Ansigtsgenkendelse

Studieområdeprojekt: Mat A og Programmering B

Udarbejdet i LaTeX af Louie Juhl
3.N NEXT Sukkertoppen Gymnasium

20/12-2019

1 Resumé

I dette projekt og denne rapport har jeg beskæftiget mig med ansigtsgenkendelse, der tilhører supervised learning paradigmet indenfor datalogiens machine learning. For at løse dette problem er jeg dykket ned i matematikken bag og undersøgt hvordan faget som redskab kan bruges til at opstille en matematisk forståelse af en hjerne, nemlig den matematiske model, det neurale netværk. For at dekomponere og forstå ansigtsgenkendelse har jeg blandt andet brugt syntetiske metoder til en visuel forståelse af problemet og formelle metoder til at repræsentere det med ren matematik. Herefter har jeg anvendt programmeringsfaget og dets metoder, såsom computational thinking og del og hersk princippet til at implementere matematikken i kode således, at jeg kunne implementere et modulært neuralt netværk der kan foretage valg på et abstrakt plan og danne sin egen forståelse af problemets rammer på baggrund af noget data. Mit netværk kunne i sidste ende foretage chiffergenkendelse med en præcision på 86%. Ansigtgenkendelse ville have krævet en langt kraftigere computer, da arkitekturen af netværket ville være mere kompleks for at løse den problemstilling. Ydermere diskuteres hvilke ændringer i netværket og dermed også koden, der kan foretages for at forbedre resultaterne.

2 Forord

Inden jeg præsenterer mit emne vil jeg fortælle om nogle nævneværdige deltajer, der er relevante i forhold til at forståelsen af denne opgave. I teksten bliver der brugt mange fremmedord, hvilket er en selvfølge eftersom, at "machine learning" ikke er særlig velkendt i Danmark og derfor findes de danske oversættelser simpelthen ikke. Først gange man støder på et fremmedord vil det blive præsenteret med gåseøjne og derefter bruges uden. Derudover er det relevant at nævne, at det ikke har været muligt at lave en funktionsdygtig ansigtsgenkendelses algoritme til mit computer system. Dog er der implementeret en algoritme, der potentielt set kunne skaleres så den kunne bruges til ansigtsgenkendelse.

Indholdsfortegnelse

1 Resumé 1
2 Forord 1
3 Indledning 3
4 Metode 3
5 Teori 4
5.1 Hvad er convolutional neural networks 4
5.2 Vektor og matrixer anvendt til at repræsentere neural netværk 5
5.3 Loss funktioner og gradient descent 7
5.4 Backpropagation 10
5.5 Convolutional neural networks 12
6 Implementering af mit convolutional neural network 14
7 Diskussion og konklusion 19
Litteraturliste 21
Bilag 22
ConvNet.py 22
program.py 26

3 Indledning

I den antropocæne epoke bliver vi i vores hverdag påvirket af et kun stigende antal teknologier, der skaber en værdi for os. Specielt det digitale har været i kraftig fremvækst i løbet af de sidste to årtier, sådan at man endda snakker om den digitale verden. Årsagen til dette er blandt andet at computere bliver hurtigere og derfor kan foretage sig stadigt mere komplekse opgaver. Indenfor de sidste 10 år har der især været en stærk fremdrift indenfor området "machine learning". Machine learning er tanken om, at maskiner kan lære af sig selv, hvilket allerede er blevet dystopiseret i film langt før de reelt set kunne gøre det på en stor skala. Men det kan de nu. Data generet af vores interaktion med digitale teknologier har givet os og maskinerne mulighed til at lære mere om os selv og verdenen omkring os. Der findes to områder, som man fristes til at kalde paradigmer indenfor machine learning: "supervised learning" og "unsupervised learning". Supervised learning er fremgangsmåden hvor vi mennesker angiver hvad det rigtige resultat af en bestemt operation burde være, så maskinen kan forbedre sine aktioner. Derimod skal maskinen selv genkende mønstre i data og forstå disse i unsupervised learning. Her fortæller vi, så at sige, ikke til maskinenen hvad det korrekte "output" burde være. Ansigtgenkendelse er en af teknologierne udsprunget af machine learnings udvikling og viden på området. Det er fordi, at det er et relevant værktøj til at løse en lang række forskellige problemer såsom, brug af ansigt som "login" til sin telefon. Ansigtsgenkendelse foretages indenfor supervised learning paradigmet. Vi lader altså algoritmen vide, hvad det forventede output er. Dette er også en såkaldt klassificeringsopgave, da vi klassificerer hvorvidt dette var den forventede person. Til det formål bruger man ofte neurale netværk, hvilket jeg også vil benytte mig af. Men hvordan kan man forstå en problemstilling som ansigtsgenkendelse og nedbryde det ved hjælp af matematik, for senere at implementere matematikken med programmering?

4 Metode

Matematik er en aksiomatisk-deduktiv videnskab. En fin beskrivelse af dette findes i bogen primus[11], der lyder: "I matematik søger man ud fra nogle grundlæggende antagelser, de såkaldte aksiomer, gennem logisk bevisførelse (deduktion) at få indsigt i tallenes egenskaber, geometriske figurer og andre abstrakte strukturer." Når man opbygger en matematisk teori sammenkæder man disse grundantagelser, som er universelt sande, ved hjælp af matematisk bevisførelse, hvorefter man kan anskue noget på et mere abstrakt niveau. Hvis en teori kan nedbrydes udelukkende til aksiomer, må den derfor være gyldig[10]. Når man skaber ny matematisk viden arbejder man ud fra den induktive metode, hvor man igennem sin viden, forsøger at lave nogle antagelser om et eller andet indenfor matematikkens rammer. Dette gør vi gymnasieelever dog sjældent. Vi anvender i stedet matematisk modellering. Vi tager udgangspunkt i et virkelighedseksempel og afgrænser det indenfor forskellige matematiske områder, hvorefter vi behandler disse problemer i de forskellige områder. Det kunne for eksempel være at beregne arealet af en trekant. For at behandle og konkludere noget om de matematiske problemstillinger bruges følgende tre typer af metoder.

- Syntetiske metoder: At konstruere figurer eller kigge på en visuel fremstilling af en

funktion for at løse problemet.

- Formelle metoder: At omskrive en ligning og løse denne eller at integrere en funktion for at finde arealet under den osv.
- Numeriske metoder: At lave regression på nogle punkter og finde regressionen, der bedst passer på problemet. En konkret numerisk metode kunne fx. være Runge-Kutta metoden til differentialligninger.[11]

Så dette er måden hvorpå man arbejder i matematik. Hvad så med programmering? I programmering snakker man om en række forskellige metoder eller tilgange til et problem.

- Computational thinking: Dekomponering, abstraktion, mønstergenkendelse og algoritmedesign. Man bruger ofte mønstergenkendelse til datalogiopgaver, da man skal se, om man kan finde et mønster og derved danne en antagelse, der løser problemet.
- Stepwise improvement: Vi udvikler en løsning, tester løsningen, vurderer hvorvidt den har opfyldt målet og gentager denne proces, indtil målet er nået.
- Divide and conquer: Man deler et problem ned i mindre dele, som man kan forstå, for derefter at kunne løse hele problemet.[2]

Alt dette har jeg anvendt således at jeg kunne løse ansigtsgenkendelse.

