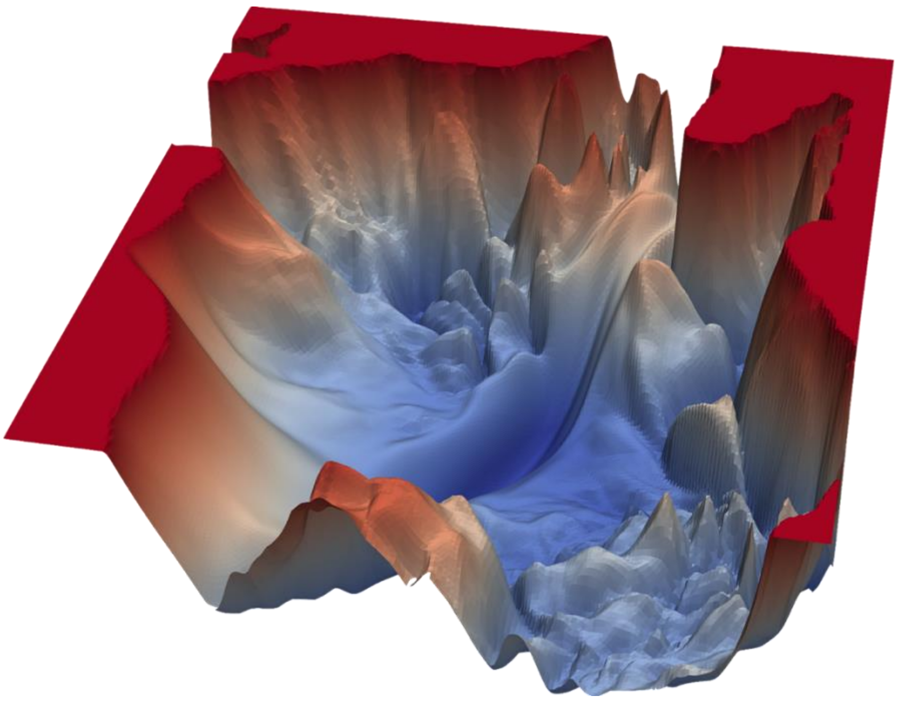
Kvantitativ analyse af Losslandskabet i Sigmoid-, ReLU- og Leaky-ReLU-aktiveringsfunktioner



Naturfag Forskerspirer 2023
Forskerkontakt: Kristoffer Stensbo-Smidt

Projekt Forskerspirer 2023	
Titel	Kvantitativ analyse af Losslandskabet i Sigmoid-, ReLU- og Leaky-ReLU-aktiveringsfunktioner
Identifikationskode	[LL52]
Navn	Laurits Kromann Larsen
Gymnasium	Svendborg Gymnasium
Fagområde	Naturvidenskab

FORSKERSPIRER

Indholdsfortegnelse

Indledning

November 2022 var en revolutionerende måned for verden: Prototypen af ChatGPT[1] for offentligheden blev lanceret[2]. Over to måneder voksede antallet af brugere til over 100 millioner mennesker[3], og ChatGPT er i dag et værktøj, som bruges eller kan bruges i stort set alle brancher. ChatGPT er opbygget af neurale netværk som repræsenterer en programmeringsmetode, der er inspireret af biologien og menneskets hjerne, hvilket giver en computer mulighed for at lære fra observerede data. Disse neurale netværk er inden for de sidste par årtier blevet meget populære, da de er gode til mønstergenkendelse. De opnår resultater, som vi i den vildeste fantasi ikke havde turde håbe på. Nye modeller og strukturer dukker op konstant. Forskellige kreative teknikker afprøves, virker de, så bruges de. Men for at kunne udvikle og forbedre modellerne er vi nødt til at have en forståelse for, hvorfor det virker.

Jeg har hele mit liv været fascineret af kunstig intelligens - især når den bliver praktisk anvendelig. Det gælder både, når den optræder på internettet og når den kobles til fysiske genstande, som f.eks. robotter. Inden for det sidste års tid har det dog været matematikken bag neurale netværk, som har optaget mig. Derfor dette forskerspireprojekt.

Problemformulering og formål

Aktiveringsfunktionerne i neurale netværk spiller en essentiel rolle i forhold til, hvor godt netværket præsterer og hvor stor en "Loss"[4] der forekommer; *" Clearly, the effect of ReLU on the loss landscape is not yet completely understood."*[5]

Jeg vil derfor undersøge følgende problemstilling:
Hvordan kan man ved hjælp af en kvantitativ analyse af losslandskabet for forskellige aktiveringsfunktioner få en større viden om disses påvirkning på loss og accuracy.

I den sammenhæng vil jeg:
- *analysere og undersøge losslandskabet for forskellige aktiveringsfunktioner.*
- *karakterisere losslandskabet, med forskellige kvantitative metrikker, såsom "antal lokale minima", "jævnheden af losslandskabet", "bredde af minima", samt metoder fra "shape analysis".*

Jeg håber, i forlængelse af ovenstående, på at kunne uddrage en mere generel forståelse for diverse aktiveringsfunktioner, hvilket måske kan føre til en bedre udnyttelse af funktionernes egenskaber.

Da der imidlertid findes utallige aktiveringsfunktioner, har jeg valgt at afgrænse mit forsøg til aktiveringsfunktionerne: **ReLU**, **Leaky-ReLU** og **Sigmoid**.

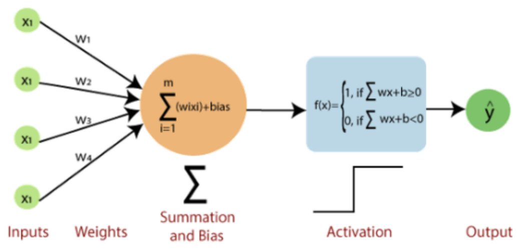
[1] OpenAi: https://chat.openai.com/
[2] (Simonsen, 2023)
[3] (Duarte, 2023)
[4] Loss er en fagterm for hvor stort tab der er i processen. Senere i projektet benyttes "losslandskab", som min egen fordanskning af det engelske ord "loss landscape"
[5] (bosman, engelbrecht, & helbig, 2023)

Teori

Single perceptron

En single perceptron er bygget op af en neuron, inputs og outputs.[6] Inputs, som bliver ganget med en vægt, fodrer neuronen med data, hvorefter neuronen bearbejder data, og kommer med et output:

$$Neuronen(x_v \dots, x_m) = \sum_{i=1}^{m} (wi \cdot xi) + bias$$


x₁
x₁
x₁
x₁
w₁
w₂
w₃
w₄
$\sum_{i=1}^{m}$(wixi)+bias
f(x)= {1, if ∑ wx+b≥0 ; 0, if ∑ wx+b<0
ŷ
∑
Inputs
Weights
Summation and Bias
Activation
Output
Figur 1: En single perceptron

Summen bliver så kørt igennem en aktiveringsfunktion, der modificerer værdien. Aktiveringsfunktionen er et af de vigtigste led i strukturen og opbygningen af de neurale netværk. Det er aktiveringsfunktionerne som er med til at give en ikke-lineær struktur i netværket, hvilket giver netværket bedre mulighed for at lære komplekse problemer.

