

Som nævnt tidligere er de afledte for det fulde netværk ikke blevet udregnet fra bunden, men derimod delvist taget fra internetkilder, mens resten er blevet regnet manuelt. Selvom beviset for disse ligger uden for opgavens niveau, er det stadig kritisk at teste disse afledte for at sikre deres korrekthed. Dette gøres ved brug af *finite difference*-metoden, som er en metode til at estimere gradienten af en funktion ved at tage forskellen mellem to punkter, der er meget tæt på hinanden. Dette kan ses på (Figur 10). Denne metode er meget simpel og kræver ikke differentialregning for at udføre. Metoden indebærer at tage to punkter med samme afstand fra x. Disse punkter er $x - \epsilon$ og $x + \epsilon$, hvor ϵ er en meget lille værdi. Nu kan gradienten estimeres som:

$$\frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \approx f'(x) \tag{73}$$



Figur 10: Finite difference metoden fra (Prancкevičius, 2015)

Værdien ϵ vil i teorien være så lille som muligt, da dette vil øge nøjagtigheden af estimatet. Dog er man typisk begrænset af maskinens præcision. Datatypen `float` har en præcision på ca. 6-9 decimaltal (Wagner, 2022), og derfor vil det være fordelagtigt at vælge ϵ i en størrelsesorden, så man ikke mister præcision. I dette tilfælde er $\epsilon = 10^{-4}$ valgt. Når man arbejder med neurale netværk, som grundlæggende er funktioner af mange variable, kan denne metode bruges til at estimere gradienten. Det gøres ved at tilføje en lille permutation til hver parameter i modellen, én ad gangen, og udføre beregningen for hver af dem. Ved at gentage dette for alle parametrene i modellen kan man sammenligne de estimerede gradienter med de gradienter, som modellen har beregnet. Hvis de to sæt gradienter stemmer godt overens, er de beregnede gradienter korrekte. Den endelige implementering af denne metode kan ses i (Kodestykke 3) på linje 85-128. Når metoden køres, ses følgende i konsollen:

```
C:/.../sop> python train.py
Loading MNIST data...
MNIST data loaded.
Gradient check for W1: min = 0.0000000000, max = 0.0000664145, mean = 0.0000001361
Gradient check for b1: min = 0.0000000080, max = 0.0000002913, mean = 0.0000001115
Gradient check for W2: min = 0.0000000060, max = 0.0000352297, mean = 0.0000004651
Gradient check for b2: min = 0.0000000940, max = 0.0000019676, mean = 0.0000003838
Gradient check for W3: min = 0.0000000684, max = 0.0000012608, mean = 0.0000001779
Gradient check for b3: min = 0.0000000890, max = 0.0000003139, mean = 0.0000001627
Gradient check execution time: 1.9709 seconds
```

Som det ses, er de estimerede gradienter meget tæt på de udregnede gradienter for alle parametrene i modellen. Dog er `W1`- og `W2`-gradienterne højere end de andre. Dette skyldes primært, at disse værdier er mere indviklede at udregne, og derfor er der en øget chance for, at usikkerhed ophober sig i værdierne. Dette er dog ikke et problem, da alle forskellende i de 2 udregninger er meget tæt på 0. Derfor kan det konkluderes, at de udregnede gradienter er korrekte, og dermed er de afledte nu vist at være korrekte.

6.2 Begrænsninger af et simpelt neuralt netværk

Selvom et simpelt neuralt netværk som det, der er implementeret i denne opgave, er vist til at være effektivt til genkendelse af håndskrevne tal i realtid, er der stadig begrænsninger og ulemper ved denne tilgang. En af de største begrænsninger ved simple neurale netværk er, at positionen af det tegnede tal på tegnefladen har stor betydning for modellens prediktering. Hvis et 2-tal tegnes i midten, har modellen typisk ingen problemer med at genkende tallet, hvorimod hvis samme 2-tal tegnes i hjørnet af tegnefladen, vil modellen sandsynligvis prediktere forkert. I simple neurale netværk kigges der på hele billedet som en lang række af pixels, som ikke er sammenhængende. Dette er ikke optimalt, idet håndskrevne tal typisk er sammenhængende, og derfor vil det være fordelagtigt at kigge på sammenhængen mellem pixels. Dette kan gøres med et Convolutional Neural Network (CNN). Et CNN er en type neuralt netværk, som er specielt designet til at lede efter mønstre i data. Denne type modeller har specialiserede lag, der kaldes convolutional lag, som er i stand til at finde mønstre i nærtliggende data ved at træne på mønstre

