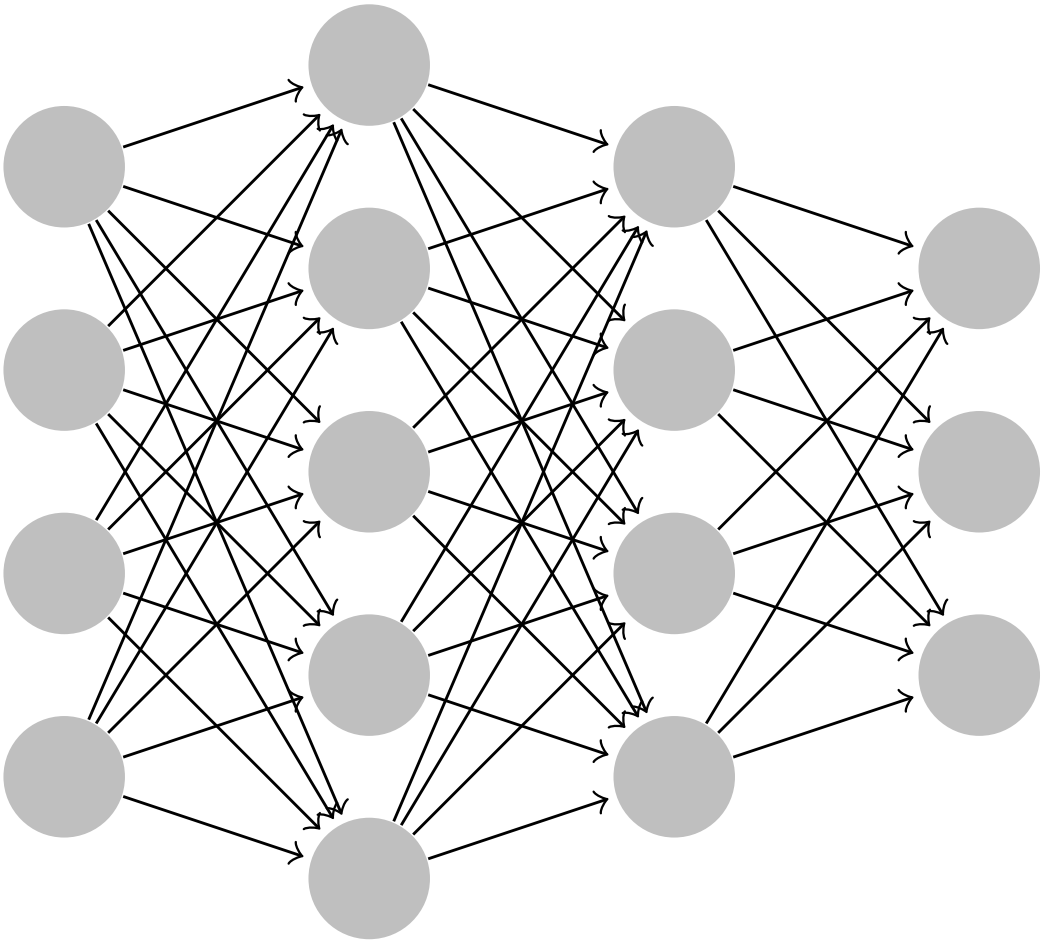
Machine Learning

Markus Ketilsø Dam
under vejledning af [SLETTET] og [SLETTET]

17. december 2020



Titelblad

Opgaveformulering

Der ønskes en redegørelse for hvad konceptet Machine Learning dækker over, med særligt fokus på neurale netværker.
Med udgangspunkt i et konkret eksempel ønskes en analyse af hvordan et kunstigt neuralt netværk er opbygget. Herunder skal centrale dele af den matematik der ligger bag kunstige neurale netværk beskrives.
Lav et program, der implementerer en algoritme, der illustrer hvordan et konkret datasæt kan behandles i et neuralt netværk. Kommentér hvilke matematiske og programmeringsmæssige overvejelser der er gjort undervejs. Vurdér betydningen af Machine Learning i moderne teknologier.

Omfang

Antal tegn (eksl. forside, indholdsfortegnelse, resume, figurer, bilag, referencer og fodnoter): 24501
Sider matematik: 5
I alt: 15.2 normalsider

Resume

Opgaven omhandler Machine Learning, en gren af datalogien som beskæftiger sig med at bruge store mængder af data til at finde mønstre automatisk. Der bliver i opgaven redegjort for de grundlæggende principper bag den lineære algebra, samt partiel differentiering, som ligger til grund for Machine Learning. Herefter bliver der redegjort for hvad et neuralt netværk er, hvordan dets output kan fortolkes, samt kendetegnene af to af de mest almindelige problemer en Machine Learning model kan støde på. Der bliver desuden redegjort for egenskaberne af et feedforward neuralt netværk, og hvordan forskellige parametre og datarepræsentationer i et neuralt netværk vælges. Efterfølgende bliver den specifikke algoritme for at træne et feedforward neuralt netværk udledt, ved hjælp af partiel differentiering af funktioner med flere inputs. Der bliver herefter implementeret et neuralt netværk i Python, som trænes på et datasæt af håndskrevne tal, hvor det opnår en præcision på ca. 92.6%.

Indhold

1 Indledning **1**

2 Problemformulering **1**

3 Grundlæggende lineær algebra **1**
3.1 Vektorer 1
3.2 Matricer 3
3.3 Funktioner 4

4 Introduktion til Machine Learning **5**
4.1 Motivation 5
4.2 Det neurale netværks egenskaber 6

5 Design af et neuralt netværk **7**
5.1 Datarepræsentationer 7
5.2 Outputfunktioner 9
5.3 Vægte 11
5.4 Potentielle problemer 11
5.4.1 Underfitting 11
5.4.2 Overfitting 11

6 Træning af et neuralt netværk **12**
6.1 Kostfunktionen 12
6.2 Gradient nedstigning 12
6.3 Udledning af træningsreglen 13
6.3.1 Outputvægte 14
6.3.2 Skjulte vægte 15
6.4 Backpropogation 16

7 Implementation **16**
7.1 Numpy 17
7.2 Programstrukur 17
7.3 Resultat 22

8 Anvendelse af neurale netværk **22**
8.1 Neurale netværk i lægekundskaben 23

9 Konklusion **23**

A Bilag **25**

A1 Kildekode til neuralt netværk 25

1 Indledning

Computere er bedre end mennesker til mange ting, men der er også mange ting som mennesker kan gøre let, mens det nærmest virker umuligt for en computer. Mennesker kan for eksempel let genkende et håndskrevet ciffer, mens det ville være svært at lave et program til at gøre det samme. Machine Learning er den gren af datalogien, som arbejder med disse problemer. Løsningen til dem er inspireret af måden som mennesker selv lærer at gøre disse ting på: Ved at se en masse eksempler, og ud fra disse udarbejde et komplekst regelsæt, som kan bruges til at løse problemet. Der vil i denne opgave arbejdes med en bestemt gren af Machine Learning kaldet *neurale netværk*, som er inspireret af den menneskelige biologi.