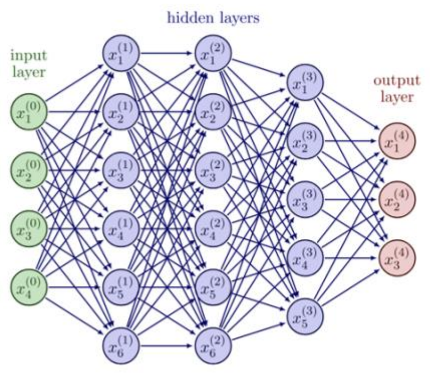
Aktiveringsfunktioner

Pierre François Verhulst udviklede igennem en række artikler[7] i årene 1838-47 det, som vi i dag kender som Sigmoid-funktionen: $\sigma(z) = \frac{1}{1+e^{-z}}$ (se bilag 10). Denne ikke-lineære funktion presser alle værdier ind mellem 0 og 1. Dette vil i visse tilfælde udløse det såkaldte ”vanishing gradient problem”[8], hvor gradienterne vil forsvinde, og dermed vil netværket ikke kunne lære. Sigmoid-funktionen anvendes stadig i nyere tid, dog har den såkaldte ReLU[9]: $ReLU(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$ (se bilag 10) overtaget populariteten og er den mest anvendte funktion i dag. Det er en ikke-lineær funktion, dog er den stadig meget simpel. Når man finder den afledte til funktionen, vil den være: $\begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$. Dette medfører, at man hurtig kan ”backpropagate”[10] igennem netværket. Et eksempel på en alternativ aktiveringsfunktion er leaky-relu funktionen: $Leaky\_Relu(z) = \begin{cases} z, & z > 0 \\ z \cdot a, & z \leq 0 \end{cases}$ Her tilføres en minimal hældning til funktionen. Dette er et forsøg på at løse problemet, ”the dying relu problem”[11] som forekommer, når inputtet er negativt, hvilket resulterer i at gradienten er nul. Dette vil medvirke til at netværket ikke kan lære.

[6] (Nielsen, 2019)
[7] (TARANOVICH, 2019)
[8] (Wang, 2019)
[9] Rectified Linear Unit(ReLU)
[10] (Olah, 2015)
[11] (lu, Shin, su, & Karniadakis, 2020)

Strukturen bag et Neuralt Netværk

På figur 2 er der vist et " multi-layer perceptron"[12], som består af mange enkelte perceptroner, eller mere præcis et fuldt forbundet ANN[13] som er opbygget af et inputlayer, et hiddenlayer og et outputlayer. Når netværket trænes, skal vægtene som bliver ganget på inputtet opdateres. Her bruges en teknik, som hedder backpropagation[14]. Da vi i vores tilfælde bruger supervised learning[15] beregner vi et loss mellem outputtet og vores labels. Labels er vores ønskede resultat eller klassifikation. Hvis vi fodrer netværket med data og vores ønskede resultat er 11, men netværket gætter på tallet 13 kan vi beregne et loss mellem vores label og hvad netværket gættede på. Cross Entropy Loss[16] er en metode til at beregne losset på:



Figur 2: Et fuldt forbundet ANN

$$H(t,p) = -\sum_{s \in S} t(s).\ log(p(s))$$

Hermed kan vi så opdatere vægtene og gætte igen. For at opdatere vægtene finder man den partielle afledte funktion. Dette gøres blandt andet ved den såkaldte "kæde regel"[17] , og hermed kan vi ved en given learning rate[18] og en algoritme[19] opdatere vægtene, så man til sidst ender ud med en forhåbentlig lav loss og høj accuracy[20].

Convolutional Neural Network

Når en person f.eks. får vist et billede af en kat, er vedkommende ikke usikker på hvad billedet forestiller. Forskellige områder af vores hjerne bliver aktiveret, når forskellige komponentprocesser inden for visuel genkendelse er i gang[21]. Derfor kan vi skelne mellem en kat og en hund. Dog bliver det meget sværere, når vi skal få en computer til at gøre det samme. Et ANN[22] kan bruges, hvis man skal træne netværket på meget små, sort-hvid-billeder, feks 28x28 pixels. Bliver billederne større, og der er forskellige farvekanaler[23] , vil dette kræve en enorm mængde inputs, og dermed også en dramatisk stigning i antal vægte. Det er her Convolutional Neural Network (CNN) kommer på banen. CNN[24] kan bestå af mange forskellige lag. Hovedsageligt er der tre lag. Et "convolutional" lag (se bilag 1), pooling lag (se bilag 2) og et fuldt forbundet lag (ANN). Imellem hvert lag er der en aktiveringsfunktion. Disse aktiveringsfunktioner benyttes konstant, men forståelsen for deres succes eller fiasko er stadig genstand for undren.[25] Følgende projekt er et forsøg på at udvide forståelsen for disse funktioner.
(Shah, 2023)

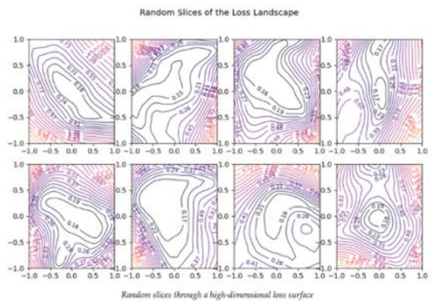
[12] (the universal approximation theorem, 2023)
[13] Artificial neural network
[14] (olah, 2015)
[15] (IBM, u.d.)
[16] (Shah, 2023)
[17] (olah, 2015)
[18] (Brownlee, 2020)
[19] (Brownlee, 2020)
[20] Accuracy: På dansk forudsigelse. Hvor mange rigtige gæt kom netværket med…
[21] (Kanwisher, Chun, McDermott, & Ledden, 1996)
[22] Artificial Neural Network
[23] RGB. Red, Green, Blue. Farvekanal
[24] Convolutional neural network
[25] (bosman, engelbrecht, & helbig, 2023)

Metode

Losslandskab

Et netværk kan bestå af flere millioner af parametre. Derfor kan vi ikke direkte visualisere loss som funktion af parametre. Dog kan jeg lave en 2d visualisering af et stykke af netværket ved at lave et såkaldt ”slice”. Hvis vi har tre punkter, w_1, w_2, w_3, i det højdimensionelle rum, kan vi lave to vektorer: $u = (w_2 - w_1)$ og $v = (w_3 - w_1)$. Vi bruger trigonometri til at gøre dem ortogonale. Ved brug af følgende formel (1) kan vi plotte punkterne ind

$$w(x, y) = w_1 + u \cdot x + v \cdot y \quad (1)$$[26]



Figur 3: Forskellige slices af losslandskab

I en udvidelse af dette bruges den såkaldte ”Principal component analysis”[27] [28]. Et eksempel på et losslandskab ses på figur 3[29]. Ud fra disse losslandskaber vil jeg udføre nogle kvantitative analyser.