i små dele af dataen. Dette gør CNN'er meget effektive til at genkende mønstre i data som billeder, lyd og tekst. Fordi positionen af det tegnede tal på tegnefladen ikke vil have nogen særlig betydning for modellens prediktering, vil en CNN-model være mere robust overfor forskellige inputs i dette brugstilfælde. Med deres mere komplekse arkitektur medfølger dog også en øget træningsbyrde og øget kompleksitet i udledningen af resultaterne. Dette er grunden til, at et simpelt neuralt netværk er valgt til denne opgave. Men CNN'er er vist til at yde markant bedre på MNIST-datasættet. En variant af CNN'er ydede en testnøjagtighed på 99.7% på MNIST-datasættet (LeCun m.fl., 1994), og derfor vil det være nyttigt at bruge en CNN-model eller en anden mere kompleks model til at genkende håndskrevne tal i en reel applikation.

7 Konklusion

I denne opgave er der blevet udviklet et neuralt netværk til genkendelse af håndskrevne tal i realtid uden brug af unødvændige abstraktioner i Python. der blev redegjort for neurale netværk såvel som de grundlæggende matematiske principper bag neurale netværk, herunder matrixmultiplikation og kædereglen. Modellen er bestående af 2 skjulte lag med 16 neuroner i hvert lag med sigmoid aktiveringsfunktioner og et outputlag med softmax aktiveringsfunktion. Modellen er trænet på MNIST-datasættet og opnåede en testnøjagtighed på 94.31% efter 100 epochs. Modellens parametre blev vurderet kvantitativt ved at bruge cross-entropy loss funktionen. For at sikre nøjagtigheden af modellens gradienter blev finite difference metoden anvendt, hvilket bekræftede de udregnede gradienters korrekthed. Begrænsningerne ved det simple neurale netværk blev også diskuteret, især modellens følsomhed overfor positionen af håndskrevne tal. Det blev bemærket, at mere komplekse modeller som Convolutional Neural Networks (CNN'er) kan overvinde disse begrænsninger og forbedre modellens ydeevne markant. Alt i alt viser denne opgave, at selv et simpelt neuralt netværk kan opnå høj nøjagtighed i genkendelse af håndskrevne tal, men der er stadig plads til forbedringer som f.eks. at anvende mere avancerede modeltyper.

Litteratur

Delclos, Y. (2021 februar). Computer vision — Image recognition via Deep Learning [Accessed: 2024-12-10]. *Artificial Intelligence in Plain English*. https://ai.plainenglish.io/computer-vision-image-recognition-via-deep-learning-a144fc45f105

IBM. (1994). What is Gradient Descent? — IBM. *IBM*. https://www.ibm.com/topics/gradient-descent

Keita, Z. (2023 december). Mastering Backpropagation: A Comprehensive Guide for Neural Networks. *DataCamp*. Hentet 7. december 2024, fra https://www.datacamp.com/tutorial/mastering-backpropagation

Khodabakhsh, H. (2019 januar). Read MNIST Dataset — Kaggle. *Kaggle*. https://www.kaggle.com/code/hojjatk/read-mnist-dataset

Kirsanov, A. (2024). The Most Important Algorithm in Machine Learning. https://www.youtube.com/watch?v=SmZmBKc7Lrs

Kumar, A. (2024 maj). Mean Squared Error vs Cross Entropy Loss Function. *Analytics Yogi*. https://vitalflux.com/mean-squared-error-vs-cross-entropy-loss-function/

Kurbiel, T. (2021 april). Derivative of the Softmax Function and the Categorical Cross-Entropy Loss. *Towards Data Science*. Hentet 8. december 2024, fra https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1

Lauritzen, N., & Bökstedt, M. (2019). Tal og Lineær Algebra 2019 4 Matricer. *Matematisk Institut, Aarhus Universitet*. https://data.math.au.dk/interactive/lintrans/Chapters/vektorerogmatricer.html

