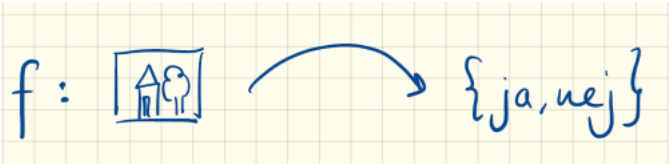
Kunstige neurale netværk

Hvad er kunstig intelligens og hvad skal vi med det?

Forestil dig at du gerne vil have en funktion f, som tager et billede som input og som output fortæller dig, om der er en hund på billedet eller ej. Det kan illustreres sådan her:



Figur 1: Funktion, der tager et billede som input, og som returnerer "ja" eller "nej" som output.

Okay, det er måske ikke så tit, at man har brug for en funktion, som kan detektere, om der er en hund på et billede, men hvad så hvis funktionen i stedet kan afgøre, om der er en kræftknude på et røntgenbillede? Eller hvis den kan genkende håndskrevet tekst? Sidstnævnte bliver f.eks. flittigt brugt til sortering af breve. Vi ønsker os i virkeligheden at være i stand til at programmere en funktion, som kan "tænke" som et menneske. Når jeg ser et billede, kan jeg på ingen tid afgøre, om der er en hund på billedet eller ej. Står jeg med et brev i hånden, kan jeg som regel også læse navn og adresse. En læge vil også kunne kigge på et røntgenbillede og afgøre, om der er en kræftknude eller ej. Det er i bund og grund, det vi forstår ved kunstig intelligens. At få computeren til at "tænke" som et menneske. Nu kunne man måske godt indvende "hvad skal det til for?". Der er vel ingen grund til at få en computer til at finde kræftknuder på et røntgenbillede, hvis vi allerede kan få en læge til det? Men hvad nu, hvis computeren faktisk kan opdage kræftknuder tidligere end

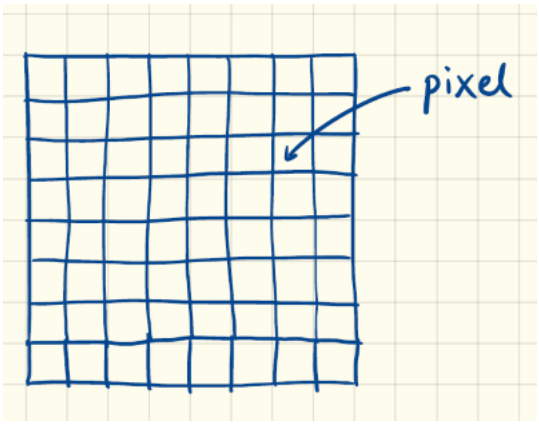
lægen? Eller hvad hvis man har så mange røntgenbilleder, at det vil være smart, at få en computer til at kigge dem igennem først?

Hvordan laver man en funktion, som kan finde "ting" på billeder?

Lad os sige, at vi nu kan se fidusen med den der lidt mærkelige funktion, som kan tage et billede som input, og som output kan fortælle et eller andet om billedet. Men det er jo ikke ligefrem den slags input og output, som vi plejer at arbejde med. Vi er f.eks. vant til at se på en eksponentialfunktion, der som input kan tage et hvilket som helst reelt tal og som output giver et positivt reelt tal. Vi plejer at sige, at funktionens defintionsmængde $Dm(f) = \mathbb{R}$ og at funktionens værdimængde $Vm(f) = \mathbb{R}_+$. Det kan også skrives sådan her:

$$f : \mathbb{R} \to \mathbb{R}_+$$

Men hvordan giver man så et billede som input? Digitale billeder består af en masse "pixels". En pixel svarer til et lille kvadratisk udsnit af billedet, som lidt forenklet er vist på figur 2.



Figur 2: Et digitalt billede består af pixels.

Du har sikkert prøvet at zoome rigtig langt ind på et digitalt billede og set, at det ender med at bestå af små "kasser". Hvis billedet er sort-hvid, så kan farven på hver enkelt pixel repræsenteres ved et tal mellem 0 og 255. Hvis værdien er

0, er pixlen sort, hvis værdien er 255 er pixlen hvid og alt imellem 0 og 255 repræsenterer gråtoner. Hvis billedet er med farver, bruger man tre værdier mellem 0 og 255 til at repræsentere farven (RGB-farver: rød, grøn og blå), men lad os bare for enkelthedens skyld antage, at vi kun betragter sort-hvid billeder. Hvis mit billede er et 8×8 pixels som på figur 2, så kan jeg altså repræsentere billedet vha. en vektor med 64 dimensioner! Det vil sige, at min funktions definitionsmængden i virkeligheden er en delmængde af:

$$Dm(f) = \mathbb{R}^{64} \quad \text{eller rettere} \quad Dm(f) = \{0, 1, \ldots, 255\}^{64}$$

Lad os sige, at vi gerne vil have vores funktion til at finde en hund eller en kræftknude på billedet. Så kunne funktionens værdimængde såmænd bare være:

$$Vm(f) = [0, 1]$$

Her vil tolkningen så f.eks. kunne være, at hvis outputværdien $o \geq 0,5$, så er svaret "ja" (det er en hund) og hvis $o < 0.5$, så er svaret "nej". Vi kunne altså bruge outputværdien til at lave følgende forudsigelse/prædiktion:

$$\text{prædiktion} = \begin{cases} \text{ja} & \text{hvis } o \geq 0,5 \\ \text{nej} & \text{hvis } o < 0,5 \end{cases} \tag{1}$$

Så i et lidt forenklet setup leder vi altså efter en funktion, som tager en 64-dimensional vektor som input, og som output leverer et tal i intervallet $[0, 1]$:

$$f : \mathbb{R}^{64} \to [0, 1]$$

Så langt så godt. Men hvad gør vi så nu? Skal vi sige, at hvis der findes nogle forholdsvis mørke pixels i midten og nogle mørke pixels, som stikker op fra de andre mørke pixels (halen), så er det en hund...? Det virker ikke umiddelbart specielt anvendeligt. Og hvad nu hvis hunden sidder på sin hale eller man kun kan se dens hoved? Inden for den tidligere klassiske kunstig intelligens ville man netop forsøge sig med denne fremgangsmåde, men AI-forskerne løb efterhånden panden mod en mur, ligesom vi noget forenklet har skitseret det her. Det virker simpelthen som en helt uoverskuelig opgave at skulle programmere en computer på den klassiske måde (a la "gør dit og dut

og dat og hvis A så B ellers C... osv.") (Sørensen and Johansen 2020). Problemet er blandt andet, at vi jo faktisk ikke en gang selv fuldstændigt og i detaljer kan forklare, hvordan vi selv genkender en hund på et billede. Det er "bare" noget vores hjerne gør (fordi den i virkeligheden har øvet sig på rigtig mange "hundebilleder"). Det var netop denne erkendelse, som førte til udviklingen af kunstige neurale netværk: Hvis vi ikke selv præcist og detaljeret kan forklare, hvad vores hjerne gør, så skulle vi måske i stedet prøve at programmere vores computer, så den efterligner den måde den menneskelige hjerne fungerer på. Man vendte sig derfor mod biologien (Baktoft 2014) og blev inspireret af den måde millionvis af neuroner i hjernen kommunikerer med hinanden på (men det må du hellere spørge en biologilærer om). Man kan derfor tænke på kunstige neurale netværk, som en forsimplet model af den menneskelige hjerne. Det får også den konsekvens, at vores funktion ender med at blive lidt magi. Vi kan ikke nødvendigvis forklare, hvorfor den præcis ser ud som den gør, men vi kan blot i sidste ende forhåbentlig konstatere, at det virker! Det er faktisk lidt grænseoverskridende. Vi er jo f.eks. vant til at kunne fortolke på de konstanter, som indgår i en lineær funktion, der er anvendt i en given sammenhæng. Men her må du glemme alt om at tillægge de konstanter, vi nu er på jagt efter, nogen som helst betydning!

VIDEO: Kunstige neurale netværk 1

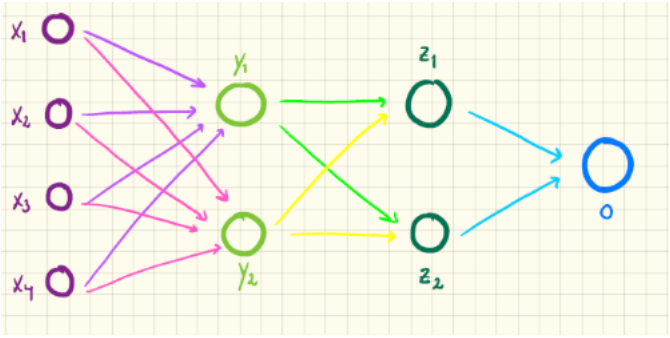
I denne video forklarer lidt om hvad et kunstigt neuralt netværk er og lidt om hvilke input- og outputværdier, man kan bruge.

https://www.youtube.com/embed/09LTr2eVOWg

Hvordan virker et kunstigt neuralt netværk?

Kunstige neurale netværk som bruges i den virkelige verden består som regel af millionvis af de såkaldte neuroner. For at holde tingene simple vil vi her begrænse os til nogle få. Forståelsesmæssigt mister man ingenting, notationen bliver blot lidt simplere. På figur 3 ser du et sådant simpelt kunstigt neuralt netværk, hvor alle cirklerne repræsenterer disse

neuroner. I første omgang ser det måske lidt uoverskueligt ud, men vi tager det et skridt ad gangen.



Figur 3: Simpelt kunstigt neuralt netværk.

Det første man kan se er, at et kunstigt neuralt netværk består af et inputlag (repræsenteret ved de fire lilla cirkler) og et outputlag (repræsenteret ved den blå cirkel). De fire værdier x_1, x_2, x_3 og x_4 svarer til de inputværdier, vi tidligere har talt om. Det kunne f.eks. være værdier, som repræsenterer gråskalaværdier på et 2×2 pixels billede (det er selvfølgelig et lidt kedeligt billede, men tænk på at princippperne her kan skaleres op). Vi har været vant til at se på vektorer i planen. Inputværdierne her svarer faktisk til en vektor i det firedimensionale rum:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

Outputværdien o kunne være et tal mellem 0 og 1 og man kan f.eks. prædiktere vha. udtrykket i (1).

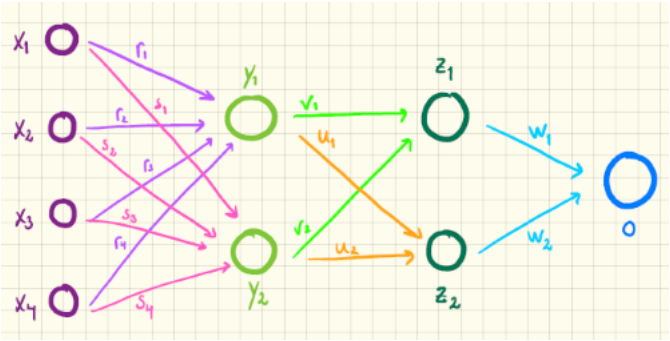
Derudover indgår der to skjulte lag, som er vist ved de lysegrønne og mørkegrønne cirkler. Det er dem, som udgør selve "maskinrummet" i det kunstige neurale netværk. Lad os starte med at se på cirklen, hvor der står y_1. Pilene op til neuronen, hvor der står y_1, viser, at den modtager alle fire inputværdier x_1, x_2, x_3 og x_4. Disse fire værdier bruges til at beregne y_1. Vi starter med at udregne:

$$r_1 \cdot x_1 + r_2 \cdot x_2 + r_3 \cdot x_3 + r_4 \cdot x_4 + r_0 \tag{2}$$

Værdierne r_1, r_2, r_3 og r_4 kalder man for vægte. Hvis f.eks. r_1 er stor og r_2, r_3 og r_4 er tæt på 0, så vil $r_1 \cdot x_1$ også blive stor

(med mindre x_1 er meget tæt på 0) og $r_2 \cdot x_2 + r_3 \cdot x_3 + r_4 \cdot x_4$ vil være tæt på 0. På den måde vil inputværdien x_1 altså få stor indflydelse på udtrykket i (2) - man siger, at x_1 kommer til at vægte højt.

Værdien r_0 kaldes for en "bias" og det er altså en konstant størrelse, som bliver lagt til uafhængig af inputværdierne. Du skal tænke på ovenstående udtryk, som du tænker på en lineær funktion $y = a \cdot x + b$. Her er der bare lige lidt flere x-værdier og a og b er skiftet ud med r'er.



Figur 4: Simpelt kunstigt neuralt netværk med navngivning af vægte.

Værdien af udtrykket i (2) kan være et hvilket som helst reelt tal. Ofte vil man være interesseret i, at den værdi der "kommer ud af" neuronen er et tal mellem 0 og 1. Til det bruges ofte den såkaldte "sigmoid"-funktion, som er defineret sådan her:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

Grafen for sigmoid-funktionen ses i figur 5. Her kan man se, hvordan ethvert reelt tal bliver afbildet over i et tal mellem 0 og 1.

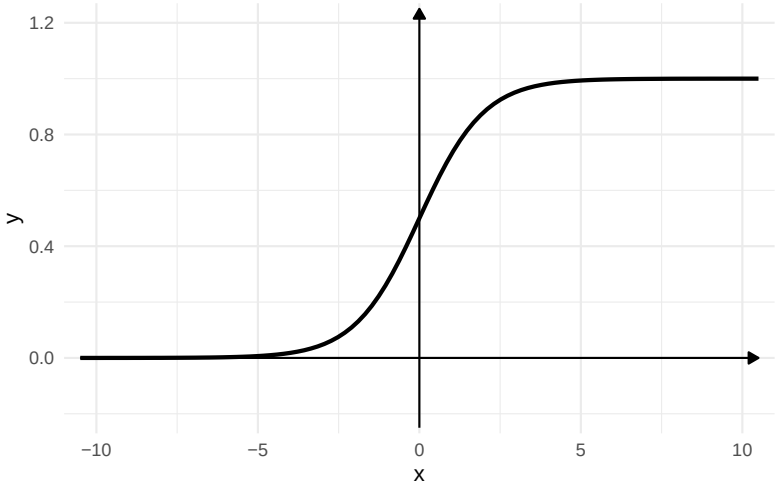
Sigmoid-funktionens differentialkvotient har en bestemt egenskab, som det fremgår af denne sætning:

Sætning 0.1. *Om sigmoid-funktionens differentialkvotient gælder*

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

Det vil sige, hvis $y = \sigma(x)$ *så er*

$$y' = y \cdot (1 - y).$$



Figur 5: Grafen for sigmoid-funktionen.

Beviset er ikke så svært og overlades til læseren.

Det er netop sigmoid-funktionen, som bruges til at udregne y_1:

$$y_1 = \sigma(r_1 \cdot x_1 + r_2 \cdot x_2 + r_3 \cdot x_3 + r_4 \cdot x_4 + r_0) \tag{4}$$

Det vil sige, at

$$y_1 = \frac{1}{1 + e^{-(r_1 \cdot x_1 + r_2 \cdot x_2 + r_3 \cdot x_3 + r_4 \cdot x_4 + r_0)}} \tag{5}$$

Nu beregnes y_2 helt tilsvarende, men med andre vægte. Før kaldte vi vægtene for r_1, r_2 osv. De nye vægte vælger vi nu at kalde for $s_1, s_2, ...$, som det også er vist på figur 4. Vi definerer altså

$$y_2 = \sigma(s_1 \cdot x_1 + s_2 \cdot x_2 + s_3 \cdot x_3 + s_4 \cdot x_4 + s_0) \tag{6}$$

Det vil sige, at

$$y_2 = \frac{1}{1 + e^{-(s_1 \cdot x_1 + s_2 \cdot x_2 + s_3 \cdot x_3 + s_4 \cdot x_4 + s_0)}} \tag{7}$$

Nu sker der det, at (de lysegrønne) neuroner i det andet lag "fyrer" deres y-værdier frem til alle neuronerne i det tredje lag. Disse værdier bliver nu brugt som inputs til beregning af de nye z-værdier (se igen figur 4. Vi kan nu, som vi har gjort ovenfor definere de nye z-værdier i det tredje lag, idet vi nu

kalder vægtene for v_1, v_2, u_1 og u_2, som det er vist på figur 4. Den eneste forskel fra tidligere er, at disse neuroner i vores eksempel kun modtager to og ikke fire inputværdier:

$$z_1 = \sigma(v_1 \cdot y_1 + v_2 \cdot y_2 + v_0) \tag{8}$$

og

$$z_2 = \sigma(u_1 \cdot y_1 + u_2 \cdot y_2 + u_0) \tag{9}$$

Nu fyrer neuroner i det tredje lag deres z-værdier til det sidste outputlag, hvor der i dette simple eksempel kun er en enkelt neuron. Hvis vi ønsker, at outputtet o bliver et tal mellem 0 og 1 sætter vi

$$o = \sigma(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0) \tag{10}$$

Vi kan nu foretage den ønskede prædiktion ved f.eks. at bruge udtrykket i (1).

Det vi her har beskrevet kaldes for **feedforward**, fordi man sender input-værdierne fremad i netværket igennem alle lagene, indtil man har beregnet outputværdien (eller outputværdierne). Hvis man har et kunstigt neuralt netværk, som er "indstillet" korrekt - det vil sige, at alle vægte og bias har de "rigtige" værdier - så kan netværket bruges til at prædiktere med. Men hvordan i alverden sørger man for at vælge de "rigtige" værdier for alle vægte og bias? Husk på i virkelighedens verden taler vi om millioner af værdier! I vores simple eksempel havde vi kun 19 vægte:

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} \quad \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} \quad \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \quad \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \quad \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix}$$

Det gøres ved hjælp af en fiks teknik, som kaldes for **back-propagation**, som vi nu skal se nærmere på.

