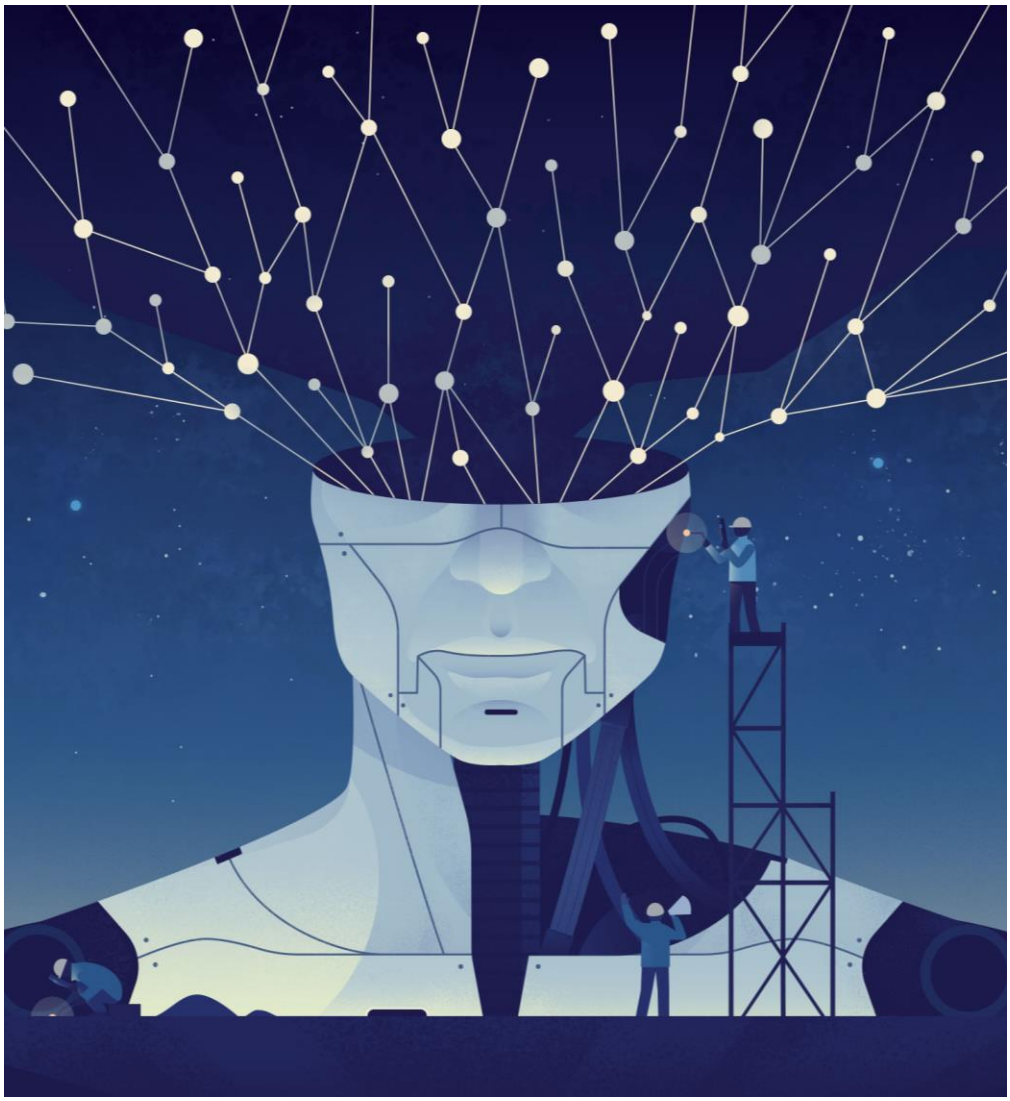


Neural Netværk

PROGRAMMERING B OG MATEMATIK A

Oliver Harboe | 3.n | 20/12-2024

Vejledere: Niels Henning Dahl og Henrik Sterner

Resume

Denne opgave udforsker den underliggende matematiske teori, der indgår i neurale netværk, herunder matrixoperationer og partielle afledte funktioner, som er fundamentet for, hvordan netværket lærer og opdaterer sine vægte gennem backpropagation. Der er også blevet undersøgt, om det er muligt at bygge et neuralt netværk fra bunden, hvor teori fra både matematik og programmering indgår. Programmeringsteorien omfatter objektorienteret programmering og computational thinking, som gør det muligt at strukturere og implementere netværkets funktionalitet effektivt. Til sidst udvikles der et fungerende neuralt netværk, som benytter teorien til at klassificere data. Dette netværk demonstrerer, hvordan teori og programmering kan kombineres for at skabe en praktisk løsning på klassificeringsproblemer.

Indhold

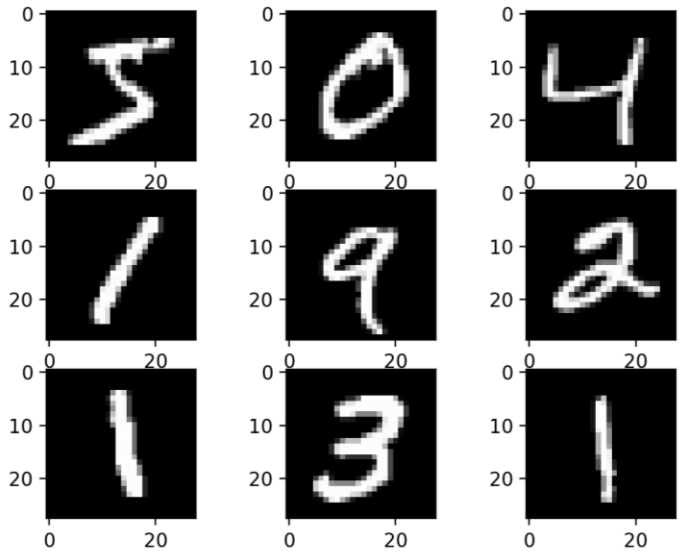
Indledning

Neurale netværk er en gren af maskinlæring, som har haft en markant betydning for udviklingen af kunstig intelligens. Arkitekturen er inspireret af, hvordan den menneskelige hjerne fungerer, ved at efterligne hjernens netværk af neuroner, som kan lave en neuronfyring ud fra input og på den måde behandle data. Det første trænede neurale netværk inden for maskinlæring kan spores tilbage til 1957, hvor Frank Rosenblatt i artiklen *The Perceptron* [1] frem viser en tidlig udgave af den type neurale netværk, som bliver anvendt i dag. Maskinlæringsarkitekturen har først fået den store opblomstring inden for de seneste år, hvilket skyldes udviklingen inden for computerkraft, som muliggør milliader af udregninger. Neurale netværks grundidéer bliver stadig benyttet i dag og er fundamentet for alt fra chatbots og selvkørende biler til oversættelsesprogrammer som Google Translate. Trenden i brugen af neurale netværk ser ikke ud til at stoppe. For at kunne fortsætte udviklingen inden for maskinlæring er det derfor vigtigt at have viden om de centrale idéer og funktioner, som et neuralt netværk bygger på.

For at kunne forstå neurale netværk skal man tage et skridt tilbage og se på maskinlæring. Maskinlæring er en metode til at designe modeller ved at give en algoritme muligheden for at lære eller forbedre sig ud fra data, uden at have decideret programmeret algoritmen til en bestemt opgave. Overordnet kan man opdele de teknikker, der bruges til at træne algoritmen, i tre hovedkategorier: forstærkende læring, usuperviseret læring og superviseret læring. I denne opgave arbejdes der kun med superviseret læring, som også er den mest anvendte metode til at træne modeller.