5 Teori

5.1 Hvad er convolutional neural networks

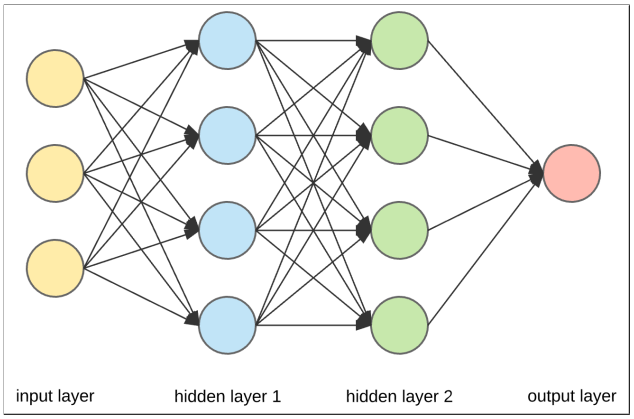

Figure 1: Visuel fremstilling af et neuralt netværk[12]

Der er forskellige måder hvorpå at man kan anskue neurale netværk, da de findes i forskellige former og versioner. Der er først og fremmest den biologiske intuition, som omhandler neuroner i hjernen, der sender og modtager signaler hvorefter kroppen foretager sig en eller anden handling på et større eller mindre niveau. Det er netop denne forståelse af hjernen,

som den anden betragtning, de såkaldte "digitale" neurale netværk er opbygget op omkring. Foroven kan der ses en visuel fremstilling af et neuralt netværk hvor vi har nogle knuder, der repræsenterer neuroner og kanterne imellem disse, som angiver hvor meget neuronens værdi i det forhenværende lag har/skal have for den individuelle neuron. Disse kanter kaldes også vægte og de bliver kun mere relevante i forståelsen for de neurale netværk. Det første lag er inputtet til dit neurale netværk og det sidste lag er outputtet. Lagene herimellem kaldes for "hidden layers". Denne betegnelse indikerer at man ofte behandler netværket, som en såkaldt "blackbox", der betyder, at man ikke har en hundrede procent klar forståelse af hvordan de skjulte lag opfatter og behandler inputtet[5]. Når man behandler billeder bruger ikke præcist denne version af neurale netværk, men derimod "convolutional neural networks" som jeg vil fortælle mere om senere, da matematikken er meget af det samme.

5.2 Vektor og matrixer anvendt til at repræsentere neural netværk

De neurale netværk kan, som antydedet tidligere, repræsenteres digitalt. Her bliver det matematiske værktøj vektorer og dets egenskaber anvendelige. Definition af en vektor i bogen Mat B htx[9] lyder således: "En vektor er en talstørrelse med en retning. Den angives som et liniestykke med en pilespids i den ene ende, der viser retningen. Størrelsen af vektoren er lig med længden af liniestykket." En af de fantastiske egenskaber ved en vektor, som også delvist kan ses i definitionen, er dens dimensionalitet. Da en vektor er en talstørrelse med en retning og retninger kan være n-dimensionelle, kan vektorer også være det. Hvilket betyder at en vektor med n-dimensioner kan opstilles, som forneden:

$$v = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{1}$$

Dette kan benyttes til at skildre vores lag som en vektor, hvor hver neuron er en værdi for en dimension:

$$lag = \begin{bmatrix} neuron_1 \\ neuron_2 \\ \vdots \\ neuron_n \end{bmatrix} \tag{2}$$

Tidligere forklarede jeg, at en neurons værdi i et lag afhang af nogle vægte og neuronerne i det tidligere lag. Faktisk hænder det, at neuronens værdi er lig summen af hver neuron i det tidligere lag ganget med vægten imellem neuronen og neuronen i det forhenværende lag. Matematisk kan dette skrives op således:

$$neuron_i = \sum_{j=1}^{j} w_{ij} \cdot neuron_j \tag{3}$$

Her er j den n'te neuron i det tidligere lag og i er den n'te neuron i dette lag. Men vi vil gerne have en ny vektor, der beskriver neuronenes værdier i det nuværende lag og til dette kan vi bruge vektor/matrix egenskaber. Vi bliver dog først nødt til at introducere matrix begrebet. Ifølge [1] er en matrix er et rektangulært array der indeholder tal, symboler eller udtryk, som er opsat med række og kolonner, $m \times n$. En matrix med størrelsen 3x2 med nogle tilfældelige tal ser ud således:

$$\begin{bmatrix} 0.9 & 2 \\ 4 & 2.3 \\ 0.2 & 3 \end{bmatrix} \tag{4}$$

I machine learning bruger vi matrixer til at repræsentere vores vægte. Derudover findes der en smart egenskab med matrixer, der hedder matrix multiplikation, hvilket minder meget om vektorens prik produkt. Denne generelle formel for matrix multiplakation ses nedstående:

$$Ax = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1+ & a_{12}x_2+ & \cdots+ & a_{1n}x_n \\ a_{21}x_1+ & a_{22}x_2+ & \cdots+ & a_{2n}x_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}x_1+ & a_{m2}x_2+ & \cdots+ & a_{mn}x_n \end{bmatrix} [7] \tag{5}$$

Vi har her en matrix A og laver matrix multiplikation med en kolonne vektor x. Vi kan faktisk anskue A, som en vægt matrix, hvor hver række indeholder individuelle vægte imellem neuronerne i det forhenværende lag og neuronen. Så når vi ganger vores vektor x, som er neuronernes værdier i det forhenværende lag, vil det stemme overens med vores tidligere definition af en neurons værdi, da man summerer værdierne over rækken. For at skabe en klarere forståelse kan et eksempel med matrix multiplikation ses forneden

$$z^l = wx = \begin{bmatrix} 0.02 & 2 \\ 4 & 2.3 \end{bmatrix} \begin{bmatrix} 2 \\ 2.5 \end{bmatrix} = \begin{bmatrix} 5.04 \\ 13.75 \end{bmatrix} \tag{6}$$

z er en vektor, der beskriver neuronernes værdier i laget l, matrixen w er vægtene og x er en vektor for neuronernes værdier i det forhenværende lag. Kanten/vægten $w_{1,1}$ imellem x_1 og z_1^l, angiver at neuronen x_1 har lille betydning for z_1^l's værdi. Derimod angiver vægten $w_{2,1}$ at x_1 har stor betydning for z_2^l's værdi.

Nu har vi næsten fået klarlagt, hvordan man bestemmer værdierne i lagene, men der resterer et enkelt problem. Lige nu udfører vi kun linære beregninger(matrix multiplikation), hvilket betyder, at vores netværk reelt set vil kollapse til en enkelt linær funktion og derfor ikke vil kunne træffe komplekse beslutninger. Derfor skal vi tilføje ikke-linære funktioner imellem vores lag. Dette gøres efter man har beregnet matrix produktet z. En af de mest brugte funktioner til dette formål er reLU funktionen.

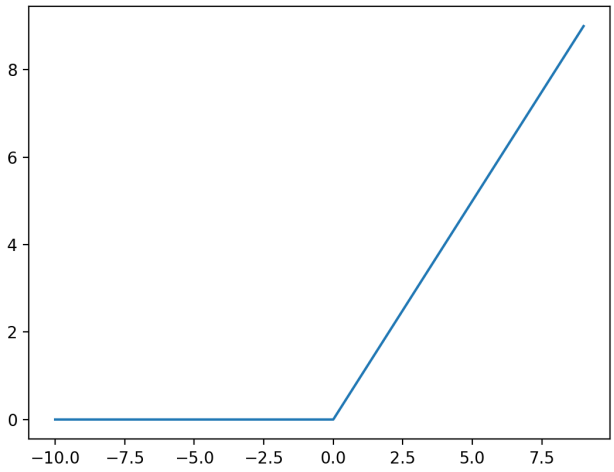

Figure 2: En afbildning af rectified linear unit

reLU funktionens matematiske notation ses forneden:

$$reLU(x) = max(0, x) \tag{7}$$

Tanken bag reLU funktionen er, at vi - ligesom i hjernen - ikke sender en negativ impuls videre, hvilket giver god mening når man kigger på funktionen. Vi anvender vores reLU funktion på hvert enkelt element i vores vector z og finder vores output fra laget, som skal gives som input til det næste lag. Dette output kalder vi for a, der står for "activation". Så neuronens egentlige værdi er ikke z, men a.

$$a = reLU(z) \tag{8}$$

Det, vores algoritme nu skal foretage sig, er at beregne hver activation i hvert lag og sende resultatet videre til næste lag, som input vektor, hvorpå vi i det sidste lag vil få et output.

5.3 Loss funktioner og gradient descent

Vi vil nu gerne finde en måde til at beskrive, hvor tæt vores output er på, hvad det burde være. Dette kaldes en "loss" funktion. Jo lavere tabet beregnet af loss funktionen er, desto bedre klarer netværket sig. En loss funktion, der ofte bliver brugte til klassifikationsopgaver er den såkaldte "average cross entropy loss function". Den er defineret således:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{9}$$

Hvad er L_i? L_i er tabet fra det individuelle træningseksempel, hvilket kan beregnes på forskellige måder. En såkaldt Softmax funktion[4] kan beregne det på følgende måde:

$$L_i = -log(p) \tag{10}$$

$$p = \frac{e^s_y}{\sum_j e^s_j} \tag{11}$$

Her er s_y scoren for den korrekte klasse over summen af alle scorene for alle klasserne. Man får altså derfor en eller anden sandsynlighed p, som man tager den negative logaritme af.