VIDEO: Kunstige neurale netværk 2

I denne video giver vi et eksempel på et simpelt kunstig neuralt netværk og forklarer **feedforward**.

https://www.youtube.com/embed/hUlyeMJGvXM

Hvordan træner man et kunstigt neuralt netværk?

Vi har altså nu set på, hvordan et kunstigt neuralt netværk virker, hvis vi kender værdierne for alle vægte og bias. Det store spørgsmål er nu, hvordan man får bestemt disse vægte, så netværket bliver så godt så muligt til at finde hundebilleder, eller hvad vi nu er på jagt efter.

Det første vi må gøre, er at opstille et mål for hvor godt et givet netværk er. Så lad os sige, at vi allerede har nogle vægte (i starten vælger man bare nogle mere eller mindre tilfældige vægte). Og lad os se på eksemplet hvor vi har fire inputværdier:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

som vi kan tænke på som pixelværdier på et billede af en hund (det er helt håbløst, at tro at et billede af en hund kan repræsenteres ved fire pixelværdier, men lige nu er det principperne, der er det vigtigste). For at træne et netværk er man også nødt til at fortælle netværket, hvad der så rent faktisk er på billedet. Ellers får vi aldrig netværket til at forstå noget som helst, hvis ikke vi sammen med inputværdierne også kender den ønskede prediktion. Tænk på da du som lille barn lærte at genkende hundebilleder i en billedbog. Det lærte du kun, fordi der i timevis inden havde siddet en voksen og fortalt dig hvilke billeder, der forestillede en hund og hvilke der ikke gjorde. Den samme information må vi også give netværket. Så sammen med input værdien $\vec{x}$, giver vi også en såkaldt *target*-værdi t. Her sætter vi:

$$t = \begin{cases} 1 & \text{hvis billedet er af en hund} \\ 0 & \text{ellers} \end{cases} \tag{11}$$

Sender vi inputværdierne repræsenteret ved vektoren $\vec{x}$ ind i netværket, så vil netværket returnere en outputværdi o mellem 0 og 1. Som tidligere vil vi tolke det på den måde, at netværket mener, at vi står med et billede af en hund hvis $o \geq 0.5$, og hvis $o < 0.5$ så vil netværket sige, at det ikke er en hund. Et netværk,

som er god til at genkende hundebilleder, vil opføre sig sådant, at hvis inputværdien $\vec{x}$ svarer til et hundebillede (dvs. $t = 1$), så vil o være tæt på 1. Og omvendt hvis inputværdien $\vec{x}$ ikke svarer til et hundebillede (dvs. $t = 0$), så vil o være tæt på 0. Det vil sige, at et godt netværk har den egenskab at

$$t - o \approx 0$$

Læg mærke til at forskellen her både kan være positiv og negativ, vi vil bare gerne have, at den er tæt på 0.

Det kan vi nu bruge til at definere det, man kalder for en *error*- eller tabsfunktion E. For et givet input $\vec{x}$ med target-værdi t og hvor netværket giver outputværdien o definerer vi tabsfunktionen E på denne måde:

$$E = \frac{1}{2}(t - o)^2 \tag{12}$$

Bemærk, at hvis der er stor forskel på t og o (hvilket vi jo ikke er interesseret i), så vil fejlen også være stor. Og omvendt hvis der er lille forskel på t og o, så vil fejlen være lille. Desuden vil fortegnet på fejlen forsvinde, fordi vi opløfter i anden. At vi ganger med $1/2$ viser sig bekvemt senere, men det er i princippet underordnet.

I virkeligheden vil man have rigtige mange træningsdata med tilhørende target- og outputværdier. Forestil dig at vi nummerer alle disse target- og outputværdier på følgende måde:

$$(t_1, o_1), (t_2, o_2), (t_3, o_3), \dots, (t_n, o_n)$$

Og da vil man definere tabsfunktionen ved at lægge alle de kvadrerede fejl sammen:

$$\begin{aligned} E &= \frac{1}{2}((t_1 - o_1)^2 + (t_2 - o_2)^2 + \cdots + (t_n - o_n)^2) \\ &= \frac{1}{2}\sum_{i=1}^{n}(t_i - o_i)^2 \end{aligned}$$

Vi vil dog se på det tilfælde, hvor vi hele tiden ser på et træningseksempel ad gangen og på den baggrund opdaterer vægtene. Derfor vil vi begrænse os til at se på fejlfunktionen i (12).

Ønsket er nu, at bestemme biasene og vægtene (lad os bare samlet set kalde dem for vægtene fremover):

$$r_0, r_1, r_2, r_3, r_4, s_0, s_1, s_2, s_3, s_4, v_0, v_1, v_2, u_0, u_1, u_2, w_0, w_1, w_2$$

sådan at tabsfunktionen E bliver så lille som mulig. Det giver jo god mening! Find de vægte som gør, at netværket begår så lille en fejl, som overhovedet mulig. Ja da - det er det, vi gør! Vi betragter nu tabsfunktionen, som en funktion af alle vægtene:

$$E(r_0, r_1, r_2, r_3, r_4, s_0, s_1, s_2, s_3, s_4, v_0, v_1, v_2, u_0, u_1, u_2, w_0, w_1, w_2) = \frac{1}{2}(t-o)^2$$

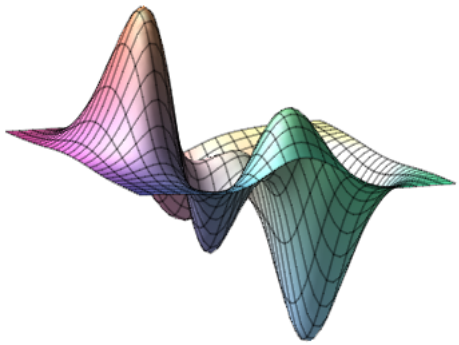
Det ser umiddelbart lidt mærkeligt ud, for vægtene ser jo ikke ud til at indgå på højreside i ovenstående udtryk, men husk på at outputværdien o jo netop bliver beregnet vha. feedforward, som er baseret på alle vægtene.

Vi vil altså gerne finde minimum for tabsfunktionen E, og vi ved jo godt, hvordan man finder minimum for en funktion af to variable: Sæt de partielle afledede lig med 0, løs de to ligninger og tjek op på at det rent faktisk er et minimum, du har fundet (og ikke f.eks. et lokalt maksimum eller et saddelpunkt).

I princippet kunne vi gøre noget tilsvarende her: E er en funktion af 19 variable (de 19 vægte). Sæt alle de 19 partielle afledede lig med 0, løs ligningerne og find ud af, at det er et (evt. lokalt) minimum, du har fundet. Problemet bliver bare, at I virkelighedens verden har vi ikke kun 19 vægte, men millionvis af vægte. Så det er en megastor opgave at finde funktionsudtryk for alle de partielle afledte og selvom vi forestiller os, vi kunne finde dem, eller de faldt ned fra månen, er der stadig et problem: At løse alle de millionvis af ligninger, som vi får, når vi sætter de partielle afledede lig med 0, viser sig simpelthen at blive alt for beregningsmæssigt tungt. Læs: Det tager for lang tid og/eller fylder for meget i hukommelsen - selv for en stor computer!

Derfor er man nødt til at gøre noget andet. Forestil dig at tabsfunktionen kun afhænger af to variable $E(v, w)$[1] og at grafen for den ser ud, som vist i figur 6. Her kan du se, at tabsfunktionen

[1] Du kan med god ret spørge: "Hvorfor nu det?". I virkeligheden afhænger den jo af millionvis af variable. Svaret er, at hvis vi skal forestille os grafen for tabsfunktionen, så er vores forestillingsevne begrænset til 3 dimensioner. Derfor må vi lige for en stund antage, at tabsfunktionen kun afhænger af to variable!



Figur 6: Eksempel på grafen for en tabsfunktion, som kun afhænger af to variable.

har to lokale minima og ét globalt minimum. Vi vil allerhelst finde vægtene sådan, at vi ender i det globale minimum, men kan vi kun finde et lokalt minimum, så kan det også gå (selvom det selvfølgelig ikke er det optimale). Forestil dig at grafen er et landskab. At finde grafens minimum svarer til, at du gerne vil ned i den dybeste dal. Din placering i landskabet svarer til, at du står i et eller andet punkt i planen (v_0, w_0) og din højde i vertikal retning er $E(v_0, w_0)$. Som du måske husker, så gælder der, at hvis du i dette punkt gerne vil bevæge dig i den retning, som er *allerstejlest*, så skal du gå i retningen givet ved gradienten:

$$\nabla E(v_0, w_0) = \begin{pmatrix} \frac{\partial E}{\partial v}(v_0, w_0) \\ \\ \frac{\partial E}{\partial w}(v_0, w_0) \end{pmatrix}$$

Det betyder også, at hvis du gerne vil gå *allermest ned ad bakke*, så skal du gå i den stik modsatte retning - altså minus gradienten:

$$-\nabla E(v_0, w_0) = \begin{pmatrix} -\frac{\partial E}{\partial v}(v_0, w_0) \\ \\ -\frac{\partial E}{\partial w}(v_0, w_0) \end{pmatrix}$$

Dét trick skal vi bruge! Vi starter altså med at vælge nogle mere eller mindre tilfældige vægte. Det svarer til, at du står et mere eller mindre tilfældigt sted i landskabet på figur 6. Så vælger vi at gå et lille stykke i den retning, hvor det går allermest ned ad bakke, ved at følge retningen angivet ved den negative gradient. Når vi står der, beregner vi gradienten i det punkt[2], og går

[2] Hov! Vi kan jo ikke finde funktionsudtryk for alle de partielle afledte, så hvad foregår der her? Jo, vi vil se, at vi ved backpropagation kan udregne de partielle afledte *i et punkt* uden at finde et funktionsudtryk.

igen et lille stykke i den retning, som den negative gradient angiver. Sådan fortsætter vi, og hvis vi sørger for ikke at tage alt for store skridt ad gangen, så vil vi til sidst ende i et lokalt minimum (der er desværre ingen garanti for, at vi vil ende i et globalt minimum[3]).

Idéen er altså at vi vælger en værdi for hver af de 19 vægte. Lad os bare som eksempel så på vægten s_1. Så beregner vi den partielle afledede

$$\frac{\partial E}{\partial s_1}$$

og så ændrer vi værdien af s_1 en lille smule i retning af den negative partielle afledede. Vi viser, at vi ændrer værdien af s_1 til en ny værdi, ved at bruge en pil:

$$s_1 \leftarrow s_1 - \eta \frac{\partial E}{\partial s_1} \tag{13}$$

Lad os lige bruge lidt tid på at forstå, hvad der står her! Venstresiden er den nye værdi af vægten s_1. Denne værdi beregnes ved hjælp af udtrykket på højreside. Her angiver s_1 den gamle/oprindelige værdi af s_1. Symbolet η (udtales "eta") er her et lille, positivt tal (f.eks. 0.05) som angiver, at vi bare gerne vil ændre s_1 en lille smule. Derfor må η ikke være al for stor. Man kalder også η for læringsraten eller på engelsk: *learning rate*. Endelig viser $-\frac{\partial E}{\partial s_1}$, at vi ønsker, at ændre alle vægte i retning af den negative gradient. Det vil nu sige, at alle 19 vægte opdateres vha. formler, som den i (13). Altså skal vi nu bare have fundet et udtryk for alle de 19 partielle afledede, og vi er i mål! Dette er netop, hvad **backpropagation algoritmen** gør, som vi gennemgår i det følgende.

[3] Hvis netværket ikke ender med at opføre sig tilfredsstillende, kan man jo prøve at starte et nyt tilfældigt sted i landskabet og gentage proceduren. Hvis man er heldig, lander man i et andet lokalt eller globalt minimum, hvor minimumsværdien er mindre end den tidligere.

VIDEO: Kunstige neurale netværk 3

I videoen her forklarer vi, hvad targetværdier er, og hvordan tabsfunktionen defineres.

https://www.youtube.com/embed/v2Jv5RTlZMw

VIDEO: Kunstige neurale netværk 4

I denne video bliver gradientnedstigning forklaret.

https://www.youtube.com/embed/ll3OO6Q2TuA

Opdatering af w-vægtene

Når man bruger backpropagation, starter man med at finde de partielle afledede for de vægte, som direkte påvirker outputværdien o. På figur 4 fremgår det, at det er vægtene w_0, w_1 og w_2 (husk at vi kalder vores bias for w_0). Lad os starte med at finde den partielle afledede for w_1. Ved at bruge kædereglen får vi:

$$\frac{\partial E}{\partial w_1} = \frac{dE}{do} \cdot \frac{\partial o}{\partial w_1}$$

Vi ved fra (12), at $E = \frac{1}{2}(t-o)^2$ og derfor er:

$$\frac{dE}{do} = \frac{1}{2} \cdot 2 \cdot (t-o) \cdot (-1) = -(t-o) \tag{14}$$

Fra (10) har vi, at $o = \sigma(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0)$ og derfor får vi

$$\frac{\partial o}{\partial w_1} = \sigma'(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0) \cdot z_1$$

Vi har tidligere vist, at $\sigma'(z) = \sigma(z)(1-\sigma(z))$ og derfor har vi

$$\frac{\partial o}{\partial w_1} = \sigma(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0)(1 - \sigma(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0)) \cdot z_1$$

Bruger vi nu, at $o = \sigma(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0)$ kan vi skrive ovenstående lidt mere kompakt:

$$\frac{\partial o}{\partial w_1} = o(1-o) \cdot z_1$$

Alt i alt får vi altså, at

$$\frac{\partial E}{\partial w_1} = \frac{dE}{do} \cdot \frac{\partial o}{\partial w_1} = -(t-o) \cdot o \cdot (1-o) \cdot z_1 \tag{15}$$

Vi kan nu udlede den første opdateringsregel for vægten w_1 ved at bruge idéen fra (13):

$$w_1 \leftarrow w_1 - \eta \cdot \frac{\partial E}{\partial w_1}$$

Indsættes udtrykket fra (15), får vi

$$w_1 \leftarrow w_1 - \eta \cdot (-(t-o) \cdot o \cdot (1-o) \cdot z_1)$$

Det vil sige, at

$$w_1 \leftarrow w_1 + \eta \cdot (t-o) \cdot o \cdot (1-o) \cdot z_1$$

Det er værd at dvæle lidt ved opdateringsleddet $\eta \cdot (t-o) \cdot o \cdot (1-o) \cdot z_1$ på højresiden fordi det faktisk giver intuitiv god mening. For det første er η, det vi som sagt kalder for vores *learning rate* - et lille positivt tal, som sørger for, at vi ikke tager for store skridt på vores vej ned i dalen (til det lokale minimum). Faktoren $t-o$ er jo netop fejlen. Nemlig forskellen mellem det vi ønsker t (target), og det som netværket giver o (output). Jo større fejl/forskel, desto mere må vi justere vægten. Ser vi på faktoren $o \cdot (1-o)$, så vil det være sådan, at hvis outputværdien o er tæt på enten 0 eller 1 (man siger at neuronen er "mættet"), så vil $o \cdot (1-o)$ være tæt på 0. Det vil sige, at hvis outputværdien er tæt på 0 eller 1, så ændrer vi heller ikke så meget på vægten. Endelig er der faktoren z_1, som er inputtet fra det foregående lag (se figur 4). Hvis værdien af denne er (numerisk) stor, så får det også stor betydning for opdateringsleddet (eller tænk på det omvendt: hvis z_1 er tæt på 0, så har z_1 alligevel ikke så stor indflydelse på outputværdien, og så giver det heller ikke mening, at justere så meget på den tilhørende vægt w_1).

Det viser sig faktisk, at faktoren $(t-o) \cdot o \cdot (1-o)$ kommer til at gå igen rigtige mange gange i det følgende. Det bliver i længden lidt tungt at slæbe rundt på. Derfor vælger vi at definere

$$\delta = (t-o) \cdot o \cdot (1-o) \tag{16}$$

og derfor kan opdateringsreglen for w_1 nu også skrives:

$$w_1 \leftarrow w_1 + \eta \cdot \delta \cdot z_1 \tag{17}$$

Helt analogt med ovenstående kan man udlede opdateringsregler for w_2 og w_0. Resultatet er samlet her.

Opdateringsregler for w-vægtene

$$w_0 \leftarrow w_0 + \eta \cdot \delta$$
$$w_1 \leftarrow w_1 + \eta \cdot \delta \cdot z_1$$
$$w_2 \leftarrow w_2 + \eta \cdot \delta \cdot z_2$$

hvor

$$\delta = (t - o) \cdot o \cdot (1 - o)$$

Men hvordan foregår det der med de opdateringsregler så egentligt? Jo altså vi starter med at sætte vægtene mere eller mindre tilfældigt. Så laver vi ved hjælp af vores træningseksempel $(\vec{x}, t)$ et **feedforward** i netværket, som det er beskrevet i afsnit . Derfor får vi beregnet outputværdien o samt z_1 og z_2 (husk at z_1 og z_2 bruges til at beregne o). Desuden kender vi jo fra vores træningsdata target-værdien t. Og voila! Alt hvad der indgår på højresiderne i ovenstående opdateringsregler har vi nu adgang til, og vi kan derfor beregne de nye w vægte.