Antal lokale minimaer

Når vi træner et netværk, ønsker vi at finde det globale minimum. Dog kan netværket i visse tilfælde ramme et lokalt minimum, hvorved netværket ”tror” det har fundet den bedste løsning. Dette vil reducere ydeevnen af netværket og kan føre til lavere accuracy og højere loss. Jo flere lokale minimaer der findes, jo større chance er der for at netværket vil ramme et lokalt minimum i stedet for et globalt minimum. Derfor kan antal af lokale minimaer have indflydelse på netværkets evne til at lære. Jeg vil derfor optælle antallet af lokale minimaer, da det kan have en effekt på losset.

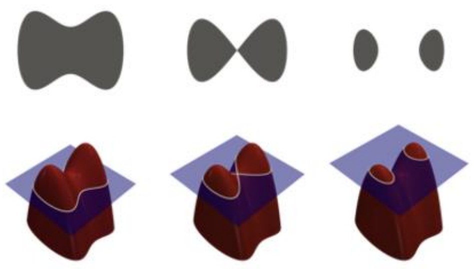
Jævnhed af losslandskab

Jeg vil også kvantitativt undersøge ”jævnheden” af losslandskabet. Jo mere jævnt losslandskabet er, desto mere generaliserende blive netværket.[30] Dette vil jeg gøre ved at beregne forskellen mellem hver pixel af losslandskabet. Jo mindre forskel der er mellem hver pixel, desto mere jævnt er landskabet. Jeg kan så plotte disse forskelle og dermed få en visualisering af hvor jævnt losslandskabet er.

[26] (Izmailov, Garipov, & Wilson, 2018)
[27] (Jaadi, 2023)
[28] (wikipidea , u.d.)
[29] (Visualizing the Loss Landscape of a Neural Network, 2020)
[30] (wen, et al., 2018)

Bredde af minima

Bredden af minimaet er også væsentligt for netværkets generalisering. Et bredt minimum kan betyde at netværket er mere robust i forhold til små ændringer i parametrene. Et bredt minimum vil også regulere for andre lokale minimaer. Et bredt minimum vil være lettere at finde og konvergere til under træning i forhold til et smalt minimum. Jeg vil bruge en såkaldt ”Level-set method”[31]. Level-set metoden gør det nemmere at håndtere geometriske former, der ændrer deres topologi over tid (se figur 4). Metoden gør det relativt enkelt at spore og beskrive disse ændringer, hvilket er nyttigt i forhold til minimaets udvikling. Jeg kan altså få en bedre visualisering af minimaet, og jeg kan så geometrisk bestemme bredden af minimaet til forskellige tidspunkter.

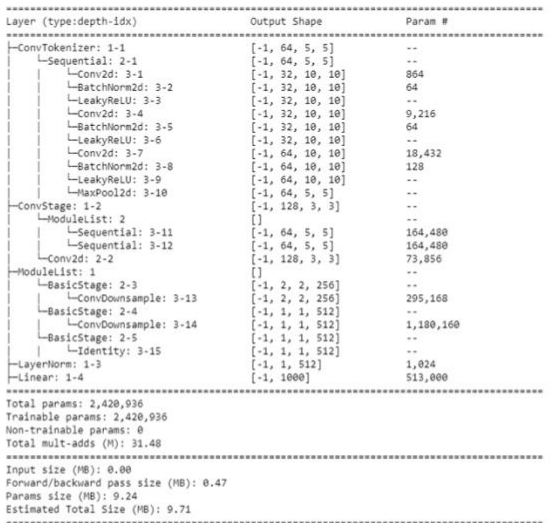

Figur 4: Level-set method

Shape Analysis

Jeg vil udover ovenstående metoder prøve at generalisere og analysere losslandskaberne ved hjælp af metoder fra: ”Shape Analysis”.[32] Her vil jeg kvantitativt beregne et ”shape index”[33] og ”curvedness”[34]. Jeg håber på, at en mere empirisk beskrivelse af losslandskabet kan være med til at generalisere og give bedre overblik over de forskellige aktiveringsfunktioners effektivitet.