2 Problemformulering

Der vil i denne opgave arbejdes med problemstillingen "Hvordan kan et neuralt netværk til billedgenkendelse implementeres?". I besvarelsen af denne problemstilling, vil der blive svaret på følgende underspørgsmål:

- Hvad er lineær algebra?
- Hvad er et neuralt netværk?
- Hvordan repræsenteres billeder i et neuralt netværk?
- Hvilke problemer kan en Machine Learning model have?
- Hvordan trænes et feedforward neuralt netværk?
- Hvordan kan neurale netværk anvendes i moderne teknologier?

3 Grundlæggende lineær algebra

Lineær algebra er læren om *vektorer* og *matricer*. Det antages, at vektorer er velkendte, i form af *geometriske vektorer* som ofte bruges til at repræsentere punkter i planet eller rummet. Geometriske vektorer skrives som regel med en pil over bogstavet, f.eks. $\vec{v}$.

3.1 Vektorer

I den lineære algebra benyttes der dog et bredere vektorbegreb: En samling af n værdier. Dette skrives også som vektorer er elementer af $\mathbb{R}^n$. En vektor

er altså en vilkårlig lang række af reelle tal. Disse skrives som regel med et lille, fedt bogstav:

$$\boldsymbol{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \in \mathbb{R}^4$$

Generelle vektorer følger generelt de samme regneregler som geometriske vektorer. De kan lægges sammen:

$$\begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ \vdots \end{bmatrix} + \begin{bmatrix} a_2 \\ b_2 \\ c_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} a_1 + a_2 \\ b_1 + b_2 \\ c_1 + c_2 \\ \vdots \end{bmatrix}$$

Vektorer kan også multipliceres med en *skalar* (et tal), $\lambda \in \mathbb{R}$:

$$\lambda \begin{bmatrix} a \\ b \\ c \\ \vdots \end{bmatrix} = \begin{bmatrix} \lambda a \\ \lambda b \\ \lambda c \\ \vdots \end{bmatrix}$$

Skalarproduktet af to vektorer kan også bestemmes med samme fremgangsmåde som ved geometriske vektorer:

$$\begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} a_2 \\ b_2 \\ c_2 \\ \vdots \end{bmatrix} = a_1 a_2 + b_1 b_2 + c_1 c_2$$

Det er desuden nyttigt at kende til transponeringsoperatoren, som omdanner kolonnevektorer til rækkevektorer og omvendt:

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \end{bmatrix}$$

$$a^T = \begin{bmatrix} a_1 & a_2 & \cdots \end{bmatrix}$$

Endeligt introduceres der det *ydre produkt*, som giver en matrix af alle kombinationer af produkter mellem de to vektorer:

$$\begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} \otimes \begin{bmatrix} a_2 \\ b_2 \\ c_2 \end{bmatrix} = \begin{bmatrix} a_1 a_2 & a_1 b_2 & a_1 c_2 \\ b_1 a_2 & b_1 b_2 & b_1 c_2 \\ c_1 a_2 & c_1 b_2 & c_1 c_2 \end{bmatrix}$$

(Deisenroth, Faisal og Ong, 2020, pp. 17-18)

3.2 Matricer

Disse principper kan nu udvides yderligere ved at introducere *matricer*. En matrix ligner en vektor, men består både af rækker og kolonner:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Omtrent de samme regler gælder matricer som der gør for vektorer.
Addition:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots \\ \vdots & \vdots & \ddots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots \end{bmatrix}$$

Multiplikation med en skalar:

$$\lambda \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} \lambda a_{11} & \lambda a_{12} & \cdots & \lambda a_{1n} \\ \lambda a_{21} & \lambda a_{22} & \cdots & \lambda a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda a_{m1} & \lambda a_{m2} & \cdots & \lambda a_{mn} \end{bmatrix}$$

Prikproduktet er en smule mere indviklet end ved vektorer, da der nu indføres endnu en dimension. Prikproduktet af to matricer, $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times k}$, vil give en matrix, $\mathbf{C} \in \mathbb{R}^{m \times k}$. Hvert element i $\mathbf{C}$, c_{ij} beregnes som:

$$\begin{aligned} c_{ij} &= \sum_{l=1}^{n} a_{il} b_{lj} \\ i &= 1, \ldots, m \\ j &= 1, \ldots, k \end{aligned}$$

Dette kan forstås at tage alle elementer i række i af $\mathbf{A}$ og multiplicere dem med det tilsvarende element i kolonne j af $\mathbf{B}$, og summere alle disse produkter. Det skal bemærkes at prikproduktet *ikke* er kommutativt, altså:

$$\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A}$$

En af grundene til at dette gælder er at, for at kunne bestemme prikproduktet af to matricer, skal deres "nabodimensioner" være ens:

$$\underbrace{\mathbf{A}}_{n\times k} \cdot \underbrace{\mathbf{B}}_{k\times m} = \underbrace{\mathbf{C}}_{n\times m}$$
$$k = k$$
$$\underbrace{\mathbf{B}}_{k\times m} \cdot \underbrace{\mathbf{A}}_{n\times k} = ?$$
$$m \neq n$$

Bemærk at prikproduktet mellem en matrix og en vektor vil give:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 \\ a_{21}b_1 + a_{22}b_2 \\ a_{31}b_1 + a_{32}b_2 \end{bmatrix}$$

Matricer kan desuden, ligesom vektorer, transponeres. Dette gøres ved at bytte om på de to akser i matricen:

$$A_{ij} = (A^T)_{ji}$$
$$A \in \mathbb{R}^{n\times m}$$
$$A^T \in \mathbb{R}^{m\times n}$$

I programmering er det ofte brugbart at bestemme det elementmæssige produkt af to vektorer eller matricer, kaldet *Hadamardproduktet*:

$$\begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} \odot \begin{bmatrix} a_2 \\ b_2 \\ c_2 \end{bmatrix} = \begin{bmatrix} a_1a_2 \\ b_1b_2 \\ c_1c_2 \end{bmatrix}$$

(Deisenroth, Faisal og Ong, 2020, pp. 22-23)

3.3 Funktioner

En del af den lineære algebra, som især er brugbar i Machine Learning, er at have $\mathbb{R}^n$ eller $\mathbb{R}^{n\times m}$ som definitionmængden og/eller værdimængden af en funktion. En mere kompakt notation for at definere en funktions definition- og værdimængde indføres her. En funktion f, med definitionmængde Dm og værdimængde Vm kan skrives som:

$$f\colon Dm \to Vm$$

F.eks. vil en funktion, som tager en vektor, $\mathbf{v} \in \mathbb{R}^3$ og giver en funktionsværdi $f(\mathbf{v}) \in \mathbb{R}$ kunne skrives som:

$$f\colon \mathbb{R}^3 \to \mathbb{R}$$

For fuldt ud at kunne bruge disse funktioner i Machine Learning, skal de kunne differentieres. Dette gøres ved hjælp af *partiel differentiering*. Ved en partiel differentiering betragter man den ene af variablerne i funktionen som en variabel, og de andre som konstanter. Eksempel:

$$\begin{aligned} f(x,y) &= xy^2 + 2 + x^2 \\ \frac{\partial f}{\partial x} &= y^2 + 2x \\ \frac{\partial f}{\partial y} &= 2xy \end{aligned}$$

Disse kan samles i en matrix for at danne det der kaldes *gradienten* af funktionen:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} y^2 + 2x & 2xy \end{bmatrix}$$

(Deisenroth, Faisal og Ong, 2020)
Når gradienten evalueres ved et punkt på grafen, vil den give retningen af den stejleste stigning (Sanderson, 2016). Det må derfor naturligt følge, at den negative gradient vil være retningen af den stejleste nedstigning.