Så mangler vi bare at finde opdateringsreglerne for de restende vægte!

VIDEO: Kunstige neurale netværk 5

I videoen her forklarer vi hvordan w-vægtene opdateres.

https://www.youtube.com/embed/X5g4h3cKSok

Opdatering af u- og v-vægtene

Vi går nu et trin længere tilbage i netværket - væk fra outputlaget. Her kan vi se neuronerne, som fyrer værdierne z_1 og z_2, som bliver påvirket af u- og v-vægtene. Lad os her starte med at bestemme opdateringsreglerne for v-vægtene. For at gøre det skal vi finde ud af hvordan v-vægtene påvirker neuronerne længere fremme i netværket. Se igen på figur 4. Her er det tydeligt, at v-vægtene påvirker den mørkegrønne neuron, som

fyrer værdien z_1, som igen påvirker outputværdien. Derfor kan vi bruge kædereglen på følgende måde:

$$\frac{\partial E}{\partial v_1} = \frac{dE}{do} \cdot \frac{\partial o}{\partial z_1} \cdot \frac{\partial z_1}{\partial v_1}$$

Vi ved allerede fra (14), at

$$\frac{dE}{do} = -(t - o)$$

Den partielle afledede af o med hensyn til z_1 finder vi ved at bruge definitionen af outputværiden o i (10)

$$\begin{aligned}\frac{\partial o}{\partial z_1} &= \sigma'(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0) \cdot w_1 \\ &= \sigma(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0) \cdot (1 - \sigma(w_1 \cdot z_1 + w_2 \cdot z_2 + w_0)) \cdot w_1 \\ &= o \cdot (1 - o) \cdot w_1\end{aligned} \tag{18}$$

hvor vi igen har brugt sætning 0.1. Og endelig ved at udnytte definitionen af z_1 i (8) får vi, at

$$\frac{\partial z_1}{\partial v_1} = \sigma'(v_1 \cdot y_1 + v_2 \cdot y_2 + v_0) \cdot y_1 \tag{19}$$

$$= \sigma(v_1 \cdot y_1 + v_2 \cdot y_2 + v_0) \cdot (1 - \sigma(v_1 \cdot y_1 + v_2 \cdot y_2 + v_0)) \cdot y_1 \tag{20}$$

$$= z_1 \cdot (1 - z_1) \cdot y_1 \tag{21}$$

Sætter vi det hele sammen får vi, at

$$\frac{\partial E}{\partial v_1} = \underbrace{-(t - o)}_{\frac{\partial E}{\partial o}} \cdot \underbrace{o \cdot (1 - o) \cdot w_1}_{\frac{\partial o}{\partial z_1}} \cdot \underbrace{z_1 \cdot (1 - z_1) \cdot y_1}_{\frac{\partial z_1}{\partial v_1}}$$

og bruger vi definitionen af δ i (16) får vi et lidt mere kompakt udtryk

$$\frac{\partial E}{\partial v_1} = -\delta \cdot w_1 \cdot z_1 \cdot (1 - z_1) \cdot y_1$$

Opdateringsreglen for v_1 bliver derfor

$$v_1 \leftarrow v_1 - \eta \cdot \frac{\partial E}{\partial v_1}$$

og med det netop udledte udtryk for $\frac{\partial E}{\partial v_1}$ får vi

$$v_1 \leftarrow v_1 - \eta \cdot (-\delta \cdot w_1 \cdot z_1 \cdot (1 - z_1) \cdot y_1)$$

Det vil sige, at

$$v_1 \leftarrow v_1 + \eta \cdot \delta \cdot w_1 \cdot z_1 \cdot (1 - z_1) \cdot y_1$$

Læg igen mærke til, at når vi har været igennem et feedforward i netværket, så kender vi alle de størrelser, som indgår i ovenstående udtryk.

På helt tilsvarende vis kan man bestemme opdateringsreglerne for v_0 og v_2. De tre opdateringsregler for v-vægtene ses her:

Opdateringsregler for v-vægtene

$$v_0 \leftarrow v_0 + \eta \cdot \delta \cdot w_1 \cdot z_1 \cdot (1 - z_1) \tag{22}$$
$$v_1 \leftarrow v_1 + \eta \cdot \delta \cdot w_1 \cdot z_1 \cdot (1 - z_1) \cdot y_1 \tag{23}$$
$$v_2 \leftarrow v_2 + \eta \cdot \delta \cdot w_1 \cdot z_1 \cdot (1 - z_1) \cdot y_2 \tag{24}$$
$$\tag{25}$$

hvor

$$\delta = (t - o) \cdot o \cdot (1 - o)$$

Opdateringsreglerne for u-vægtene findes på præcis samme måde. Her skal man blot se, at u-vægtene har indflydelse på outputtet via z_2 (se figur 4). Derfor skal man f.eks. finde den partielle afledede af E med hensyn til u_1 ved at bruge kædereglen på denne måde

$$\frac{\partial E}{\partial u_1} = \frac{dE}{do} \cdot \frac{\partial o}{\partial z_2} \cdot \frac{\partial z_2}{\partial u_1}$$

Udregninger svarende til det netop gennemgåede giver os

Opdateringsregler for u-vægtene

$$u_0 \leftarrow u_0 + \eta \cdot \delta \cdot w_2 \cdot z_2 \cdot (1 - z_2) \tag{26}$$
$$u_1 \leftarrow u_1 + \eta \cdot \delta \cdot w_2 \cdot z_2 \cdot (1 - z_2) \cdot y_1 \tag{27}$$
$$u_2 \leftarrow u_2 + \eta \cdot \delta \cdot w_2 \cdot z_2 \cdot (1 - z_2) \cdot y_2 \tag{28}$$
$$\tag{29}$$

hvor

$$\delta = (t - o) \cdot o \cdot (1 - o)$$

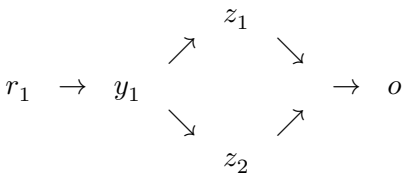
VIDEO: Kunstige neurale netværk 6

I videoen her forklarer vi, hvordan v-vægtene opdateres.

https://www.youtube.com/embed/CVG2lh9T6lg

Opdatering af r- og s-vægtene

Så er vi endelig fremme ved r- og s vægtene. Start lige med at tage en dyb indånding! Nu bliver det lidt mere kompliceret. Se på figur 4. Lad os starte med at finde den partielle afledede af E med hensyn til r_1. Når man ser på netværket, kan man se, at r_1 i første omgang påvirker y_1, y_1 påvirker både z_1 og z_2, som så til sidst påvirker outputværdien o. Det kan illustreres sådan her



Balladen er, at y_1 både påvirker z_1 og z_2, og det gør det hele lidt mere kompliceret. Lad os lige starte med at se bort fra det. Ifølge kædereglen får vi så:

$$\frac{\partial E}{\partial r_1} = \frac{dE}{do} \cdot \frac{\partial o}{\partial y_1} \cdot \frac{\partial y_1}{\partial r_1}$$

Men så var det jo, at o i virkeligheden afhænger af y_1 både via z_1 og z_2. Man kunne skrive det sådan her:

$$o(z_1(y_1), z_2(y_1))$$

Bemærk, at z_1 og z_2 jo også afhænger af y_2, men når vi skal differentiere med hensyn til y_1, så er y_2 at betragte som en konstant. Og når konstanter bliver differentieret, så giver det som bekendt 0.

Derfor: For at finde den partielle afledede af o med hensyn til y_1 må vi benytte kædereglen for funktioner af flere variable. Den siger, at

$$\frac{\partial o}{\partial y_1} = \frac{\partial o}{\partial z_1} \cdot \frac{\partial z_1}{\partial y_1} + \frac{\partial o}{\partial z_2} \cdot \frac{\partial z_2}{\partial y_1}$$

Det samlede udtryk for den partielle afledede af E med hensyn til r_1 bliver derfor

$$\frac{\partial E}{\partial r_1} = \frac{dE}{do} \cdot \left(\frac{\partial o}{\partial z_1} \cdot \frac{\partial z_1}{\partial y_1} + \frac{\partial o}{\partial z_2} \cdot \frac{\partial z_2}{\partial y_1} \right) \cdot \frac{\partial y_1}{\partial r_1} \tag{30}$$

Vi finder hver af de afledede, som indgår i ovenstående udtryk én ad gangen. Vi ved allerede fra (14), at

$$\frac{dE}{do} = \frac{1}{2} \cdot 2 \cdot (t - o) \cdot (-1) = -(t - o)$$

Vi ved også fra (18), at

$$\frac{\partial o}{\partial z_1} = o \cdot (1 - o) \cdot w_1$$

Differentieres z_1 (se (8)) med hensyn til y_1 får vi

$$\frac{\partial z_1}{\partial y_1} = \sigma'(v_1 \cdot y_1 + v_2 \cdot y_2 + v_0) \cdot v_1 \tag{31}$$

$$= z_1 \cdot (1 - z_1) \cdot v_1 \tag{32}$$

hvor vi igen har brugt sætning 0.1 og definitionen af z_1 i (8). Helt tilsvarende kan vi finde $\frac{\partial o}{\partial z_2}$ og $\frac{\partial z_2}{\partial y_1}$ (se (9))

$$\frac{\partial o}{\partial z_2} = o \cdot (1 - o) \cdot w_2$$

og

$$\frac{\partial z_2}{\partial y_1} = z_2 \cdot (1 - z_2) \cdot u_1$$

Den sidste partielle afledede $\frac{\partial y_1}{\partial r_1}$ finder vi ved at differentiere udtrykket for y_1 i (4), hvor vi endnu engang udnytter sætning 0.1.

Indsætter vi nu alle de udtryk, som vi netop har udledt, i (30) får vi et temmelig langt udtryk for $\frac{\partial E}{\partial r_1}$:

$$\frac{\partial E}{\partial r_1} = \underbrace{-(t-o)}_{\frac{dE}{do}} \cdot \tag{33}$$

$$\Big(\underbrace{o \cdot (1-o) \cdot w_1}_{\frac{\partial o}{\partial z_1}} \cdot \underbrace{z_1 \cdot (1-z_1) \cdot v_1}_{\frac{\partial z_1}{\partial y_1}} + \underbrace{o \cdot (1-o) \cdot w_2}_{\frac{\partial o}{\partial z_2}} \cdot \underbrace{z_2 \cdot (1-z_2) \cdot u_1}_{\frac{\partial z_2}{\partial y_1}} \Big) \cdot \tag{34}$$

$$\underbrace{y_1 \cdot (1-y_1) \cdot x_1}_{\frac{\partial y_1}{\partial r_1}} \tag{35}$$

Og sætter vi $o \cdot (1-o)$ uden for parentesen og erstatter $(t-o) \cdot o \cdot (1-o)$ med δ får vi

$$\frac{\partial E}{\partial r_1} = -\delta \cdot \Big(w_1 \cdot z_1 \cdot (1-z_1) \cdot v_1 + w_2 \cdot z_2 \cdot (1-z_2) \cdot u_1 \Big) \cdot y_1 \cdot (1-y_1) \cdot x_1$$

Helt i tråd med tidligere får vi altså følgende opdateringsregel for r_1

$$r_1 \leftarrow r_1 - \eta \cdot \frac{\partial E}{\partial r_1}$$

Det vil sige

$$r_1 \leftarrow r_1 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1-z_1) \cdot v_1 + w_2 \cdot z_2 \cdot (1-z_2) \cdot u_1 \Big) \cdot y_1 \cdot (1-y_1) \cdot x_1$$

Udleder man tilsvarende opdateringsregler for r_2, r_3, r_4 og r_0 vil man se, at det eneste som kommer til at ændre sig i ovenstående er den sidste faktor x_1, som bliver erstattet med henholdsvis x_2, x_3, x_4 og 1. Derfor får vi samlet set

Opdateringsregler for r-vægtene

$$r_0 \leftarrow r_0 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_1 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_1 \Big) \cdot y_1 \cdot (1 - y_1) \tag{36}$$

$$r_1 \leftarrow r_1 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_1 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_1 \Big) \cdot y_1 \cdot (1 - y_1) \cdot x_1 \tag{37}$$

$$r_2 \leftarrow r_2 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_1 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_1 \Big) \cdot y_1 \cdot (1 - y_1) \cdot x_2 \tag{38}$$

$$r_3 \leftarrow r_3 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_1 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_1 \Big) \cdot y_1 \cdot (1 - y_1) \cdot x_3 \tag{39}$$

$$r_4 \leftarrow r_4 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_1 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_1 \Big) \cdot y_1 \cdot (1 - y_1) \cdot x_4 \tag{40}$$

$$\tag{41}$$

hvor

$$\delta = (t - o) \cdot o \cdot (1 - o)$$

Opdateringen af s-vægtene foregår på samme måde. Hvis du ser på figur 4, kan du se, at alle s-vægtene påvirker y_2, som så påvirker både z_1 og z_2, som i sidste ende påvirker outputtet o. Ser vi generelt på vægten s_i, hvor $i = 0, 1, 2, 3$ eller 4, har vi altså

$$\begin{array}{ccccccc} & & & & z_1 & & \\ & & & \nearrow & & \searrow & \\ s_i & \rightarrow & y_2 & & & & \rightarrow \quad o \\ & & & \searrow & & \nearrow & \\ & & & & z_2 & & \end{array}$$

Som tidligere kan vi starte med at skrive

$$\frac{\partial E}{\partial s_i} = \frac{dE}{do} \cdot \frac{\partial o}{\partial y_2} \cdot \frac{\partial y_2}{\partial s_i}$$

og bruger vi igen kædreglen for funktioner af flere variable, får vi

$$\frac{\partial E}{\partial s_i} = \frac{dE}{do} \cdot \left(\frac{\partial o}{\partial z_1} \cdot \frac{\partial z_1}{\partial y_2} + \frac{\partial o}{\partial z_2} \cdot \frac{\partial z_2}{\partial y_2} \right) \cdot \frac{\partial y_2}{\partial s_i}$$

I ovenstående udtryk bliver det klart, at opdateringsreglerne vil blive ens bortset fra den sidste faktor.

Nu udledes alle de partielle afledede, fuldstændig som for r-vægtene og vi ender med følgende opdateringsregler for s-vægtene:

Opdateringsregler for s-vægtene

$$s_0 \leftarrow s_0 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_2 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_2\Big) \cdot y_2 \cdot (1 - y_2) \tag{42}$$

$$s_1 \leftarrow s_1 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_2 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_2\Big) \cdot y_2 \cdot (1 - y_2) \cdot x_1 \tag{43}$$

$$s_2 \leftarrow s_2 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_2 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_2\Big) \cdot y_2 \cdot (1 - y_2) \cdot x_2 \tag{44}$$

$$s_3 \leftarrow s_3 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_2 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_2\Big) \cdot y_2 \cdot (1 - y_2) \cdot x_3 \tag{45}$$

$$s_4 \leftarrow s_4 + \eta \cdot \delta \cdot \Big(w_1 \cdot z_1 \cdot (1 - z_1) \cdot v_2 + w_2 \cdot z_2 \cdot (1 - z_2) \cdot u_2\Big) \cdot y_2 \cdot (1 - y_2) \cdot x_4 \tag{46}$$

(47)

hvor

$$\delta = (t - o) \cdot o \cdot (1 - o)$$

Det var faktisk det! Altså det blev jo en værre omgang bogstavgymnastik, men faktum er, at vi er i mål med at udlede backpropagation algoritmen for vores simple netværk i figur 4. Hurra for det!

VIDEO: Kunstige neurale netværk 7

I denne video forklares hvordan r-vægtene opdateres.

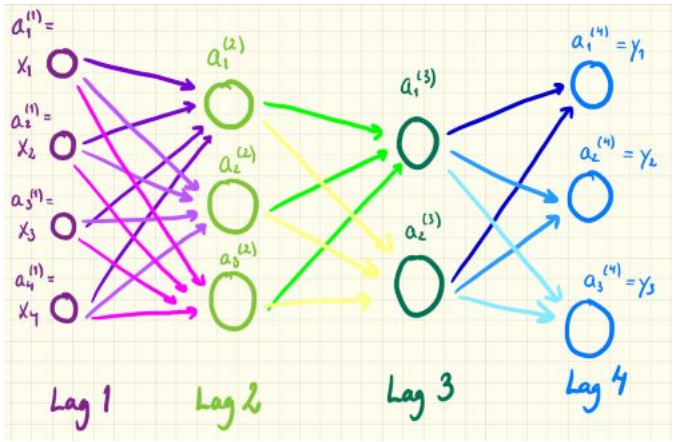
https://www.youtube.com/embed/fUFH8hVvGMU

Fra bogstaver til indekser

Det står på nuværende tidspunkt nok klart for de fleste, at den notation, som vi har anvendt i de foregående afsnit, er en lille smule tung. Der er bare rigtig mange bogstaver, og det kan være svært at huske om et givet bogstav betegner en outputværdi fra en neuron eller, om det er en vægt. Desuden skalerer notationen ufattelig dårligt! Forestil dig, at vi tilføjer $2-3$ ekstra lag til vores netværk - det bliver svært at blive ved med at finde nye bogstaver!

Derfor griber man traditionelt set notationen i forbindelse med kunstige neurale netværk lidt anderledes an, så det skalerer bedre, og så det bliver nemmere at læse (i hvert tilfælde når man lige har vænnet sig til de ekstra indekser, som vi bliver nødt til at indføre).