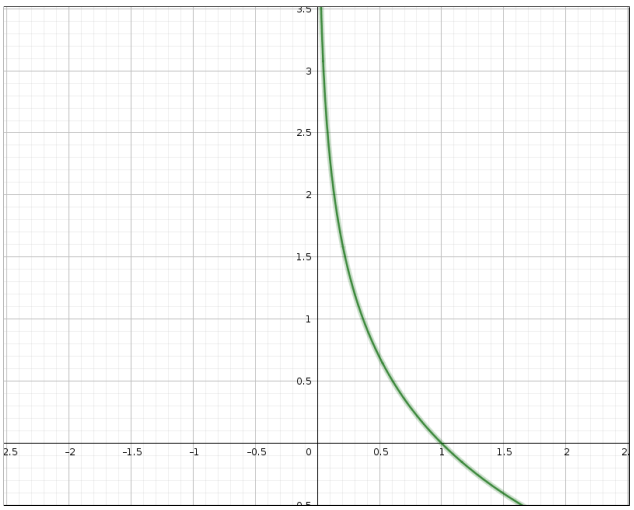

Figure 3: En afbildning af -log(p)

Dette giver intuitivt mening, da tabet vil stige jo mindre vores sandsynlighed for den korrekte klasse er og vores minimale tab vil være 0, når sandsynligheden p er 1. Det er netop her, at ideen om supervised learning indtræder, fordi man skal vide hvad den korrekte klasse y er, for at kunne udregne tabet.

Nu har vi beskrevet en måde, der kan bestemme tabet af en funktion, hvilket leder os videre til hvordan optimeringen af funktionen skal foregå, således at man opnår mindre forkerte resultater, altså at tabet minimeres. En ofte brugt analogi for hvordan man formindsker sit tab, er forestillingen om en dal. Du står et eller andet sted i en dal, hvor din placering er inputtet til loss funktionen. Punktet du står på, har en højde, som er tabet af din funktion(funktionsværdien af loss funktionen) ved netop dette input. Målet er - ligesom tidligere - at mindske tabet og derved finde et lavpunkt i dalen. Men hvordan gør man dette? Jo, det er her hvor ideen om "gradient descent" indtræffer. Gradient descent er tanken om, at du gerne vil gå i den retning hvor højden(tabet) formindskes mest. Så hvordan gøres dette? Det er her, hvor differentialregning gør sit indtog. Med differentialregning kan man bestemme hældningen af tangenten til et punkt. Men hvordan var det lige man gjorde dette? Jo, det kan vi gøre med tretrinsreglen. Tretrinsreglen har følgende skridt.

1. Trin 1: Bestem differenskvotienten
2. Trin 2: Omroker udtrykket så trin 3 bliver lettere
3. Trin 3: Bestem grænseværdien for $\Delta x \to 0$ [9]

Med funktionen $f(x) = ax^2$ gøres det på følgende måde.

Trin 1:

$$\frac{\Delta y}{\Delta x} = \alpha = \frac{f(x_0 + \Delta x) - f(x_0)}{\delta x} = \frac{a \cdot (x + \Delta x)^2 - ax^2}{\Delta x} \tag{12}$$

Man har her valgt to punkter $f(x_0)$ og $f(x_0 + \Delta x$, hvor at Δx er en tilvækst i x. Derefter har man i tælleren fundet forskellen i funktionsværdi imellem de to punkter Δy. Imellem disse to punkter kan der tegnes en ret linje, som også kaldes sekanten. Vi ved, at hældningen af en sekant er funktionstilvæksten Δy over en tilvækst Δx. Derfor har man divideret med tilvæksten Δx for at finde ud af, hvad sekantens hældning er. Derefter kan vi bare indsætte vores funktionsværdi på pladserne.

Trin 2: Vi omrokerer nu vores udtryk

$$\frac{a \cdot (x + \Delta x)^2 - ax^2}{\Delta x} = \frac{a \cdot (x^2 + \Delta x^2 + 2 \cdot x \cdot \Delta x) - ax^2}{\Delta x} = \frac{a \cdot x^2 + a \cdot \Delta x^2 + 2 \cdot a \cdot x \cdot \Delta x - ax^2}{\Delta x} \tag{13}$$

Vi har først brugt kvadratsætningen til at udregne indholdet i vores parantes, derefter har vi ganget a ind på vores parantes.

$$\frac{a \cdot x^2 + a \cdot \Delta x^2 + 2 \cdot a \cdot x \cdot \Delta x - ax^2}{\Delta x} = a \cdot \Delta x + 2 \cdot a \cdot x \tag{14}$$

Vi ser her at ax^2 går ud med $-ax^2$ og at vores division medΔx får Δx^2 til at blive til Δx og vores andet Δx i det andet led forsvinder.

Trin 3:

$$f'(x) = \lim_{\Delta x \to 0} a \cdot \Delta x + 2 \cdot a \cdot x = 2 \cdot a \cot x \tag{15}$$

Vi tager her limitfunktionen, som går ind og formindsker afstanden, Δx, imellem vores punkter, således at forskellen imellem dem er nul, så vi så at sige befinder os i punktet x_0. Så tanken er, at man gør afstanden så lille, at man kan sige at hældningen er linær og derfor kan antage hældningen af sekanten. Da Δx nu er 0 forsvinder hele leddet, da $a \cdot 0 = 0$. Vi kan nu til sidst konkludere, at differentialkvotienten af denne funktion er[9]:

$$\frac{df}{dx} = f'(x) = 2 \cdot a \cdot x \tag{16}$$

Vi har nu beskrevet hvordan hældningen af tangenten til et punkt kan beskrives. Dette er gjort af en funktion med en variabel. I vores tilfælde har vi at gøre med flere dimensioner og derfor tager vores funktion ikke bare en variabel, men mange flere. Hvis vi har at gøre med funktioner af flere variabler, som fx en vægt matrix w og en input vektor x, bliver vi nødt til at bruge partielt afledede. Så hvad er partielt afledede? En partielt afledt er, at man differentierer med hensyn til den ene variabel og lader den anden agere konstant. Hvis vi fx har funktionen: $f(h, t) = 3h^2 - 2t$ og vi beregner den partielt afledede med hensyn til h:

$$\frac{\partial f}{\partial h} = 2 \cdot 3h \tag{17}$$

Da t anskues som en konstant, forsvinder ledet med t og vi bruger konklusionen fra tretrinsreglen fra eksemplet forinden, at en funktion på formen $f(x) = ax^2$ har differentialkvotienten $f'(x) = 2 \cdot a \cdot x$. Her finder vi den partielt afledede med hensyn til h. Hvis vi derimod vil beregne den partielt afledede med hensyn til t, lader vi h være en konstant og foretager samme fremgangsmetode[8]. Ordet gradient, som jeg nævnte tidligere, betyder faktisk bare en vektor med de partielt afledte, hvilket angiver retningen hvori hældningen er størst [4]. Hvis vi vender tilbage til vores forestilling om en dal, har vi jo faktisk en funktion, loss funktionen, som tager en vektor som variabel og giver os et tab. Vi kan derfor tager gradient af denne funktionen og finde ud af, hvilken retning vi skal bevæge os i for at maksimere vores tab, da gradienten angiver hældningen. Hvis vi tager den negative gradient, vil vi få en vektor af partielt afledte, som har den modsatte retning af hældningen. Derefter skal vintage et skridt i denne retning og bevæge os imod et lavere tab, altså en bølgedal. Men for at minimere tabet kræver det, at vi justerer alle vores vægte imellem vores neuroner så vi får den ønskede effekt af et skridt i retningen af den negative gradient af vores tab. Så vi skal beregne hvor stor en betydning de individuelle vægte har for tabet, således at vi kan justere de vægte, der forårsager et stort tab i den anden retning. Til dette bruger man "backpropagation".