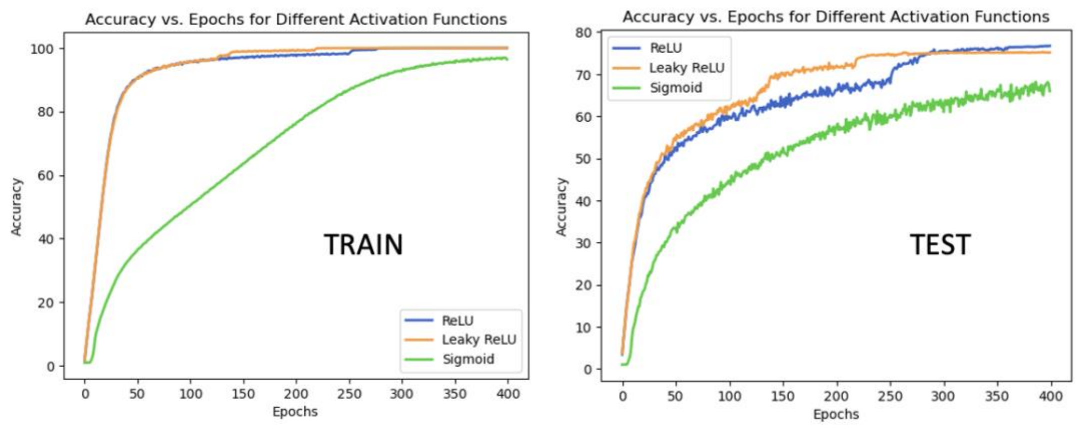
Pilotforsøg

Som forarbejde til forsøget har jeg lavet et pilotforsøg. Ideen har været selv at udforme et neuralt netværk og efterfølgende træne det. Formålet var at teste aktiveringsfunktionerne: ReLU, Leaky-ReLU og Sigmoid. Jeg ville sammenligne loss og accuracy mellem funktionerne, og dette ville give et godt overblik over om det overhovedet var grundlag for en videre undersøgelse. Jeg kodede mit neurale netværk i python med pytorch (Se bilag 4 og evt fodnote[35]). Jeg har her benyttet den såkaldte ConvMLP-s[1] model. På figur 5 ses modellen fra pilotforsøget inklusive de forskellige lag. Jeg trænede netværket på datasættet Cifar-100[36]. Alle resultaterne kan ses i bilag 5 og 6 samt aktiveringsfunktionerne med


```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─ConvTokenizer: 1-1                     [-1, 64, 5, 5]            --
|    └─Sequential: 2-1                   [-1, 64, 5, 5]            --
|    |    └─Conv2d: 3-1                  [-1, 32, 10, 10]          864
|    |    └─BatchNorm2d: 3-2             [-1, 32, 10, 10]          64
|    |    └─LeakyReLU: 3-3               [-1, 32, 10, 10]          --
|    |    └─Conv2d: 3-4                  [-1, 32, 10, 10]          9,216
|    |    └─BatchNorm2d: 3-5             [-1, 32, 10, 10]          64
|    |    └─LeakyReLU: 3-6               [-1, 32, 10, 10]          --
|    |    └─Conv2d: 3-7                  [-1, 64, 10, 10]          18,432
|    |    └─BatchNorm2d: 3-8             [-1, 64, 10, 10]          128
|    |    └─LeakyReLU: 3-9               [-1, 64, 10, 10]          --
|    |    └─MaxPool2d: 3-10              [-1, 64, 5, 5]            --
├─ConvStage: 1-2                         [-1, 128, 3, 3]           --
|    └─ModuleList: 2                     []                        --
|    |    └─Sequential: 3-11             [-1, 64, 5, 5]            164,480
|    |    └─Sequential: 3-12             [-1, 64, 5, 5]            164,480
|    └─Conv2d: 2-2                       [-1, 128, 3, 3]           73,856
├─ModuleList: 1                          []                        --
|    └─BasicStage: 2-3                   [-1, 2, 2, 256]           --
|    |    └─ConvDownsample: 3-13         [-1, 2, 2, 256]           295,168
|    └─BasicStage: 2-4                   [-1, 1, 1, 512]           --
|    |    └─ConvDownsample: 3-14         [-1, 1, 1, 512]           1,180,160
|    └─BasicStage: 2-5                   [-1, 1, 1, 512]           --
|    |    └─Identity: 3-15               [-1, 1, 1, 512]           --
├─LayerNorm: 1-3                         [-1, 1, 512]              1,024
├─Linear: 1-4                            [-1, 1000]                513,000
==========================================================================================
Total params: 2,420,936
Trainable params: 2,420,936
Non-trainable params: 0
Total mult-adds (M): 31.48
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.47
Params size (MB): 9.24
Estimated Total Size (MB): 9.71
==========================================================================================
```
Figur 5: Model ConvMLP-s med Leaky-Relu som aktiveringsfunktion

[31] (wikipedia , u.d.)
[32] (wikipedia, u.d.)
[33] (Koenderink & Doorn, 1992)
[34] (Koenderink & Doorn, 1992)
[35] Hver af aktiveringsfunktionerne blev trænet i 400 epochs[35] med en batch-size[35] på 100 og en learning rate på henholdsvis: 10^{-4} for Relu og Leaky_Relu og 10^{-5} for Sigmoid. Jeg har her benyttet den såkaldte ConvMLP-s[35] model(se figur 5). Da koden var hardcoded[35] var jeg nødt til manuelt at omprogrammere koden så jeg kunne tilføje de ønskede aktiveringsfunktioner (se bilag 4). Derudover brugte jeg en scheduler: reduceLrOnPlatteau[35]. Den skulle skulle sænke learningraten med faktor på 0,1 når losset ikke var faldet i 15 epochs.
[36] (Canadian Institute for Advanced Research, 100 classes): Cifar-100 er et kendt datasæt med 60000 forskellige billeder hvoraf der er 100 klasser og 20 ”super-klasser”. Super klasser er større kategorier. Feks; dyr, møbler osv…

forskellige seeds i bilag 7. Nedenfor ses resultaterne for accuracy for hhv. træning og test.



Figur 6: Accuracy ved train og test for aktiveringsfunktionerne, ReLU, Leaky-ReLU og Sigmoid. Gennemsnit af 3 forskellige seeds for at regulere for støj.

Konklusion på pilotforsøg

Ud fra de visualiserede resultater fra pilotforsøget, (Se bilag 5) er det tydeligt at se en forskel på ydeevnen for de tre aktiveringsfunktioner. På x-aksen har vi Epochs, antal gange vi træner netværket fuldt igennem, og på y-aksen har vi enten accuracy eller loss. ReLU og Leaky-ReLU klarer sig betydeligt bedre end Sigmoid-funktionen. De opnår hurtigere og bedre resultater. Inden for 50 epochs har ReLU og Leaky-ReLU allerede ramt +90% accuracy. Hvorimod det tager Sigmoid omkring 300 epochs at nå samme resultat. Derudover rammer Sigmoid aldrig samme acurracy som ReLU og Leaky-ReLU. På test-datasættet hvor funktionerne bliver udsat for data de aldrig har set før, præsterede ReLU og Leaky-ReLU stadig markant bedre (Se bilag 6). Dette kunne derfor tyde på, at ReLU og Leaky-ReLU er bedre til at generalisere end Sigmoid funktionen.

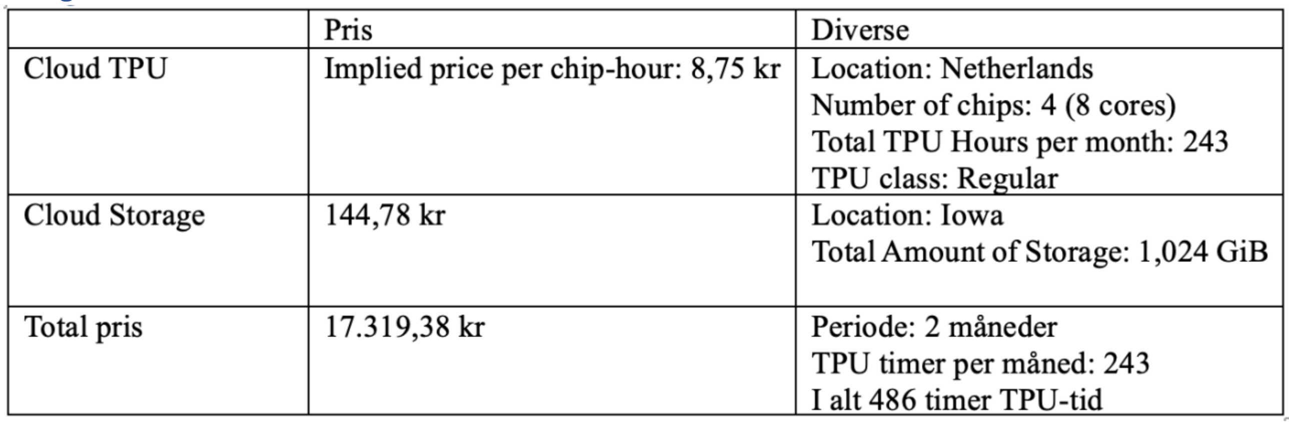
Men hvorfor? For at kunne analysere hvorfor ReLU og Leaky-ReLU fungerer bedre er det nødvendigt med et større analysearbejde, en større datamængde og en større computerkraft.