4 Introduktion til Machine Learning

I dette afsnit vil der redegøres for motivationen af Machine Learning, herunder problemerne som løses af Machine Learning.

4.1 Motivation

En computers styrke er generelt problemer hvor løsningen kan skrives som en liste af instruktioner. Der kan eksempelvis beskrives en algoritme for at finde kvadratroden af et tal. Det er dog sværere, at beskrive en algoritme for at bestemme om et billede indeholder et tal, og hvilket. Selvom det i princippet er muligt at beskrive og implementere en sådan algoritme, ville det være særdeles upraktisk. Her kan der i stedet ses til mennesket for at finde ud af hvordan tal genkendes. Når en person ser et tal, gør personen ikke brug af nogen bestemt algoritme, det er noget personen ved intuitivt. Denne intuition er dannet på baggrund af de tusindvis af tal personen har set i løbet af sin levetid, der har gjort at personen kan identificere bestemte

træk som hvert tal indeholder. Ideen bag Machine Learning er altså at kunne vise[1] et program et datasæt af billeder af tal, samt et svarark, som indeholder hvilket tal der er blevet vist. Hvis programmet forudsiger det forkerte svar, justerer det sine interne regler, således at det er tættere på at svare rigtigt. Machine Learning er det område indenfor datalogi der beskæftiger sig med disse problemer. En underkategori af Machine Learning er kunstige neurale netværk, som er inspireret af hjernen. Ligesom hjernen består af et komplekst forbundet netværk af nerver, består et neuralt netværk af knuder som er forbundne med hinanden. (Mitchell, 1997)

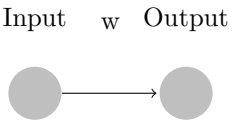
4.2 Det neurale netværks egenskaber

Hver knude har en inputværdi og en outputværdi. I knuder med en foregående knude bestemmes inputværdien som en vægtet sum af de foregående knuder, samt et bias, for at rette op på systematiske fejl. Bemærk at vægten skrives som $w_{til,fra}$.

$$x_i = b_i + \sum_{k=0}^{n} w_{ik} o_k \tag{1}$$

Outputværdien kan bestemmes som funktionsværdien af en aktiveringsfunktion (også kaldet en outputfunktion), $\alpha(x)$ til inputværdien

$$o_i = \alpha(x_i) \tag{2}$$



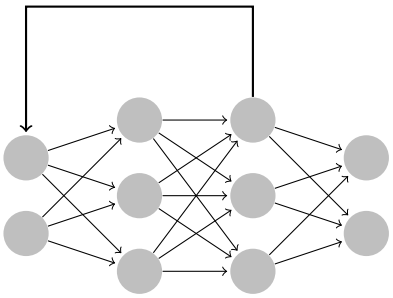
Figur 1: Et simpelt neuralt netværk, bestående af en inputknude og en outputknude

Der findes flere forskellige typer neurale netværk, men der vil her være fokus på såkaldte *feed-forward netværk*.

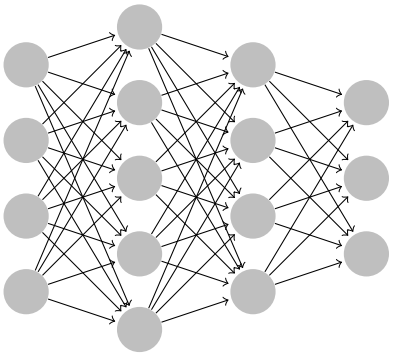
[1]Ordet "vise" er en personificering. Når det benyttes, menes der at give programmet datasættet som parameter

Definition 4.1 (Feed-forward netværk)**.** Et feed-forward netværk har følgende egenskaber (Clabaugh, Myszewski og Pang, 2000):

1. Netværket er *acyklisk* (Se figur 2 og 3)
2. Netværket består af *lag* af knuder. Hver knude i et lag er forbundet til alle knuder i det forrige og næste lag.
3. Knuder i samme lag er ikke forbundet med hinanden.



Figur 2: Et cyklisk netværk. Bemærk at en af de skjulte knuder forbinder til en tidligere knude



Figur 3: Et acyklisk neuralt netværk. Dette er desuden et gyldigt feed-forward netværk

Lagene i et neuralt netværk kan deles ind i tre forskellige typer:

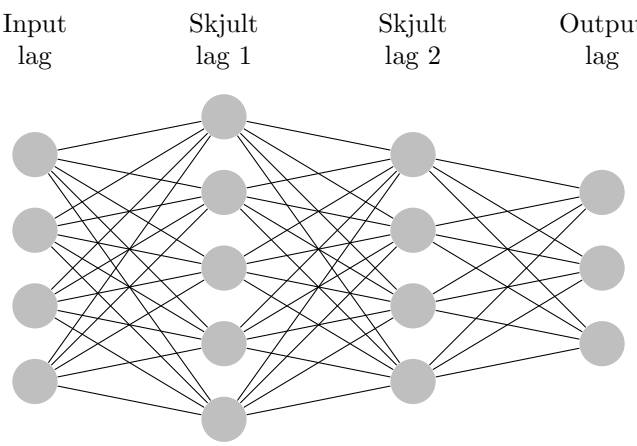
1. *Inputlag*, hvor data indføres. Inputlaget har ingen foregående lag. Et feed-forward netværk har nøjagtig et inputlag.
2. *Outputlag*, hvor resultatet kan aflæses. Outputlaget er det sidste lag i netværket. Et feed-forward netværk har nøjagtig et outputlag.
3. *Skjulte lag*, som ligger imellem inputlaget og outputlaget.