LeCun, Y., Cortes, C., & Burges, C. J. (1994). MNIST handwritten digit database. *Courant Institute NYU, Google Labs, Microsoft Research*. https://yann.lecun.com/exdb/mnist/

Nielsen, M. (2019a). How the backpropagation algorithm works. http://neuralnetworksanddeeplearning.com/chap2.html

Nielsen, M. (2019b). Using neural nets to recognize handwritten digits. http://neuralnetworksanddeeplearning.com/chap1.html

Pranckevičius, A. (2015 august). Finite Differences. *The blog at the bottom of the sea*. https://blog.demofox.org/2015/08/02/finite-differences/

Sanderson, G. (2017a oktober). 3Blue1Brown - But what is a Neural Network? *3Blue1Brown*. https://www.3blue1brown.com/lessons/neural-networks

Sanderson, G. (2017b). Gradient descent, how neural networks learn. *3Blue1Brown*. https://www.3blue1brown.com/lessons/gradient-descent

Simonson, M. (2015 oktober). Matrix Multiplication Made Easy. *American Mathematical Society*. https://blogs.ams.org/mathgradblog/2015/10/19/matrix-multiplication-easy/

Smith, W. (2024 februar). Derivative of Sigmoid Function - Simplified Explanation for Better Understanding. *The Story of Mathematics*. Hentet 8. december 2024, fra https://www.storyofmathematics.com/derivative-of-sigmoid-function/

St. Clair, B. (2021 april). Explainer: What is a neuron? *Science News*. https://www.snexplores.org/article/explainer-what-is-a-neuron

Verma, A. (2020 juli). Building A Neural Net from Scratch Using R - Part 1. *R Views*. Hentet 8. december 2024, fra https://rviews.rstudio.com/2020/07/20/shallow-neural-net-from-scratch-using-r-part-1/

Wagner, B. (2022 september). Floating-point numeric types - C# reference. *Microsoft Learn*. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types

Bilag

mnist_dataloader.py

```python
from array import array
import struct
import numpy as np

class MnistDataloader(object):
    def __init__(self, training_images_filepath,training_labels_filepath,
                 test_images_filepath, test_labels_filepath):
        self.training_images_filepath = training_images_filepath
        self.training_labels_filepath = training_labels_filepath
        self.test_images_filepath = test_images_filepath
        self.test_labels_filepath = test_labels_filepath

    def read_images_labels(self, images_filepath, labels_filepath):
        labels = []
        with open(labels_filepath, 'rb') as file:
            magic, size = struct.unpack(">II", file.read(8))
            if magic != 2049:
                raise ValueError('Magic number mismatch, expected 2049, got {}'.format(magic))
            labels = array("B", file.read())

        with open(images_filepath, 'rb') as file:
            magic, size, rows, cols = struct.unpack(">IIII", file.read(16))
            if magic != 2051:
                raise ValueError('Magic number mismatch, expected 2051, got {}'.format(magic))
            image_data = array("B", file.read())
        images = []
        for i in range(size):
            images.append([0] * rows * cols)
        for i in range(size):
            img = np.array(image_data[i * rows * cols:(i + 1) * rows * cols])
            img = img.reshape(28, 28)
            images[i][:] = img

        return images, labels

    def load_data(self):
        x_train, y_train = self.read_images_labels(self.training_images_filepath,
        ↪  self.training_labels_filepath)
        x_test, y_test = self.read_images_labels(self.test_images_filepath,
        ↪  self.test_labels_filepath)
        return (x_train, y_train),(x_test, y_test)
```

Kodestykke 1: Python kode til indlæsning af MNIST datasættet

activations.py

```python
import numpy as np

def sigmoid_activation(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return np.exp(-x) / (1 + np.exp(-x))**2

def sigmoid_derivative_from_sigmoid_output(sigmoid_output):
    # a faster way to calculate sigmoid_derivative
    return sigmoid_output * (1 - sigmoid_output)

def softmax(x):
    if x.ndim == 1:
        exp_x = np.exp(x - np.max(x))
        return exp_x / np.sum(exp_x)
    else:
        exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)
```