Tanken om, hvordan en maskine kan lære, kan virke indviklet. For at hjælpe med forståelsen, kan man tænke på, hvordan man som menneske ville lære en færdighed. Man kunne forestille sig, at man er en tennisspiller, som gerne vil lære at slå en god forhånd. Hvis man virkelig vil være sikker på at blive god, ville man søge træning hos en, der allerede har forstand på, hvordan man laver et godt forhåndsslag. Herefter ville man gentagne gange slå et forhåndsslag og få feedback fra træneren om, hvordan man kan forbedre sig. Til at starte med vil man måske ikke engang ramme banen, men efter mange gentagelser vil man kunne se forbedring. Feedbackmetoden er tankegangen bag superviseret læring. Ved superviseret læring har vi noget data, som er inputtet til vores model. For at sammenligne dette med vores tennisanalog kan dette være, hvad vi ser med vores øjne og hvor på banen vi står. Vi har også et output, som er resultatet, vores model kommer frem til. Dette kunne være det slag, vi slår. Det kan være svært at vide, om det slag man laver, faktisk er godt. Det samme problem opstår indenfor maskinlæring, og på samme måde som man har en træner, der retter på de slag, man laver, bruger man labelled data, som er det korrekte svar til at rette på modellen. På denne måde kan man sammenligne det svar, modellen giver, med det rigtige svar. For at samle op: superviseret lærings centrale pointe er, at man har noget data og et svar til dataen. Man giver modellen noget data og sammenligner modellens output med det rigtige svar. Man retter på

modellen, så forskellen mellem modellens output og det rigtige svar bliver så lille som muligt. Formålet med modellen er, at den vil være i stand til at generalisere data, hvilket betyder, at den vil kunne komme med det korrekte svar på data, den ikke har set før. Tænk på, hvis man kun har trænet at slå forhånden under specifikke forhold, på præcist samme sted, hvor bolden kommer på samme måde, og man slår det samme sted hen hver gang. Hvis tennisspilleren får en bold, som ikke er præcis som de andre, vil spilleren have meget svært ved at slå et godt slag, da omstændighederne ikke har været en del af træningen. Samme problem kan opstå for en maskine, der udfører en opgave. Hvis modellen er meget god til at udføre en opgave på data, den er trænet på, men er meget dårlig på eksempler, den ikke har set, er modellen overfittet [2]. En anden årsag til et dårligt slag fra tennisspilleren kunne være, at spilleren simpelthen ikke har trænet nok, og derfor ikke kan lave et godt slag. Dette kaldes underfitting. Underfitting kan også være en konsekvens af en for simpel model, hvilket betyder, at man ikke har mulighed for at lave små justeringer, der kan opfange en mere kompleks struktur i datasættet.



Figur 1 MNIST dataeksempler

Der vil i denne opgave blive lagt fokus på at udforske teorien bag neurale netværk. Dette indebærer både den matematiske teori og hvordan man kan inddrage den matematiske teori ved hjælp af programmering. Opgavens fokus er ikke på at forsøge at fremstille den bedste model til opgaven, men derimod på at undersøge neurale netværk som en model for en opgave, hvor brugen af modellen tidligere har vist fornuftige resultater. Der vil blive set på et klassificeringsproblem, som går ud på at tildele data til en foruddefineret kategori. I opgaven vil der blive set på et specifikt eksempel, hvor modellen bliver trænet på MNIST-datasættet. MNIST-datasættet (Modified National Institute of Standards and Technology) er en stor samling af håndskrevne tal fra nul til ni. De håndskrevne tal bliver repræsenteret ved et 28 x 28 gråtonebillede, hvor hver pixel kan være mellem 0 og 255, hvor 255 er hvid og 0 er sort. I opgaven vil der blive brugt en del af det fulde datasæt, som indeholder 42.000 dataeksempler [3].

Matematisk Analyse af Neurale Netværk

Et neuralt netværk består, som navnet antyder, af et netværk af neuroner, der sender information til hinanden for at træffe beslutninger. Men hvad er et neuron, og hvordan er de koblet sammen til et netværk?

Neuroner som funktioner

Man kan tænke på et neuron som en matematisk funktion, der modtager et input, bearbejder det og giver et output. I den mest simple form kan et neuron beskrives som den simple lineære funktion, man kender fra matematik.

$$z(x) = a \cdot x + b$$

Netværket bliver skabt ved at bruge outputtet fra et neuron som inputtet til et andet neuron. Et neuron er derfor en sammensat funktion af tidligere neuroner. Lige nu har vores neuron kun én variabel, men i virkeligheden har det flere variabler, hvilket gør det muligt for netværket at finde frem til komplekse sammenhænge i datasættet. Vi kan udvide formlen for neuronet, så den kan indeholde flere variabler:

$$z(x) = \left(\sum_{i=1}^{n} w_i \cdot x_i\right) + b$$

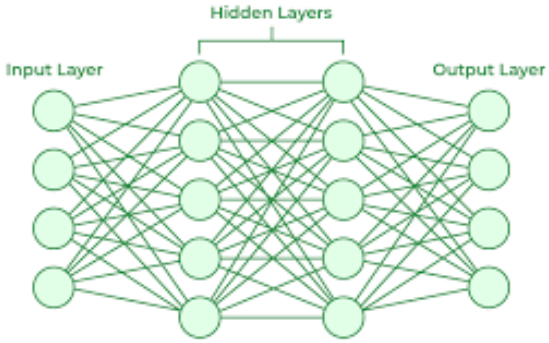
Her er x_i inputtet, som kunne være fra et tidligere neuron, w_i er en koefficient, der bestemmer, hvor meget inputtet skal vægtes i outputtet, og b er en konstant, der hjælper med at justere resultatet i funktionen. Vi kan ved at justere på w og b ændre på outputtet af funktionen. Resultatet af denne lineære kombination bliver herefter sendt gennem aktiveringsfunktionen, som giver neuronet mulighed for at lære avancerede mønstre og relationer. Ikke-linearitet i aktiveringsfunktionen er afgørende, fordi den gør det muligt for netværket at modellere komplekse mønstre.

$$a(x) = \sigma\left(\left(\sum_{i=1}^{n} w_i \cdot x_i\right) + b\right)$$

Netværks lag

Et netværk består af forskellige lag, som arbejder sammen for at bearbejde data. Man kan opdele disse lag i tre typer: inputlaget, de skjulte lag og outputlaget. Inputlaget er det første lag, hvor dataene bliver introduceret for netværket. Inputlaget er ikke et ”neuralt lag”, hvor vi laver operationer på dataene; i stedet sender inputlaget det videre til de næste neuroner. Man kan i stedet opfatte neuronerne i dette lag som en enhed, der holder på en talværdi. Hvis vi skulle designe et neuralt netværk til MNIST-datasættet, ville inputlaget have et neuron for hver pixel i billedet. Da billedet er 28x28 pixels, ville det betyde, at

netværket starter med 784 neuroner, hvor hvert neuron har en værdi mellem 0 og 255, hvor 0 er sort og 255 er hvid. Disse værdier ville blive sendt videre til det næste lag.



Figur 2 eksempel på et neural netværk

De lag, som ikke er det første eller sidste lag i netværket, kaldes skjulte lag. Skjulte lag kaldes 'skjulte', fordi deres indre processer ikke er direkte synlige for den, der bruger netværket. Selvom de spiller en central rolle i læring og beslutningstagning i neurale netværk, er deres funktioner kun synlige gennem vægtene og aktiveringerne, der bliver opdateret under træning. Skjulte lag er de lag, der er ansvarlige for at bearbejde og lære komplekse mønstre i dataene.

I modsætning til inputlaget har de skjulte lag ikke et fast krav om antallet af neuroner, der skal indgå i laget. Dette er noget, som udvikleren, der modellerer netværket, selv kan bestemme. I et neuralt netværk er lagene forbundet på den måde, at hvert neuron i et lag modtager outputtet fra alle neuronerne i det forrige lag som input. Hver forbindelse mellem neuronerne har en vægt, der bestemmer, hvor meget indflydelse en neurons output har på inputtet til et andet neuron. Når et neuron modtager input fra et tidligere lag, beregner det en vægtet sum af alle inputtene og anvender en aktiveringsfunktion på resultatet for at bestemme dets eget output.

I inputlaget modtager neuronerne de rå data og sender dem videre til de første skjulte lag, hvor hvert neuron i det skjulte lag modtager input fra alle neuronerne i inputlaget. På samme måde modtager hvert neuron i de efterfølgende skjulte lag input fra alle neuronerne i det forrige lag. Når informationen når outputlaget, modtager hvert neuron i dette lag input fra alle neuronerne i det sidste skjulte lag og producerer netværkets endelige output.