5 Design af et neuralt netværk

5.1 Datarepræsentationer

En computer arbejder kun med tal. Det er derfor nødvendigt at kunne repræsentere et billede som tal. Et billede inddeles i diskrete enheder kaldet *pixels*. Da der vil kun betragtes gråskalabilleder, vil hver pixel, p kun have en værdi, $0 \leq p \leq 1$, $p \in \mathbb{R}$, hvor en pixel med værdien 0 repræsenterer en helt sort pixel og en pixel med værdien 1 repræsenterer en helt hvid pixel[2]. Billedet kan nu repræsenteres som et gitter af reelle tal, $0 \leq p \leq 1$. Billedets

[2]I det oprindelige datasæt gælder det at $0 \leq p \leq 255$, $p \in \mathbb{Z}$, men dette skaleres

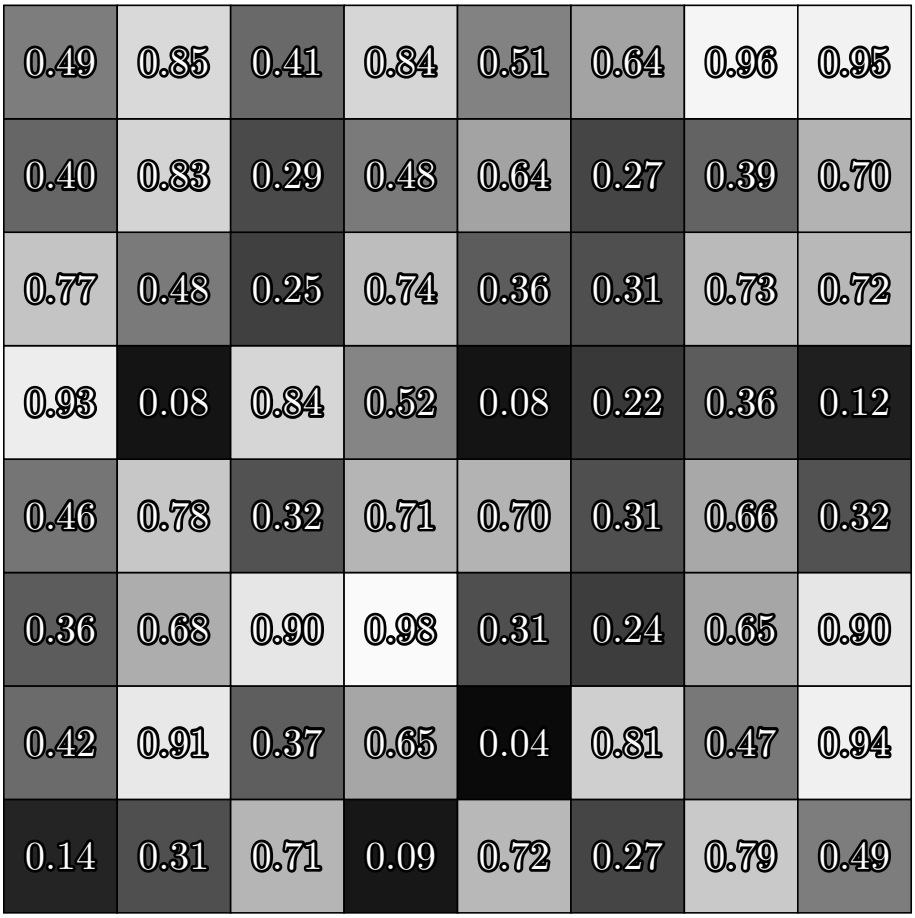


Figur 4: Et feed-forward netværk

rækker lægges nu side om side, for at gøre datastrukturen endimensionel, så den kan repræsenteres simpelt på computeren.

Nu hvor inputdataen kan repræsenteres, mangler formaten for netværkets output. En umiddelbar mulighed vil være at at give netværket en enkelt outputknude, hvis værdi vil fortolkes som netværkets "gæt". Her er det dog vigtigt at overveje hvilke værdier der skal være mulige for netværket at gætte. I tilfældet af den enkelte outputknude vil netværkets mulige gæt, g, være $g \in Vm(\alpha)$. Dette er ikke ideelt, da det som regel vil gælde at $Vm(\alpha) \in \mathbb{R}$, og netværkets gæt vil ideelt være $g \in \mathbb{Z}$, $0 \leq g \leq 9$. En mulig løsning er at runde resultatet til et heltal, men dette kan være problematisk, da der nu kasseres information fra netværket. Desuden vil der senere være brug for en funktion der bestemmer, hvor langt netværket har været fra det korrekte svar. Her vil denne løsning vise sig problematisk. Hvis det korrekte svar eksempelvis er 3, og netværket har svaret 7, vil dens svar, ifølge kostfunktionen, være "mere forkert"[3] end hvis netværket havde svaret 5. Det giver dog ikke mening at se på svarets numeriske værdi for at finde ud af hvor tæt det er på det korrekte svar, idet hvert tal i dette tilfælde kan betragtes som forskellige kategorier.
Løsningen til dette problem er at gøre netværkets outputlag til 10 knuder, tilsvarende de 10 mulige svar fra netværket. Hver outputknude fortolkes derfor som chancen, ifølge netværket, for at billedet indeholder det tal. Netværkets svar vil fortolkes som indekset af den knude der har den største værdi. Denne teknik kaldes *one-hot encoding*.

[3]Dvs. fejlfunktionen vil have en større værdi



Figur 5: Et eksempel på et billede, med pixelværdier indsat

5.2 Outputfunktioner

Indtil videre svarer hver vægt til en lineær funktion, og en knude kan repræsenteres som en sum af lineære funktioner. Summen af et vilkårligt antal lineære funktioner, vil også give en lineær funktion (Se bevis 5.1).

Bevis 5.1 (Summen af lineære funktioner)**.** Der er givet en funktion, $f(x)$, som summen af to linære funktioner:

$$f(x) = a_1x + b_1 + a_2x + b_2$$

a_1 og a_2 samles som faktorer af x:

$$f(x) = (a_1a_2)x + (b_1 + b_2)$$
$$f(x) = ax + b$$

□

Derudover vil hver knude, der ikke ligger umiddelbart før inputlaget, kunne repræsenteres som en kæde af linære funktioner:

$$O_i = \mathbf{W_i} \sum O_{i-1}(O_{i-2}(\dots))$$

Bevis 5.2 (Kædning af linære funktioner)**.** Der er givet en funktion, $f(x)$, som er en kæde af linære funktioner:

$$f(x) = a_1(a_2x + b_2) + b_1$$
$$f(x) = a_1a_2x + a_1b_2 + b_1$$
$$f(x) = ax + b$$

□

Da alle enkelte elementer af netværket kan beskrives af linære funktioner, vil outputlaget også kunne beskrives som en vektor af linære funktioner. Dette er ikke ideelt, da det er sandsynligt, at dataen der ønskes at lære ikke er lineært. Der skal derfor introduceres et ikke-lineært element til netværket. Dette er hvad outputfunktioner bruges til. I stedet for at sende den vægtede sum af de forgående værdier direkte videre til det næste lag, bliver de først behandlet af en outputfunktion. To eksempler på almindelige outputfunktioner er sigmoid-funktionen og ReLU (Shamara, 2017):

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
$$ReLU(x) = \begin{Bmatrix} x > 0, & x \\ x \leq 0, & 0 \end{Bmatrix}$$

Der vil her benyttes sigmoid-funktionen som outputfunktion, primært fordi dens værdimængde er $0 < \sigma(x) < 1$, hvilket giver en elegant repræsentation for netværkets output som sandsynligheder.