Lad os se på netværket i figur 7. Dette netværk har 3 outputværdier i stedet for én, og der er ændret på antallet af neuroner i det ene af de skjulte lag i forhold til tidligere.



Figur 7: Kunstigt neuralt netværk med flere outputneuroner.

Hvert lag i netværket er nu nummeret fortløbende fra 1 til 4. Desuden giver vi nu en samlet betegnelse for den værdi, som hver neuron "spytter ud". F.eks. vil den 3. neuron i det 2. lag "outputte" eller "fyre" værdien

$$a_3^{(2)}$$

Det vil altså sige, at det tal, som står hævet i parentesen, refererer til laget og det tal, som er sænket, refererer til nummeret på rækken i det givne lag.

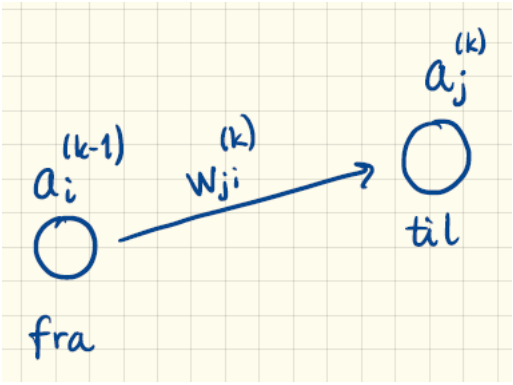
Bemærk også at inputværdierne i det første lag nu har to forskellige betegnelser for det samme:

$$x_i = a_i^{(1)}$$

og tilsvarende har outputværdierne i det sidste og fjerde lag også to forskellige betegnelser:

$$y_i = a_i^{(4)}$$

Lad os nu se på hvordan **feedforward** virker med vores nye notation. For det første er vi nødt til at være lidt smartere end tidligere i forhold til, hvad vi kalder vores vægte og bias. Der er tradition for, at man navngiver vægtene, som det ses på figur 8.



Figur 8: Navngivning af vægte.

Her er tanken, at vi gerne vil beregne outputværdien fra den j'te neuron i det k'te lag. Det gør vi ved at vægte alle outputværdierne fra det foregående lag: $(k-1)$. Den vægt, som vi ganger outputværdien fra den i'te neuron i det $(k-1)'te$ lag $(a_i^{(k-1)})$ med, og som skal bruges for at beregne $a_j^{(k)}$, vælger vi at kalde for

$$w_{ji}^{(k)}$$

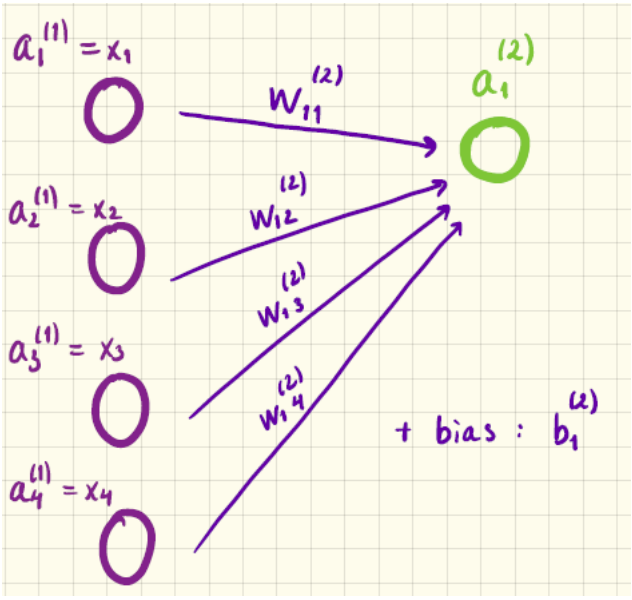
Hvis man tænker på, at vi skal *til* række j i *lag* k *fra* række i, så kan man måske huske på notationen sådan her:

$$w_{\text{til fra}}^{(\text{lag})}$$

Se også hvordan det passer med figur 8.

Feedforward med indekser

Vi vil nu se på, hvordan de forskellige $a_j^{(k)}$-værdier beregnes. Det vil sige, at vi altså skal se på, hvordan de forskellige feedforward-ligninger ser ud med vores nye notation.



Figur 9: Udregning af $a_1^{(2)}$.

Lad os se på et konkret eksempel, så bliver det lidt nemmere at forholde sig til. Vi starter med at udregne outputværdien $a_1^{(2)}$ for den første neuron i det andet lag. Denne neuron får input fra alle neuroner i det foregående lag (som her er inputlaget). Bruger vi den notation for vægtene, som vi lige har indført, så starter vi med at beregne:

$$z_1^{(2)} = w_{11}^{(2)} \cdot x_1 + w_{12}^{(2)} \cdot x_2 + w_{13}^{(2)} \cdot x_3 + w_{14}^{(2)} \cdot x_4 + b_1^{(2)}$$

Der er to ting at bemærke her: 1) Vi vælger, at kalde udtrykket på højreside for $z_1^{(2)}$ og, 2) vi har kaldt biasen for $b_1^{(2)}$.

Bruger vi nu de mere generelle udtryk for inputværdierne

$a_1^{(1)}, a_2^{(1)}, \ldots, a_4^{(1)}$ kan vi skrive:

$$z_1^{(2)} = w_{11}^{(2)} \cdot a_1^{(1)} + w_{12}^{(2)} \cdot a_2^{(1)} + w_{13}^{(2)} \cdot a_3^{(1)} + w_{14}^{(2)} \cdot a_4^{(1)} + b_1^{(2)} \tag{48}$$

$$= \sum_{i=1}^{4} w_{1i}^{(2)} a_i^{(1)} + b_1^{(2)} \tag{49}$$

Og endelig finder vi outputværdien $a_1^{(2)}$ for den første neuron i det andet lag ved som tidligere at anvende sigmoid-funktionen på ovenstående udtryk:

$$a_1^{(2)} = \sigma(z_1^{(2)}) \tag{50}$$

$$= \sigma\left(\sum_{i=1}^{4} w_{1i}^{(2)} a_i^{(1)} + b_1^{(2)}\right) \tag{51}$$

Det her er faktisk notationsmæssigt selve idéen. Folder vi det ud til hele det andet lag får vi derfor:

Feedforwardligninger til lag 2

Beregn først:

$$z_1^{(2)} = \sum_{i=1}^{4} w_{1i}^{(2)} a_i^{(1)} + b_1^{(2)} \tag{52}$$

(53)

$$z_2^{(2)} = \sum_{i=1}^{4} w_{2i}^{(2)} a_i^{(1)} + b_2^{(2)} \tag{54}$$

(55)

$$z_3^{(2)} = \sum_{i=1}^{4} w_{3i}^{(2)} a_i^{(1)} + b_3^{(2)} \tag{56}$$

(57)

Outputværdierne for neuroner i det andet lag udregnes

dernæst på denne måde:

$$a_1^{(2)} = \sigma(z_1^{(2)}) \tag{58}$$
$$\tag{59}$$
$$a_2^{(2)} = \sigma(z_2^{(2)}) \tag{60}$$
$$\tag{61}$$
$$a_3^{(2)} = \sigma(z_3^{(2)}) \tag{62}$$
$$\tag{63}$$

Og vover vi pelsen, kan vi helt generelt skrive:

Feedforward-ligninger til lag 2

Beregn først:

$$z_j^{(2)} = \sum_{i=1}^{4} w_{ji}^{(2)} a_i^{(1)} + b_j^{(2)} \tag{64}$$

Outputværdierne for neuroner i det andet lag udregnes dernæst på denne måde:

$$a_j^{(2)} = \sigma(z_j^{(2)}) \tag{65}$$

for $j \in \{1, 2, 3\}$.

Fordelen ved denne notation er, at det nu er utrolig nemt at opskrive feedforward-ligningerne for lag 3 og 4 - det er blot nogle indekser, som skal ændres lidt. I det tredje lag er der to neuroner, hvis outputværdier beregnes på denne måde:

Feedforward-ligninger til lag 3

Beregn først:

$$z_j^{(3)} = \sum_{i=1}^{3} w_{ji}^{(3)} a_i^{(2)} + b_j^{(3)} \tag{66}$$

Outputværdierne for neuroner i det tredje lag udregnes dernæst på denne måde:

$$a_j^{(3)} = \sigma(z_j^{(3)}) \tag{67}$$

for $j \in \{1, 2\}$.

Og endelig beregnes outputtet fra hele netværket i det fjerde lag:

Feedforward-ligninger til lag 4

Udregn først:

$$z_j^{(4)} = \sum_{i=1}^{2} w_{ji}^{(4)} a_i^{(3)} + b_j^{(4)} \tag{68}$$

Outputværdierne for neuroner i det tredje lag udregnes dernæst på denne måde:

$$y_j = a_j^{(4)} = \sigma(z_j^{(4)}) \tag{69}$$

for $j \in \{1, 2, 3\}$.

Det fremgår nu tydeligt, at feedforward-ligningerne er på fuldstændig samme form, og vi vil derfor helt generelt kunne skrive:

Feedforward-ligninger generelt

Beregn først:

$$z_j^{(k)} = \sum_{i} w_{ji}^{(k)} a_i^{(k-1)} + b_j^{(k)} \tag{70}$$

Outputværdierne for neuroner i det k'te lag udregnes dernæst på denne måde:

$$a_j^{(k)} = \sigma(z_j^{(k)}) \tag{71}$$

for $k \in \{2, 3, 4\}$.

Når man bruger feedforward, starter man altså med at udregne outputværdierne for det første skjulte lag, dernæst for det andet skjulte lag og så videre, indtil man når til outputværdierne for selve netværket (deraf navnet: *feedforward*). Bemærk her, at det ikke giver mening at udregne $a_j^{(1)}$, fordi det svarer til

inputværdierne til netværket.

Backpropagation med indekser

Lad os nu se på hvordan backpropagation fungerer. Vi skal altså have opskrevet vores opdateringsregler med den nye notation, og vi vil gribe det an, ligesom vi gjorde det i afsnit . Nemlig ved at starte i det sidste lag (her lag 4) og finde opdateringsreglerne for de vægte og bias, som har direkte indflydelse på outputværdierne fra lag 4.

Opdateringsregler for lag 4

Vi er altså i første omgang på jagt efter

$$\frac{\partial E}{\partial w_{ji}^{(4)}} \quad \text{og} \quad \frac{\partial E}{\partial b_j^{(4)}},$$

for $j \in \{1,2,3\}$, $i \in \{1,2\}$. Tabsfunktionen E, som hører til netværket i figur 7, bliver her:

$$E = \frac{1}{2}\sum_{j=1}^{3}\left(t_j - y_j\right)^2 = \frac{1}{2}\sum_{j=1}^{3}\left(t_j - a_j^{(4)}\right)^2 \tag{72}$$

hvor igen t_j er den ønskede target-værdi for den j'te outputneuron.

Lad os starte med at bestemme $\frac{\partial E}{\partial w_{ji}^{(4)}}$. Vi må derfor først se på, hvordan $w_{ji}^{(4)}$ påvirker tabsfunktionen E. Da $w_{ji}^{(4)}$ kun indgår i udtrykket for beregningen af $z_j^{(4)}$, som igen bruges til beregningen af $a_j^{(4)}$, som dernæst direkte påvirker tabsfunktionen, kan vi skrive:

$$w_{ji}^{(4)} \to z_j^{(4)} \to a_j^{(4)} \to E$$

Bruger vi først kædereglen én gang, får vi derfor

$$\frac{\partial E}{\partial w_{ji}^{(4)}} = \frac{\partial E}{\partial z_j^{(4)}} \cdot \frac{\partial z_j^{(4)}}{\partial w_{ji}^{(4)}} \tag{73}$$

og bruges kædereglen igen, kan første faktor udfoldes yderligere

$$\frac{\partial E}{\partial z_j^{(4)}} = \frac{\partial E}{\partial a_j^{(4)}} \cdot \frac{\partial a_j^{(4)}}{\partial z_j^{(4)}}$$

Lad os starte med at udregne $\frac{\partial E}{\partial z_j^{(4)}}$ ved at udregne hver faktor på højresiden i ovenstående udtryk for sig. Fra (72) får vi, at

$$E = \frac{1}{2}\sum_{j=1}^{3}\left(t_j - a_j^{(4)}\right)^2 = \frac{1}{2}\left(\left(t_1 - a_1^{(4)}\right)^2 + \left(t_2 - a_2^{(4)}\right)^2 + \left(t_3 - a_3^{(4)}\right)^2\right)$$

Hvis vi f.eks. skal differentiere ovenstående med hensyn til $a_2^{(4)}$, kan vi se at alle de led, som ikke indeholder $a_2^{(4)}$, vil være at betragte som konstanter, når vi differentierer - og når vi differentierer konstanter, får vi som bekendt 0. Derfor får vi, at

$$\frac{\partial E}{\partial a_2^{(4)}} = \frac{1}{2} \cdot 2 \cdot (t_2 - a_2^{(4)}) \cdot (-1) = -(t_2 - a_2^{(4)})$$

På tilsvarende vis har vi derfor generelt, at

$$\frac{\partial E}{\partial a_j^{(4)}} = -(t_j - a_j^{(4)}) \tag{74}$$

Vi ved også, at

$$a_j^{(4)} = \sigma(z_j^{(4)})$$

Og bruger vi endnu en gang resultatet fra sætning 0.1 får vi, at

$$\frac{\partial a_j^{(4)}}{\partial z_j^{(4)}} = \sigma'(z_j^{(4)}) = \sigma(z_j^{(4)}) \cdot (1 - \sigma(z_j^{(4)})) = a_j^{(4)} \cdot (1 - a_j^{(4)}) \tag{75}$$

Indtil videre har vi altså, at

$$\frac{\partial E}{\partial z_j^{(4)}} = \frac{\partial E}{\partial a_j^{(4)}} \cdot \frac{\partial a_j^{(4)}}{\partial z_j^{(4)}} \tag{76}$$

$$= -(t_j - a_j^{(4)}) \cdot a_j^{(4)} \cdot (1 - a_j^{(4)}) \tag{77}$$

I forhold til det videre arbejde viser det sig hensigtsmæssigt, at lave en samlet betegnelse for

$$\frac{\partial E}{\partial z_j^{(4)}} = -(t_j - a_j^{(4)}) \cdot a_j^{(4)} \cdot (1 - a_j^{(4)}) \tag{78}$$

Vi sætter derfor

$$\delta_j^{(4)} = \frac{\partial E}{\partial z_j^{(4)}} = -(t_j - a_j^{(4)}) \cdot a_j^{(4)} \cdot (1 - a_j^{(4)})$$

Udtrykket $\delta_j^{(4)}$ kalder man også for fejlleddet for det fjerde lag, men det kommer vi tilbage til senere.

Vi har nu fundet den første faktor i (73), og mangler derfor kun at bestemme $\frac{\partial z_j^{(4)}}{\partial w_{ji}^{(4)}}$. Bruger vi (68) ser vi, at

$$z_j^{(4)} = \sum_{i=1}^{2} w_{ji}^{(4)} a_i^{(3)} + b_j^{(4)} = w_{j1}^{(4)} a_1^{(3)} + w_{j2}^{(4)} a_2^{(3)} + b_j^{(4)}$$

Skal vi f.eks. differentiere dette udtryk med hensyn til $w_{j1}^{(4)}$, får vi (fordi de fleste led i ovenstående, vil være at betragte som konstanter)

$$\frac{\partial z_j^{(4)}}{\partial w_{j1}^{(4)}} = a_1^{(3)}$$

Og helt tilsvarende hvis vi differentierer med hensyn til $w_{j2}^{(4)}$, får vi

$$\frac{\partial z_j^{(4)}}{\partial w_{j2}^{(4)}} = a_2^{(3)}$$

Generelt har vi derfor, at

$$\frac{\partial z_j^{(4)}}{\partial w_{ji}^{(4)}} = a_i^{(3)} \tag{79}$$

Samler vi nu de tre udtryk, som vi netop har udledt og indsætter i (73) får vi

$$\frac{\partial E}{\partial w_{ji}^{(4)}} = -(t_j - a_j^{(4)}) \cdot a_j^{(4)} \cdot (1 - a_j^{(4)}) \cdot a_i^{(3)}$$

og med den lidt kortere notation, som vi indførte ovenfor, kan vi nu skrive

$$\frac{\partial E}{\partial w_{ji}^{(4)}} = \delta_j^{(4)} \cdot a_i^{(3)}$$

For at finde opdateringsreglerne for biasene, må vi først bestemme de partielle afledede af E med hensyn til $b_j^{(4)}$. På helt tilsvarende vis får vi, at

$$\frac{\partial E}{\partial b_j^{(4)}} = \frac{\partial E}{\partial z_j^{(4)}} \cdot \frac{\partial z_j^{(4)}}{\partial b_j^{(4)}}$$

Vi ved allerede, at

$$\frac{\partial E}{\partial z_j^{(4)}} = \delta_j^{(4)} \tag{80}$$

og ser man på ligningen i (68), ses det nemt, at

$$\frac{\partial z_j^{(4)}}{\partial b_j^{(4)}} = 1$$

og derfor har vi, at

$$\frac{\partial E}{\partial b_j^{(4)}} = \delta_j^{(4)}$$

Opdateringsreglerne for de vægte og bias, som hører til outputlaget (lag 4) er derfor

$$w_{ji}^{(4)} \leftarrow w_{ji}^{(4)} - \eta \cdot \frac{\partial E}{\partial w_{ji}^{(4)}} = w_{ji}^{(4)} - \eta \cdot \delta_j^{(4)} \cdot a_i^{(3)}$$

og

$$b_j^{(4)} \leftarrow b_j^{(4)} - \eta \cdot \frac{\partial E}{\partial b_j^{(4)}} = b_j^{(4)} - \eta \cdot \delta_j^{(4)}$$

Vi kan altså opsummere:

Opdateringsregler til vægte og bias i outputlaget (lag 4)

Vægtene i outputlaget opdateres på denne måde:

$$w_{ji}^{(4)} \leftarrow w_{ji}^{(4)} - \eta \cdot \delta_j^{(4)} \cdot a_i^{(3)}$$

Biasene i outputlaget opdateres på denne måde:

$$b_j^{(4)} \leftarrow b_j^{(4)} - \eta \cdot \delta_j^{(4)}$$

hvor

$$\delta_j^{(4)} = \frac{\partial E}{\partial z_j^{(4)}} = -(t_j - a_j^{(4)}) \cdot a_j^{(4)} \cdot (1 - a_j^{(4)}) \qquad (81)$$

Udtrykket $\delta_j^{(4)}$ kalder man, som nævnt tidligere, også for fejlleddet i den j'te række i det fjerde lag, og man kan se på ovenstående opdateringsregler, at dette fejlled netop indgå i opdateringen af både vægtene og biasene. Faktisk kan vi præcis, som vi gjorde det på side , tillægge dette fejlled en intuitiv god mening. Det kommer vi tilbage til igen senere!