Outputlaget er det sidste lag i modellen og giver den endelige forudsigelse. Outputlagets funktion er at transformere det arbejde, modellen har lavet i de skjulte lag, til en form, som kan forstås. En typisk struktur for outputlaget afhænger af den specifikke opgave, netværket er trænet på. I en klassifikationsopgave, som MNIST-opgaven, ville outputlaget bestå af 10 neuroner, som repræsenterer tallene fra 0 til 9. Outputlaget ville repræsentere en sandsynlighed for, hvilket tal der er blevet givet som input. Man kan derfor finde den endelige forudsigelse ved at tage den største værdi blandt neuronerne.

I dybden med Aktiverings funktioner

I opgaven bliver der benyttet to forskellige slags aktiveringsfunktioner, som er ReLU og SoftMax. Begge funktioner har hver deres formål i netværket. ReLU anvendes i de skjulte

lag for at hjælpe netværket med at lære ikke-lineære mønstre. ReLU virker ved at tage den største værdi mellem x og 0, hvilket betyder, at hvis inputtet er under nul, vil det blive ændret til 0; ellers er outputtet det samme som inputtet. Man kan beskrive funktionen matematik:

$$ReLU(x) = \begin{cases} x & if x > 0, \\ 0 & if\ x \leq 0 \end{cases}$$

Softmax bliver derimod kun brugt i outputlaget til at transformere forudsigelsen til et bestemt interval, som nemt kan forstås. Softmax er en hyppigt anvendt aktiveringsfunktion, som bruges til klassifikationsopgaver med flere kategorier. Outputtet fra softmax-funktionen kan aflæses som sandsynligheden for, at den specifikke kategori er svaret. Formlen for Softmax-funktionen er:

$$Softmax(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

$$z_i = er\ ouputtet\ for\ den\ i'te\ nuron$$
$$k = er\ antalet\ af\ neuroner$$

Inputtet I softmax funktionen er en vektor, som indeholder outputtet fra alle neuroner i outputlaget.

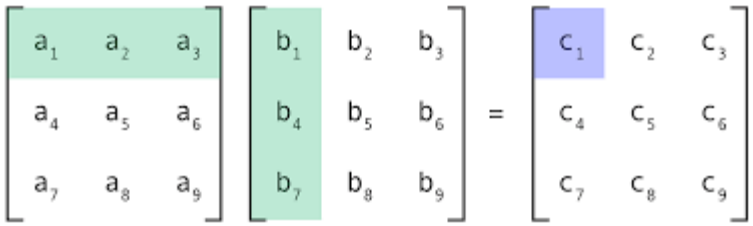
Matrix Multiplikation

Neurale netværk bliver ofte brugt på enorme datasæt, hvilket betyder, at beregningerne, som er nødvendige for at træne netværket, er meget ressourcekrævende. Den neurale netværksstruktur muliggør brugen af matrix- og vektorberegning, som nyere grafikkort er blevet optimeret til at kunne udføre. Ved at repræsentere dataene som en matrice med dimensionerne 784 gange antal dataeksempler, kan vi også repræsentere vægte og bias som matricer og vektorer.

$$W = \begin{bmatrix} w_{0,0} & \cdots & w_{i,0} \\ \vdots & \ddots & \vdots \\ w_{0,j} & \cdots & w_{i,j} \end{bmatrix}$$

$$X = \begin{bmatrix} x_{0,0} & \cdots & x_{784,0} \\ \vdots & \ddots & \vdots \\ x_{0,j} & \cdots & x_{784,j} \end{bmatrix}$$

I stedet for at bruge den traditionelle tilgang til at udregne outputtet for et neuron, hvor man tager summen af produktet af to tal, kan der anvendes matrixmultiplikation Resultatet af matrixmultiplikationen er en ny matrix, hvor elementet c_{ij} er prikproduktet af



Figur 3 matrix multiplikations eksempel

matrix a's række og matrix b's kolonne. Kravet til matrixmultiplikation er, at rækkerne i matrix a skal have samme dimension som kolonnerne i matrix b [4].

Prikproduktet for flere dimensioner bliver udregnet således:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$$

I stedet for at udregne outputtet for hvert neuron i separate udregninger, kan der anvendes matrixmultiplikation. Neuronernes output i et lag kan skrives med matrixnotation:

$$a(x) = ReLU(W \cdot x + b)$$

Her er W vægtene i laget, x inputtet til alle neuroner i laget, og b bias for alle neuroner i laget.

Der er tilfælde, hvor dimensionerne i matrix a og b ikke passer. Her bruges matrixtransformation, som bytter rundt på rækker og kolonner. Denne transformation noteres med et opløftet T, som står for transpose [5].

Figur 4 transpose eksempel

Backprobagation

Der er tidligere blevet forklaret, hvordan et netværk kan behandle data fra inputlaget til outputlaget og på den måde finde outputtet fra netværket. Backpropagation er en grundlæggende metode, der bruges til at træne neurale netværk. Træningen sker ved at præsentere et dataeksempel, som behandles af netværket, hvor det endelige resultat opnås i outputlaget. For at kunne ændre på netværket, så det klarer sig bedre, er det vigtigt at vide, hvilke ændringer netværket skal lave. Ligesom ved tennis er det meget svært at blive bedre, hvis man ikke ved, hvad man gør forkert. "Træneren" for det neurale netværk er loss-funktionen [6], Loss-funktionen sammenligner det rigtige svar med netværkets output. Ud fra dette bliver vægtene i netværket justeret. Vi vil gerne justere netværket mere, hvis netværket laver en forudsigelse, der er meget langt fra det rigtige svar. Den loss funktion, som ofte bliver anvendt i multi klassifikation er kategorisk Cross Entropy [7]. funktionen er ofte anvendt, da den er fremragende når outputtet er mellem 0 og 1, til at straffe dårlige forudsigelse fra modellen. Formlen for kategorisk Cross Entropy er [8]:

$$J(y, \hat{y}) = -\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i)$$

Her er n *antallet af neuroner i outputlaget, y er det korrekte svar og ŷ er modellens output*

Målet med træningen er at mindske tabet, hvilket er outputtet af funktionen. Ved et perfekt netværk ville tabet være 0, da det ville forudsige svaret korrekt 100% af tiden. Ved at bruge den afledte funktion af loss-funktionen i forhold til vægtene i netværket, kan man finde ud af, hvordan de enkelte vægte skal justeres. Matematisk set handler dette om partiell differentiation, da den funktion, der skal differentieres, har flere variable. Når man laver partiell differentiation, differentierer man ud fra én variabel og opfatter de andre som konstanter. I et neuralt netværk kan vores tabsfunktion afhænge af mange variable. For eksempel, hvis et netværk har 100 vægte og bias, er vores tabsfunktion en funktion med 100 variable [9].

Som tidligere nævnt er neuroner koblet sammen, hvilket skaber en sammensat funktion. Dette er afgørende, når der skal differentieres. Når man differentierer sammensatte funktioner, vil der blive gjort brug af kædereglen. Når vi differentierer gennem et komplekst neuralt netværk, skal vi hele tiden kæde delresultaterne sammen. For hver vægt beregner vi, hvor meget den bidrager til det samlede tab. Et eksempel på kædereglen med tre sammensatte funktioner ville være

$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dq} \cdot \frac{dq}{dx}$$

Når vi bruger dette i sammenhæng med et neuralt netværk vil notationen, for at finde en vægts betydning på tabet i det næst sidst lag i netværket være:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

J er tabs funktionen, a er outputtet af neutronen efter aktiveringsfunktionen, z er neuron før aktiverings funktion og w er vægten.

$$z(w) = wx + b\ ,\ z'(w) = x$$
$$z(b) = wx + b\ , z'(b) = 1$$

Ved brug af Kategorisk Cross Entropy sammen med softmax funktionen i outputlaget bliver gradienten af loss funktionen med hensyn til outputtet af neuronerne i det sidste lag rigtig simpel [8]:

$$\frac{\partial J}{\partial z} = a - y$$

Formlen er udledt ved at simplificere kategorisk cross entropy. Eftersom der kun er en af de 10 neuroner, som skal være aktiveret, betyder det, at labelvektoren kun har et element, som ikke er nul. Tabet kan derfor skrives som:

$$\frac{\partial J}{\partial a} = -\log(\widehat{y_c})$$

Her er c for rigtig eller correct

Ved at indsætte softmax funktionen for den forudsete værdi bliver det til:

$$\frac{\partial J}{\partial a} = -\log\left(\frac{e^{z_c}}{\sum_{j=1}^{k} e^{z_j}}\right)$$

ved at opdele logaritme udtrykke ved hjælp af regne reglen [10]

$$\frac{\partial J}{\partial a} = -\left(\log(e^{z_c}) - \log\left(\sum_{j=1}^{k} e^{z_j}\right)\right) = -z_c + \log\left(\sum_{j=1}^{k} e^{z_j}\right)$$