5.4 Backpropagation

For at forklare backpropagation vil jeg starte med at introducere en såkaldt "computational graph"[5], der opdeler et regnestykke i mindre dele. Vi opstiller en funktion af tre variable:

$$f(t, h, k) = (h + k) \cdot t \tag{18}$$

Vi kan her dele vores regnestykke op i mindre dele, da vi først lægger h og k sammen hvorefter vi multiplicerer resultatet med t, hvilket giver os funktionsværdien. Vi kan derfor opdele vores funktion i to andre funktioner:

$$v(h, k) = h + k \tag{19}$$

Her er v en funktion, der beskriver h og k lagt sammen. Nu kan vi omskrive vores funktion f til følgende.

$$f(t, v) = v \cdot t \tag{20}$$

Vi kan nu tegne følgende graf:

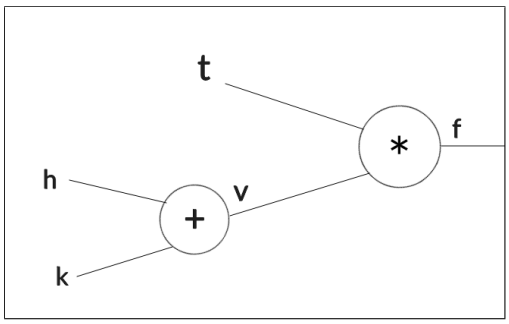

Figure 4: En afbildning vores computational graph

Da vi nu har forstået hvad en computational graph er, vil jeg forklare backpropagation nærmere. I backpropagation regner vi tilbage i vores computational graph for at finde ud af, hvor stor en indflydelse de individuelle variable havde på vores output f. Det bliver klarere undervejs hvad der menes med dette. Vi vil nu tage et eksempel hvor vores variabler antager værdierne: t = 2, h = 4, k = -5 og udføre backpropagation

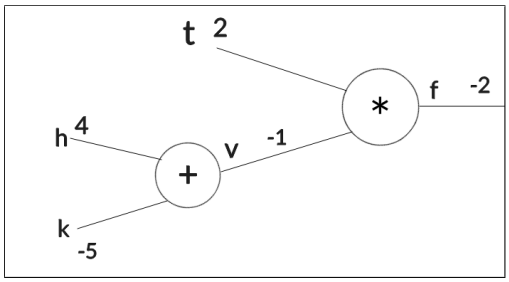

Figure 5: En afbildning vores computational graph

Vi vil nu gerne bestemme de partielt afledede. Altså hvor stor betydning henholdvis t, h og k har på f. Vi starter den partielt afledede af f med hensyn til f.

$$\frac{\partial f}{\partial f} = 1 \tag{21}$$

Vi vil herefter bestemme den partielt afledede af f med hensyn til t.

$$\frac{\partial f}{\partial t} = v => -1 \tag{22}$$

Vi har her brugt vores differentialregneregel, som siger, at hvis vi har at gøre med en funktion på formen $f(x) = ax$ så vil differentialkvotienten være: $f'(x) = a$. Dette kan gøres, fordi vores v agerer konstant. Vi vil nu bestemme den partielt afledte af f med hensyn til v

$$\frac{\partial f}{\partial v} = t => 2 \tag{23}$$

Det præcis samme gennemføres, som i det tidliger tilfælde med t, men hvor det nu er t som agerer konstant. Nu skal vi finde den partielt afledede af f med hensyn til h. Men h og

f er ikke forbundet direkte. Derfor bruges den såkaldte "chain rule". Kædereglen kan ses forneden.

$$\frac{\partial f}{\partial h} = \frac{\partial f}{\partial v}\frac{\partial v}{\partial h} \tag{24}$$

Så tanken bag dette er kort sagt, at vi først udregner den partielt afledte af f med hensyn til v("upstream gradient"), hvorefter vi skalerer denne værdi med partielt afledede af v med hensyn til h("local gradient"). Så vi starter med at beregne den partielt afledte af v med hensyn til h.

$$\frac{\partial v}{\partial h} = 1 \tag{25}$$

Da k agerer konstant forsvinder ledet og i h's led står der implicit 1*h, altså er vores a værdi 1 og derfor også vores partielt afledte det. Da vi i forvejen kender den partielt afledte af f med hensyn til v kan vi gøre følgende:

$$\frac{\partial f}{\partial h} = \frac{\partial f}{\partial v}\frac{\partial v}{\partial h} => 2 \cdot 1 = 2 \tag{26}$$

Det præcis samme gøres for vores variabel k, hvor vi dog blot skalerer den partielt afledte af f med hensyn til v med den partielt afledte af v med hensyn til k:

$$\frac{\partial f}{\partial k} = \frac{\partial f}{\partial v}\frac{\partial v}{\partial k} => 2 * 1 = 2 \tag{27}$$

Nu har vi faktisk foretaget os backpropagation, da vi har bevæget os tilbage igennem vores regnestykke og fundet ud af, hvor stor en betydning vores variable har på vores resultat. Man foretager sig næsten det samme når man laver matrix multiplikation, som man også kan opstille som en computational graph. Så vi vælger at opstille et 2-lag neuralt netværk, som en computational graph.

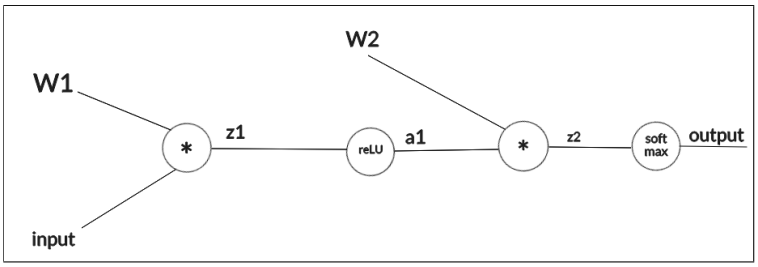

Figure 6: En computational graph af et 2 lag neuralt netværk med softmax loss og relu funktion

Hvis vi skulle beregne W1's vægte ville vi skulle tage den partielt afledtede af z1 med hensyn til W og gange det med den partielt afledede af outputtet med hensyn til z1. Så vil vi finde W1 betydning for outputtet(tabet) og justere W1 således, at tabet formindskes.

5.5 Convolutional neural networks

Som nævnt tidligere bruger man convolutional neural networks til ansigtsgenkendelse og andre klassifikationsopgaver hvor man behandler billeder. Årsagen til dette er, at man

mister en semantisk forståelse når man bare konverterer sit billede til en kolonne-vektor med antallet af pixels i billedet. Der er altså ingen sammenhæng imellem pixels i vektoren, som der i billedet.

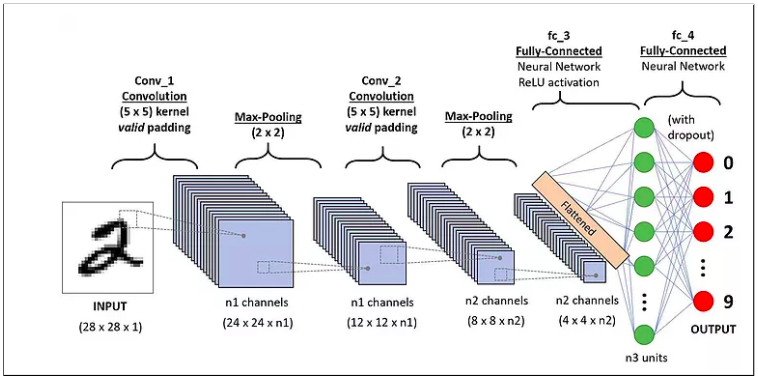

Figure 7: En visualisering af et convolutional neural netværk

Det man gør i et convolutional neural netværk er vi beholder billedets format og lader det være en $N \times Mmatrix$. Derefter foretager vi os convolutions på netværket. Det man gør her er, at man tager et antal "filtre/kernels" og lader det passere over billedet. En kernel er $N \times N$ matrix hvor hvert element i matrixen er en vægt. Måden den fungerer på er ved at sætte den et sted på billedet og lave matrix multiplikation imellem billedet og kernelen. Forståelsen for, hvad kernelen gør er, at den samler noget information et sted i billedet og sender denne information videre til det næste lag. Forneden kan en vi se en visuel repræsentation af hvad en kernel gør.