Bemærk, at hvis vi i vores netværk starter med at vælge mere eller mindre tilfældige vægte, så kan vi på baggrund af dem bruge feedforwardligningerne til at udregne, $a_j^{(4)}$- og $a_i^{(3)}$-værdierne. Samtidig kender vi target-værdierne t_j, og vi kan derfor også udregne fejlleddene $\delta_j^{(4)}$. Vi har altså alt, hvad vi skal bruge for at benytte ovenstående opdateringsregler.

Opdateringsregler for lag 3

Vi bevæger os nu et trin længere bagud i netværket og udleder opdateringsreglerne for det næstsidste lag - lag 3. Altså skal vi have bestemt

$$\frac{\partial E}{\partial w_{ji}^{(3)}} \quad \text{og} \quad \frac{\partial E}{\partial b_j^{(3)}},$$

for $j \in \{1, 2\}$, $i \in \{1, 2, 3\}$. Vi må igen se på, hvordan $w_{ji}^{(3)}$ påvirker tabsfunktionen E. Ser vi på figur figur 7, kan vi se, at $w_{ji}^{(3)}$ direkte påvirker $z_j^{(3)}$, som igen direkte påvirker $a_j^{(3)}$. Nu vil den j'te neuron i det tredje lag fyre værdien $a_j^{(3)}$ til alle neuroner i det fjerde lag. Altså vil $a_j^{(3)}$ påvirke $z_1^{(4)}, z_2^{(4)}$ og $z_3^{(4)}$, som bruges til beregning af $a_1^{(4)}, a_2^{(4)}$ og $a_3^{(4)}$, som så igen vil påvirke tabsfunktionen E. Det kan illustreres på denne måde

$$w_{ji}^{(3)} \rightarrow z_j^{(3)} \rightarrow a_j^{(3)} \begin{array}{ccccc} & \nearrow & z_1^{(4)} \rightarrow a_1^{(4)} & \searrow & \\ \rightarrow & & z_2^{(4)} \rightarrow a_2^{(4)} & & \rightarrow \; E \\ & \searrow & z_3^{(4)} \rightarrow a_3^{(4)} & \nearrow & \end{array}$$

I første omgang kan vi skrive

$$\frac{\partial E}{\partial w_{ji}^{(3)}} = \frac{\partial E}{\partial z_j^{(3)}} \cdot \frac{\partial z_j^{(3)}}{\partial w_{ji}^{(3)}} \tag{82}$$

og så gentagne gange bruge kædereglen til at udfolde dette udtryk.

Lad os starte med det nemmeste, nemlig $\frac{\partial z_j^{(3)}}{\partial w_{ji}^{(3)}}$. Ser vi på definitionen af $z_j^{(3)}$ i (66), kan vi argumentere helt tilsvarende, som da vi ovenfor udledte udtrykket i (79) og får

$$\frac{\partial z_j^{(3)}}{\partial w_{ji}^{(3)}} = a_i^{(2)}$$

Lad os nu kaste os over den første faktor i (82). Vi kan starte med at udnytte denne lidt overordnede måde, som $z_j^{(3)}$ påvirker E på

$$z_j^{(3)} \to a_j^{(3)} \to E$$

Kædereglen giver os derfor i første omgang

$$\frac{\partial E}{\partial z_j^{(3)}} = \frac{\partial E}{\partial a_j^{(3)}} \cdot \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \tag{83}$$

Igen er sidste faktor nem nok, idet

$$a_j^{(3)} = \sigma(z_j^{(3)})$$

og derfor er

$$\frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} = \sigma'(z_j^{(3)}) = \sigma(z_j^{(3)}) \cdot (1 - \sigma(z_j^{(3)})) = a_j^{(3)} \cdot (1 - a_j^{(3)}),$$

hvor vi endnu en gang har benytte sætning 0.1.

Når vi skal bestemme $\frac{\partial E}{\partial a_j^{(3)}}$ kommer vi ikke udenom kædereglen for funktioner af flere variable. Det bliver tydeligt, når vi zoomer ind på hvordan $a_j^{(3)}$ påvirker E:

$$\begin{array}{ccccccc} & & z_1^{(4)} \to a_1^{(4)} & & & & \\ & \nearrow & & & \searrow & & \\ a_j^{(3)} & \to & z_2^{(4)} \to a_2^{(4)} & \to & & \to & E \\ & \searrow & & & \nearrow & & \\ & & z_3^{(4)} \to a_3^{(4)} & & & & \end{array}$$

For at vi senere kan udnytte nogle af de ligninger, som vi udledte i lag 4, vil vi faktisk bare nøjes med at se på det, på denne måde:

$$\begin{array}{ccccc} & & z_1^{(4)} & & \\ & \nearrow & & \searrow & \\ a_j^{(3)} & \rightarrow & z_2^{(4)} & \rightarrow & E \\ & \searrow & & \nearrow & \\ & & z_3^{(4)} & & \end{array}$$

Nu er vi endelig klar til at bruge kædereglen for funktioner af flere variable:

$$\frac{\partial E}{\partial a_j^{(3)}} = \frac{\partial E}{\partial z_1^{(4)}} \cdot \frac{\partial z_1^{(4)}}{\partial a_j^{(3)}} + \frac{\partial E}{\partial z_2^{(4)}} \cdot \frac{\partial z_2^{(4)}}{\partial a_j^{(3)}} + \frac{\partial E}{\partial z_3^{(4)}} \cdot \frac{\partial z_3^{(4)}}{\partial a_j^{(3)}} \tag{84}$$

$$= \sum_{k=1}^{3} \frac{\partial E}{\partial z_k^{(4)}} \cdot \frac{\partial z_k^{(4)}}{\partial a_j^{(3)}} \tag{85}$$

Se nu dukker der noget op, som vi har set før! Nemlig det fejlled, som vi definerede i (81), og som vi allerede har regnet ud, da vi opdaterede vægtene og biasene i lag 4. Tænk lige over det - det er faktisk ret fedt! Dvs. at vi kan skrive:

$$\frac{\partial E}{\partial a_j^{(3)}} = \sum_{k=1}^{3} \delta_k^{(4)} \cdot \frac{\partial z_k^{(4)}}{\partial a_j^{(3)}} \tag{86}$$

Så mangler vi kun lige at finde $\frac{\partial z_k^{(4)}}{\partial a_j^{(3)}}$! Fra (68) har vi, at

$$z_k^{(4)} = \sum_i w_{ki}^{(4)} a_i^{(3)} + b_k^{(4)}$$

så

$$\frac{\partial z_k^{(4)}}{\partial a_j^{(3)}} = \frac{\partial}{\partial a_j^{(3)}} \left(\sum_i w_{ki}^{(4)} a_i^{(3)} + b_k^{(4)} \right)$$

Når vi skal differentierer summen i ovenstående udtryk, får vi kun et led med, når $i = j$, fordi i alle andre tilfælde, vil vi med hensyn til $a_j^{(3)}$ skulle differentiere en konstant. Og da

$$\frac{\partial}{\partial a_j^{(3)}} \left(w_{kj}^{(4)} a_j^{(3)} \right) = w_{kj}^{(4)}$$

har vi altså, at

$$\frac{\partial z_k^{(4)}}{\partial a_j^{(3)}} = w_{kj}^{(4)}$$

Indsætter vi dette i (86), har vi nu

$$\frac{\partial E}{\partial a_j^{(3)}} = \sum_{k=1}^{3} \delta_k^{(4)} \cdot \frac{\partial z_k^{(4)}}{\partial a_j^{(3)}} = \sum_{k=1}^{3} \delta_k^{(4)} \cdot w_{kj}^{(4)} \tag{87}$$

Nu skal vi i første omgang tilbage til (83) og indsætte det vi netop er kommet frem til:

$$\frac{\partial E}{\partial z_j^{(3)}} = \underbrace{\left(\sum_{k=1}^{3} \delta_k^{(4)} \cdot w_{kj}^{(4)} \right)}_{\frac{\partial E}{\partial a_j^{(3)}}} \cdot \underbrace{a_j^{(3)} \cdot (1 - a_j^{(3)})}_{\frac{\partial a_j^{(3)}}{\partial z_j^{(3)}}} \tag{88}$$

Som vi gjorde i afsnit 4, vil vi også give dette lidt lange udtryk en særlig betegnelse, nemlig

$$\delta_j^{(3)} = \frac{\partial E}{\partial z_j^{(3)}} = \left(\sum_{k=1}^{3} \delta_k^{(4)} \cdot w_{kj}^{(4)} \right) \cdot a_j^{(3)} \cdot (1 - a_j^{(3)}) \tag{89}$$

Det kan vist godt være lidt svært at bevare overblikket her, men nu er vi faktisk i mål! Vi indsætter i (82)

$$\frac{\partial E}{\partial w_{ji}^{(3)}} = \frac{\partial E}{\partial z_j^{(3)}} \cdot \frac{\partial z_j^{(3)}}{\partial w_{ji}^{(3)}} \tag{90}$$

$$= \delta_j^{(3)} \cdot a_i^{(2)} \tag{91}$$

Det er nu en smal sag at bestemme $\frac{\partial E}{\partial b_j^{(3)}}$, da

$$\frac{\partial E}{\partial b_j^{(3)}} = \frac{\partial E}{\partial z_j^{(3)}} \cdot \frac{\partial z_j^{(3)}}{\partial b_j^{(3)}} \tag{92}$$

$$= \delta_j^{(3)} \cdot \frac{\partial z_j^{(3)}}{\partial b_j^{(3)}} \tag{93}$$

Fra (66) har vi, at

$$z_j^{(3)} = \sum_{i=1}^{3} w_{ji}^{(3)} a_i^{(2)} + b_j^{(3)}$$

og derfor er

$$\frac{\partial z_j^{(3)}}{\partial b_j^{(3)}} = 1$$

Altså får vi

$$\frac{\partial E}{\partial b_j^{(3)}} = \delta_j^{(3)} \tag{94}$$

Glæden er stor, da vi nu har alle ingredienser til at opskrive opdateringsreglerne for det tredje lag!

Opdateringsregler til vægte og bias i lag 3

Vægtene i outputlaget opdateres på denne måde:

$$w_{ji}^{(3)} \leftarrow w_{ji}^{(3)} - \eta \cdot \delta_j^{(3)} \cdot a_i^{(2)}$$

Biasene i outputlaget opdateres på denne måde:

$$b_j^{(3)} \leftarrow b_j^{(3)} - \eta \cdot \delta_j^{(3)}$$

hvor

$$\delta_j^{(3)} = \frac{\partial E}{\partial z_j^{(3)}} = \left(\sum_{k=1}^{3} \delta_k^{(4)} \cdot w_{kj}^{(4)} \right) \cdot a_j^{(3)} \cdot (1 - a_j^{(3)}) \tag{95}$$

Bemærk, at udgangspunktet for ovenstående er, at vi først har lavet et feedforward i netværket, så vi har alle $a_i^{(2)}$- og $a_j^{(3)}$-værdier. Derudover har vi allerede opdateret vægtene og biasene i lag 4. Derfor kender vi også fejleddene $\delta_k^{(4)}$ fra lag 4, som indgår i beregningen af $\delta_j^{(3)}$ i (95). Altså er det muligt at foretage de beregninger, som opdateringsreglerne i lag 3 kræver.

Opdateringsregler for lag 2

Vi er nu fremme ved det sidste lag, hvor vi skal have opdateret vægte og bias (husk på at $a_i^{(1)}$-værdierne jo ikke skal beregnes,

men er inputværdierne til netværket). Den gode nyhed her er, at der absolut intet nyt er under solen! Vi vil derfor heller ikke gå i drabelige deltaljer med alle udregninger her, men blot skitsere idéen.

Vi er nu på jagt efter

$$\frac{\partial E}{\partial w_{ji}^{(2)}} \quad \text{og} \quad \frac{\partial E}{\partial b_j^{(2)}}$$

og kigger vi på vores netværk i figur 7 kan vi se følgende afhængigheder:

$$w_{ji}^{(2)} \quad \rightarrow \quad z_j^{(2)} \rightarrow a_j^{(2)} \quad \begin{array}{ccc} & \nearrow & z_1^{(3)} \\ & & \searrow \\ & \searrow & z_2^{(3)} \\ & & \nearrow \end{array} \quad E$$

Vi kan nu igen skrive

$$\frac{\partial E}{\partial w_{ji}^{(2)}} = \frac{\partial E}{\partial z_j^{(2)}} \cdot \frac{\partial z_j^{(2)}}{\partial w_{ji}^{(2)}} \tag{96}$$

Helt analogt til tidligere ses det nemt, at

$$\frac{\partial z_j^{(2)}}{\partial w_{ji}^{(2)}} = a_i^{(1)}$$

og kædreglen giver igen, at

$$\frac{\partial E}{\partial z_j^{(2)}} = \frac{\partial E}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(2)}}{\partial z_j^{(2)}} \tag{97}$$

Her fås også uden problemer, at den sidste faktor kan skrives som

$$\frac{\partial a_j^{(2)}}{\partial z_j^{(2)}} = a_j^{(2)} \cdot (1 - a_j^{(2)})$$

og bruger man kædereglen for funktioner af flere variable, kommer man frem til, at

$$\frac{\partial E}{\partial a_j^{(2)}} = \sum_{k=1}^{2} \frac{\partial E}{\partial z_k^{(3)}} \cdot \frac{\partial z_k^{(3)}}{\partial a_j^{(2)}} \tag{98}$$

$$= \sum_{k=1}^{2} \delta_k^{(3)} w_{kj}^{(3)}, \tag{99}$$

hvor vi allerede har udregnet $\delta_k^{(3)}$, da vi opdaterede vægtene og biasene i lag 3.

Indsætter vi i (97) og samtidig definerer fejlleddet $\delta_j^{(2)}$ for det andet lag, får vi

$$\delta_j^{(2)} = \frac{\partial E}{\partial z_j^{(2)}} = \left(\sum_{k=1}^{2} \delta_k^{(3)} w_{kj}^{(3)} \right) \cdot a_j^{(2)} \cdot (1 - a_j^{(2)})$$

Alt i alt ender vi med

$$\frac{\partial E}{\partial w_{ji}^{(2)}} = \delta_j^{(2)} \cdot a_i^{(1)} \tag{100}$$

$$= \delta_j^{(2)} \cdot x_i \tag{101}$$

fordi alle $a_i^{(1)}$-værdierne svarer til selve inputværdierne x_i til netværket.

Det er nu ikke svært at se, at

$$\frac{\partial E}{\partial b_j^{(2)}} = \delta_j^{(2)}$$

og vi får derfor følgende opdateringsregler for lag 2:

Opdateringsregler til vægte og bias i lag 2

Vægtene i outputlaget opdateres på denne måde:

$$w_{ji}^{(2)} \leftarrow w_{ji}^{(2)} - \eta \cdot \delta_j^{(2)} \cdot a_i^{(1)} \tag{102}$$

Biasene i outputlaget opdateres på denne måde:

$$b_j^{(2)} \leftarrow b_j^{(2)} - \eta \cdot \delta_j^{(2)} \tag{103}$$

hvor

$$\delta_j^{(2)} = \frac{\partial E}{\partial z_j^{(2)}} = \left(\sum_{k=1}^{2} \delta_k^{(3)} w_{kj}^{(3)} \right) \cdot a_j^{(2)} \cdot (1 - a_j^{(2)}) \quad (104)$$

{#eq-error_term_j_2}

Var det så egentlig smart med alle de indekser?

Hvis man er nået hertil, kan man godt følge sig en lille smule forpustet. Der har godt nok været mange indekser at holde styr på! Både nogle der var sænkede, og nogle der var hævede og sat i parenteser! Alligevel kan man måske godt se fidusen nu.