Udførsel

Når losslandskabet skal analyseres, er det vigtigt, at det har en tilstrækkelig høj opløsning. Hvis opløsningen ikke er god nok, vil der mangle detaljer, små lokale minimaer, som ikke vil blive visualiseret. Hver pixel i losslandskabet repræsenterer en epoch. Hvis jeg vil have et losslandskab med 500x500 pixels skal jeg træne 250000 epochs. Dette skal jeg gøre én gang for hver aktiveringsfunktion. I alt skal jeg derfor træne 750000 epochs. Derfor har jeg brug for en stærk computer. Jeg vil derfor bruge de 20.000 kr på Cloud TPU[37] (se bilag 3 og budget). Alt efter hvor lang tid det tager at træne én epoch på Cloud TPU'en, vil jeg regulere opløseligheden af de tre losslandskaber jeg laver, og dermed reducere antallet af epochs, jeg skal træne.

[37] https://cloud.google.com/products/calculator/#id=8c9e8b7d-13d6-46bc-afb7-5c8a7f6104d5

Budget



	Pris	Diverse
Cloud TPU	Implied price per chip-hour: 8,75 kr	Location: Netherlands Number of chips: 4 (8 cores) Total TPU Hours per month: 243 TPU class: Regular
Cloud Storage	144,78 kr	Location: Iowa Total Amount of Storage: 1,024 GiB
Total pris	17.319,38 kr	Periode: 2 måneder TPU timer per måned: 243 I alt 486 timer TPU-tid

Tidsplan

Jeg vil starte med at lave de tre losslandskaber, som visualiseres i form af et heatmap. Forventet tid 486 timer. Derpå vil jeg påbegynde den kvantitative undersøgelse af losslandskaberne (som beskrevet i metodeafsnittet). Jeg vil starte med at optælle antal minimaer, hvorefter jævnheden beregnes. Derefter vil jeg beregne bredden af det globale minima og prøve empirisk at kategorisere landskabet ved hjælp af shape analysis.

Konklusion

Jeg håber, at jeg ud fra forsøget kan være med til at danne et større grundlag og forståelse for hvordan de forskellige aktiveringsfunktioner fungerer. Hvis det viser sig, at ReLU- og Leaky-ReLU-funktionerne har færre lokale minimaer, bredere globale minimaer og er mere jævne, så kunne dette danne grundlag for videre forskning. Er det muligt at lave nogle aktiveringsfunktioner, som opfylder de kriterier og fungerer lige så godt hvis ikke bedre? Derudover vil den såkaldte shape analysis som bliver lavet også være med til at kategorisere funktionerne. Til fremtidig forskning kunne man træne en model på andre modeller og deres kvantitative resultater ud fra diverse parametre og dermed komme med bud på aktiveringsfunktioner inden for forskellige problemer. En form for symbiose. Dog kræver dette ekstremt mange losslandskaber og data.

Anerkendelse

Jeg vil gerne uddele en stor tak til min forskerkontakt, Kristoffer Stensbo-Smidt, postdoc: Department of Applied Mathematics and Computer Science (DTU). Derudover vil jeg gerne uddele en tak til min vejleder Christian Maaløv Andreasen (Svendborg Gymnasium) og de andre forskerspiredeltagere, som har været med til at motivere og inspirere mig.

Litteraturliste:

- bosman, a. s., engelbrecht, a., & helbig, m. (28. juni 2023). Empirical Loss Landscape Analysis of Neural Network Activation Functions.

- Brownlee, J. (12. september 2020). *machinelearningmastery*. Hentet fra machine learning mastery: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

- Duarte, F. (13. july 2023). *explodingtopics*. Hentet fra https://explodingtopics.com/blog/chatgpt-users

- IBM. (u.d.). *IBM*. Hentet fra https://www.ibm.com/topics/supervised-learning

- bosman, a. s., engelbrecht, a., & helbig, m. (28. juni 2023). Empirical Loss Landscape Analysis of Neural Network Activation Functions.

- Brownlee, J. (12. september 2020). *machinelearningmastery*. Hentet fra machine learning mastery: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

- *Dot Product*. (u.d.). Hentet fra wikipedia: https://en.wikipedia.org/wiki/Dot_product

- Duarte, F. (13. july 2023). *explodingtopics*. Hentet fra https://explodingtopics.com/blog/chatgpt-users

- IBM. (u.d.). *IBM*. Hentet fra https://www.ibm.com/topics/supervised-learning

- Izmailov, P., Garipov, T., & Wilson, A. G. (2018). *Visualizing Mode Connectivity*. Hentet fra izmailovpavel.github: https://izmailovpavel.github.io/curves_blogpost/

- Jaadi, Z. (29. marts 2023). *builtin*. Hentet fra https://builtin.com/data-science/step-step-explanation-principal-component-analysis

- Jadon, S. (16. marts 2018). *Introduction to Different Activation Functions for Deep Learning*. Hentet fra medium.com: https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092

- *javatpoint*. (u.d.). Hentet fra https://www.javatpoint.com/single-layer-perceptron-in-tensorflow

- Kanwisher, N., Chun, M., McDermott, J., & Ledden, P. (december 1996). Functional imaging of human visual recognition. *sciencedirect*, s. 55-67.

- Koenderink, J. J., & Doorn, A. J. (oktober 1992). Surface shape and curvature scales. *ScienceDirect*, s. 557-564.

- krisnhamurthy, b. (28. oktober 2022). *builtin*. Hentet fra https://builtin.com/machine-learning/relu-activation-function

- Li, H., Xu, Z., Taylor, G., Studer, C., & Goldstein, T. (7. november 2018). Visualizing the Loss Landscape of Neural Nets. *cornell university*, s. 1-18.

- Li, J., Hassani, A., Walton, S., & Shi, H. (18. september 2021). ConvMLP: Hierarchical Convolutional MLPs for Vision. s. 1-12.

- lu, l., Shin, Y., su, Y., & Karniadakis, G. E. (14. oktober 2020). Dying ReLU and Initialization: Theory and Numerical Examples. *global science press*, s. 1671-1706.

- *Max Pooling* . (u.d.). Hentet fra paperswithcode: https://paperswithcode.com/method/max-pooling

- Nielsen, M. (December 2019). *neuralnetworksanddeeplearning*. Hentet fra http://neuralnetworksanddeeplearning.com/chap1.html

- olah, c. (31. august 2015). *colah's blog* . Hentet fra http://colah.github.io/posts/2015-08-Backprop/

- Shah, D. (26. januar 2023). *Cross Entropy Loss: Intro, Applications, Code*. Hentet fra v7labs : https://www.v7labs.com/blog/cross-entropy-loss-guide

- Simonsen, J. F. (20. juli 2023). *Somejuan* . Hentet fra https://somejuan.dk/blog/hvad-er-chatgpt-og-hvordan-bruger-du-ai-chatbotten/

- TARANOVICH, S. (27. april 2019). *EDN*. Hentet fra https://www.edn.com/how-the-sigmoid-function-is-used-in-ai/

- *the universal approximation theorem*. (26. marts 2023). Hentet fra deep-mind: https://www.deep-mind.org/2023/03/26/the-universal-approximation-theorem/

- *Visualizing the Loss Landscape of a Neural Network*. (30. december 2020). Hentet fra Math for Machines: https://mathformachines.com/posts/visualizing-the-loss-landscape/

- Wang, C.-F. (8. jan 2019). *towardsdatascience*. Hentet fra https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

- wen, w., wang, y., yan, f., xu, C., wu, c., & chen, y. (2. december 2018). SmoothOut: Smoothing Out Sharp Minima to Improve Generalization in Deep Learning.