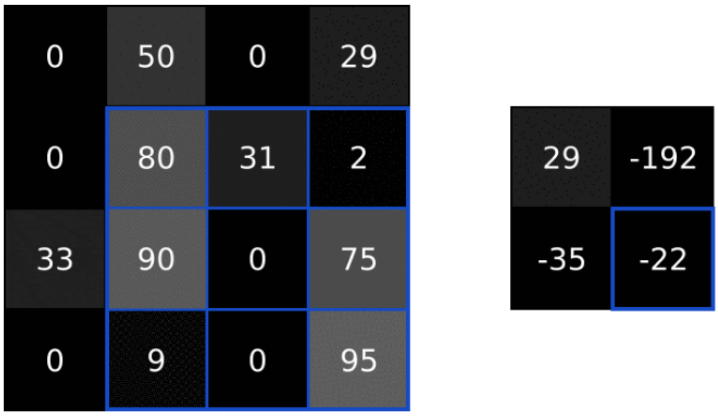

Figure 8: En 3x3 kernel anvendt på et 4x4 matrix

Ovenfor til venstre kan der ses et 4x4 billede repræsenteret med nogle pixel værdier. Det blå omrids er kernelens nuværende position og billedet til højre outputtet af kernelen ved det specifikke sted. Det der er gjort er, at man har lavet matrix multiplikation imellem kernelen og værdierne på kernelens nuværende position. Hvorefter man har summeret alle værdierne i kernelen og repræsenteret den i outputtet, hvilket kan ses til højre. Man vil altså få et

output med størrelsen 2x2, hvor hvert element i outputtet er summen af værdier af matrix multiplikationen imellem kernelens vægte og pixel værdierne ved kernelen placeret på unik position på 4x4 pixel matrixen.

6 Implementering af mit convolutional neural network

Nu er de helt centrale principper redegjort for, så vi kan gå i gang med implementeringen. Jeg har valgt at foretage implementeringen i python, da numpy tillader hurtige matematiske operationer på vektorer. Det skal her nævnes, at min nuværende arkitektur af netværket laver chiffergenkendelse og ikke ansigtsgenkendelse. Det skyldes at træningstiden af et netværk, der skal foretage sig ansigtsgenkendelse ligger på flere tusinde timer[6].

```
ConvolutionalNet = CNN.ConvNet([["conv", 1, 8, 3, 1, 1], ["MaxPool"], ["
    FCLayer", 13 * 13 * 8, 10, "softmax"]]) #Creates a convolutional neural
     network object
```

Listing 1: Convolutional network object

Jeg starter med at lave et objekt fra min ConvNet klasse med arkitekturen givet som parametre til vores kunstruktør.

```
class ConvNet:
    def __init__(self, architechture):
        self.layers = [] #Initializes empty list of layers
        for layer in architechture: #Iterates over each layer in
    architechture lsit
            if(layer[0] == "conv"): #If the first element in the layer,
    which is a list, is "conv" then it creates a convolutional neural
    network
                self.layers.append(ConvLayer(layer[1],layer[2], layer[3],
    layer[4], layer[5]))
            elif(layer[0] == "MaxPool"): #Does exactly the same just for
    maxpool layer
                self.layers.append(MaxPoolLayer())
            elif(layer[0] == "FCLayer"): #Same as above
                self.layers.append(FCLayer(layer[1], layer[2], layer[3]))
```

Listing 2: ConvNet class

Det min konstruktørfunktion foretager sig er, at itererer over alle elementerne i listen architechture, som er en liste af lister, hvor elementerne heri er de individuelle lags nødvendige variabler for opsætningen af deres struktur. Til dette kigger vi på det første element i listen for laget, hvorefter man opretter et lag, som sættes for enden af det hidtidige sidste lag i vores liste layers. De tre konkrete lag vi kan oprette er et convolutional lag, et MaxPool lag, som bare skalerer vores convolutional lag ned i en mindre størrelse, og til sidst et softmax lag.

```
    def train(self):
        correctly_classified = 0 #The number of images correctly
    classified within the last 100 examples
```

```
        loss = 0 #The sum of the loss within the last 100 examples
        for training_example, (image, label) in enumerate(zip(train_images
    , train_labels)): #Runs over each training in example in the MNIST
    dataset
            if training_example % 100 == 99: #If we have completed the
    training of a 100 examples print out the accuracy and the average loss
                print(
                    '[Training example %d] The last 100 training examples:
     The average loss was %.3f | Accuracy:%d%%' % (training_example + 1,
    loss / 100, correctly_classified) #Prints out the average loss and the
    accuracy
                )
                loss = 0 #Resets the loss for the next 100 examples
                correctly_classified = 0 #Resets the number of correctly
    for the next 100 examples
            l_i, acc = self.backprop(image, label) #Computes the
    backpropagation of of the i'th example
            loss += l_i #Adds the loss of the example to the current loss
            correctly_classified += acc #Adds 1 if the acc was one which
    is calculated in the backpropagation function.
```

Listing 3: train metode i vores ConvNet klasse

Vi starter med at initialisere to variable, der henholdsvis skal holde styr på hvad vores samlede tab er for de 100 træningseksempler og hvor mange vi har klassificeret rigtigt. Derefter iterere vi over alle vores træningseksempler og skriver til consolen, hvad det gennemsnitslige tab var og hvor stor en præcision netværket havde. Vi sætter disse variabler til 0, så de kan bruges til de næste 100 eksempler. For hvert eksempel laver vi backpropagation, som giver os tabet af træningseksemplet og hvorvidt den klassificerede det rigtigt. Vi adderer dette til vores to variabler, der holder styr på tabet og antallet af rigtigt klassificerede.

```
    def backprop(self, x, y):
        loss, acc = self.forward(x, y) #Feeds forward the input
    throughtout the layers
        i = len(self.layers) - 1 #A variable that indicates our current
    layer in the backpropagation process.
        dX = np.zeros(10) #is initialized with a random placeholder value
        while (i >= 0): #Iterates backward through the layers, which is
    also clear in the name backpropagation
            dX = self.layers[i].backward(dX, y) #Calculates the input
    gradient, which is the upstream gradient the next layer needs to
    calculates its gradient with respects to the loss
            i -= 1 #Decreases the value of i
        return loss, acc #returns the loss and the accuracy of the
    training example
```

Listing 4: backpropagation metode i vores ConvNet klasse

Vi starter her med at kalde vores forward funktion, der lader inputtet passere igennem vores lag og som beregner et output og et tab på dette output. Derudover får vi også at vide hvorvidt vores netværk klassificerede vores eksempel korrekt med hensyn til vores output label y. Derefter initialiseres variablen i til at være det sidste lag i vores netværk hvorefter vi opretter en placeholder value for dX, som er den såkaldte upstream gradient, som den lokale gradient skal skaleres med. Denne placeholder værdi bruges overhovedet ikke i det

sidste lag i netværket(første lag i backpropagationen), da den partielt afledte af tabet med hensyn til tabet, er 1. Vi kalder vores backward metode på vores i'ende lag og formindsker derefter i med 1. Til sidst returnerer vi tabet og hvorvidt vi klassificerede rigtigt eller forkert. Vi bevæger os nu videre til hvordan selve "feed forward" processen fungerer, hvorefter vi beskriver backward processen.

```
    def forward(self, image, label):
        out = (image/255)-0.5 #Scales our input values to an interval between -0.5 and 0.5
        for layer in self.layers: #iterates through each layer and calculates the output of the layer and sends this output as input to the next layerl.
            out = layer.forward(out)
        loss = -np.log(out[label]) #Calculates the loss of the network on the training example
        acc = 0
        if(np.argmax(out) == label): #Checks whether position the highest output was equal to the label. If True then the accuracy on the training example was 100%
            acc = 1
        return loss, acc
```