Hvis vi ser på de opdateringsregler, som vi lige har udledt, så kan man se, at selve opdateringsreglerne af vægte og bias følger *præcis* samme form. Faktisk kan man, hvis man sammenligner opdateringsreglerne for de tre lag se, at opdateringsreglerne er på denne form:

Generelle opdateringsregler til vægte og bias

Vægtene opdateres generelt på denne måde:

$$w_{ji}^{(\text{lag})} \leftarrow w_{ji}^{(\text{lag})} - \eta \cdot \delta_j^{(\text{lag})} \cdot a_i^{(\text{lag-1})} \quad (105)$$

Biasene i outputlaget opdateres på denne måde:

$$b_j^{(\text{lag})} \leftarrow b_j^{(\text{lag})} - \eta \cdot \delta_j^{(\text{lag})} \quad (106)$$

Bemærk her, at da vi allerede har lavet en feedforward i netværket, så kender vi outputværdierne $a_i^{(\text{lag})}$ i alle lag. Det vil sige, at vi kan opdatere vægtene og biasene, når blot vi kan beregne fejlleddene.

Den eneste reelle forskel på opdateringsreglerne er, at fejlleddene udregnes lidt forskelligt, alt efter om der er tale om outputlaget eller et skjult lag:

Beregning af fejlleddene

Fejlleddene i outputlaget beregnes på denne måde:

$$\delta_j^{(\text{outputlag})} = -(t_j - y_j) \cdot y_j \cdot (1 - y_j), \qquad (107)$$

idet outputværdierne fra netværket netop er $y_1, y_2, ...$
Fejlleddene i et skjult lag beregnes på denne måde:

$$\delta_j^{(\text{lag})} = \left(\sum_k \delta_k^{(\text{lag}+1)} w_{kj}^{(\text{lag}+1)} \right) \cdot a_j^{(\text{lag})} \cdot (1 - a_j^{(\text{lag})}) \quad (108)$$

Bemærk her, at fejlleddene fra outputlaget uden videre kan beregnes, da vi kender target-værdierne t_j og outputværdierne y_j fra netværket (fordi vi allerede har lavet en feedforward). Vi kan også beregne fejlleddene i alle skjulte lag, idet vi hele tiden arbejder bagud i netværket (*backpropagation*). Det vil sige, at vi hele tiden har adgang til fejlleddene i laget længere fremme (lag+1), hvor (lag+1) første gang vil svare til outputlaget. Desuden kender vi pga. feedforward alle outputværdier $a_j^{((\text{lag}))}$ og alle vægte $w_{kj}^{((\text{lag}+1))}$. Derfor kan vi også beregne fejlleddene i alle de skjulte lag.

Denne indsigt og den generelle overordnede struktur på opdateringsreglerne, var meget svær at indse med fremgangsmåde i afsnit . Her druknede alt bare i et sandt bogstavshelvede!

Der er et par andre interessante ting at sige om beregningen af fejlleddene. Lad os først se på outputlaget:

$$\delta_j^{(\text{outputlag})} = -(t_j - y_j) \cdot y_j \cdot (1 - y_j)$$

Hvis der er stor forskel på target-værdien t_j og outputværdien y_j, så bliver forskellen $t_j - y_j$ numerisk stor. Altså vil en stor forskel på det, vi ønsker, og det vi får ud af netværket betyde, at fejlleddet bliver større og i sidste ende, at de vægte, som direkte påvirker outputtet, også vil blive opdateret meget. Endelig ser vi igen, at hvis outputneuronen er mættet (dvs. at y_j enten er tæt på 0 eller 1), så vil fejlleddet ikke blive opdateret i samme grad, som hvis outputneuronen ikke havde været mættet (fordi hvis y_j enten er tæt på 0 eller 1, så vil $y_j \cdot (1 - y_j)$ være tæt på 0).

Vi ser altså, at fejlleddet fra det sidste lag direkte afhænger af hvor stor forskellen er på target-værdi og outputværdi.

Ser vi så på fejlleddene fra de skjulte lag:

$$\delta_j^{(\text{lag})} = \left(\sum_k \delta_k^{(\text{lag}+1)} w_{kj}^{(\text{lag}+1)} \right) \cdot a_j^{(\text{lag})} \cdot (1 - a_j^{(\text{lag})})$$

Så kan vi igen se, at hvis den tilhørende outputneuron, som fyrer værdien $a_j^{(\text{lag})}$, er mættet, så vil fejlleddet være tættere på 0, end hvis neuronen ikke havde været mættet. Samtidig kan vi også se, at der i fejlleddet indgår en vægtet sum af alle fejlleddene fra laget længere fremme:

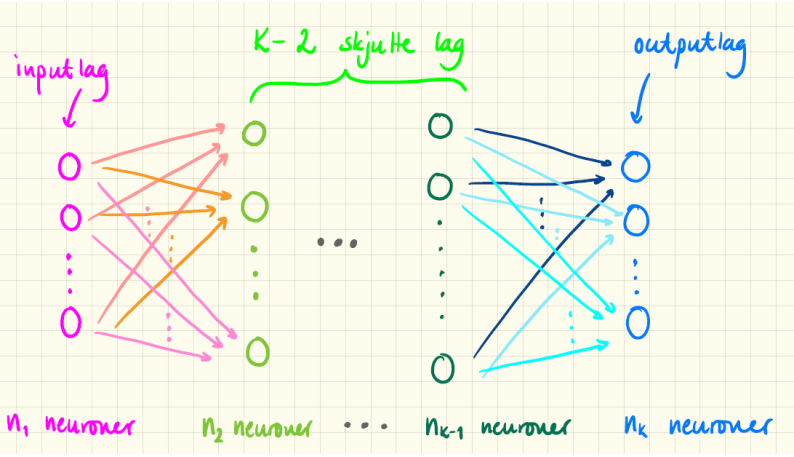
$$\sum_k \delta_k^{(\text{lag}+1)} w_{kj}^{(\text{lag}+1)}$$

På den måde vil store fejl i laget længere fremme også få indflydelse på fejlleddet i det nuværende lag.

Kunstige neurale netværk helt generelt

Med det benarbejde vi lige har været igennem, ligger det faktisk lige til højrebenet at generalisere overstående til et vilkårligt kunstigt neuralt netværk.

Vi forestiller os nu, at vi har K lag i netværket, som det er illustreret på figur 10.



Figur 10: Generelt kunstigt neuralt netværk med K lag.

Det vil sige ét inputlag, ét outputlag og $K-2$ skjulte lag. Antallet af neuroner i hvert lag kan variere og behøver ikke at være det samme. Så lad os navngive antallet af neuroner i de K lag på denne måde:

$$n_1, n_2, \dots, n_k, \dots, n_K$$

Inputværdierne til netværket betegner vi:

$$a_1^{(1)}, a_2^{(1)}, \dots, a_{n_1}^{(1)}$$

Feedforward ligningerne er nu helt identiske med dem vi opstillede i afsnit 4:

Feedforwardligninger - helt generelt

Beregn først:

$$z_j^{(k)} = \sum_i w_{ji}^{(k)} a_i^{(k-1)} + b_j^{(k)} \tag{109}$$

Outputværdierne for neuroner i det k'te lag udregnes dernæst på denne måde:

$$a_j^{(k)} = \sigma(z_j^{(k)}) \tag{110}$$

for $k \in \{2, 3, \dots, K\}$.

Hvis vi også kalder outputværdierne fra netværket for

$$y_1, y_2, \dots, y_{n_K}$$

så beregnes disse værdier altså ved

$$y_j = a_j^{(K)} = \sigma(z_j^{(K)}) \tag{111}$$

for $j \in \{1, 2, \dots, n_K\}$.

Backpropagation - generelt

Lad os så se på backpropagation. Vi ved fra det foregående, at der reelt set kun er to ting, vi skal gøre:

1. Finde opdateringsreglerne for vægte og bias i outputlaget
2. Finde opdateringsreglerne for vægte og bias i et vilkårligt skjult lag

Opdateringsregler i outputlaget

Vores tabsfunktion er stadig

$$E = \frac{1}{2}\sum_{i=1}^{n_K}\left(t_i - a_i^{(K)}\right)^2,$$

hvor t_i igen er targetværdierne. Vi skal bestemme

$$\frac{\partial E}{\partial w_{ji}^{(K)}} \quad \text{og} \quad \frac{\partial E}{\partial b_j^{(K)}}$$

Vi gør præcis som i afsnit 4. Vi indser først, at vi har denne direkte afhængighed fra $w_{ji}^{(K)}$ til E:

$$w_{ji}^{(K)} \to z_j^{(K)} \to a_j^{(K)} \to E$$

Derfor får vi

$$\frac{\partial E}{\partial w_{ji}^{(K)}} = \frac{\partial E}{\partial z_j^{(K)}} \cdot \frac{\partial z_j^{(K)}}{\partial w_{ji}^{(K)}} \tag{112}$$

På grund feedforwardligningen i (109) får vi for det første, at

$$\frac{\partial z_j^{(K)}}{\partial w_{ji}^{(K)}} = a_i^{(K-1)} \tag{113}$$

Nu bruger vi kædereglen til at bestemme

$$\frac{\partial E}{\partial z_j^{(K)}} = \frac{\partial E}{\partial a_j^{(K)}} \cdot \frac{\partial a_j^{(K)}}{\partial z_j^{(K)}} \tag{114}$$

På grund af feedforwardligningen i (110) og sætning 0.1 får vi sidste faktor til

$$\frac{\partial a_j^{(K)}}{\partial z_j^{(K)}} = a_j^{(K)} \cdot (1 - a_j^{(K)})$$

Endelig får vi, ved at differentiere tabsfunktionen med hensyn til $a_j^{(K)}$

$$\frac{\partial E}{\partial a_j^{(K)}} = -(t_j - a_j^{(K)})$$

Vi definerer nu igen fejlleddet for outputlaget $\delta_j^{(K)}$, som tidligere

$$\delta_j^{(K)} = \frac{\partial E}{\partial z_j^{(K)}}$$

og indsætter vi det, vi netop har udledt, i (114) får vi

$$\delta_j^{(K)} = -(t_j - a_j^{(K)}) \cdot a_j^{(K)} \cdot (1 - a_j^{(K)})$$

Indsætter vi nu det hele i (112), har vi altså:

$$\frac{\partial E}{\partial w_{ji}^{(K)}} = \delta_j^{(K)} \cdot a_i^{(K-1)}$$

Det er ikke svært at overbevise sig selv om, at

$$\frac{\partial E}{\partial b_j^{(K)}} = \delta_j^{(K)}$$

og derfor har vi:

Generelle opdateringsregler til vægte og bias i outputlaget (lag K)

Vægtene i outputlaget opdateres på denne måde:

$$w_{ji}^{(K)} \leftarrow w_{ji}^{(K)} - \eta \cdot \delta_j^{(K)} \cdot a_i^{(K-1)} \tag{115}$$

Biasene i outputlaget opdateres på denne måde:

$$b_j^{(K)} \leftarrow b_j^{(K)} - \eta \cdot \delta_j^{(K)} \tag{116}$$

hvor

$$\delta_j^{(K)} = \frac{\partial E}{\partial z_j^{(K)}} = -(t_j - a_j^{(K)}) \cdot a_j^{(K)} \cdot (1 - a_j^{(K)}) \tag{117}$$

Opdateringsregler i et vilkårligt skjult lag

Vi ser nu på et vilkårligt skjult lag k, som hverken er inputlaget eller outputlaget. Det vil sige, at $k \in \{2, 3, \ldots, K-1\}$. Vi antager, at vi har kørt backpropagation på alle lag, der ligger

længere fremme i netværket, og specielt har vi altså beregnet fejlleddene i lag $k+1$:

$$\delta_j^{(k+1)} = \frac{\partial E}{\partial z_j^{(k+1)}}$$

Vi indser først, at vi har denne afhængighed fra $w_{ji}^{(k)}$ til tabsfunktionen E:

$$w_{ji}^{(k)} \quad \to \quad z_j^{(k)} \to a_j^{(k)} \quad \begin{array}{ccc} \nearrow & z_1^{(k+1)} & \searrow \\ \to & \vdots & \to \\ \searrow & z_j^{(k+1)} & \nearrow \\ & \vdots & \\ & z_{n_{k+1}}^{(k+1)} & \end{array} \quad E \tag{118}$$

Vi starter som tidligere med at bruge kædereglen én gang:

$$\frac{\partial E}{\partial w_{ji}^{(k)}} = \frac{\partial E}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k)}} \tag{119}$$

Fra feedforwardligningen i (109) får vi for det første, at

$$\frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k)}} = a_i^{(k-1)}$$

Endnu en anvendelse af kædereglen, og hvor vi også i samme hug definerer fejlleddet $\delta_j^{(k)}$ for det k'te skjulte lag, giver:

$$\delta_j^{(k)} = \frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial E}{\partial a_j^{(k)}} \cdot \frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} \tag{120}$$

Den sidste partielle afledede kan vi udlede fra feedforwardligningen (110) og sætning 0.1:

$$\frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} = a_j^{(k)} \cdot (1 - a_j^{(k)})$$

For at beregne $\frac{\partial E}{\partial a_j^{(k)}}$ må vi have fat i kædereglen for funktioner af flere variable (se illustrationen i (118)):

$$\frac{\partial E}{\partial a_j^{(k)}} = \sum_{i=1}^{n_{k+1}} \frac{\partial E}{\partial z_i^{(k+1)}} \cdot \frac{\partial z_i^{(k+1)}}{\partial a_j^{(k)}} \tag{121}$$

Vi udnytter nu, at vi allerede kender fejlleddene fra lag $k+1$ og kan derfor omskrive til

$$\frac{\partial E}{\partial a_j^{(k)}} = \sum_{i=1}^{n_{k+1}} \delta_i^{(k+1)} \cdot \frac{\partial z_i^{(k+1)}}{\partial a_j^{(k)}} \tag{122}$$

Endelig får vi fra feedforwardligningen (109), at

$$\frac{\partial z_i^{(k+1)}}{\partial a_j^{(k)}} = w_{ij}^{(k)} \tag{123}$$

og derfor er

$$\frac{\partial E}{\partial a_j^{(k)}} = \sum_{i=1}^{n_{k+1}} \delta_i^{(k+1)} \cdot w_{ij}^{(k)} \tag{124}$$

Ved at indsætte i (120) får vi nu fejlleddet i det k'te lag

$$\delta_j^{(k)} = \frac{\partial E}{\partial z_j^{(k)}} = \left(\sum_{i=1}^{n_{k+1}} \delta_i^{(k+1)} \cdot w_{ij}^{(k)} \right) \cdot a_j^{(k)} \cdot (1 - a_j^{(k)})$$

Altså er

$$\frac{\partial E}{\partial w_{ji}^{(k)}} = \delta_j^{(k)} \cdot a_i^{(k-1)} \tag{125}$$

og tilsvarende får vi også, at

$$\frac{\partial E}{\partial b_j^{(k)}} = \delta_j^{(k)}$$

Opdateringsreglerne for et vilkårligt skjult lag bliver så:

Opdateringsregler til vægte og bias i et vilkårligt skjult lag k

Vægtene i outputlaget opdateres på denne måde:

$$w_{ji}^{(k)} \leftarrow w_{ji}^{(k)} - \eta \cdot \delta_j^{(k)} \cdot a_i^{(k-1)} \tag{126}$$

Biasene i outputlaget opdateres på denne måde:

$$b_j^{(k)} \leftarrow b_j^{(k)} - \eta \cdot \delta_j^{(k)} \tag{127}$$

hvor

$$\delta_j^{(k)} = \frac{\partial E}{\partial z_j^{(k)}} = \left(\sum_{i=1}^{n_{k+1}} \delta_i^{(k+1)} \cdot w_{ij}^{(k)} \right) \cdot a_j^{(k)} \cdot (1 - a_j^{(k)}) \tag{128}$$

Stokastisk gradientnedstigning

Vi har faktisk snydt lidt… Okay – indrømmet – det er lidt træls at komme at sige nu! Men i alt hvad vi har lavet indtil nu, har vi kun kigget på ét træningseksempel. Vi har ladet inputværdierne for det ene træningseksempel "kører igennem" netværket (feedforward), beregnet tabsfunktionen og brugt resultatet herfra til at opdatere alle vægtene (backpropagation). Men vi har jo ikke kun ét træningseksempel. Vi har faktisk rigtig mange! Måske ligefrem tusindvis af træningsdata. Men hvad gør man så?

Lad os lige genopfriske den tabsfunktion, som vi endte med i det helt generelle tilfælde:

$$E = \frac{1}{2} \sum_{i=1}^{n_K} \left(t_i - a_i^{(K)} \right)^2 . \tag{129}$$

Her er t_i target-værdien for den i'te outputneuron for lige præcis det træningseksempel vi står med. Husk på at et givet træningseksempel består af inputværdierne

$$x_1, x_2, \dots, x_{n_1}$$

og de ønskede target-værdier

$$t_1, t_2, \dots, t_{n_K}.$$

Når vi kører disse inputværdier igennem netværket, får de selvfølgelig i sidste ende direkte betydning for outputværdierne i det sidste lag (K):

$$a_1^{(K)}, a_2^{(K)}, \cdots, a_{n_K}^{(K)}.$$

Det vil sige, at i vores tabsfunktion i (129), så afhænger både t_i'erne og $a_i^{(K)}$'erne af træningseksemplet. Hvis vi sådan lidt

generelt benævner vores træningseksempel med x, så vil det kunne udtrykkes sådan her:

$$E_x = \frac{1}{2}\sum_{i=1}^{n_K}\left(t_{x,i} - a_{x,i}^{(K)}\right)^2,$$

hvor så $t_{x,i}$ er target-værdien for den i'te outputneuron fra træningsdata x og $a_{x,i}^{(K)}$ er outputværdien for den i'te outputneuron, som er beregnet på baggrund af inputværdierne fra træningsdata x.