Finder den afledte funktion med hensyn på z:

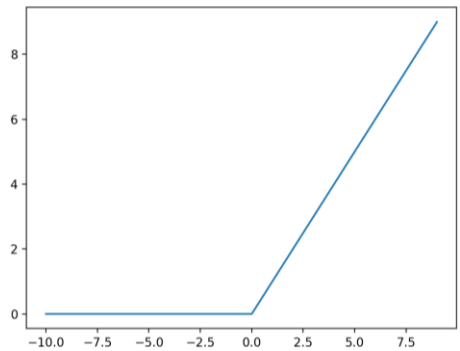
$$\frac{\partial J}{\partial z_c} = -1 + \frac{e^{z_c}}{\sum_{j=1}^{k} e^{z_j}} = \frac{e^{z_c}}{\sum_{j=1}^{k} e^{z_j}} - 1$$

Dette er også outputtet fra softmax funktionen minus labelled

$$\frac{\partial J}{\partial z} = a - y$$

Når der differentieres, skal der tages højde for ReLU-funktionen, da der optræder et "knæk" i grafen, hvilket gør, at den matematisk ikke er differentiérbar i punktet (0,0). Dette skyldes, at hældningen skifter fra 0 til 1 ved dette punkt, hvilket betyder, at den afledte funktion af ReLU ikke er veldefineret. I praksis bruges ReLUs afledte funktion som en simpel stykvis funktion:

$$ReLU' = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$



Figur 5 ReLU funktion

Matematisk ville et neuralt netværk som har et skjult lag, med ReLU funktion og har softmax funktionen som aktiverings funktion på outputlaget, ville se således ud.

Fremad:

$$Z^{[1]} = W^{[1]} \cdot X + b^{[1]}$$
$$A^{[1]} = ReLU\left(Z^{[1]}\right)$$
$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + b$$
$$A^{[2]} = Softmax(Z^{[2]})$$

Back propagation:

$$dZ^{[2]} = A^{[2]} - Y^T$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]}$$
$$dB^{[2]} = \frac{1}{m} \sum dZ^{[2]} A^{[1]}$$
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} \cdot ReLU'\left(Z^{[1]}\right)$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]}$$
$$dB^{[1]} = \frac{1}{m} \sum dZ^{[1]}$$

opdatering af parameter:

$$W^{[2]} := W^{[2]} - \alpha \cdot dW^{[2]}$$
$$b^{[2]} := b^{[2]} - \alpha \cdot db^{[2]}$$

$$W^{[1]} := W^{[1]} - \alpha \cdot dW^{[1]}$$

$$b^{[1]} := b^{[1]} - \alpha \cdot db^{[1]}$$

Laget bliver repræsenteret ved [i] og m er antal trænings eksempler

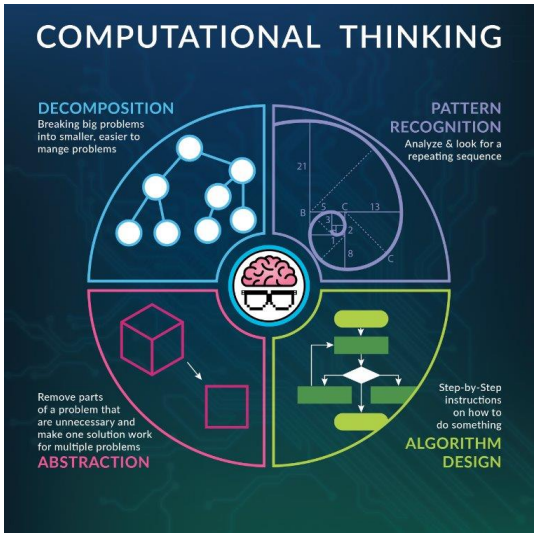
Efter der er blevet fundet ud af hvor meget hver vægt skal ændres bliver der brugt en optimerings funktion, som står for at ændre vægtende til en ny værdi. I opgaven bliver der brugt en optimerings funktion ved navn gradient decent, som ser således ud:

$$w^{t+1} = w - \alpha \cdot \nabla \mathrm{J}(w)$$

For at ændringerne ikke bliver for store tilføjer man Alpha, som bliver kaldt for learning raten. Derefter minusser du produktet af gradienten matricen og Alpha med den tidligere vægt, hvilket giver en forbedret vægt.

Computational Thinking og design valg

Når man designer neurale netværk fra bunden, spiller computational tænkning en afgørende rolle i at strukturere og håndtere kompleksiteten af opgaven. Denne tilgang til problemløsning giver os et systematisk framework for at nedbryde, analysere og implementere de mange komponenter, der udgør et neuralt netværk. At designe et neuralt netværk fra bunden er en kompleks opgave, hvor struktur i programmet er alt afgørende for at holde styr på programmets komponenter og hvilken rolle de spiller. Computational thinking omhandler 4 områder, som er blevet inddraget i konstruktionen af det neurale netværk. [11].



Figur 6 computational thinking billed

Decomposition er et af områderne i computational thinking. Ved at anvende decomposition nedbryder vi det komplekse problem i mindre, mere overskuelige opgaver, som hver især kan tackles individuelt. For et neuralt netværk som program betyder det forarbejdning af data, netværksdesign, efterbehandling af data. Hver af disse komponenter kan yderligere nedbrydes i mindre delopgaver. For eksempel kan netværket deles op i flere delopgaver som inputlaget, skjulte lag, outputlag, men også funktioner som backpropagation og implementering af aktiveringsfunktionerne, der bliver anvendt i netværket. Denne systematiske nedbrydning gør det muligt at fokusere på én komponent ad gangen og sikre, at hver del fungerer korrekt, før de integreres i den større helhed.

Mønstergenkendelse, som er en central del af computational tænkning, hjælper os med at identificere gentagende strukturer og processer i netværket. For eksempel kan vi se, hvordan beregninger gentages gennem forskellige lag, eller hvordan gradientberegninger følger bestemte mønstre. Denne indsigt fører til mere effektiv kode og bedre forståelse af netværkets opførsel.

Ved at abstrahere komplekse detaljer kan vi fokusere på de væsentlige aspekter af hver komponent, hvilket gør koden mere vedligeholdelig og fleksibel. Dette er det tredje område inden for computational thinking, hvilket er vigtigt at have i tankerne. Inden for maskinlæring findes der utallige måder at optimere en model på, ved at tilføje alt fra omfattende dataforbehandling til brugen af eksotiske arkitekturer og optimeringsteknikker. Men ved hjælp af abstraktion kan man fokusere på specifikke løsninger omkring det grundlæggende bag et neuralt netværk og undgå at blive distraheret af de mange alternative muligheder. Dette sikrer, at ressourcerne koncentreres om at forfine den valgte metode og opnå optimale resultater.

Den algoritmiske tænkning, der er kernen i den computational tilgang, er generelt vigtig, når man programmerer, da uklare instruktioner kan medføre forkerte svar, som ikke bliver taget højde for. Dette er et kæmpe problem i et neuralt netværk, da der kan opstå en stor mængde gentagende operationer, hvis svarene ikke intuitivt kan valideres af et menneske. Dette kan i værste fald give fejl i programmet. Det er derfor vigtigt, når vi skal implementere træningsalgoritmer og optimere netværkets ydeevne. Det hjælper os med at designe komponenterne i programmet, såsom forward og backpropagation.

Den matematiske formalisering, der følger med computational tænkning, hjælper os med at bygge bro mellem den præsenterede teori og en praktisk anvendelse af netværket. Dette er en stor hjælp, da de matematiske formler skal modificeres for at kunne bruges i en programmeringssammenhæng.