Listing 5: forward metode i vores ConvNet klasse

Vi starter med at skalere vores pixel værdier til et interval imellem -0.5 og 0.5. Derefter itererer vi over vores lag og beregner outputtet af et lag, for dernæst at sende det videre som input til det næste. Herefter beregnes tabet ved at tage den negative logaritme af sandsynligheden af den korrekte klassificering. Dernæst kigger vi på, hvorvidt den største værdi i vores output(som er en sandsynligheds vector) var den klassificering, som vi forventede og hvis det var, sætter vi vores acc variabel lig 1, da den hermed foretog den rigtige klassifikation. Til sidst returnerer vi tabet og vores acc variabel.

```
class ConvLayer:
    def forward(self, x):
        self.x = x #Gets the input and stores it for later use
        self.height, self.width = x.shape #Gets the height and width of our input
        self.filters = Filters(self.S, self.F, self.K) #Creates a filter matrix with a depth of K aka the amount of filters
        output = np.zeros(((int)((self.height-self.F)/1)+1, (int)((self.width-self.F)/1)+1, self.K)) #We use a formula that calculates the size of the output by using the width and the height and our filter size
        for i in range((int)((self.height-self.F)/1)+1): #We iterate in a range of the output matrix columns
            for j in range((int)((self.width - self.F) / 1) + 1): #We iterate in a range of the output matrix rows
                position = x[i:(i+self.F), j:(j+self.F)] #We get the position current position of our filter/kernel on our input
                output[i, j] = np.sum(position*self.filters.filters, axis = (1, 2)) #We do a matrix multiplication between the kernel and its current location on the input and sum the values of the elements.
        return output #Returns the output
```

Listing 6: forward metode for vores convolutional lag

Vi starter med at gemme vores input x, da vi skal bruge det til backpropagation senere. Dernæst tager vi højden og længden af vores input, hvilket skal bruges til at udregne størrelsen på vores matrix i det næste lag. Dernæst opretter vi et filter ved hjælp af en Filter klasse, som kan ses i bilaget for den samlede kode. Vi opretter en nulmatrix output med en størrelse, som vi beregner hjælp af en formel, der beskriver hvad størrelsen af outputtet er med en bestemt størrelse kernel filter F. Dernæst vil vi gerne beregne hvert element i vores output. Dette gøres ved at lave et for-loop, der itererer over rækkerne og et "nested" for-loop over kolonnerne. Vi laver en variabel, som vi kalder position, som vi sætter lig med værdierne af vores input x ved kernelens nuværende position på x. Derefter tager vi matrix produktet imellem de førnævnte værdier og vores kernel, og tager summen af elementerne i resultatet fra førnævnte beregning hvorefter vi tildeler det i,j element i output matrixen med resultatet efter summeringen. Dette gøres for alle vores output værdier og til sidst returnerer vi outputtet. Det næste lag i vores netværk er et maxpool lag, men koden vil jeg ikke beskrive i detaljer, da den bare nedskalerer outputtet af laget forinden. Derfor springer vi direkte til vores softmax lag, som er et næsten ordinært lag i et neuralt netværk.

```
class FCLayer:
    def forward(self, x):
        self.last_x = x.shape #Gets the shape of the input, which is used later in backprop
        self.x = x.flatten() #Makes the input a column vector
        self.z = np.dot(self.x, self.w)+self.biases #Calculates our z value by taking the dot product/matrix multiplication between our input and our weight matrix. Then adds a bias.
        if(self.activation_func == "sigmoid"): #Is it a sigmoid function
            self.a = sigmoidGate(self.z) #Activation is calculated with a sigmoid gate
        elif(self.activation_func == "softmax"): #Is it a softmax function
            self.a = softmaxGate(self.z) #calculates activation
        return self.a #returns activation
```

Listing 7: forward metode for vores fully-connected lag

Vi starter med at gemme dimensionerne af vores input, da det skal bruges i backpropagationen. Derefter gør vi vores input til en kolonne vector, da vi kun arbejder på 1 dimensionelle inputs i fully-connected lag. Herefter tag vi prik produktet/matrix multiplication imellem vores matrix w og vores input x, hvorefter der lægges en bias til. Matrix w og bias vectoren blev initialiseret i konstruktør funktion for laget. Jeg har også implementeret en anden non-lineær funktion, nemlig sigmoid funktionen, således at man kan vælge imellem enten at have sigmoid funktion, som vores non-lineære funktion eller en softmax funktion. Med denne if-else struktur vil man let kunne implementere yderligere non-lineære funktioner, såsom en reLU. I dette tilfælde beregner vi dog vores activation med en softmax funktion og returner outputtet fra softmax funktionen, som det endegyldige output for laget.

```
def softmaxGate(z):  #L is the loss vector
    exp = np.exp(z) #Calculated a vector scores for each class
    return exp/np.sum(exp, axis=0) #Calculated the probability of each class by normalizing with the total score
```

Listing 8: Softmax funktion

Vi beregner i vores softmax funktion først en score vector, hvorefter vi normaliserer denne score vector med summen af score vectoren, så vi altså får en sandsynligheds vector for de forskellige klasser. Nu har vi beregnet outputtet af vores netværk og vil nu gerne justere vores vægte så netværket opnår bedre resultater. Så vi vender tilbage til vores backpropagation, som starter med det sidste lag, altså softmax laget.

```
1     def backward(self, dA, y):
2         elif(self.activation_func == "softmax"):
3             self.dZ = softmaxGrad(self.a, y) #Calculates the softmax
    gradient
4             self.dW = np.dot(self.x[np.newaxis].T, self.dZ[np.newaxis]) #
    Calculates the weight gradient getting local gradient which is the
    input transposed and scaling it by the upstream gradient dZ
5             self.dB = self.dZ #The bias impact turns out to be 1*
    upstreamgradient
6             self.dX = np.dot(self.dZ, self.w.T)
7             self.w = self.w -0.005*self.dW #Adjusts the weight with a
    learning rate of 0.005.
8             self.biases = self.biases - 0.005*self.dB #Adjusts the biases
    with a learning rate of 0.005.
9             self.dX = self.dX.reshape(self.last_x) #Reshapes the input
    gradient so it matches the shape of our previous layer
10        return self.dX
```

Listing 9: Backward funktion i FCLayer

Vi starter her med at tjekke, hvorvidt vi har at gøre med en softmax funktion, hvilket vi har. Derfor beregner vi først softmax gradienten. Da den er forbundet direkte med outputtet behøves kædereglen ikke. Derefter beregner vi den partielt afledede af outputtet af vores funktion med hensyn til vægtene ved at gange den lokale gradient af dZ med hensyn til vægtene, som viser sig at være inputtet transponeret, med opstrøms gradienten dZ. Derefter beregnes den partielt afledede af outputtet med hensyn til biasene, hvilket viser sig bare at være 1*opstrømsgradienten. Til sidst beregnes input gradienten, hvor det viser sig, at den lokalt partielt afledede af z med hensyn til x er w matrixen transponeret hvorefter vi har ganget opstrømsgradienten på, som kædereglen fortæller os. Gradienten af inputtet sendes videre til det foregående lag i det korrekte form, da laget skal bruge denne, som sin opstrøms gradient. Vi springer dog igen over MaxPool laget og fortsætter videre til vores convolutional lag.

```
1     def backward(self, dA, y):
2         dF = np.zeros(self.filters.filters.shape) #Creates a matrix of the
     size of the filters.
3         for i in range((int)((self.height - self.F) / 1) + 1): #We iterate
     in a range of the output matrix columns
4             for j in range((int)((self.width - self.F) / 1) + 1): #We
    iterate in a range of the output matrix rows
5                 for k in range(self.K): #We iterate over each individual
    filter in our filter weight matrix
6                     position = self.x[i:(i + self.F), j:(j + self.F)] #We
    do a matrix multiplication between the kernel and its current location
    on the input and sum the values of the elements.
7                     dF = dA[i, j, k] * position #Takes the local gradient
    which is equivalent to the position and takes the matrix product
```

```
    between it and the upstream gradient dA
8         self.filters.filters = self.filters.filters-0.005*dF #Readjusts
    the filters weights
9         return None
```

Listing 10: Backward funktion i Convolutional Lag

Vi starter med at initialisere en matrix, som er vores gradient for vores filtre/kernels. Vi har her tre for-loops, der henholdvis løber over rækkerne og kolonnerne i outputtet og filter matrixens dybde altså antallet af forskellige filtre. Den lokale gradient er lig inputtets position når kernelen befinder sig på positionen, hvilket skaleres med opstrøms gradienten, som er lig input gradienten fra laget tidligere i backpropagation processen. Vi finder ved at itererer igennem, hvilken betydning de k filtre har for vores output og justere derefter filterets vægte. Til sidste returnerer vi none, da det er vores første lag i vores convolutional neural network og input gradienten altså ikke skal bruges videre i forløbet. Nu skal vi så træne vores netværk og vurdere hvor godt det klarer sig på noget test data.