- *wikipedia*. (u.d.). Hentet fra Shape analysis (digital geometry): https://en.wikipedia.org/wiki/Shape_analysis_(digital_geometry)

- *wikipedia* . (u.d.). Hentet fra Level-set method: https://en.wikipedia.org/wiki/Level-set_method

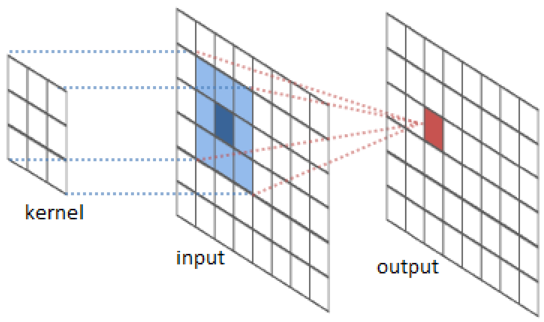
- *wikipidea* . (u.d.). Hentet fra Principal component analysis: https://en.wikipedia.org/wiki/Principal_component_analysis

- Li, H., Xu, Z., Taylor, G., Studer, C., & Goldstein, T. (7. november 2018). Visualizing the Loss Landscape of Neural Nets. *cornell university*, s. 1-18.

Bilag

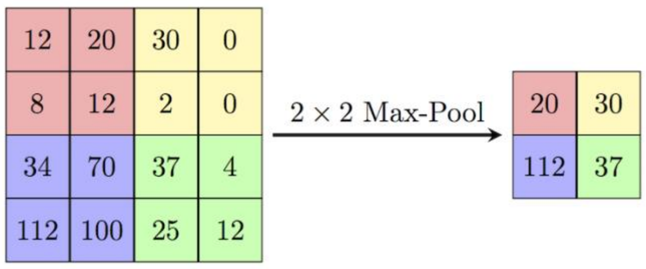
Bilag 1
I ”convolutional layer” bruges et såkaldt filter eller kernel til at finde mønstre i billedet. Filteret som anvist på (Figur 1) har dimensionerne 3x3. Filteret kører over billedet pixel for pixel og beregner prikproduktet:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

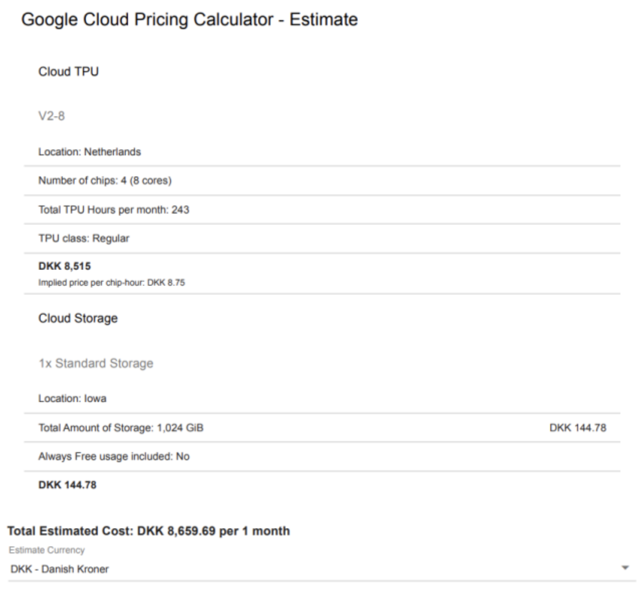


Figur 1

Bilag 2
Pooling layer:
Max Pooling laget er med til at nedskære billedets størrelse. Et filter med N x N dimensioner kører over billedet. Den største værdi bliver udvalgt og hermed nedskaleres billedet.



Bilag 3: Google Cluod TPU



Google Cloud Pricing Calculator - Estimate

Cloud TPU

V2-8

Location: Netherlands

Number of chips: 4 (8 cores)