Den samlede tabsfunktion, som er den, vi i virkeligheden ønsker at minimere, bliver så gennemsnittet af tabsfunktionerne hørende til de enkelte træningsdata:

$$E = \frac{1}{n}\sum_x E_x = \frac{1}{n}\sum_x\left(\frac{1}{2}\sum_{i=1}^{n_K}\left(t_{x,i} - a_{x,i}^{(K)}\right)^2\right). \tag{130}$$

Husk på at vi er ude efter

$$\frac{\partial E}{\partial w_{ji}^{(k)}} \quad \text{og} \quad \frac{\partial E}{\partial b_j^{(k)}}$$

for $k \in \{2, 3, \ldots, K\}$ og hvor E nu er summen i (130). Heldigvis kan vi differentiere ledvis, og der gælder derfor

$$\frac{\partial E}{\partial w_{ji}^{(k)}} = \frac{1}{n}\sum_x \frac{\partial E_x}{\partial w_{ji}^{(k)}}$$

og tilsvarende

$$\frac{\partial E}{\partial b_j^{(k)}} = \frac{1}{n}\sum_x \frac{\partial E_x}{\partial b_j^{(k)}}$$

Det kommer så til at betyde, at opdateringsreglerne nu generelt bliver på formen

$$w_{ji}^{(k)} \leftarrow w_{ji}^{(k)} - \eta \cdot \frac{\partial E}{\partial w_{ji}^{(k)}} = w_{ji}^{(k)} - \eta \cdot \frac{1}{n}\sum_x \frac{\partial E_x}{\partial w_{ji}^{(k)}} \tag{131}$$

og tilsvarende for biasene

$$b_j^{(k)} \leftarrow b_j^{(k)} - \eta \cdot \frac{\partial E}{\partial b_j^{(k)}} = b_j^{(k)} - \eta \cdot \frac{1}{n}\sum_x \frac{\partial E_x}{\partial b_j^{(k)}} \tag{132}$$

Alle leddene $\frac{\partial E_x}{\partial w_{ji}^{(k)}}$ og $\frac{\partial E_x}{\partial b_j^{(k)}}$, som indgår i opdateringsreglerne, svarer netop til hvad vi har udledt i de foregående afsnit, fordi vi jo netop her kun så på ét træningseksempel ad gangen. Hvis vi overfører dette til opdateringsreglerne i outputlaget, så vil vi f.eks. få

Generelle opdateringsregler til vægte og bias i outputlaget (lag K) med brug af alle træningsdata

Vægtene i outputlaget opdateres på denne måde:

$$w_{ji}^{(K)} \leftarrow w_{ji}^{(K)} - \eta \cdot \frac{1}{n} \sum_x \left(\delta_{x,j}^{(K)} \cdot a_{x,i}^{(K-1)} \right) \tag{133}$$

Biasene i outputlaget opdateres på denne måde:

$$b_j^{(K)} \leftarrow b_j^{(K)} - \eta \cdot \frac{1}{n} \sum_x \left(\delta_{x,j}^{(K)} \right) \tag{134}$$

hvor

$$\delta_{x,j}^{(K)} = \frac{\partial E_x}{\partial z_j^{(K)}} = -(t_{x,j} - a_{x,j}^{(K)}) \cdot a_{x,j}^{(K)} \cdot (1 - a_{x,j}^{(K)}) \tag{135}$$

Og helt tilsvarende vil det se ud for opdateringsreglerne i de skjulte lag.

Lad os lige dvæle lidt ved, hvad det her, det egentlig betyder. Lad os sige at vi har 1000 træningsdata. Så skal vi lade de 1000 træningsdata kører igennem netværket, så vi kan beregne de 1000 led, som indgår i de ovenstående summer. Herefter kan vi opdatere alle vægte og bias én gang. Det vil blot være ét lille skridt på vej ned i dalen mod det lokale minimum, som vi er på jagt efter. Dette lille skridt skal gentages rigtig mange gange indtil værdierne af alle vægte og bias ser ud til at begynde at konvergere – svarende til at vi har ramt det lokale minimum.

Så selvom gradientnedstigning kan bruges til at finde et lokalt minimum for tabsfunktionen E, så er det faktisk også en beregningsmæssig stor og tung opgave! Derfor er der forsket meget videre i at gøre det endnu bedre og endnu hurtigere. I algoritmer som disse er der ofte et trade-off: Man kan gøre noget hurtigere ved at bruge mere hukommelse – eller bruge mindre

hukommelse ved at gøre det en smule langsommere. En af de teknikker, der er kommet ud af den forskning, er, at man kan bruge mindre hukommelse ved i hvert opdateringsskridt kun at bruge en tilfældigt udvalgt del af træningsdata – det kunne f.eks. være 10% af alle træningsdata. Så vil man i hvert skridt stadig bruge opdateringsreglerne i (131) og (132), men hvor der nu kun summeres over de 10% af træningsdatene. Hver gang man laver et nyt opdateringsskridt, vil man tage en ny tilfældigt udvalgt del af træningsdata. Denne teknik kalder man *stokastisk gradientnedstigning* (stochastic gradient descent). Og der er endnu flere af sådanne små ændringer, der enten gør algoritmen hurtigere eller at den bruger mindre hukommelse. Det vil komme an på den enkelte anvendelse, hvad der er vigtigst her.

Valg af tabsfunktion

I afsnit definerede vi tabsfunktionen ved

$$E = \frac{1}{2}(t-o)^2$$

mens vi afsnit 4 og afsnit 4 så, hvordan denne tabsfunktionen kan generaliseres, når outputlaget har mere end én neuron. I alle tilfælde er der tale om en *kvadratisk* tabsfunktion, fordi vi ser på den kvadrerede forskel på target-værdi t og outputværdi o. Men det er et *valg* at definere tabsfunktionen på denne måde, man kunne have defineret mange andre tabsfunktioner i stedet for. Egentlig ønsker vi blot en tabsfunktion E med følgende egenskaber:

- $E > 0$ - fordi det giver bedst mening at tale om et positivt tab.
- Hvis vi har et netværk, som klassificerer godt, så er E tæt på 0, og omvendt hvis netværket er dårligt til at klassificere, så er E langt væk fra 0.

Har en tabsfunktion disse egenskaber, så giver det mening at bestemme vægtene i netværket, så tabsfunktionen minimeres.

Faktisk viser det sig, at valget af den kvadratiske tabsfunktion kombineret med sigmoid aktiveringsfunktionen har nogle ulemper. For at forstå det, skal vi se på opdateringsreglerne for vægtene tættest på outputlaget. Men den notation, som vi anvendte i afsnit , fik vi f.eks. at

$$w_1 \leftarrow w_1 + \eta \cdot (t - o) \cdot o \cdot (1 - o) \cdot z_1$$

Når vi træner netværket, ved vi jo ikke, hvilke værdier vægtene skal have - så vi starter med at vælge nogle mere eller mindre tilfældige værdier. Det betyder også, at indtil netværket er trænet, så vil den andel af træningseksemplerne, som klassificeres korrekt, ikke nødvendigvis være så stor. Det gør ikke så meget, hvis bare netværket lærer hurtigt - det vil sige, at netværket hurtigt får opdateret vægtene så andelen af træningseksemplerne, som klassificeres korrekt, bliver stor. Problemet med den kvadratiske tabsfunktion er, at man i nogle tilfælde kan komme ud for at dette ikke sker, men derimod at netværket lærer langsomt.

Problemet opstår, hvis startvægtene fejlagtigt kommer til at give en outputværdi o, som enten er tæt på 0 eller 1. Hvis det sker, vil $t-o$ være "stor" (fordi outputværdien er fejlagtig), men samtidig vil faktoren $o \cdot (1-o)$ være tæt på 0 (fordi o enten er tæt på 0 eller på 1). Derfor kan vi risikere, at w_1-vægten ikke bliver opdateret særlig meget, fordi $t - o$ bliver ganget med et tal, som er tæt på 0. Sker dette kalder man det for *slow learning*.

Nu kunne man måske godt tænke, at det da ikke har noget med tabsfunktionen at gøre, men det har lige præcis noget med tabsfunktionen *i kombination* med sigmoid-funktionen at gøre. For at se det må vi genkalde os, hvor ovenstående opdateringsregel kom fra. Faktisk svarede $t - o$ til $-dE/do$ og $o \cdot (1 - o)$ var sigmoid-funktionen $\sigma(w_0 + w_1 \cdot z_1 + w_2 \cdot z_2)$ differentieret. Det vil sige:

$$w_1 \leftarrow w_1 + \eta \cdot \underbrace{(t - o)}_{-dE/do} \cdot \underbrace{o \cdot (1 - o)}_{\sigma'(\cdots)} \cdot z_1 \tag{136}$$

Hvis man derimod kunne vælge en tabsfunktion, som passer bedre sammen med sigmoid-funktionen, på den måde at forstå, at faktoren $o \cdot (1-o)$ ville forkorte ud i ovenstående udtryk, så ville man slippe af med problemet omkring langsom læring.

Nok ikke helt overraskende så kan man faktisk godt diske op med en sådan tabsfunktion! Den tabsfunktion, som passer godt sammen med sigmoid-funktionen, kaldes for *cross-entropy* tabsfunktionen og er defineret sådan her

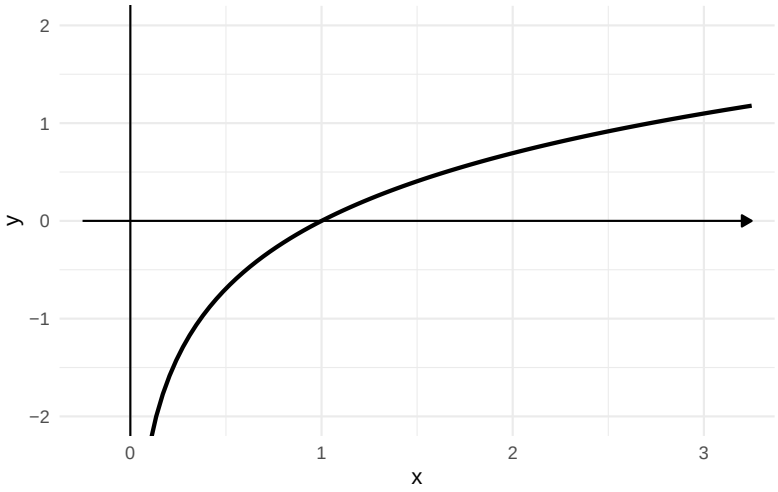
$$E = -\left(t \cdot \ln(o) + (1-t) \cdot \ln(1-o)\right) \tag{137}$$

Det ser jo ikke umiddelbart videre behageligt ud, og det er slet ikke oplagt, at det overhovedet er en tabsfunktion. Det vil sige, opfylder cross-entropy tabsfunktionen betingelserne listet ovenfor? Lad os starte med at undersøge om $E > 0$. For det første er target-værdien t altid enten 0 eller 1. Derfor er der følgende to muligheder:

$$t = 0: \quad E = -\ln(1-o) \tag{138}$$

$$t = 1: \quad E = -\ln(o) \tag{139}$$

Vi ved også, at outputværdien o ligger mellem 0 og 1. Altså $0 < o < 1$. Derfor vil også $1-o$ ligge mellem 0 og 1. På figur 11 ses grafen for den naturlige logaritmefunktion $\ln(x)$. Her ser vi, at hvis $0 < x < 1$, så vil $\ln(x)$ være negativ. Derfor kan vi fra ovenstående udtryk se, at E vil være positiv (fordi både $\ln(1-o)$ og $\ln(o)$ vil være negative).



Figur 11: Grafen for den naturlige logaritmefunktion $\ln(x)$.

Vi mangler nu at redegøre for, at $E \approx 0$, hvis netværket er god til at klassificere. Der er igen to muligheder alt efter om $t = 0$ eller om $t = 1$. Hvis $t = 0$ og netværket er godt, så vil o også

være tæt på 0, og $1-o$ vil være tæt på 1. I det tilfælde får vi

$$E = -\left(\underbrace{t}_{=0} \cdot \ln(o) + \underbrace{(1-t)}_{=1} \cdot \underbrace{\ln(1-o)}_{\approx 0} \right),$$

hvor vi bruger, at $\ln(x) \approx 0$, hvis $x \approx 1$ (se figur 11). Alt i alt får vi i dette tilfælde, at E er tæt på 0. Hvis derimod $t = 1$ og netværket er godt, så o er tæt på 1, så får vi:

$$E = -\left(\underbrace{t}_{=1} \cdot \underbrace{\ln(o)}_{\approx 0} + \underbrace{(1-t)}_{=0} \cdot \ln(1-o) \right)$$

Igen får vi også i dette tilfælde, at E er tæt på 0. På tilsvarende vis kan man argumentere for, at hvis netværket er dårligt, så enten $t = 0$ og $o \approx 1$ eller $t = 1$ og $o \approx 0$, så vil E blive stor.

Alt i alt er vi altså kommet frem til, at cross-entropy tabsfunktionen opfylder betingelserne ovenfor og dermed, at cross-entropy tabsfunktionen rent faktisk *er* en tabsfunktion! Så langt så godt! Nu mangler vi at vise, at cross-entropy tabsfunktionen løser det potentielle problem omkring langsom læring. Det gør vi ved at finde et nyt udtryk for opdatering af w_1-vægten. Hvis vi ser på udtrykket i (136), så skal vi bare have fundet et nyt udtryk for dE/do, da vi holder fast i sigmoid aktiveringsfunktionen. Så lad os differentiere cross-entropy tabsfunktionen defineret i (137) med hensyn til o:

$$\frac{dE}{do} = -\left(t \cdot \frac{1}{o} - (1-t) \cdot \frac{1}{1-o} \right),$$

hvor vi har husket, at $\ln(1-o)$ er en sammensat funktion. Vi sætter nu på fælles brøkstreg

$$\frac{dE}{do} = -\frac{t \cdot (1-o) - o \cdot (1-t)}{o \cdot (1-o)} \tag{140}$$

$$= -\frac{t - t \cdot o - o + t \cdot o}{o \cdot (1-o)} \tag{141}$$

$$= -\frac{t-o}{o \cdot (1-o)} \tag{142}$$

Og derfor er

$$-\frac{dE}{do} = \frac{t-o}{o \cdot (1-o)}$$

som ved indsættelse i (136) giver

$$w_1 \leftarrow w_1 + \eta \cdot \underbrace{\frac{t-o}{o\cdot(1-o)}}_{-dE/do} \cdot \underbrace{o\cdot(1-o)}_{\sigma'(\cdots)} \cdot z_1$$

Vi ser nu, at udtrykket $o \cdot (1-o)$, som var den faktor, der kunne give anledning til langsom læring, forkorter ud, og vi får i stedet

$$w_1 \leftarrow w_1 + \eta \cdot (t-o) \cdot z_1$$

Her ses det tydeligt, at hvis der er stor forskel på target-værdi t og outputværdi o, så vil det give anledning til en stor opdatering af w_1-vægten[4] - også selvom o er tæt på 0 eller 1. På den måde passer cross-entropy tabsfunktionen bedre sammen med sigmoid aktiveringsfunktionen end den kvadratiske tabsfunktion gør. Simpelthen fordi det potentielle problem omkring langsom læring undgås.

[4] Med mindre selvfølgelig at z_1 er tæt på 0, men i det tilfælde vil w_1 alligevel ikke have særlig stor indflydelse på outputværdien o.

Billedgenkendelse og kunstige neurale netværk

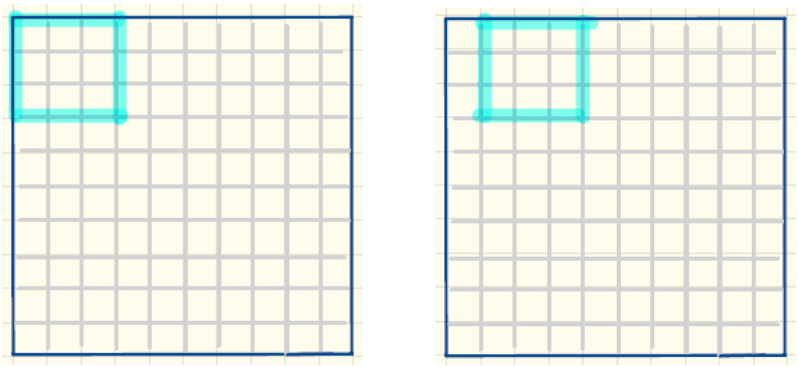
Vi startede egentlig med at sige, at vi godt kunne tænke os at træne et kunstigt neuralt netværk, så det kan bruges til at afgøre, om der er en hund på et billede eller ej. Og egentlig kan det godt lade sig gøre, med det vi har lært indtil nu. Der er bare én afgørende ting ved billedgenkendelse, som vi ikke har taget højde for. I afsnit forklarede vi, hvordan man kan repræsentere et sort/hvidt billede ved hjælp af en vektor. F.eks. vil et billede på 10×10 pixels kunne repræsenteres ved en 100 dimensional vektor. Vi kan så lave et kunstigt neuralt netværk med 100 input neuroner, et antal skjulte lag og en outputneuron. Men gør vi det, så tillægger vi det ingen som helst betydning, at nogle af de 100 pixels i billedet er tætte på hinanden, mens andre igen er langt væk fra hinanden. I stedet behandler vi alle pixels fuldstændig ens uden at tage højde for den indbyrdes placering, de enkelte pixels har i forhold til hinanden. Det giver ikke ret meget mening, når man behandler billeder. Det vil med andre ord sige, at vi faktisk smider værdifuld information ud med badevandet!