Kodestykke 2: Python kode til aktiveringsfunktioner

train.py

```
import numpy as np
from mnist_dataloader import MnistDataloader  # Added import statement
from activations import sigmoid_activation, sigmoid_derivative,
↪  sigmoid_derivative_from_sigmoid_output, softmax
import time

print("Loading MNIST data...")
mnist_dataloader = MnistDataloader(
        'mnist/train-images.idx3-ubyte', 'mnist/train-labels.idx1-ubyte',
        'mnist/t10k-images.idx3-ubyte', 'mnist/t10k-labels.idx1-ubyte'
    )
(x_train, y_train), (x_test, y_test) = mnist_dataloader.load_data()
print("MNIST data loaded.")

x_train = np.array(x_train).reshape(-1, 28*28) / 255.0
y_train_one_hot = np.zeros((len(y_train), 10))
y_train_one_hot[np.arange(len(y_train)), y_train] = 1

x_test = np.array(x_test).reshape(-1, 28*28) / 255.0
y_test_one_hot = np.zeros((len(y_test), 10))
y_test_one_hot[np.arange(len(y_test)), y_test] = 1

class NeuralNetwork:
    def __init__(self):
        self.noNodes = [16, 16]
        self.nI = x_train.shape[1]
        self.nO = y_train_one_hot.shape[1]
        self.W1 = np.random.uniform(-1, 1, (self.nI, self.noNodes[0]))
        self.b1 = np.random.uniform(-1, 1, self.noNodes[0])
        self.W2 = np.random.uniform(-1, 1, (self.noNodes[0], self.noNodes[1]))
        self.b2 = np.random.uniform(-1, 1, self.noNodes[1])
        self.W3 = np.random.uniform(-1, 1, (self.noNodes[1], self.nO))
        self.b3 = np.random.uniform(-1, 1, self.nO)

    def predict(self, X):
        Z1 = X @ self.W1 + self.b1
        A1 = sigmoid_activation(Z1)
        Z2 = A1 @ self.W2 + self.b2
        A2 = sigmoid_activation(Z2)
        Z3 = A2 @ self.W3 + self.b3
        A3 = softmax(Z3)
        return {'A1': A1, 'A2': A2, 'A3': A3, 'Z1': Z1, 'Z2': Z2, 'Z3': Z3}

    def update(self, X, Y, pred, rate=1):
        m = X.shape[0]
        A1, A2, A3 = pred['A1'], pred['A2'], pred['A3']
        dZ3 = A3 - Y
        dW3 = (A2.T @ dZ3) / m
        db3 = np.sum(dZ3, axis=0) / m
        dZ2 = (dZ3 @ self.W3.T) * sigmoid_derivative_from_sigmoid_output(A2)
        dW2 = (A1.T @ dZ2) / m
        db2 = np.sum(dZ2, axis=0) / m
        dZ1 = (dZ2 @ self.W2.T) * sigmoid_derivative_from_sigmoid_output(A1)
        dW1 = (X.T @ dZ1) / m
```

```
        db1 = np.sum(dZ1, axis=0) / m
        self.W1 -= rate * dW1
        self.b1 -= rate * db1
        self.W2 -= rate * dW2
        self.b2 -= rate * db2
        self.W3 -= rate * dW3
        self.b3 -= rate * db3
        return {'W1': dW1, 'b1': db1, 'W2': dW2, 'b2': db2, 'W3': dW3, 'b3': db3}

    def train(self, X, Y, epochs):
        idx = np.random.permutation(X.shape[0])
        X, Y = X[idx], Y[idx]

        X_batches = np.array_split(X, X.shape[0] // 1000)
        Y_batches = np.array_split(Y, Y.shape[0] // 1000)
        for epoch in range(1, epochs + 1):
            for X_batch, Y_batch in zip(X_batches, Y_batches):
                pred = self.predict(X_batch)
                self.update(X_batch, Y_batch, pred, rate=1)
            if epoch % 10 == 0:
                loss = -np.mean(np.sum(Y_batch * np.log(pred['A3'] + 1e-8), axis=1))
                accuracy = np.mean(np.argmax(pred['A3'], axis=1) == np.argmax(Y_batch, axis=1))
                test_pred = self.predict(x_test)
                test_accuracy = np.mean(np.argmax(test_pred['A3'], axis=1) == y_test)
                print(f"Epoch {epoch} Loss: {loss:.4f}, Accuracy: {accuracy * 100:.2f}%, Test
                ↪  Accuracy: {test_accuracy * 100:.2f}%")
            if epoch % 100 == 0:
                self.export_to_file("model.npz")

    def export_to_file(self, filename):
        np.savez(filename, W1=self.W1, b1=self.b1, W2=self.W2, b2=self.b2, W3=self.W3,
        ↪  b3=self.b3)

    def gradient_check(self, X, Y, epsilon=1e-4):
        start_time = time.time()

        # gradient med backpropagation
        pred = self.predict(X)
        grads = self.update(X, Y, pred, rate=0) # uden at opdatere vægtene

        # gradient med finite difference
        params = ['W1', 'b1', 'W2', 'b2', 'W3', 'b3']
        grad_approx = {}
        original_params = {}
        for param in params:
            theta = getattr(self, param)
            grad_approx[param] = np.zeros_like(theta)
            original_params[param] = np.copy(theta)
            it = np.nditer(theta, flags=['multi_index'], op_flags=['readwrite'])
            while not it.finished:
                idx = it.multi_index
                theta_plus = np.copy(theta)
                theta_minus = np.copy(theta)
                theta_plus[idx] += epsilon
                theta_minus[idx] -= epsilon