Total TPU Hours per month: 243

TPU class: Regular

DKK 8,515

Implied price per chip-hour: DKK 8.75

Cloud Storage

1x Standard Storage

Location: Iowa

Total Amount of Storage: 1,024 GiB DKK 144.78

Always Free usage included: No

DKK 144.78

Total Estimated Cost: DKK 8,659.69 per 1 month

Estimate Currency

DKK - Danish Kroner

Bilag 4: Kode til mit neurale netværk. Heriblandt ConvMLP-s model

```python
from torch.hub import load_state_dict_from_url
import torch
from torch.nn import Module, ModuleList, \
    Sequential, \
    Linear, \
    LayerNorm, \
    Conv2d, \
    BatchNorm2d, \
    ReLU, \
    LeakyReLU, \
    Sigmoid, \
    GELU, \
    Identity
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

import torch.nn as nn


def drop_path(x, drop_prob: float = 0., training: bool = False):
    """
    Obtained from: github.com:rwightman/pytorch-image-models
    Drop paths (Stochastic Depth) per sample (when applied in main path of residual blocks).
    This is the same as the DropConnect impl I created for EfficientNet, etc networks, however,
    the original name is misleading as 'Drop Connect' is a different form of dropout in a separate paper...
    See discussion: https://github.com/tensorflow/tpu/issues/494#issuecomment-532968956 ... I've opted for
    changing the layer and argument names to 'drop path' rather than mix DropConnect as a layer name and use
    'survival rate' as the argument.
    """
    if drop_prob == 0. or not training:
        return x
    keep_prob = 1 - drop_prob
    shape = (x.shape[0],) + (1,) * (x.ndim - 1)  # work with diff dim tensors, not just 2D ConvNets
    random_tensor = keep_prob + torch.rand(shape, dtype=x.dtype, device=x.device)
    random_tensor.floor_()  # binarize
    output = x.div(keep_prob) * random_tensor
    return output
```

```
class DropPath(nn.Module):
    """
    Obtained from: github.com:rwightman/pytorch-image-models
    Drop paths (Stochastic Depth) per sample  (when applied in main path of residual blocks).
    """

    def __init__(self, drop_prob=None):
        super(DropPath, self).__init__()
        self.drop_prob = drop_prob

    def forward(self, x):
        return drop_path(x, self.drop_prob, self.training)

class ConvStage(Module):
    def __init__(self,
                 num_blocks=2,
                 embedding_dim_in=64,
                 hidden_dim=128,
                 embedding_dim_out=128,
                 activation=nn.ReLU):
        super(ConvStage, self).__init__()
        self.conv_blocks = ModuleList()
        act_fn = activation() if activation == Sigmoid else activation(inplace=True)
        for i in range(num_blocks):
            block = Sequential(
                Conv2d(embedding_dim_in,
                       hidden_dim,
                       kernel_size=(1, 1),
                       stride=(1, 1),
                       padding=(0, 0),
                       bias=False),
                BatchNorm2d(hidden_dim),
                act_fn,#inplace=True
                Conv2d(hidden_dim, hidden_dim, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False),
                BatchNorm2d(hidden_dim),
                act_fn,#inplace=True
                Conv2d(hidden_dim, embedding_dim_in, kernel_size=(1, 1), stride=(1, 1), padding=(0, 0), bias=False),
                BatchNorm2d(embedding_dim_in),
                act_fn#inplace=True
            )
            self.conv_blocks.append(block)
        self.downsample = Conv2d(embedding_dim_in,
                                 embedding_dim_out,
                                 kernel_size=(3, 3),
                                 stride=(2, 2),
                                 padding=(1, 1))

    def forward(self, x):
        for block in self.conv_blocks:
            x = x + block(x)
        return self.downsample(x)


class Mlp(Module):
    def __init__(self,
                 embedding_dim_in,
                 hidden_dim=None,
                 embedding_dim_out=None,
                 activation=GELU):
        super().__init__()
        hidden_dim = hidden_dim or embedding_dim_in
        embedding_dim_out = embedding_dim_out or embedding_dim_in
        self.fc1 = Linear(embedding_dim_in, hidden_dim)
        self.act = activation()
        self.fc2 = Linear(hidden_dim, embedding_dim_out)

    def forward(self, x):
        return self.fc2(self.act(self.fc1(x)))


class ConvMLPStage(Module):
    def __init__(self,
                 embedding_dim,
                 activation,
                 dim_feedforward=2048,
                 stochastic_depth_rate=0.1):
        super(ConvMLPStage, self).__init__()
        self.norm1 = LayerNorm(embedding_dim)
        self.channel_mlp1 = Mlp(embedding_dim_in=embedding_dim, hidden_dim=dim_feedforward, activation=activation)
        self.norm2 = LayerNorm(embedding_dim)
        self.connect = Conv2d(embedding_dim,
                              embedding_dim,
                              kernel_size=(3, 3),
                              stride=(1, 1),
                              padding=(1, 1),
                              groups=embedding_dim,
                              bias=False)
        self.connect_norm = LayerNorm(embedding_dim)
        self.channel_mlp2 = Mlp(embedding_dim_in=embedding_dim, hidden_dim=dim_feedforward, activation=activation)
        self.drop_path = DropPath(stochastic_depth_rate) if stochastic_depth_rate > 0 else Identity()

    def forward(self, src):
        src = src + self.drop_path(self.channel_mlp1(self.norm1(src)))
        src = self.connect(self.connect_norm(src).permute(0, 3, 1, 2)).permute(0, 2, 3, 1)
        src = src + self.drop_path(self.channel_mlp2(self.norm2(src)))
        return src



class ConvDownsample(Module):
    def __init__(self, embedding_dim_in, embedding_dim_out):
        super().__init__()
        self.downsample = Conv2d(embedding_dim_in, embedding_dim_out, kernel_size=(3, 3), stride=(2, 2),
                                 padding=(1, 1))

    def forward(self, x):
        x = x.permute(0, 3, 1, 2)
        x = self.downsample(x)
        return x.permute(0, 2, 3, 1)


class BasicStage(Module):
    def __init__(self,
                 num_blocks,
                 embedding_dims,
                 activation,
                 mlp_ratio=1,
                 stochastic_depth_rate=0.1,
                 downsample=True):
        super(BasicStage, self).__init__()
        self.blocks = ModuleList()
        dpr = [x.item() for x in torch.linspace(0, stochastic_depth_rate, num_blocks)]
        for i in range(num_blocks):
            block = ConvMLPStage(activation=activation,
                                 embedding_dim=embedding_dims[0],
                                 dim_feedforward=int(embedding_dims[0] * mlp_ratio),
                                 stochastic_depth_rate=dpr[i],
                                 )
            self.blocks.append(block)

        self.downsample_mlp = ConvDownsample(embedding_dims[0], embedding_dims[1]) if downsample else Identity()

    def forward(self, x):
        for blk in self.blocks:
            x = blk(x)
        x = self.downsample_mlp(x)
        return x




class ConvTokenizer(nn.Module):
    def __init__(self, embedding_dim=64, activation=nn.ReLU):
        super(ConvTokenizer, self).__init__()
        act_fn = activation() if activation == Sigmoid else activation(inplace=True)
        self.block = nn.Sequential(
            nn.Conv2d(3,
                      embedding_dim // 2,
                      kernel_size=(3, 3),
                      stride=(2, 2),
                      padding=(1, 1),
                      bias=False),
            nn.BatchNorm2d(embedding_dim // 2),
            act_fn,
            nn.Conv2d(embedding_dim // 2,
                      embedding_dim // 2,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=(1, 1),
                      bias=False),
            nn.BatchNorm2d(embedding_dim // 2),
            act_fn,
            nn.Conv2d(embedding_dim // 2,
                      embedding_dim,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=(1, 1),
                      bias=False),
            nn.BatchNorm2d(embedding_dim),
            act_fn,
            nn.MaxPool2d(kernel_size=(3, 3),
                         stride=(2, 2),
                         padding=(1, 1),
                         dilation=(1, 1))
        )

    def forward(self, x):
        return self.block(x)
__all__ = ['ConvMLP', 'convmlp_s', 'convmlp_m', 'convmlp_l']


model_urls = {
    'convmlp_s': 'https://shi-labs.com/projects/convmlp/checkpoints/convmlp_s_imagenet.pth',
    'convmlp_m': 'https://shi-labs.com/projects/convmlp/checkpoints/convmlp_m_imagenet.pth',
    'convmlp_l': 'https://shi-labs.com/projects/convmlp/checkpoints/convmlp_l_imagenet.pth',
}


class ConvMLP(nn.Module):
    def __init__(self,
                 blocks,
                 dims,
            mlp_ratios,
            channels=64,
            n_conv_blocks=3,
            activation = nn.ReLU,
            classifier_head=True,
            num_classes=1000,
            *args, **kwargs):
    super(ConvMLP, self).__init__()
    assert len(blocks) == len(dims) == len(mlp_ratios), \
        f"blocks, dims and mlp_ratios must agree in size, {len(blocks)}, {len(dims)} and {len(mlp_ratios)} passed."

    self.activation = activation
    self.tokenizer = ConvTokenizer(embedding_dim=channels,activation=activation)
    self.conv_stages = ConvStage(n_conv_blocks,
                                 embedding_dim_in=channels,
                                 hidden_dim=dims[0],
                                 embedding_dim_out=dims[0], activation=activation)

    self.stages = nn.ModuleList()
    for i in range(0, len(blocks)):
        stage = BasicStage(num_blocks=blocks[i],
                           activation=activation,
                           embedding_dims=dims[i:i + 2],
                           mlp_ratio=mlp_ratios[i],
                           stochastic_depth_rate=0.1,
                           downsample=(i + 1 < len(blocks)))
        self.stages.append(stage)
    if classifier_head:
        self.norm = nn.LayerNorm(dims[-1])
        self.head = nn.Linear(dims[-1], num_classes)
    else:
        self.head = None
    self.apply(self.init_weight)

def forward(self, x):
    x = self.tokenizer(x)
    x = self.conv_stages(x)
    x = x.permute(0, 2, 3, 1)
```