Det faktum tages der højde for i de såkaldte *convolutional neural networks*. Som illustration ser vi på et 10×10 pixels billede, hvis pixelværdier vi vil repræsentere på denne måde:

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$	$x_{1,7}$	$x_{1,8}$	$x_{1,9}$	$x_{1,10}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$	$x_{2,7}$	$x_{2,8}$	$x_{2,9}$	$x_{2,10}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$	$x_{3,7}$	$x_{3,8}$	$x_{3,9}$	$x_{3,10}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$	$x_{4,7}$	$x_{4,8}$	$x_{4,9}$	$x_{4,10}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$	$x_{5,7}$	$x_{5,8}$	$x_{5,9}$	$x_{5,10}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$	$x_{6,7}$	$x_{6,8}$	$x_{6,9}$	$x_{6,10}$
$x_{7,1}$	$x_{7,2}$	$x_{7,3}$	$x_{7,4}$	$x_{7,5}$	$x_{7,6}$	$x_{7,7}$	$x_{7,8}$	$x_{7,9}$	$x_{7,10}$
$x_{8,1}$	$x_{8,2}$	$x_{8,3}$	$x_{8,4}$	$x_{8,5}$	$x_{8,6}$	$x_{8,7}$	$x_{8,8}$	$x_{8,9}$	$x_{8,10}$
$x_{9,1}$	$x_{9,2}$	$x_{9,3}$	$x_{9,4}$	$x_{9,5}$	$x_{9,6}$	$x_{9,7}$	$x_{9,8}$	$x_{9,9}$	$x_{9,10}$
$x_{10,1}$	$x_{10,2}$	$x_{10,3}$	$x_{10,4}$	$x_{10,5}$	$x_{10,6}$	$x_{10,7}$	$x_{10,8}$	$x_{10,9}$	$x_{10,10}$

Det vil sige, at $x_{5,8}$ er pixelværdien i række 5 og kolonne 8 i billedet. Husk at når billedet er sort/hvidt, er denne værdi et heltal mellem 0 og 255.

Vi forestiller os nu, at vi lader et vindue på 3×3 pixels kører henover billedet, som vist på figur 5.



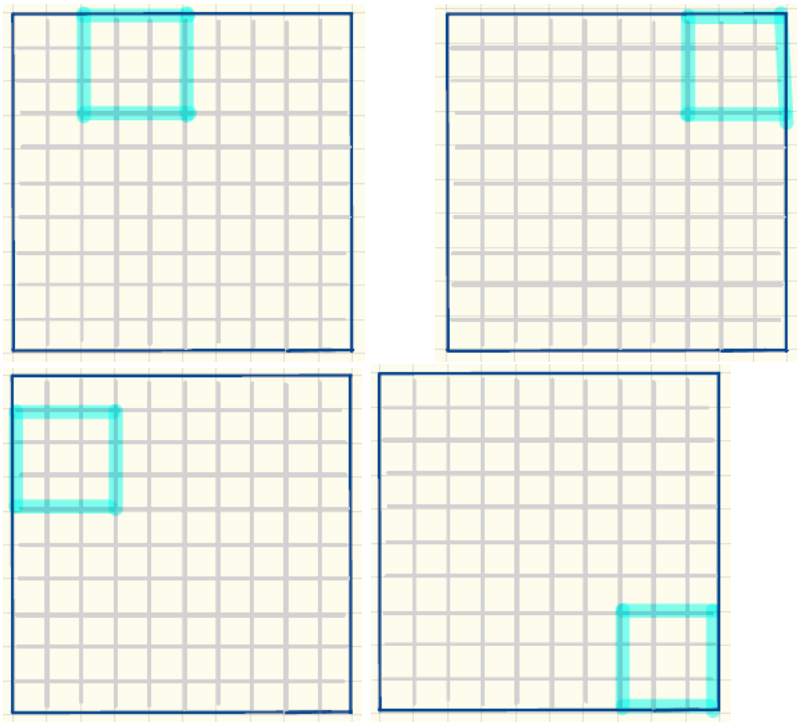
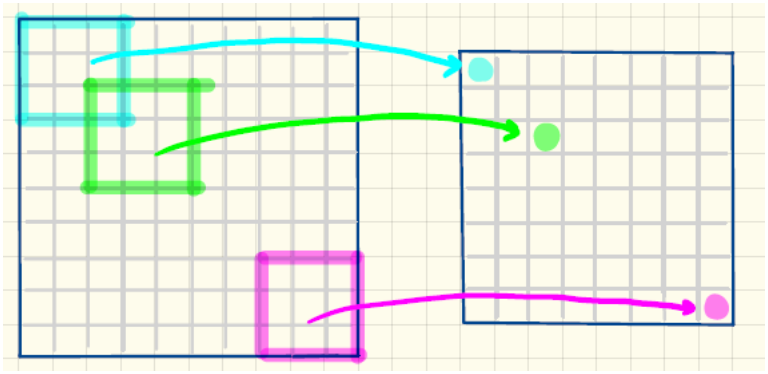


Illustration af vindue på 3×3 pixels, som glider henover billedet.

Hvis man tæller efter, vil der i alt kunne glide 8 vinduer vandret og 8 vinduer lodret. Det giver i alt 64 vinduer. Pixelværdierne i hvert af disse vinduer sendes nu frem til en skjult neuron i et nyt lag i netværket. Dette er illustreret på figur 12.



Figur 12: Local receptive fields for de skjulte neuroner.

Bemærk, at neuronerne i det første skjulte lag er repræsenteret som et 8×8 pixels billede (i stedet for neuroner i en vertikal

søjle som vi har gjort tidligere). Hvert af vinduerne til venstre i figur 12 kalder man for et *local receptive field* for den tilhørende skjulte neuron til højre i figuren. For at beregne den værdi som den første skjulte neuron sender videre i netværket, gør vi som tidligere - men med den vigtige undtagelse, at vi kun vægter de 9 værdi fra det tilhørende *local receptive field*. Hvis vi kalder den værdi, som den første skjulte neuron sender videre, for $y_{1,1}$, så kommer det til se sådan her ud:

$$y_{1,1} = \sigma(v_0 + v_1 \cdot x_{1,1} + v_2 \cdot x_{1,2} + v_3 \cdot x_{1,3} \tag{143}$$
$$+ v_4 \cdot x_{2,1} + v_5 \cdot x_{2,2} + \cdots + v_9 \cdot x_{3,3}), \tag{144}$$

her er v_0 en bias, $v_1, v_2, \ldots, v_9$ er vægte og σ er igen sigmoid-funktionen.

Det vil altså sige, at hver neuron i det første skjulte lag *kun* afhænger af 9 af de i alt 100 input-neuroner. På den måde får vi konstrueret et netværk, hvor vi eksplicit indbygger i netværket, at pixels som ligger tæt på hinanden, skal have noget med hinanden at gøre. Og omvendt hvis to pixels ligger langt væk fra hinanden, skal de ikke have noget med hinanden at gøre (fordi de ikke kommer til at indgå i den samme vægtede sum).

En anden ting, der er ny, er, at vi for alle 64 vinduer bruger de *samme* 10 vægte: $v_0, v_1, \ldots, v_9$. Det vil sige, at når vi f.eks. skal udregne den sidste værdi, så bliver det

$$y_{8,8} = \sigma(v_0 + v_1 \cdot x_{8,8} + v_2 \cdot x_{8,9} + v_3 \cdot x_{8,10} \tag{145}$$
$$+ v_4 \cdot x_{9,8} + v_5 \cdot x_{9,9} + \cdots + v_9 \cdot x_{10,10}) \tag{146}$$

Resultatet bliver 8×8 skjulte neuroner, som er vist til højre på billedet i figur 12. Dette lag i netværket kalder man for et *convolutional layer*. De i alt 10 vægte kaldes for et *feature map*, og idéen er, at de bruges til at finde en bestemt egenskab (feature) i billedet. Lad os se på et eksempel. Nedenfor er illustreret et 10×10 pixels billede med en vertikal mørk linje i midten. Værdierne i hver celle er egentlig ikke en del af billedet, men blot den tilhørende gråskalaværdi.

Den vertikale linje opstår, fordi der på hver række er store forskelle i gråskalaværdierne (fra høje værdier til lavere værdier og igen tilbage til høje værdier). Omvendt er der ingen horisontale linjer, fordi gråskalaværdierne i hver kolonne ikke ændrer

240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240
240	240	240	150	100	100	150	240	240	240

sig. Som input til et convolutional neural netværk vil man som regel standardisere inputværdierne, så de alle ligger mellem 0 og 1. Det gør man, fordi det viser sig, at backpropagation algoritmen konvergerer hurtigere. Vi standardiserer blot ved at dividere ovenstående værdier med 255 og får

Et feature map, som kan bruges til at finde henholdsvis lodrette og vandrette linjer i et billede er vist i figur 5 og figur 5.

Et feature map f_1, som kan finde vandrette linjer i billeder.

Et feature map f_2, som kan finde lodrette linjer i billeder.

Det ses hurtigt, at hvis vi anvender feature map f_1 til at vægte værdierne i et vilkårligt 9×9 vindue i det originale billede, så får man 0 (såfremt vi sætter biasværdien $v_0 = 0$). Anvender vi f.eks. f_1 på det tredje vindue fås

$$0 + 1 \cdot 0.94 + 1 \cdot 0.59 + 1 \cdot 0.39 + 0 \cdot 0.94 \tag{147}$$

$$+ 0 \cdot 0.59 + 0 \cdot 0.39 - 1 \cdot 0.94 - 1 \cdot 0.59 - 1 \cdot 0.39 = 0 \tag{148}$$

Da $\sigma(0) = 0.5$ ender vi altså med et skjult lag, hvor alle 64 skjulte neuroner sender 0.5 frem i netværket. Det svarer til,

0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94
0.94	0.94	0.94	0.59	0.39	0.39	0.59	0.94	0.94	0.94

1	1	1
0	0	0
-1	-1	-1

at dette feature map ikke har fundet nogle vandrette linjer i billedet.

Bruger vi nu derimod f_2 på det samme vindue, så får vi

$$0 + 1 \cdot 0.94 + 0 \cdot 0.59 - 1 \cdot 0.39 + 1 \cdot 0.94 \\ + 0 \cdot 0.59 - 1 \cdot 0.39 + 1 \cdot 0.94 + 0 \cdot 0.59 - 1 \cdot 0.39 = 1.65$$

Anvendes sigmoid-funktionen fås $\sigma(1.65) = 0.84$. Sådan fortsættes for alle 64 vinduer (local receptive fields), og man ender med nedenstående convolutional lag, hvor værdierne også er vist med de tilhørende farver (i den allermørkeste søjle står der 0.16 - det er bare lidt svært at se!):

Den lodrette linje i det oprindelige billede kommer her til udtryk som henholdsvis en lysegrå og en mørkegrå lodret stribe. Den lysegrå stribe viser, at der på det oprindelige billeder har været en overgang fra lys til mørk og omvendt

1	0	-1
1	0	-1
1	0	-1

0.5	0.74	0.84	0.65	0.35	0.16	0.26	0.5
0.5	0.74	0.84	0.65	0.35	0.16	0.26	0.5
0.5	0.74	0.84	0.65	0.35	0.16	0.26	0.5
0.5	0.74	0.84	0.65	0.35	0.16	0.26	0.5
0.5	0.74	0.84	0.65	0.35	0.16	0.26	0.5
0.5	0.74	0.84	0.65	0.35	0.16	0.26	0.5
0.5	0.74	0.84	0.65	0.35	0.16	0.26	0.5
0.5	0.74	0.84	0.65	0.35	0.16	0.26	0.5

viser den mørkegrå stribe, at der på det oprindelige billede har været en overgang fra mørk til lys.

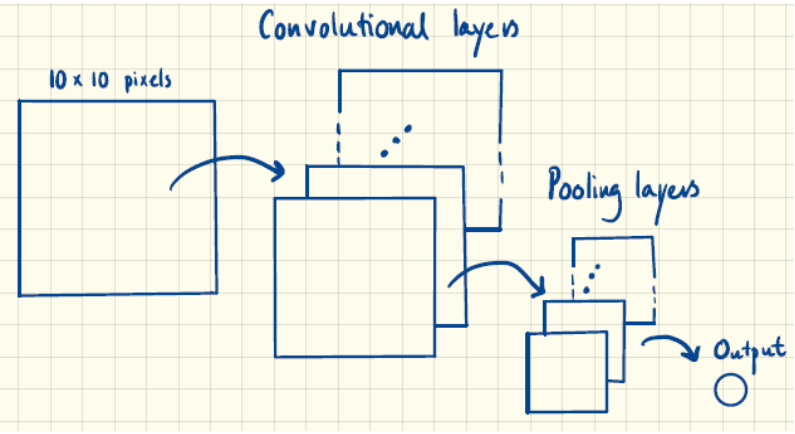
Herefter kondenseres billedet yderligere i det man kalder for et *pooling layer*. Det gør man ved at inddele ovenstående billede i f.eks. 2×2 pixels vinduer, som *ikke* overlapper hinanden. Herefter kan man gøre forskellige ting - en simpel mulighed er at tage maksimumsværdien af de 4 pixels i hvert billede. Dette kaldes for *max pooling*. Gør man det får man følgende 4×4 pixels billede (igen vist med de tilhørende farver):

0.74	0.84	0.35	0.5
0.74	0.84	0.35	0.5
0.74	0.84	0.35	0.5
0.74	0.84	0.35	0.5

Læg mærke til hvordan man på dette kondenserede billede tydeligt kan se den lodrette stribe, som var i det oprindelige

billede.

Så dette er den overordnede idé, men der er flere ting at bemærke. For det første bruges et feature map til at finde egenskaber på et billede som vist ovenfor, men man vil typisk ikke kun lede efter én egenskab men flere. Det vil sige, at der ikke kun er ét convolutional lag, men måske 5 eller 20 lag. For hvert af disse lag fås et tilhørende pooling lag. Dette er illustreret på figur 13.



Figur 13: Convolutional neural network.

Alle neuroner i pooling lagene forbindes nu til outputneuronen. Igen udregnes sigmoid-funktionen taget på en vægtet sum af alle neuronerne fra pooling lagene. Der er også mulighed for at indlægge flere convolutional og pooling lag i netværket. Ligesom der også kan indlægges flere "almindelige" neuroner (som vi tidligere har set det) efter pooling lagene.

For det andet vil man ikke anvende feature maps som på forhånd er defineret (som vi gjorde det i eksemplet ovenfor). For hvert convolutional layer trænes i stedet for de fælles vægte, som skal benyttes i det pågældende feature map. Selve træningen af netværket foregår igen ved hjælp af backpropagation, men nu justeret til de begrænsninger der er lagt ind i netværket.

For det tredje bliver antallet af vægte, som skal fittes, drastiske reduceret i et convolutional neuralt netværk sammenlignet med et almindelig neuralt netværk. Lad os som eksempel se på klassifikation af et 10×10 pixels billede. Antag, at vi anvender et local receptive field (vindue) på 3×3 pixels. Det vil sige, at

hvert convolutional lag vil bestå af 8×8 neuroner. Da vægtene er delte, vil vi for hvert convolutional lag få bruge for $3 \cdot 3 + 1 = 10$ vægte. Antag, at vi har 5 convolutional lag (bestående af $5 \cdot 8 \cdot 8 = 320$ neuroner), så får vi i alt brug for $5 \cdot 10 = 50$ vægte. For at komme til pooling laget anvendes ingen vægte. Men pooling laget vil bestå af 5 pooling lag med i alt 4×4 neuroner. Det giver 80 neuroner i alt. Forbindes disse til én outputneuron skal vi bruge 81 vægte (én for hver neuron plus en bias). Alt i alt ender vi med 131 vægte.

Havde vi i stedet lavet et fuldt forbundet kunstigt neuralt netværk med 100 input neuroner, ét skjult lag med 64 neuroner samt én outputneuron, så skulle vi først have bruge $101 \cdot 64 = 6464$ vægte for at forbinde input neuronerne med samtlige neuroner i det første skjulte lag. Herefter har vi brug for 65 vægte for at forbinde det skjulte lag med outputneuronen. Alt i alt giver det 6529 vægte! Der er altså en voldsom forskel i antallet af vægte, som skal læres. For at undgå overfitting må vi have mange flere træningsdata, end vi har vægte. Det betyder, at vi typisk vil kunne nøjes med færre træningseksempler i et convolutional neural netværk sammenlignet med et klassisk fuldt forbundet kunstigt neuralt netværk. Ydermere viser det sig, at convolutional neurale netværk kan trænes til at klassificere billeder langt bedre end et almindeligt kunstigt neuralt netværk. Netop fordi at de forskellige pixels placeringer i forhold til hinanden tages i betragtning, når et convolutional neuralt netværk konstrueres.

Videre læsning

Baktoft, Allan. 2014. *Matematik i Virkeligheden. Bind 2.* Forlaget Natskyggen.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning.* Springer.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning with Applications in r. Second Edition.* Springer.

Mitchell, Tom M. 1997. *Machine Learning.* The McGraw-Hill Companies, Inc.

Nielsen, Michael A. 2015. *Neural Networks and Deep Learning.*

Determination Press. http://neuralnetworksanddeeplearning.com/index.html.

Sørensen, Henrik Kragh, and Mikkel Willum Johansen. 2020. *"Invitation Til de Datalogiske Fags Videnskabsteori". Lærebog Til Brug for Undervisning Ved Institut for Naturfagenes Didaktik, Københavns Universitet*. Under udarbejdelse.