5.3 Vægte

Hvert lag af vægte kan repræsenteres som en matrix, $\mathbf{W}$, hvor hvert element w_{jk} repræsenterer vægten fra knude k til knude j. Dette betyder at inputværdierne for et lag kan bestemmes som:

$$\mathbf{x} = \begin{bmatrix} x_j \\ x_{j+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0k} \\ w_{10} & w_{11} & \cdots & w_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{j0} & w_{j1} & \cdots & w_{jk} \end{bmatrix} \begin{bmatrix} o_0 \\ o_1 \\ \vdots \\ o_k \end{bmatrix} \tag{3}$$

Mere simpelt skrives dette som:

$$\mathbf{x} = \mathbf{W}\mathbf{o}_{-1} \tag{4}$$

Hvor $\mathbf{o}_{-1}$ er outputværdierne af det forgående lag.

5.4 Potentielle problemer

5.4.1 Underfitting

Underfitting refererer til, når en model ikke kan lære det givne datasæt. Dette sker som regel enten fordi modellen ikke er stor nok (F.eks. hvis et neuralt netværk har for lidt knuder/lag), eller fordi modellen ikke kan modellere egenskaber af det givne datasæt (F.eks. ved at forsøge at bruge et neuralt netværk til at modellere sekventiel data, som tekst). Underfitting er relativt let at identificere, idet det kendetegnes ved en lav præcision/høj kostværdi, både ved træning af netværket og ved evaluering af netværket. (Brownlee, 2016)

5.4.2 Overfitting

Overfitting referer til når en Machine Learning model lærer datasættet for godt. Her indlærer modellen også støjen i datasættet, hvilket gør at den har lært datasættet "udenad". Dette skyldes som regel en for stor model, eller et for lille datasæt. Overfitting kan identificeres ved at have to datasæt: Et træningssæt, som bruges til at træne modellen, og et testsæt, som udelukkende bruges til at evaluere modellen. Da modellen aldrig trænes på testsættet, vil den have en lav præcision på testsættet hvis modellen er overfit. (Brownlee, 2016)

6 Træning af et neuralt netværk

Der er nu konstrueret et neuralt netværk, som kan tage et billede som input og give et resultat. Dette resultat er dog ikke brugbart endnu, da netværket ikke er blevet trænet.

6.1 Kostfunktionen

For at kunne træne et netværk, skal der først defineres en funktion som fortæller hvor tæt netværket er på det korrekte svar. Denne funktion kaldes *kostfunktionen* eller *fejlfunktionen*, og vil her defineres som:

$$E_t(\mathbf{o}) = \frac{1}{2}\sum_{k=1}^{n}(t_k - o_k)^2 \tag{5}$$
$$E_t \colon \mathbb{R}^n \to \mathbb{R}$$

Her er $\mathbf{o}$ netværkets outputs, og $\mathbf{t}$ den forventede værdi af den tilsvarende knude.
Denne funktion kan udvides for at bestemme fejlen for et helt datasæt:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{t\in T}\sum_{k\in outputs}(t_{kd} - o_{kd})^2 \tag{6}$$

Kostfunktionen er dog ikke specielt brugbar til at vurdere netværkets præstation for mennesker, da den ikke siger noget om hvor ofte netværket gætter rigtigt. Her indføres præcisionen af netværket:

$$P = \frac{c}{t}$$

Her er c antallet af korrekte gæt netværket har lavet, og t er det totale antal gæt netværket har lavet. Bemærk at denne funktion ikke direkte bruges til at træne netværket, men er blot en måde at kunne sammenligne modeller.

6.2 Gradient nedstigning

Eftersom netværkets præstation nu kan måles ud fra værdien af en enkelt funktion, er målet nu at få værdien af denne funktion så tæt på 0 som muligt, for alle træningseksempler. Da dette er et optimeringsproblem, vil den åbenlyse løsning være at differentiere funktionen og sætte den differentierede lig med 0. Dette er dog ikke en praktisk løsning, da denne funktion vil være meget kompleks når den udvides, idet det vil vise sig at, et netværk som skal

kunne genkende relativt små billeder, vil have tusindvis af vægte, hvilket gør den upraktisk at differentiere. Løsningen som Machine Learning præsenterer er at bruge gradienten til at gradvist nærme sig et lokalt minimum. Denne algoritme kaldes *gradient nedstigning*. Først bestemmes gradienten, ved partiel differentiering:

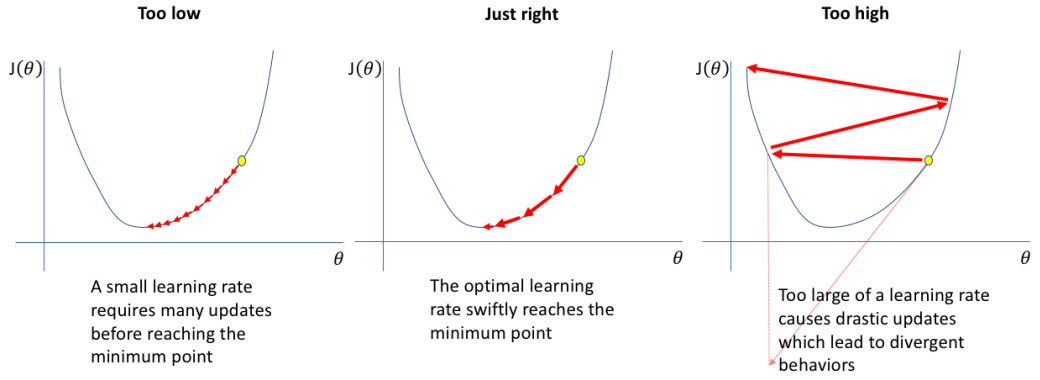
$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0} \quad \frac{\partial E}{\partial w_1} \quad \dots \quad \frac{\partial E}{\partial w_n}\right] \tag{7}$$

Ud fra dette kan ændringen i vægtene bestemmes som:

$$\Delta \mathbf{w} = -\eta \nabla E(\mathbf{w}) \tag{8}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w} \tag{9}$$

Her er η *læringsraten*, som er et tal, $\eta > 0$, $\eta \in \mathbb{R}$, som bestemmer hvor hurtigt modellen bevæger sig i retningen af den stejleste nedstigning (Mitchell, 1997). Denne parameter er vigtig, da en for lav læringsrate vil resultere i at netværket nærmer sig et lokalt minimum for langsomt, mens en for høj værdi vil resultere i at netværket "rammer forbi" det lokale minimum (Jordan, 2018). Eksempler på disse ses på figur 6.



Figur 6: En visualisering af læringsratens indvirkning på et neuralt netværk (Jordan, 2018)

6.3 Udledning af træningsreglen

For at bestemme hvor meget hver vægt skal ændres, skal gradienten, som vist i ligning 7, bestemmes. $\frac{\partial E}{\partial w_{jk}}$ skal derfor bestemmes. Dette gøres på to forskellige måder, afhængigt af om knude j er en output knude eller en skjult knude.