7 Diskussion og konklusion

Vi træner vores netværk til at genkende tal fra 0 til 9 med 10000 træningseksempler og derefter tester vi vores netværk på noget ukendt data. Med en learning rate på 0.005 fik jeg følgende resultater på mit testdata:

```
[On 1000 test examples] The average loss was 0.448 | Accuracy 86.4%
```

Figure 9: Performance af netværk på ukendt data

Dette er ok, men man kan opnå langt højere præcision. Så hvordan kunne vi forbedre dette netværk og forhøje præcisionen? Hvis man vil ændre udkommet af et netværk, er der bestemte variable man kan skrue på, disse kaldes hyperparametrerne. Hyperparametrene til et netværk er de variabler, som er fastsat inden træning. Det er fx. learning raten, som angiver hvor stort et skridt man vælger at tage i gradientens retning. Derudover er det også parametre såsom den basale arkitektur af netværket, antallet af neuroner i lagene og hvornår man vælger at opdatere sine vægte. Vi justerer i vores netværk vægtene efter hvert enkelt træningseksempel, men når man træner netværkene i virkeligheden vil man opdele ens træningsdata i såkaldte minibatches. Disse minibatches indeholder en eller anden mængde træningseksempler som ved konvention er et base 2 tal. Man summerer derfor gradienterne af vægtene for alle træningseksemplerne og tager gennemsnittet af disse, hvorefter man justerer vægtene. Dette gøres for at simulere det samlede tab for at kunne justere vægtene mere præcist. Det kan for eksempel være, at vi i vores netværk har justeret en vægt i en matrix, der egentlig var i orden på længere sigt[3].

Lidt tidligere introducerede jeg hyperparametrene og definitionen af dem. Der er to forskellige måder hvorpå, at man indstiller disse parametre således at ens netværk klarer sig godt. Den første metode er, at man inddeler alt sit data i tre dele: træningsdata, validationsdata og test data. Man træner sit netværk med nogle valgte hyperparametre og vurderer hvorvidt

disse er gode, ved at lade netværket køre på validationsdataet og se hvor godt de klarer sig på dette. Man vælger derfor de hyperparametre, der klarer sig bedst på validationsdataet. Derefter kører netværket på test data'en, hvilket er dit endegyldige resultat på hvor godt dit netværk klarede sig. Årsagen til, at man ikke kigger på hvilke hyperparametre, der klarer sig bedst på test-dataet er fordi, at man så bevidst "fitter" på test-dataet, hvilket ikke er en korrekt videnskabelig tilgang. Dermed har man ingen viden om hvordan netværket vil klare sig på nyt data[3].

En anden måde man kan bestemme nogle gode hyperparametre på er med metoden cross-validation. Her deler du dit dataset op i træningsdata og testdata. Herefter deler du dit træningsdata op i fem folde hvor en af disse folde agerer validationsdata. Man træner herfra sit netværk med nogle bestemte hyperparametre på 4 af foldende og validerer hyperparametrene på den sidste fold. Man roterer hvilken fold, der fungerer validationsdata og får en samlet vurdering for de pågældende hyperparametre. Dette gøres for en række forskellige hyperparametre og man afgør hvilken selektion af hyperparametre, der klarede sig bedst på alle foldende og de forskellige rotationer af validationsfolden. Denne metode er især brugt på mindre dataset, da det tager kræver en del tid at foretage denne process.

Så hvad kan vi konkludere? Ansigtsgenkendelse er en algoritme, der konstrueres med matematik og implementeres med programmering. Matematikken er kompleks, men modulær og netværket skal være stort hvis man gerne vil have ansigtsgenkendelse med nogenlunde resultater. Derfor er det nødvendigt, at man træner sit netværk med computere, der kan foretage sig mange "floating point operations per second[6]". På trods af, at ansigtgenkendelse ikke er muligt, at implementere med min computer, ville min kode - dette netværk - let kunne skaleres, fordi koden er modulær, hvilket giver mulighed for at eksperimentere med netværkets arkitektur. Vi har dog fået vores netværk til at løse chiffergenkendelse klassifikation med en forholdsvis god præcision. Ydermere har vi lært nogle relevante metoder, der bruges indenfor machine learning industrien til at træne neurale netværk og vælge de rigtige hyperparametre. Vi kan her på falderebet stille os selv spørgsmålet om, hvorvidt vi programmører og matematikere, burde tage stilling de konsekvenser vores teknologier eventuelt kan forårsage? Skal vi lade de store tech-giganter og staten have adgang til disse teknologier, der potentielt set kunne indskrænke privatlivet for den enkelte?

Litteraturliste

[1] Wiki authors. Matrix. https://en.wikipedia.org/wiki/Matrix_(mathematics), 2019. [En længere beskrivelse af en matrix samt dets egenskaber].

[2] Jesper Buch. Programmering. https://programmering.systime.dk/index.php?id=230, 2017. [Beskrivelse af programmerings metoders].

[3] Justin Johnson og Serena Young Fei-Fei Li. Lecture 3 — Image Classification. https://www.youtube.com/watch?v=0oUX-nOEjG0&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=2, 2017. [Video; set undervejs i projektforløbet].

[4] Justin Johnson og Serena Young Fei-Fei Li. Lecture 3 — Loss functions Optimization. https://www.youtube.com/watch?v=h7iBpEHGVNc&t=2439s, 2017. [Video; set undervejs i projektforløbet].

[5] Justin Johnson og Serena Young Fei-Fei Li. Lecture 4 — Introduction to Neural Networks. https://www.youtube.com/watch?v=d14TUNcbn1k&t=2905s, 2017. [Video; set undervejs i projektforløbet].

[6] Dmitry Kalenichenko og James Philbin Florian Schroff. FaceNet: A Unified embedding for Face Recognition og clustering. https://arxiv.org/pdf/1503.03832.pdf, Årstal: Ukendt. [En rapport om face recognition].

[7] Math insigt. Matrix multiplikation. https://mathinsight.org/matrix_vector_multiplication, 2019. [En beskrivelse af matrix multiplikation].

[8] Math is fun. Partial derivatives. https://www.mathsisfun.com/calculus/derivatives-partial.html, 2019. [Beskrivelse af partielt afledte].

[9] John Schødt Pedersen Niels Padkjær Pedersen og Peter Hansen. Klaus Marthinus, Michael Jensen. Mat B htx. https://matbhtx.systime.dk/index.php?id=95, 2017. [bog; læst undervejs i projektforløbet].

[10] Line Holst og Anders Damsgaard. Almen studieforbederelse. https://at.systime.dk/, 2008. [Beskrivelse matematik fagets metoder].

[11] Peter Føge og Bonnie Hegner. Primus. https://primus.systime.dk/index.php?id=frontpage, 2009. [Beskrivelse matematik fagets metoder].

[12] Yang S. Build up a Neural Network with python. https://towardsdatascience.com/build-up-a-neural-network-with-python-7faea4561b31, 2019. [Billede fra artikel].