Denne strukturerede tilgang muliggør en gradvis udvikling af neurale netværk. Dette hjælper med at verificere, at de enkle dele i programmet virker. Vi kan starte med simple implementeringer, teste hver komponent grundigt og systematisk tilføje kompleksitet efterhånden. Det er denne metodiske fremgangsmåde, der gør det muligt at bygge et robust og effektivt neuralt netværk fra bunden, selv når vi arbejder med komplekse arkitekturer og store datasæt. Strukturen er også til stor hjælp, når programmet ikke opfører sig som forventet.

Programmeringssproget, som programmet bliver skrevet i, er Python. Der er mange grunde til, at Python er et fremragende valg til denne opgave, og en af dem er Pythons enkle og letlæselige syntaks, som sproget er kendt for. Dette gør det nemmere at forstå koden og logikken bag koden. Dette er især værdifuldt indenfor neurale netværk, hvor der arbejdes med komplekse algoritmer. Python er også særdeles god til at give klare og forståelige fejlmeddelelser, når der opstår problemer i koden, hvilket adskiller sig fra mere low-level sprog som C++, hvor der ofte kræves en bedre indsigt i koden. Den enkle syntaks og Pythons opbygning gør, at Pythons er langsommere end sprog som C++, som i modsætning kompileres direkte til maskinkode. Pythons hastighed bliver en udfordring, når det gælder intensive beregninger, som f.eks. træning af neurale netværk. Denne udfordring bliver taklet ved hjælp af biblioteker som NumPy, som er et numerisk bibliotek i Python. Størstedelen af NumPy er skrevet i C, som er et andet sprog, der kan kompileres direkte til maskinkode. NumPy giver adgang til at lave vektor- og matrixberegninger i Python. Python har også andre kraftfulde biblioteker, som Pandas, der bruges til at behandle data. Ved at kombinere Python og numeriske biblioteker kan man beholde Pythons enkle syntaks uden at gå på kompromis med beregningshastigheden.

Indenfor maskinlæring vil man ofte træne flere modeller med samme arkitektur for at optimere ydeevnen. Det kunne være, at man vil sammenligne en model med 5 lag med en model med 2 lag og se, om de ekstra lag var bedre til at løse opgaven. Ved at lave netværket som en klasse kan vi oprette flere objekter, som alle sammen er neurale netværk, men hvor der er forskel på antal neuroner.

Implementering af Neurale Netværk

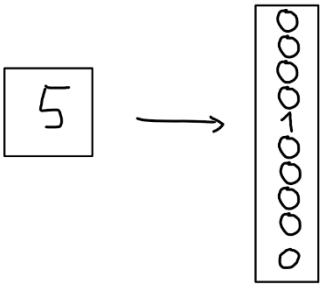
Forbehandling af data

```python
def load_data(path:str) -> tuple[np.ndarray,np.ndarray]:
    '''
    loading data from csv
    '''
    train_df = pd.read_csv(path)

    y = train_df.loc[:,'label']
    X = train_df.drop('label',axis=1)

    return X.to_numpy(),y.to_numpy()
```

Før man kan gå i gang med at implementere netværket, skal man først loade MNIST-datasættet ind. MNIST-datasættet er downloadet fra Kaggle.com igennem en CSV-fil, som er en komma-separeret fil. For at kunne arbejde med dataene bliver pandas-biblioteket i Python benyttet, da det indeholder effektive værktøjer til at manipulere data. Datasættet består af 785 kolonner, hvor den første er label for det håndskrevne tal, og de 784 andre er pixels i billedet. Dataene bliver delt op i labels og inputdata. Udover at opdele dataene i labels og inputdata, vil vi yderligere dele det op i data, som modellen skal trænes på, og data, som skal validere modellen og teste dens evne til at forudsige det rigtige svar. De data, som modellen skal trænes på, er train-data, og de data, modellen skal testes på, er test-data. Datasættet bliver delt op, så 95% af datasættet bliver brugt til at træne modellen, og 5% bliver brugt til at validere den.

```python
def split_data(X:pd.DataFrame,y:pd.DataFrame) -> tuple[np.ndarray,np.ndarray,np.ndarray,np.ndarray]:
    '''
    splits data into train and test
    '''
    X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.05,random_state=0,stratify=y)
    return X_train,y_train,X_test,y_test
```

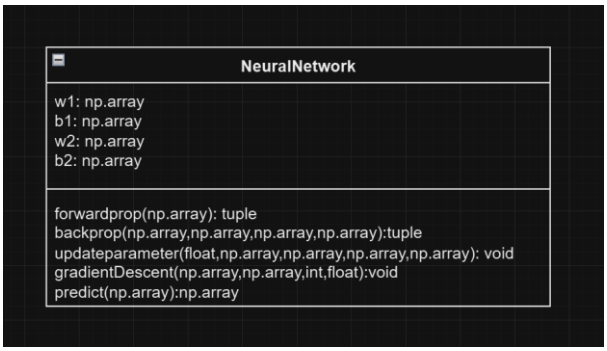
Labelled for dataeksemplerne er et heltal, der repræsenterer det håndskrevne tal i eksemplet. Dette passer ikke med, hvordan vores outputlag er opbygget, og hvordan vores loss-funktion fungerer. Som nævnt tidligere er outputlaget i netværket 10 neuroner, hvor aktiveringsfunktionen er softmax. Det er derfor nødvendigt at transformere heltallet til en one-hot encoded vektor. En one-hot encoded vektor er en vektor, som i dette tilfælde har 10 dimensioner, der repræsenterer de 10 neuroner. Labelled bliver repræsenteret ved at have 1 ved det relevante indeks. Dette gøres ved først at lave en tom matrice med de korrekte dimensioner, hvilket er antal dataeksempler gange 10. Derefter bliver det korrekte indeks fundet, og værdien bliver erstattet med 1. Dette bliver gentaget for alle dataeksempler.



Figur 7 onehot encode eksempel

```python
oneHot = np.zeros((y.shape[0],10))
oneHot[np.arange(y.shape[0]),y] = 1
```

Neurale netværk



Figur 8 klasse diagram

Til det neurale netværk vil der kun blive brugt NumPy, som er et matematikbibliotek, der laver vektor- og matrixoperationer. Til at starte med opretter vi en klasse ved navn NeuralNetwork, som har en constructor, der opretter vægtene og bias i netværket. Netværket har et skjult lag, der består af 16 neuroner. Vægtene og bias tildeles en tilfældig værdi ved hjælp af NumPys randn-funktion, som outputter en tilfældig værdi ud fra en standard normalfordeling, der har et gennemsnit på 0 og en varians på 1.

```python
import numpy as np

Qodo Gen: Options | Test this class
class NeuralNetwork:
    Qodo Gen: Options | Test this method
    def __init__(self,hidden_size) -> None:
        '''
        Initalizere parameters (weights and biases)
        '''
        self.w1 = np.random.randn(hidden_size, 784)
        self.b1 = np.random.randn(hidden_size, 1)
        self.w2 = np.random.randn(10, hidden_size)
        self.b2 = np.random.randn(10, 1)
```

Efter vægtene er blevet initialiseret, laver vi forward propagation, hvor vi opretter lagene i netværket. Her bliver np.dot brugt til at lave matrixmultiplikation. For at kunne udføre matrixmultiplikation korrekt må vi transponere inputdataene. Da vores inputdata har formen 784 gange antal datapunkter, er vi nødt til at transponere matricen for at kunne multiplicere korrekt med vægtmatricen. Resultatet bliver inputtet i ReLU-funktionen. Som det ses i koden spiller NumPys matrixoperationer en vigtig rolle i programmet, eftersom ReLU-funktionen bliver brugt på alle elementerne i inputtet.

```python
def forwardProp(self,X:np.ndarray) -> tuple:
    '''
    Forward propagation
    '''
    z1 = np.dot(self.w1,X.T) + self.b1
    a1 = ReLU(z1)
    z2 = np.dot(self.w2,a1) + self.b2
    a2 = softmax(z2)

    return z1,a1,z2,a2
```

Ved implementeringen af softmax-funktionen opstod der et problem, da formlen kunne medføre, at værdierne blev ekstremt store, hvilket resulterede i, at programmet ikke kunne køre. For at undgå dette problem var det nødvendigt at ændre på den matematiske formel for at sikre numerisk stabilitet. Softmax-funktionen ud fra den matematiske formel ville se således ud:

```python
np.exp(x) / np.sum(np.exp(x), axis=0)
```