6.3.1 Outputvægte

Først skal fejlen, som funktion af vægten bestemmes. Da vægten kun kan have indflydelse på en outputknude, kan de andre ignoreres.

$$E(w_{jk}) = \frac{1}{2}(t_j - \sigma(w_{jk}o_k))^2$$

Her kan kædereglen bruges til at differentiere udtrykket, ved hjælp af de værdier der allerede er definerede:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial x_j} \cdot \frac{\partial x_j}{\partial w_{jk}}$$

Det sidste udtryk kan bestemmes som:

$$\frac{\partial x_j}{\partial w_{jk}} = o_k$$

Herfra skal $\frac{\partial E}{\partial o_j}$ bestemmes:

$$\begin{aligned}\frac{\partial E}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= (t_j - o_j)\frac{\partial}{\partial o_j}(t_j - o_j) \\ &= (t_j - o_j) \cdot -1 \\ &= -(t_j - o_j)\end{aligned}$$

Det gælder at:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

$\frac{\partial o_j}{\partial x_j}$ kan derfor bestemmes som:

$$\frac{\partial o_j}{\partial x_j} = \frac{\partial \sigma(x_j)}{\partial x_j} = o_j \cdot (1 - o_j)$$

Alt dette kan samles til:

$$\frac{\partial E}{\partial w_j k} = (t_j - o_j) \cdot o_j \cdot (1 - o_j) \cdot o_k \tag{10}$$

Dette samles med ligning 8:

$$\Delta w_{jk} = \eta \cdot (t_j - o_j) \cdot o_j \cdot (1 - o_j) \cdot o_k \tag{11}$$

Ud fra dette indføres en ny variabel, δ_j:

$$\delta_j = -\frac{\partial E}{\partial x_j} = o_j \cdot (1 - o_j) \cdot (t_j - o_j) \tag{12}$$

6.3.2 Skjulte vægte

Ændringen af vægten i skjulte knuder er lidt mere indviklet, da hver vægt kan have indvirkning på mere end en outputknude. Her vil det være brugbart at referere til alle knuder, hvis værdi direkte afhænger af værdien af knude j. Her indføres $Ds(j)$. Kædereglen bruges igen til at finde $\frac{\partial E}{\partial w_{jk}}$. Her vil det dog også gælde at:

$$\frac{\partial x_j}{\partial w_{jk}} = o_k$$

Hvilket kan genbruges senere.

Der skal nu bestemmes $\frac{\partial E}{\partial x_j}$:

$$\frac{\partial E}{\partial x_j} = \sum_{k \in Ds(j)} \frac{\partial E}{\partial x_k} \cdot \frac{\partial x_k}{\partial x_j}$$

δ_k kan nu indføres, jf. ligning 12

$$\begin{aligned}
\frac{\partial E}{\partial x_j} &= \sum_{k \in Ds(j)} -\delta_k \cdot \frac{\partial x_k}{\partial x_j} \\
&= \sum_{k \in Ds(j)} -\delta_k \cdot \frac{\partial x_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial x_j} \\
&= \sum_{k \in Ds(j)} -\delta_k \cdot w_{kj} \cdot \frac{\partial o_j}{\partial x_j} \\
&= \sum_{k \in Ds(j)} -\delta_k \cdot w_{kj} \cdot o_j \cdot (1 - o_j) \\
&= o_j \cdot (1 - o_j) \sum_{k \in Ds(j)} -\delta_k \cdot w_{kj}
\end{aligned}$$

Dette opskrives som $\delta_j = -\frac{\partial E}{\partial x_j}$:

$$\delta_j = o_j \cdot (1 - o_j) \cdot \sum_{k \in Ds(j)} \delta_k \cdot w_{kj} \tag{13}$$

Jf. ligning 8, vil ændringen af vægten w_{jk} være:

$$\Delta w_{jk} = \eta \cdot \delta_j \cdot o_k \tag{14}$$

6.4 Backpropogation

Der kan nu beskrives en algoritme for hvordan et neuralt netværk trænes (Mitchell, 1997):
For hvert træningseksempel, med input $\mathbf{x}$ og forventede værdi $\mathbf{t}$:
Feed-forward:

1. Sæt værdien af inputlaget af netværket til $\mathbf{x}$
2. Bestem outputværdien for hver knude i alle lag, indtil alle værdier af outputknuderne er bestemte.

Backpropogation:

3. For hver outputknude k, bestem δ_k (Se ligning 12):

$$\delta_k = o_k \cdot (1 - o_k) \cdot (t_k - o_k)$$

4. For hver skjulte knude, bestem δ_h (Se ligning 13):

$$\delta_h = o_h \cdot (1 - o_h) \sum_{k \in Ds(h)} w_{kh} \delta_k$$

5. Beregn ændringen for hver vægt (Se ligning 14):

$$\Delta w_{jk} = \eta \cdot \delta_j \cdot o_k$$

6. Opdater hver vægt:

$$w_{jk} \leftarrow \Delta w_{jk} + w_{jk}$$

Denne specifikke algoritme kaldes *stokastisk gradient nedstigning*. I modsætning til almindelig gradient nedstigning, justeres vægtene efter hvert træningseksempel, hvilket gør træningen markant hurtigere.

7 Implementation

Når der er dannet et matematisk grundlag for virkemåden af neurale netværk, kan et sådant netværk implementeres. Dette vil gøres i Python 3.8, dels på grund af den kompakte syntaks, men primært på grund af biblioteket `numpy`, som tillader hurtige matematiske operationer når der arbejdes med matricer. Netværket der implementeres vil bestå af 3 lag: Et inputlag på 784 knuder (tilsvarende træningseksemplernes 28x28 pixels), et skjult lag på 32 knuder og et ouputlag på 10 knuder. Denne implementation understøtter både et vilkårligt antal lag og knuder i lagene.

7.1 Numpy

`numpy`-biblioteket (som i programmet importeres med navnet `np`) tilbyder klassen `ndarray`, som kan repræsentere et array med et vilkårligt antal dimensioner. I dette tilfælde vil denne klasse bruges til at repræsentere vektorer og matricer. Der defineres dog nogle operationer, som går imod den almindelige matematiske praksis:
*-operatoren bruges til det elementmæssige produkt:

```
1  import numpy as np
2  a = np.array([1,2,3,4])
3  b = np.array([0,1,2,3])
4  c = a * b
5  print(c)
6  # Output:
7  # array([ 0,  2,  6, 12])
```

Prikproduktet bestemmes ved funktionen `np.dot(a,b)`.