Bilag

Bilag 1: ConvNet.py

```
    import numpy as np
import mnist
train_images = mnist.train_images()[:10000]
train_labels = mnist.train_labels()[:10000]
test_images = mnist.test_images()[:1000]
test_labels = mnist.test_labels()[:1000]
def softmaxGate(z):  #L is the loss vector
    exp = np.exp(z) #Calculated a vector scores for each class
    return exp/np.sum(exp, axis=0) #Calculated the probability of each class by normalizing with the total score
def softmaxGrad(a, y):
    for i in range(a.size):
        if(i == y):
            a[i] -= 1
            return a
def sigmoidGate(z):
    return 1.0 / (1.0 + np.exp(-z))

def sigmoid_derivative(z):
    return sigmoidGate(z)*(1-sigmoidGate(z))

class FCLayer: #Fully connected layer
    def forward(self, x):
        self.last_x = x.shape  # Gets the shape of the input, which is used later in backprop
        self.x = x.flatten()  # Makes the input a column vector
        self.z = np.dot(self.x, self.w) + self.biases  # Calculates our z value by taking the dot product/matrix multiplication between our input and our weight matrix. Then adds a bias.
        if (self.activation_func == "sigmoid"):  # Is it a sigmoid function
            self.a = sigmoidGate(self.z)  # Activation is calculated with a sigmoid gate
        elif (self.activation_func == "softmax"):  # Is it a softmax function
            self.a = softmaxGate(self.z)  # calculates activation
        return self.a  # returns activation
    def backward(self, dA, y):
        if(self.activation_func == "sigmoid"):
            self.dZ = sigmoid_derivative(self.z)*dA #Calculates the local gradient by taking the derivative of the sigmoid with respects to the output z.
            self.dW = self.last_x[np.newaxis] @ self.dZ[np.newaxis] # Calculates the local weight gradient
```

```
            self.dB = self.dZ
            self.dX = np.dot(self.w.T, self.z) #Caculates the local input gradient
            return self.dX

        elif(self.activation_func == "softmax"):
            self.dZ = softmaxGrad(self.a, y)
            self.dW = np.dot(self.x[np.newaxis].T, self.dZ[np.newaxis]) #Calculates the weight gradient
            self.dB = self.dZ
            self.dX = np.dot(self.dZ, self.w.T)
            self.w = self.w -0.005*self.dW
            self.biases = self.biases - 0.005*self.dB
            self.dX = self.dX.reshape(self.last_x)
        return self.dX

    def __init__(self, prev_neurons, neurons, activation_func): #Opstning er korrekt
        self.neurons = neurons
        self.activation_func = activation_func
        self.prev_neurons = prev_neurons
        self.w = np.random.randn(prev_neurons, neurons)/prev_neurons
        self.biases = np.zeros(neurons)
class ConvLayer:
    # K is number of filters, F is the filters spacial extent F*F*D, S is the stride aka interval which you pass filter over, P is the amount of zeropadding
    def __init__(self, depth, K, F, S, P):
        self.depth = depth
        self.K = K
        self.F = F
        self.S = S
        self.P = P
    def forward(self, x):
        self.x = x #Gets the input and stores it for later use
        self.height, self.width = x.shape #Gets the height and width of our input
        self.filters = Filters(self.S, self.F, self.K) #Creates a filter matrix with a depth of K aka the amount of filters
        output = np.zeros(((int)((self.height-self.F)/1)+1, (int)((self.width-self.F)/1)+1, self.K)) #We use a formula that calculates the size of the output by using the width and the height and our filter size
        for i in range((int)((self.height-self.F)/1)+1): #We iterate in a range of the output matrix columns
            for j in range((int)((self.width - self.F) / 1) + 1): #We iterate in a range of the output matrix rows
                position = x[i:(i+self.F), j:(j+self.F)] #We get the position current position of our filter/kernel on our input
                output[i, j] = np.sum(position*self.filters.filters, axis = (1, 2)) #We do a matrix multiplication between the kernel and its current location on the input and sum the values of the elements.
        return output #Returns the output
    def backward(self, dA, y):
```

```
        dF = np.zeros(self.filters.filters.shape) #Creates a matrix of the size of the filters.
        for i in range((int)((self.height - self.F) / 1) + 1): #We iterate in a range of the output matrix columns
            for j in range((int)((self.width - self.F) / 1) + 1): #We iterate in a range of the output matrix rows
                for k in range(self.K): #We iterate over each individual filter in our filter weight matrix
                    position = self.x[i:(i + self.F), j:(j + self.F)] #We do a matrix multiplication between the kernel and its current location on the input and sum the values of the elements.
                    dF = dA[i, j, k] * position #Takes the local gradient which is equivalent to the position and takes the matrix product between it and the upstream gradient dA
        self.filters.filters = self.filters.filters-0.005*dF #Readjusts the filters weights
        return None



class MaxPoolLayer:
    def forward(self, x):
        self.x = x #Gets the input and stores it for later use
        self.height, self.width, self.depth = x.shape
        self.newH = self.height // 2
        self.newW = self.width // 2
        output = np.zeros((self.height // 2, self.width // 2, self.depth))
        for i in range(self.newH):
            for j in range(self.newW):
                position = x[(i * 2):(i*2+2), (j * 2):(j * 2 + 2)]
                output[i, j] = np.amax(position, axis=(0, 1))
        return output
    def backward(self, dA, y):
        dX = np.zeros(self.x.shape)
        for i in range(self.newH):
            for j in range(self.newW):
                position = self.x[(i * 2):(i * 2 + 2), (j * 2):(j * 2 + 2)]
                height, width, depth = position.shape
                amax = np.amax(position, axis=(0, 1))
                for i2 in range(height):
                    for j2 in range(width):
                        for f2 in range(depth):
                            if position[i2, j2, f2] == amax[f2]:
                                dX[i*2 +i2, j*2+j2, f2] = dA[i, j, f2]
        return dX
class Filters:
    def __init__(self, S, F, num_filters):
        self.filters = np.random.rand(num_filters, F, F)/(F*F)
        self.stride = S
class ConvNet:
    def __init__(self, architechture):
        self.layers = [] #Initializes empty list of layers
```

```
117         for layer in architechture: #Iterates over each layer in
    architechture lsit
118             if(layer[0] == "conv"): #If the first element in the layer,
    which is a list, is "conv" then it creates a convolutional neural
    network
119                 self.layers.append(ConvLayer(layer[1],layer[2], layer[3],
    layer[4], layer[5]))
120             elif(layer[0] == "MaxPool"): #Does exactly the same just for
    maxpool layer
121                 self.layers.append(MaxPoolLayer())
122             elif(layer[0] == "FCLayer"): #Same as above
123                 self.layers.append(FCLayer(layer[1], layer[2], layer[3]))
124     def forward(self, image, label):
125         out = (image/255)-0.5 #Scales our input values to an interval
    between -0.5 and 0.5
126         for layer in self.layers:
127             out = layer.forward(out)
128         loss = -np.log(out[label]) #Calculates the loss of the network
129         acc = 0
130         if(np.argmax(out) == label): #Checks whether position the highest
    output was equal to the label. If True then the accuracy on the
    training example was 100%
131             acc = 1
132         return loss, acc
133     def backprop(self, x, y):
134         loss, acc = self.forward(x, y)
135         i = len(self.layers) - 1
136         dX = np.zeros(10)
137         while (i >= 0):
138             dX = self.layers[i].backward(dX, y)
139             i -= 1
140         return loss, acc
141     def train(self):
142         correctly_classified = 0
143         loss = 0
144         for training_example, (image, label) in enumerate(zip(train_images
    , train_labels)):
145             if training_example % 100 == 99:
146                 print(
147                     '[Training example %d] The last 100 training examples:
     The average loss was %.3f | Accuracy:%d%%' % (training_example + 1,
    loss / 100, correctly_classified)
148                 )
149                 loss = 0
150                 correctly_classified = 0
151             l_i, acc = self.backprop(image, label)
152             loss += l_i
153             correctly_classified += acc
154     def test(self):
155         loss = 0
156         correctly_classified = 0
157         for test_example, (image, label) in enumerate(zip(test_images,
    test_labels)):
158             l_i, acc = self.forward(image, label)
```

```
159             loss += l_i
160             correctly_classified += acc
161         print('[On %d test examples] The average loss was %.3f | Accuracy
    ' % (test_labels.size, loss/test_labels.size) +(str)(
    correctly_classified*100/test_labels.size)+'%')
```

Listing 11: Min ConvNet.py fil

Bilag 2: program.py

```
1 import ConvNet as CNN
2 import mnist_loader
3 training_data, validation_data, test_data = \
4 mnist_loader.load_data_wrapper()
5 ConvolutionalNet = CNN.ConvNet([["conv", 1, 8, 3, 1, 1], ["MaxPool"], ["
    FCLayer", 13 * 13 * 8, 10, "softmax"]])
6 ConvolutionalNet.train()
7 ConvolutionalNet.test()
```

Listing 12: Filen hvori jeg kører mit program