```

```
                setattr(self, param, theta_plus)
                J_plus = self.loss_function(X, Y)

                setattr(self, param, theta_minus)
                J_minus = self.loss_function(X, Y)

                grad_approx[param][idx] = (J_plus - J_minus) / (2 * epsilon)
                setattr(self, param, theta)  # sæt parameteren tilbage til original værdi før næste iteration
                it.iternext()
            setattr(self, param, original_params[param])  # sæt parameteren tilbage til original værdi

        # Compare gradients
        for param in params:
            grads_diff = np.abs(grads[param] - grad_approx[param])
            grads_sum = np.abs(grads[param]) + np.abs(grad_approx[param]) + 1e-8
            relative_difference = grads_diff / grads_sum
            print(f"Gradient check for {param}: min = {np.min(relative_difference):.10f}, max = {np.max(relative_difference):.10f}, mean = {np.mean(relative_difference):.10f}")


        end_time = time.time()
        print(f"Gradient check execution time: {end_time - start_time:.4f} seconds")

    def loss_function(self, X, Y):
        pred = self.predict(X)
        A3 = pred['A3']
        cost = -np.mean(np.sum(Y * np.log(A3 + 1e-8), axis=1))
        return cost

nn = NeuralNetwork()
nn.gradient_check(x_train[:10], y_train_one_hot[:10])  # Test the gradient before training
nn.train(x_train, y_train_one_hot, epochs=10000)
```