Der bliver specificeret, hvilken akse der skal summeres over, hvilket gør, at programmet tillader at køre flere dataeksempler gennem på én gang. Derfor er det vigtigt, at man ikke summerer på den forkerte akse. Man kan forestille sig, hvordan np.sum(np.exp(x)) kan blive meget stor ved tidlige træningsiterationer, da der er 10 neuroner i det sidste lag, og hver af de 10 neuroner har 16 tidligere neuroner, og dette er opløftet i Eulers tal. Ved at subtrahere maksimum før exp() sikrer vi, at værdierne ikke bliver for store. Matematisk ser det således ud:

$$Softmax(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Der bruges potens regne reglen, til at introducere maksimumværdien:

$$x^{a+b} = x^a \cdot x^b$$

$$e^{z_i} = e^{z_i - \max(z)} \cdot e^{\max(z)}$$

Der kan blive gjort det samme i nævneren:

$$\sum_{j=1}^{k} e^{z_j} = e^{\max(z)} \cdot \sum_{j=1}^{k} e^{z_j - \max(z)}$$

Det vil sige at den nye softmax funktion er:

$$Softmax(\vec{z})_i = \frac{e^{z_i - \max(z)} \cdot e^{\max(z)}}{e^{\max(z)} \sum_{j=1}^{k} e^{z_j - \max(z)}}$$

Dette kan reduceres til:

$$Softmax(\vec{z})_i = \frac{e^{z_i - \max(z)}}{\sum_{j=1}^{k} e^{z_j - \max(z)}}$$

Med den nye formel bliver potensen begrænset til at være lig med eller under 1. Ved brug af Python og NumPy kan vi implementere softmax-funktionen.

```python
def softmax(x):
    # Softmax functionen
    # returns probability distribution
    exp_x = np.exp(x - np.max(x, axis=0, keepdims=True))
    return exp_x / np.sum(exp_x, axis=0, keepdims=True)
```

Netværket er nu i stand til at sende data fremad gennem lagene i netværket. Eftersom vægtene og bias er blevet initialiseret med tilfældige værdier, betyder det, at netværket gætter tilfældigt, da modellen ikke har været igennem træning endnu. For at give netværket muligheden for at forbedre sig, skal der implementeres backpropagation.

Ved at bruge kædereglen kan vi effektivt beregne, hvordan ændringer i de enkelte vægte og biases i et neuralt netværk påvirker den samlede fejl. Kædereglen bliver brugt til at gå baglæns igennem netværket og udregne en variabels indflydelse på tabet. Vi vil gerne finde den afledte funktion af loss-funktionen med hensyn til en vægt i outputlaget. Til at starte finder vi den afledte funktion af loss-funktionen med hensyn til z2, som er neuronerne i det sidste lag før softmax-funktionen $\frac{\partial J}{\partial z2}$. Den ydre funktion er nu blevet differentieret, og det næste skridt er derfor at differentiere den indre. Da et neuron uden en aktiveringsfunktion er en lineær funktion med flere variabler, kan der relativt let blive differentieret med hensyn til vægten. For at finde den afledte funktion af loss-funktionen

med hensyn til b, kræves der minimalt arbejde, da den ydre funktion er den samme som før, og den indre er 1 ifølge teorien. Derefter bliver den ydre funktion ganget med den indre. Det er i denne proces, at matrixmultiplikation kommer i spil. Matrix multiplikation gør at hele datasættet bliver trænet på samtidigt. For at dimensionerne går op til at lave matrixmultiplikation, er det nødvendigt at transpose a2. Derefter bliver der taget gennemsnittet for vægtens gradient. Dette gøres, da vi først opdaterer vægtene og bias, efter netværket er trænet på hele datasættet. Da den indre funktion med hensyn til b er lig med 1, tager vi summen af vektoren. For at kunne differentiere det skjulte lags vægte og bias, er det først nødvendigt at differentiere ReLU funktionen. Ved at behandle ReLU mærke som en stykvis funktion, er dette forholdsvist simpelt. Dette kan nemt gøres ved hjælp af numpys where funktion, som ud fra om et givet udtryk er rigtig eller falsk indsætter et element. Så hvis x er større end nul, er hældningen 1 ellers er den 0. Da ReLu funktioen er differentieret er vi i stand til at differentiere med hensyn på outputtet fra neuronen i førstelag. Eftersom netværkets arkitektur ikke ændre sig undervej kan vi genbruge tidligere differentieret funktioner. Matrix multriplikationen med w2 gør at, fejlene fra output laget bliver korrekt sendt tilbage til det skjultelag. Derefter bliver der ganget med ReLUs afledte funktion, hvilket skyldes kædereglen. For at finde gradienten af vægtende i det første lag kan, man gentage metoden brugt til at differentiere vægtende i det andet lag. Gradienterne for vægtende og biasesne bliver returnet. De returnet gradienter bliver brugt til at optimere modellen ved graident descent. Ved at subtrahere gradienten ganget med læringsraten, bliver vægten forbedret. Gradient Descent fungerer ved først at sende dataene igennem netværkket forlæns, hvor der bliver holdt styr på resultaterne undervejs i netværket. Derefter går man baglæns, hvor resultaterne fra lagene og før og efter aktiverings funktionerne, bliver sammenlignet, ved backpropagation. Derefter bliver vægtende opdateret. Undervejs i trænings processen bliver udvikleren opdateret omkring netværks nøjagtighed på trænings dataene. Modellen outputter stadig en sandsynligheds fordelling, hvilket ikke er brugervenligt. For at konvetere dette til en heltal, som repræsentere den endelige

```python
def backProp(self,z1,a1,z2,a2,X,y) -> tuple[np.ndarray,np.ndarray,np.ndarray,
    """
    Backpropagation
    using categoricalcrossentropy loss function which is simplyfied A2 - y
    """

    m = y.shape[0]
    dz2 = a2 - y.T
    dw2 = 1/m * np.dot(dz2,a1.T)
    db2 = 1/m * np.sum(dz2, axis=1, keepdims=True)
    dz1 = np.dot(self.w2.T,dz2) * ReLU_m(z1)
    dw1 = 1/m * np.dot(dz1,X)
    db1 = 1/m * np.sum(dz1, axis=1, keepdims=True)

    return dw1,db1,dw2,db2
```

```python
def ReLU_m(x:float) -> float:
    # derivative of relu funktio
    # if x > 0 return 1 else 0
    return np.where(x > 0, 1,0)
```

```python
def update_parameters(self,alpha,dw1,db1,dw2,db2) -> None:
    """
    Update parameters
    """
    self.w1 -= alpha * dw1
    self.b1 -= alpha * db1
    self.w2 -= alpha * dw2
    self.b2 -= alpha * db2
```

```python
def gradientDescent(self, X: np.ndarray, y: np.ndarray, epochs: int, alpha: float) -> list:
    '''
    Gradient Descent
    '''

    for epoch in range(epochs):
        z1, a1, z2, a2 = self.forwardProp(X)
        dw1, db1, dw2, db2 = self.backProp(z1, a1, z2, a2, X, y)
        self.update_parameters(alpha, dw1, db1, dw2, db2)


        if epoch % 5 == 0:
            predictions = self.predict(X)
            accuracy = get_accuracy(predictions, y)
            print(f'Epoch: {epoch}, accuracy: {accuracy:.4f}')
```

```python
def predict(self,X:np.ndarray) -> np.ndarray:
    '''
    changes from a onehot encoded vector to a number
    outputs matrix (10,)
    '''

    _, _, _, a2 = self.forwardProp(X)
    return np.argmax(a2, axis=0)
```

forudsigelse, bliver der brugt argmax hvilket retunere index, som har den største værdi.