7.2 Programstrukur

Den primære klasse i programmet er klassen `Layer`, som repræsenterer et lag i det neurale net. Den indeholder værdierne af knuderne, samt vægtene som går ind i laget. Desuden indeholder den en metode til at beregne værdien af hele netværket rekursivt. Netværket skal dog først initialiseres, hvilket gøres med funktionerne `createLayers(template)` og `randomizeLayers(layers, variance)`:

```
36 layersTemplate = [len(dpTrain.images[0].normalizedData), 32,
   ↪  10]
37 layers = createLayers(layersTemplate)
38 if weights == None:
39     randomizeLayers(layers, 0.05)
```

Funktionen `createLayers` laver en liste af lag med de givne størrelser:

```
155 def createLayers(layers: list):
156     layerList = []
157     l = Layer(layers[0])
158     layerList.append(l)
```

```
159     for i in range(1, len(layers)):
160         l = Layer(layers[i], l)
161         layerList.append(l)
162         layerList[i - 1].next = l
163     return layerList
```

`randomizeLayers` giver herefter hver vægt i lagene en tilfældig værdi:

```
166 def randomizeLayers(layers: typing.List[Layer], variance:
    ↪ float):
167     rV = np.vectorize(lambda _: random.uniform(-variance,
        ↪ variance))
168     for l in layers:
169         if not hasattr(l, "weights"):
170             continue
171         l.weights = rV(l.weights)
```

Her bruges funktionen `np.vectorize` til at køre en funktion på alle elementer i en matrix på en kompakt måde. `vectorize` tager en funktion med et argument, og returnerer en funktion med et argument. Når man kører `np.vectorize(f)(m)`, vil den returnere en ny matrix, som er `m`, med `f` anvendt på alle værdier. I dette tilfælde gives `vectorize` en lambdafunktion som argument, som er en simpel måde at skrive en kort funktion som kun skal bruges en gang. Syntaksen:

```
1 h = np.vectorize(lambda x: g(x))
```

Svarer til syntaksen:

```
1 def f(x):
2   return g(x)
3 h = np.vectorize(f)
```

I dette tilfælde ignoreres argumentet af lambdafunktionen, og i stedet returneres et tilfældigt tal i intervallet $[-variance; variance]$
Når netværket er initialiseret, kan outputværdierne af et lag beregnes:

```
34 def calculateValues(self, data: np.ndarray) -> np.ndarray:
35     """
```

```
36      Calculate the values of the layer, and set them on the
    ↪   layer object
37
38      Returns
39      -------
40      The calculated values
41
42      """
43
44      # If this is the input layer, the values will simply be
        ↪   the input values
45      if self.previous == None:
46          self.inputValues = data
47          self.outputValues = data
48          return self.outputValues
49
50      # Otherwise, calculate the input values as weights ·
        ↪   previous
51      prevValues = self.previous.calculateValues(data)
52      self.inputValues = np.dot(self.weights, prevValues)
53      self.outputValues = mlmath.sigmoid(self.inputValues)
54
55      return self.outputValues
```

For at beregne alle outputværdier i netværket, kaldes denne funktion på outputlaget, hvorefter beregningen vil bevæge sig bagud gennem det rekursive kald på linje 51. Outputværdien beregnes med funktionen `mlmath.sigmoid(x)`, som er implementeret som:

```
4   def sigmoid(x):
5       return 1 / (1 + np.exp(-x))
```

Bemærk at, der i denne funktion bruges `numpy`s eksponentialfunktion, `np.exp(x)`, da den kan arbejde med matricer og vektorer hurtigere end den indbyggede `math.exp(x)` (Lebigot, 2010). Ud fra outputværdierne, kan en liste af værdierne δ_j bygges, ved først at bygge en liste fyldt med nuller, som har samme længde som antallet af outputværdier. Denne liste kan herefter fyldes ud ved ligning 12:

```
error = [0] * len(outputLayer.outputValues)
for i in range(len(outputLayer.outputValues)):
  error[i] = (
    outputLayer.outputValues[i]
    * (1 - outputLayer.outputValues[i])
    * (target[i] - outputLayer.outputValues[i])
  )
```

Her bør det dog bemærkes at alt indekseres ud fra værdien `i`. Dette kan derfor omskrives til en enkelt vektoroperation for en stor forbedring i hastigheden:

```
error = (
  outputLayer.outputValues
  * (1 - outputLayer.outputValues)
  * (target - outputLayer.outputValues)
)
```

Herfra kan ændringen af hver vægt bestemmes som det ydre produkt af `error`-vektoren og outputværdien af den knude som vægten går fra (Se ligning 14):

```
changes = []
for layer in layers[::-1]:  # Loop through the layers from
↪  the back
    if layer.previous == None:
        continue
    changes = [
        learningRate * np.outer(error,
        ↪  layer.previous.outputValues)
    ] + changes  # Put a matrix of the output values
    ↪  multiplied by the error at the front of the changes
    ↪  list
```

δ-værdierne for det næste lag beregnes nu, med en omskreven udgave af ligning 13, som udnytter hastigheden af `numpy`s operationer på vektorer og matricer:

```
    error = (
        np.dot(layer.weights.transpose(), error)
```

```
113         * layer.previous.outputValues
114         * (1 - layer.previous.outputValues)
115     )
```

Efter listen af ændringer er fyldt ud, anvendes disse ændringer til netværket. Dette gøres ved at gå igennem listen af lag og listen af ændringer samtidigt, ved hjælp af `zip`-funktionen. Bemærk at inputlaget ekskluderes fra listen, ved operationen `layers[1:]`, da der ikke er nogen vægte som går ind i inputlaget, og der kan derfor ikke foretages ændringer på dens vægte.

```
116 for change, layer in zip(changes, layers[1:]):
117     if layer.previous != None:
118         layer.weights += change
```

Herfra kan netværket testes. Præcisionen måles ved at sammenligne indekset af en største værdi i outputvektoren, som findes ved numpy-funktionen `np.argmax(v)`, med det korrekte svar, `t.label`:

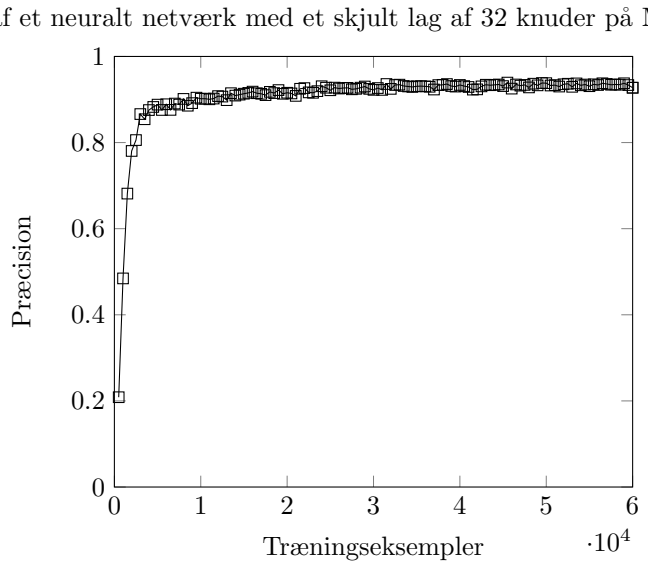
```
121 def testPass(layers: typing.List[Layer], dpTest: Dataparser):
122     costSum = 0
123     correctGuesses = 0
124     for t in tqdm(dpTest.images, leave=False, desc="Testing"):
125         layers[-1].calculateValues(np.array(t.normalizedData))
126         guess = np.argmax(layers[-1].outputValues)
127         if guess == t.label:
128             correctGuesses += 1
129         costSum += layers[-1].cost(t.expectedVector())
130 
131     return correctGuesses / len(dpTest.images)
```

Funktionen `tqdm` fra biblioteket `tqdm` bruges her til at vise en statuslinje i konsollen. Funktionelt har det samme betydning som:

```
for t in dpTest.images
```

7.3 Resultat

Efter blot 60,000 iterationer af stokastisk gradient nedstigning igennem MNIST-datasættet, bestående af 60,000 gråskalabilleder af tal i en opløsning af 28x28 pixels (LeCun, Cortes og Burges, 2020), har modellen opnået en præcision på ca. 92.7%. En graf af præcisionen over tid, kan ses på figur 7. Instruktioner



Figur 7: På x-aksen vises antallet af træningseksempler modellen er blevet vist, på y-aksen vises gættepræcisionen på de 10,000 testeksempler. Modellen er blevet testet på de 10,000 testeksempler hver femhundrede træningseksempel.

til at hente programmets kildekode og køre den findes i bilag A1.