Kodestykke 3: Python kode til træning af neuralt netværk

run.py

```
from array import array
import pygame
import sys
import numpy as np
from mnist_dataloader import MnistDataloader  # Added import statement
from activations import sigmoid_activation, softmax

pygame.init()
WIDTH, HEIGHT = 800, 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Drawing Pad")

BUTTON_COLOR = (70, 70, 70)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)


print("Loading MNIST data...")
mnist_dataloader = MnistDataloader(
        'mnist/train-images.idx3-ubyte', 'mnist/train-labels.idx1-ubyte',
        'mnist/t10k-images.idx3-ubyte', 'mnist/t10k-labels.idx1-ubyte'
    )
(x_train, y_train), _ = mnist_dataloader.load_data()
print("MNIST data loaded.")

class DrawingPad:
    def __init__(self):
        self.dx = 28
        self.dy = 28
        self.last_pixel_x = None
        self.last_pixel_y = None
        self.pixel_size = min(WIDTH // self.dx, HEIGHT // self.dy)
        self.pixel_values = [
            [0 for _ in range(self.dy)] for _ in range(self.dx)]

    def handle_event(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN:
            x, y = event.pos
            grid_x = x // self.pixel_size
            grid_y = y // self.pixel_size
            if (self.last_pixel_x == grid_x and self.last_pixel_y == grid_y):
                return
            if 0 <= grid_x < self.dx and 0 <= grid_y < self.dy:
                self.last_pixel_x = grid_x
                self.last_pixel_y = grid_y
                for i in range(-1, 2):
                    for j in range(-1, 2):
                        new_x = grid_x + i
                        new_y = grid_y + j
                        if 0 <= new_x < self.dx and 0 <= new_y < self.dy:
                            distance = (
                                (new_x - grid_x) ** 2 + (new_y - grid_y) ** 2
                            ) ** 0.5
                            newPixelValue = min(
```

```
                                255,
                                self.pixel_values[new_x][new_y]
                                + int(255 * (1 - distance / 3)),
                            )
                            if newPixelValue > self.pixel_values[new_x][new_y]:
                                self.pixel_values[new_x][new_y] = newPixelValue

    def draw(self):
        for i in range(self.dx):
            for j in range(self.dy):
                color_value = self.pixel_values[i][j]
                rect = pygame.Rect(
                    i * self.pixel_size,
                    j * self.pixel_size,
                    self.pixel_size,
                    self.pixel_size,
                )
                pygame.draw.rect(
                    screen, (color_value, color_value, color_value), rect)
                pygame.draw.rect(screen, (50, 50, 50), rect, 1)

    def clear(self):
        self.pixel_values = [
            [0 for _ in range(self.dy)] for _ in range(self.dx)
        ]

    def set_image(self, image):
        self.pixel_values = np.array(image).T

    def get_image(self):
        pixel_values = self.pixel_values
        image = np.array(pixel_values).T
        return image

def predict(image, W1, b1, W2, b2, W3, b3):
    x = np.array(image).reshape(-1) / 255.0
    Z1 = x @ W1 + b1
    A1 = sigmoid_activation(Z1)
    Z2 = A1 @ W2 + b2
    A2 = sigmoid_activation(Z2)
    Z3 = A2 @ W3 + b3
    A3 = softmax(Z3)
    return np.argmax(A3), A3


def main():
    model = None
    W1, b1, W2, b2, W3, b3 = None, None, None, None, None, None
    try:
        model = np.load("model.npz")
        W1, b1 = model['W1'], model['b1']
        W2, b2 = model['W2'], model['b2']
        W3, b3 = model['W3'], model['b3']
    except FileNotFoundError:
        print("Model file not found. Please run train.py to train the model.")
        sys.exit(1)
```

```
111
112     screen.fill(BLACK)
113     drawing_pad = DrawingPad()
114     clear_button = pygame.Rect(10, HEIGHT - 60, 100, 50)
115     font = pygame.font.Font(None, 36)
116     button_text = font.render("Clear", True, WHITE)
117     predicted_digit = None
118     probabilities = None
119
120     drawing_pad.set_image(x_train[1337]) # Set an example image from the training set, in this
        ↪  case the image is a 6
121     print("Label of the image:", y_train[1337])
122
123     while True:
124         for event in pygame.event.get():
125             if event.type == pygame.QUIT:
126                 pygame.quit()
127                 sys.exit()
128             elif event.type == pygame.MOUSEBUTTONDOWN and clear_button.collidepoint(
129                 event.pos
130             ):
131                 drawing_pad.clear()
132
133         image = drawing_pad.get_image()
134         predicted_digit, probabilities = predict(image, W1, b1, W2, b2, W3, b3)
135         print("Predicted digit:", predicted_digit)
136
137         mouse_clicked = pygame.mouse.get_pressed()[0]
138         if mouse_clicked:
139             drawing_pad.handle_event(
140                 pygame.event.Event(
141                     pygame.MOUSEBUTTONDOWN, {"pos": pygame.mouse.get_pos()}
142                 )
143             )
144         screen.fill(BLACK)
145         drawing_pad.draw()
146         pygame.draw.rect(screen, BUTTON_COLOR, clear_button)
147         screen.blit(button_text, (clear_button.x + 10, clear_button.y + 10))
148         if predicted_digit is not None:
149             prediction_text = font.render(f"Predicted: {predicted_digit}", True, WHITE)
150             screen.blit(prediction_text, (WIDTH - 200, 10))
151             for i, prob in enumerate(probabilities):
152                 prob_text = font.render(f"{i}: {prob * 100:.2f}%", True, WHITE)
153                 screen.blit(prob_text, (WIDTH - 200, 40 + i * 30))
154
155         pygame.display.flip()
156
157
158 if __name__ == "__main__":
159     main()
```

Kodestykke 4: Python kode til kørsel af neuralt netværk