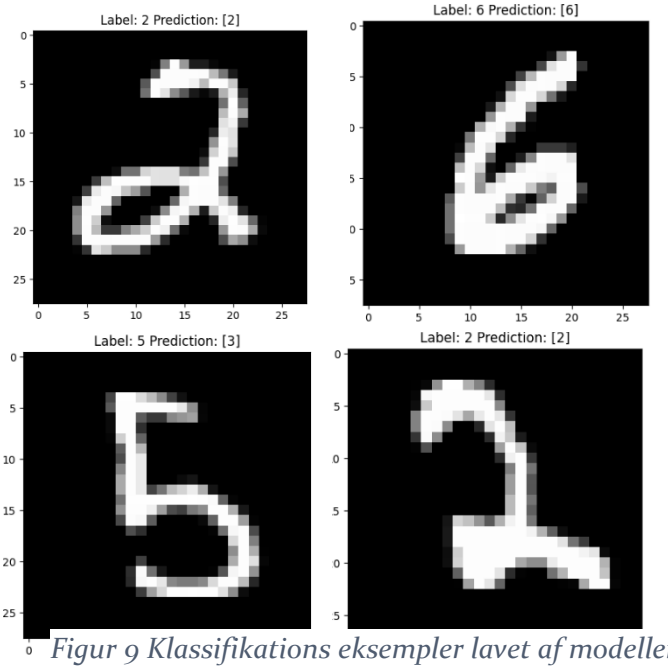
Efter teorien burde dette program være i stand til at træne på noget data og ændre vægtende i modellen for at opnå bedre præcision. Dette var ikke tilfældet ved første træningsforsøg, da modellen viste et besvær ved at optimere vægtende i netværket igennem en fejlfindings proces kom det frem til at det var en gradient fejl, under træningen, hvilket resulteret i at vægtende ikke blev træning, som det ellers var forventet. Grunden til fejlen var at dataene ikke var normaliseret, da en pixel kunne have en værdi mellem 0 og 255, dette gøre det svære for modellen at optimere parametrene. Ved at dividere inputs dataene med 255, så det lå mellem 0 og 1, resulteret det i at netværket var i stand til at optimere parametrene effektivt.

Model Evaluering

For at sikre, at netværket fungerer optimalt og ikke bare husker svarene under træningen, bliver modellen evalueret på data, som ikke indgår i træningssættet. Dette vil være med til at vurdere præstationen og effektiviteten af det neurale netværk, når det kommer til at klassificere håndskrevne cifre. Evalueringen vil fokusere på netværkets nøjagtighed på testdatasættet.

Model	Præcision på test
16 neuroner	61.46%
32 neuroner	72.51%
64 neuroner	80.85%

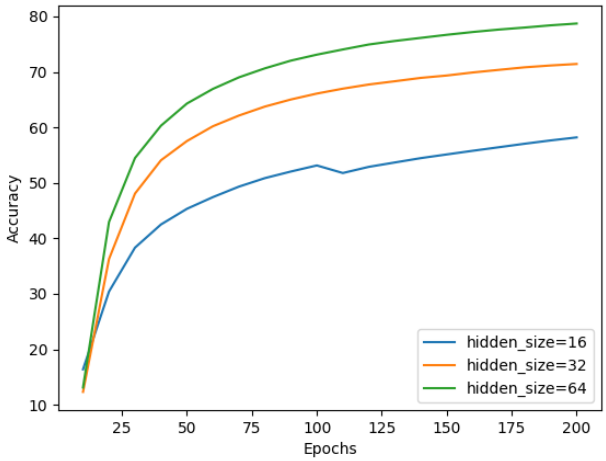
Det første netværk har 16 neuroner i det skjulte lag, og netværket er blevet trænet i 200 epochs. Efter træningen blev netværkets klassificeringsevne testet på testdatasættet, hvor netværket fik 61,46% korrekte forudsigelser. Sammenlignet med et tilfældigt gæt, som er 10%, er dette væsentligt bedre. Dette tolkes som, at netværket har været i stand til at træne på data og generalisere til nye, usete dataeksempler. Ved at se på konkrete eksempler kan man observere, hvordan netværket er blevet trænet til at genkende former som buer og lige streger. I eksemplet, hvor det håndskrevne tal var 5(figur9), kan man se, hvordan den nedre bue i femtallet kunne ligne den nedre bue i tallet 3. Derudover var netværket også i stand til at klassificere korrekt på det samme tal



Figur 9 Klassifikations eksempler lavet af modellen

skrevet på to forskellige måder. Det ene 2-tal har en krølle, som den anden ikke har.

Der blev yderlige undersøgt hvordan det skjulte lags antal neuroner havde en konsekvens på nøjagtigheden af netværket, der blev fundet frem til at ved at øge antal af neuroner i det skjulte lager, vil det forbedre netværket klassifikations evner. Ved at fordoble neuronerne i det skjulte lag fra 16 til 32 gav det en forbedring på 18%, på uset data. Ved at øge til 64 gav en nøjagtighed på 80.85% hvilket er en forbedring på 31.5%, fra det originale netværk. Dette kan betyde at den forholdsvise lave nøjagtighed på det originale netværket skyldes et lille antal neuroner. Ved at se på nøjagtigheden under træningen på træningsdata, kan man se hvordan netværket optimere tabsfunktionen, hvor gradienten i starten er meget stor hvilket betyder at den tager store skidt, hvor modellen efter flere epochs har svære ved at forbedre sig. Grafen viser også hvordan det store netværk forbedrer sig hurtigere.



Figur 10 Nøjagtighed under træningsprocess

Konklusion

Det kan konkluderes, at det er muligt at opbygge et neuralt netværk fra bunden ved brug af Python og NumPy. Ved at anvende den underliggende matematiske teori er der blevet konstrueret en fungerende model, som er i stand til at klassificere håndskrevne tal fra MNIST-datasættet med en høj nøjagtighed. Selv et lille neuralt netværk præsterer markant bedre end en model, der gætter tilfældigt, hvilket demonstrerer kraften og effektiviteten af selv en grundlæggende implementering af neurale netværk. Dette viser, at det ikke kun er muligt at forstå, men også at bygge komplekse maskinlæringsmodeller uden afhængighed af avancerede frameworks. Ud fra resultaterne ser det ud til, at der er potentiale for yderligere forbedring ved at tilføje flere neuroner i det skjulte lag og eventuelt tilføje flere lag i netværket. Ud over forbedring gennem forhøjelse ville det også være oplagt at finjustere hyperparametre for at opnå højere præcision fra modellen.

Matematikken er fundamentet for maskinlæring og neurale netværk. Det er igennem matematiske formler og ideer, at tankerne bag neurale netværk-algoritmer bliver forklaret. Det er derfor meget vigtigt at have gode matematiske kompetencer, når man bygger et neuralt netværk. Mange af de centrale principper, som gradienter og matrixoperationer, er kernen til, at et neuralt netværk kan fungere. Matematik optræder også i mange andre områder inden for maskinlæring, som opgaven ikke går i dybden med. Inden for maskinlæring bliver statistik ofte brugt i databehandling, da modellering af datasættet kan optimere modellens præstationer. Matematik kan også bruges til at optimere programmet; ved at reducere udtryk kan man spare på computerkraft, og den sparet computerkraft kan i stedet bruges til at træne modellen endnu mere.

Eftersom maskinlæring, og især neurale netværk, kræver hundredvis af iterationer for at træne modellerne, er programmering et uundværligt værktøj. Igennem programmering kan man opdatere vægtene i netværket superhurtigt og træne modellen på enorme datasæt. Programmering gør det muligt at automatisere træningsprocessen, så det kræver lidt til ingen menneskelige kræfter.

Matematik og programmering er begge essentielle fag, når det gælder neurale netværk. Ved at kombinere fagene kan man benytte matematikkens universelle sprog til at formidle de centrale ideer bag neurale netværk, hvor man benytter programmmeringens evne til at udføre mange beregninger på kort tid.

Litteraturliste

[1] Frank, »The Perceptron,« Cornell Aeronautical Laboratory, 1958.

[2] Google, »developers.google.com,« google, 9 10 2024. [Online]. Available: https://developers.google.com/machine-learning/crash-course/overfitting/overfitting. [Senest hentet eller vist den 10 12 2024].

[3] Kaggle, »kaggle.com,« National Institute of Standards and Technology, [Online]. Available: https://www.kaggle.com/competitions/digit-recognizer.

[4] M. i. fun, »Math is fun,« [Online]. Available: https://www.mathsisfun.com/algebra/matrix-multiplying.html.

[5] cuemath.com, »cuemath.com,« cuemath.com, [Online]. Available: https://www.cuemath.com/algebra/transpose-of-a-matrix/.

[6] D. B. o. C. Stryker, »IBM.com,« IBM, [Online]. Available: https://www.ibm.com/think/topics/loss-function.