8 Anvendelse af neurale netværk

Modellen der heri er trænet kan kun genkende håndskrevne tal i en relativt lille opløsning og uden farver. Konceptet af neurale netværk kan dog udvides til at strække sig over mange områder i den virkelige verden. En af fordelene ved Machine Learning er at det, i modsætning til traditionelle programmer, kræver en minimal mængde justering for at kunne behandle andre datasæt med andre regler. Dette kan gøres ved blot at justere antallet af input-, output-, og skjulte knuder i netværket, samt give det et passende datasæt. Eksemplet med håndskrevne tal kan for eksempel udvides til at inkludere bogstaver ved at forlænge outputlaget til at bestå af 36 knuder (39 hvis æ, ø

og å skal inkluderes), i stedet for 10.

8.1 Neurale netværk i lægekundskaben

Neurale netværk behøves ikke nødvendigvis kun at bruges til billedbehandling. Neurale netværk er også blevet brugt inden for medicin til at finde hjertesygdomme ud fra *elektrokardiogrammer* (også kaldet et EKG)[4]. Netværkets inputs har været måleværdierne, og outputværdierne har repræsenteret en af de mulige sygdomme i datasættet, eller normal (ingen sygdom). Datasættet har bestået af 3266 tilfælde, samt køn og alder, hvori hvert tilfælde lider af en eller ingen af de mulige sygdomme. På grundlag af dette datasæt har netværket kunne opnå en præcision på 68.8%. (Bortolan, Degani og Willems, 1991)

9 Konklusion

Der er i opgaven redegjort for de grundlæggende principper i den lineære algebra, heriblandt vektorer og matricer. Der er desuden redegjort for partiel differentiering, og hvordan hældningen af en funktion med flere argumenter, eller matricer eller vektorer som argumenter, bestemmes. Herefter er der forklaret de egenskaber som kendetegner et neuralt netværk, samt hvordan billeder repræsenteres i det. De mest almindelige problemer i Machine Learning, underfitting og overfitting, er forklaret. Herefter udledes de essentielle formler, som bruges til at træne et neuralt netværk. Alt dette er blevet brugt til at implementere et neuralt netværk i programmeringssproget *Python*. Dette netværk er blevet trænet på et datasæt af 60,000 håndskrevne tal, og har opnået en præcision på ca. 92.7% på et separat evalueringssæt. Endeligt er der perspektiveret til hvordan neurale netværk kan bruges i moderne teknologier, mere specifikt deres brug i lægevidenskaben til analyse af elektrokardiogrammer, for at identificere potentielle hjertesygdomme.

[4]Et EKG er en graf over hjertets elektriske impulser (Kastrup, 2019)

Litteratur

Bortolan, Giovanni, R. Degani og Johannes Willems (okt. 1991). "ECG classification with neural networks and cluster analysis". I: s. 177–180. ISBN: 0-8186-2485-X. DOI: 10.1109/CIC.1991.169074.

Brownlee, Jason (mar. 2016). *Overfitting and Underfitting With Machine Learning Algorithms*. URL: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/ (bes. 16.12.2020).

Clabaugh, Caroline, Dave Myszewski og Jimmy Pang (2000). *Neural Networks - Architecture*. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Architecture/feedforward.html (bes. 08.12.2020).

Deisenroth, Marc Peter, A Aldo Faisal og Cheng Soon Ong (2020). *Mathematics for machine learning*. Cambridge University Press. ISBN: 9781108455145.

Jordan, Jeremy (mar. 2018). *Setting the learning rate of your neural network*. URL: https://www.jeremyjordan.me/nn-learning-rate/ (bes. 14.12.2020).

Kastrup, Jens (sep. 2019). *Ekg*. URL: https://www.sundhed.dk/borger/patienthaandbogen/undersoegelser/undersoegelser/ekg/ekg/ (bes. 16.12.2020).

Lebigot, Eric O (sep. 2010). *Are NumPy's math functions faster than Python's?* URL: https://stackoverflow.com/a/3651058 (bes. 15.12.2020).

LeCun, Yann, Corinna Cortes og Christopher J.C. Burges (2020). *The MNIST database of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/ (bes. 15.12.2020).

Mitchell, Tom M. (1997). *Machine Learning*. 1. udg. McGraw-Hill series in computer science. McGraw-Hill. ISBN: 9780070428072,0070428077.

Sanderson, Grant (maj 2016). *Gradient and graphs*. URL: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/gradient-and-directional-derivatives/v/gradient (bes. 11.12.2020).

Shamara, Sagar (sep. 2017). *Activation Functions in Neural Networks*. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 (bes. 09.12.2020).

A Bilag

A1 Kildekode til neuralt netværk

Kildekoden til programmet kan tilgås ved `https://github.com/TheMagzuz/SOPML`. Programmet kan hentes med

```
git clone https://github.com/TheMagzuz/SOPML
```

Krævede pakker kan installeres med

```
pip install -r requirements.txt
```

Hvorefter programmet kan køres med

```
python3 main.py
```

For at programmet kan køre, skal der være følgende filer i samme mappe som `main.py`:

- `train-images.idx3-ubyte` og `t10k-images.idx3-ubyte`, som er billedfilerne for hhv. træningssættet og testsættet i MNIST formaten.
- `train-labels.idx1-ubyte` og `t10k-labels.idx1-ubyte`, som er de korrekte svar for de tilsvarende billeder

Det benyttede datasæt kan tilgås her: `http://yann.lecun.com/exdb/mnist/` Bemærk at filerne som hentes herfra vil være komprimerede, og skal udpakkes Programmet tager følgende argumenter, som alle er valgfrie:

-i Modellen som programmet skal benytte. Hvis denne værdi ikke er sat, initialiseres en ny model med tilfældige vægte

-m Filen hvor modellen skal gemmes til. Modellen gemmes efter hver epoke (Fulde runde gennem træningssættet)

-c Filen hvor historikken af modellens præcision skal gemmes. Filen er en kommaadskilt lise af punkter, med koordinaterne (træningseksempler, præcision)

-f Antallet af træningseksempler mellem hver evaluering af modellen

-e Antallet af epoker programmet skal køre før det stopper