[7] M. Sorokin, »medium,« [Online]. Available: https://medium.com/@vergotten/categorical-cross-entropy-unraveling-its-potentials-in-multi-class-classification-705129594a01.

[8] D. J. &. J. H. Martin, »chapter 7 Neural networks,« i *Speech and language processing*, Draft, 2024.

[9] webmatematik, »webmatematik,« [Online]. Available: https://www.webmatematik.dk/lektioner/matematik-a/funktioner-af-to-variable/partielle-afledede.

[10] systime, »systime,« [Online]. Available: https://matbhtx.systime.dk/?id=1403.

[11] U. o. york, »https://online.york.ac.uk/,« [Online]. Available: https://online.york.ac.uk/what-is-computational-thinking/.

[12] systime, »systime,« [Online]. Available: https://matbhtx.systime.dk/?id=1454.

Bilag

Koden kan også tilgås igennem github: https://github.com/oliverharboe

Kode:

```
import numpy as np


class NeuralNetwork:
    def __init__(self,hidden_size) -> None:
        '''
        Initalizere parameters (weights and biases)
        '''
        self.w1 = np.random.randn(hidden_size, 784)
        self.b1 = np.random.randn(hidden_size, 1)
        self.w2 = np.random.randn(10, hidden_size)
        self.b2 = np.random.randn(10, 1)

    def forwardProp(self,X:np.ndarray) -> tuple:
        '''
        Forward propagation
        '''
        z1 = np.dot(self.w1,X.T) + self.b1
        a1 = ReLU(z1)
```

```
        z2 = np.dot(self.w2,a1) + self.b2
        a2 = softmax(z2)

        return z1,a1,z2,a2

    def backProp(self,z1,a1,z2,a2,X,y) ->
tuple[np.ndarray,np.ndarray,np.ndarray,np.ndarray]:
        """
        Backpropagation
        using categoricalcrossentropy loss function which is simplyfied A2 - y
        """

        m = y.shape[0]
        dz2 = a2 - y.T
        dw2 = 1/m * np.dot(dz2,a1.T)
        db2 = 1/m * np.sum(dz2,axis=1,keepdims=True)
        dz1 = np.dot(self.w2.T,dz2) * ReLU_m(z1)
        dw1 = 1/m * np.dot(dz1,X)
        db1 = 1/m * np.sum(dz1,axis=1,keepdims=True)
        return dw1,db1,dw2,db2

    def update_parameters(self,alpha,dw1,db1,dw2,db2) -> None:
        """
```

```
        Update parameters
        """
        self.w1 -= alpha * dw1
        self.b1 -= alpha * db1
        self.w2 -= alpha * dw2
        self.b2 -= alpha * db2



    def gradientDescent(self, X: np.ndarray, y: np.ndarray, epochs: int, alpha: float) ->
None:
        '''
        Gradient Descent
        '''
        accuracy_arr = []
        for epoch in range(epochs):
            z1, a1, z2, a2 = self.forwardProp(X)
            dw1, db1, dw2, db2 = self.backProp(z1, a1, z2, a2, X, y)
            self.update_parameters(alpha, dw1, db1, dw2, db2)


            if epoch % 10 == 0:
                predictions = self.predict(X)
                accuracy = get_accuracy(predictions, y)
                accuracy_arr.append(accuracy)
                print(f'Epoch: {epoch}, accuracy: {accuracy:.4f}')
        return np.array(accuracy_arr)
```

```
    def predict(self,X:np.ndarray) -> np.ndarray:
        '''
        changes from a onehot encoded vector to a number
        outputs matrix (10,)
        '''
        _, _, _, a2 = self.forwardProp(X)
        return np.argmax(a2, axis=0)

def ReLU_m(x:float) -> float:
    # derivative of relu funktion
    # if x > 0 return 1 else 0
    return np.where(x > 0, 1,0)

def softmax(x: np.ndarray) -> np.ndarray:
    # Softmax functionen
    # returns probability distribution
    exp_x = np.exp(x - np.max(x, axis=0, keepdims=True))
    return exp_x / np.sum(exp_x, axis=0, keepdims=True)

def ReLU(x:np.ndarray) -> np.ndarray:
    # ReLu functionen
    return np.maximum(0,x)

def get_accuracy(predictions: np.ndarray, y: np.ndarray) -> float:
```

```
    true_labels = np.argmax(y, axis=1)
    return np.mean(predictions == true_labels)


import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
from neuralnetwork import NeuralNetwork
def main():
    FILEPATH = 'Data/mnist_data.csv'
    X,y = load_data(FILEPATH)
    X = normalize_data(X)
    X_train,y_train,X_test,y_test = split_data(X,y)
    y_train = oneHotlabel(y_train)
    model = NeuralNetwork(hidden_size=16)
    #model2 = NeuralNetwork(hidden_size=32)
    #model3 = NeuralNetwork(hidden_size=64)
    accuracy = model.gradientDescent(X_train,y_train,epochs=400,alpha=0.1)
    #accuracy2 = model2.gradientDescent(X_train,y_train,epochs=400,alpha=0.1)
    #accuracy3 = model3.gradientDescent(X_train,y_train,epochs=350,alpha=0.05)
    #plot_accuracy(accuracy,accuracy2,accuracy3)
    #prediction1 = model3.predict(np.array(X_test[49:50]))
    #prediction2 = model3.predict(np.array(X_test[20:21]))
    #prediction1 = model.predict(np.array(X_test[1:]))
    #prediction2 = model2.predict(np.array(X_test[1:]))
    #prediction3 = model3.predict(np.array(X_test[1:]))
```

```
    #print(f'16n accuracy: {test_accuracy(prediction1,y_test[1:])}, 32n accuracy:
{test_accuracy(prediction2,y_test[1:])}, 64n accuracy:
{test_accuracy(prediction3,y_test[1:])}')

    #plot_numbers(X_test[49],y_test[49],prediction1)

    #plot_numbers(X_test[20],y_test[20],prediction2)


def load_data(path:str) -> tuple[np.ndarray,np.ndarray]:
    '''
    loading data from csv
    '''
    train_df = pd.read_csv(path)


    y = train_df.loc[:,'label']
    X = train_df.drop('label',axis=1)


    return X.to_numpy(),y.to_numpy()


def split_data(X:pd.DataFrame,y:pd.DataFrame) ->
tuple[np.ndarray,np.ndarray,np.ndarray,np.ndarray]:
    '''
    splits data into train and test
    '''
    X_train,X_test,y_train,y_test =
train_test_split(X,y,test_size=0.05,random_state=0,stratify=y)
    return X_train,y_train,X_test,y_test


def normalize_data(data):
```

```
    # change interval from [0;255] to [0;1]
    return data/255.


def oneHotlabel(y:np.ndarray) -> np.ndarray:
    """
    Onehot encode labels output shape is n * 10
    where n is the number of labels
    arange runs through the rows. then it indexes to the y[i] value og places a 1 in that place


    """
    oneHot = np.zeros((y.shape[0],10
))
    oneHot[np.arange(y.shape[0]),y] = 1
    return oneHot


def plot_accuracy(accuracy:np.ndarray,accuracy2:np.ndarray,accuracy3:np.ndarray) ->
None:
    x = np.arange(start=10,stop=accuracy.shape[0]*10+10,step=10)
    accuracy = accuracy*100
    accuracy2 = accuracy2*100
    accuracy3 = accuracy3*100
    plt.plot(x,accuracy)
    plt.plot(x,accuracy2)
    plt.plot(x,accuracy3)
    plt.legend(['hidden_size=16','hidden_size=32','hidden_size=64'])
    plt.xlabel('Epochs')
```

```
    plt.ylabel('Accuracy')
    plt.show()


def plot_numbers(num:np.ndarray,label:np.ndarray,prediction:np.ndarray) -> None:
    '''
    Creating a gray scale image of the number (with and without label)
    '''
    image = num.reshape(28,28)
    plt.imshow(image, cmap='gray')
    if label != None:
        plt.title(f'Label: {label} Prediction: {prediction}')
    plt.show()


def test_accuracy(pre,y):
    correct = (pre == y).sum()
    total = len(y)
    return correct / total


if __name__ == '__main__':
    main()
```