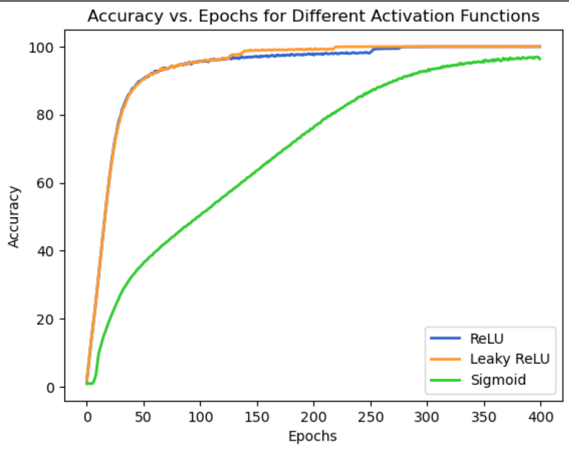
```
        for stage in self.stages:
            x = stage(x)
        if self.head is None:
            return x
        B, _, _, C = x.shape
        x = x.reshape(B, -1, C)
        x = self.norm(x)
        x = x.mean(dim=1)
        x = self.head(x)
        return x

    @staticmethod
    def init_weight(m):
        if isinstance(m, (nn.Linear, nn.Conv1d)):
            nn.init.trunc_normal_(m.weight, std=.02)
            if isinstance(m, (nn.Linear, nn.Conv1d)) and m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.LayerNorm):
            nn.init.constant_(m.bias, 0)
            nn.init.constant_(m.weight, 1.0)
        elif isinstance(m, nn.Conv2d):
            if isinstance(m, nn.Conv2d):
                #nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                #nn.init.xavier_normal_(m.weight)
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='leaky_relu')
            else:
                raise ValueError(f"No defined weight initialization for {type(m).__name__}")
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1.)
            nn.init.constant_(m.bias, 0.)


def _convmlp(arch, activation, pretrained, progress, classifier_head, blocks, dims, mlp_ratios, *args, **kwargs):
    model = ConvMLP(activation=activation, blocks=blocks, dims=dims, mlp_ratios=mlp_ratios,
                    classifier_head=classifier_head, *args, **kwargs)
    if pretrained and arch in model_urls:
        state_dict = load_state_dict_from_url(model_urls[arch],
                                              progress=progress)
        model.load_state_dict(state_dict)
    return model


def convmlp_s(pretrained=False, activation=nn.ReLU, progress=False, classifier_head=True, *args, **kwargs):
    return _convmlp('convmlp_s',activation=activation, pretrained=pretrained, progress=progress,
                    blocks=[2, 4, 2], mlp_ratios=[2, 2, 2], dims=[128, 256, 512],
                    channels=64, n_conv_blocks=2, classifier_head=classifier_head,
                    *args, **kwargs)


def convmlp_m(activation=nn.ReLU, pretrained=False, progress=False, classifier_head=True, *args, **kwargs):
    return _convmlp('convmlp_m', activation= activation, pretrained=pretrained, progress=progress,
                    blocks=[3, 6, 3], mlp_ratios=[3, 3, 3], dims=[128, 256, 512],
                    channels=64, n_conv_blocks=3, classifier_head=classifier_head,
                    *args, **kwargs)


def convmlp_l(activation=nn.ReLU, pretrained=False, progress=False, classifier_head=True, *args, **kwargs):
    return _convmlp('convmlp_l', activation=activation, pretrained=pretrained, progress=progress,
                    blocks=[4, 8, 3], mlp_ratios=[3, 3, 3], dims=[192, 384, 768],
                    channels=96, n_conv_blocks=3, classifier_head=classifier_head,
                    *args, **kwargs)

model = convmlp_s(activation=LeakyReLU).to(device)

import torch
import numpy as np
import torchvision
import torch.nn as nn
from torchvision import datasets
from torchvision import transforms
import torch.nn.functional as F
from torchsummary import summary
import pandas as pd
import pickle
import matplotlib.pyplot as plt
import numpy as np
from torch.optim import lr_scheduler
```

```
#transformer
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

num_epoch = 400
learning_rate = 0.0001 # PÅ SMALL KØRTE JEG 0.0001 LR
batch_size = 100

train_dataset = torchvision.datasets.CIFAR100(root='./data/', download=True, train=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR100(root='./data/',transform=transforms.ToTensor())
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, factor=0.1, patience=15, verbose=True)

loss_values = []
acc_values = []
test_loss_values = []
test_acc_values = []

#42,250,300
torch.manual_seed(300)
np.random.seed(300)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(300)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

for epoch in range(num_epoch):
    loss_opsamling = 0
    n_correct = 0
    n_samples = 0
    for i, (image, label) in enumerate(train_loader):
        image = image.to(device)
        label = label.to(device)
        pred = model(image)
        loss = criterion(pred,label)

        #backward

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        loss_opsamling += loss.item()

        _, predicted = torch.max(pred, 1)
        n_samples += label.size(0)
        n_correct += (predicted == label).sum().item()

    loss_opsamling = loss_opsamling/(i+1)
    acc = 100.0 * n_correct / n_samples
    loss_values.append(loss_opsamling)
    acc_values.append(acc)
    print(f'{epoch}:loss: {loss_opsamling}')
    print(f'{epoch}:acc: {acc}')
    scheduler.step(loss_opsamling)

    #test loss og acc
    with torch.no_grad():
        n_correct = 0
        n_samples = 0
        loss_opsamling_test = 0
        for b, (images, labels) in enumerate(test_loader):
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            loss = criterion(pred,label)
            loss_opsamling_test += loss.item()
            # max returns (value ,index)
            _, predicted = torch.max(outputs, 1)
            n_samples += labels.size(0)
            n_correct += (predicted == labels).sum().item()
    acc_test = 100.0 * n_correct / n_samples
    loss_opsamling_test = loss_opsamling_test/(b+1)
    test_loss_values.append(loss_opsamling_test)
    test_acc_values.append(acc_test)

with open('loss_values_leaky_relu.pkl', 'wb') as file:
    pickle.dump(loss_values, file)

with open('acc_values_leaky_relu.pkl', 'wb') as file:
    pickle.dump(acc_values, file)

with open('test_loss_values_leaky_relu.pkl', 'wb') as file:
    pickle.dump(test_loss_values, file)

with open('test_acc_values_leaky_relu.pkl', 'wb') as file:
    pickle.dump(test_acc_values, file)
```

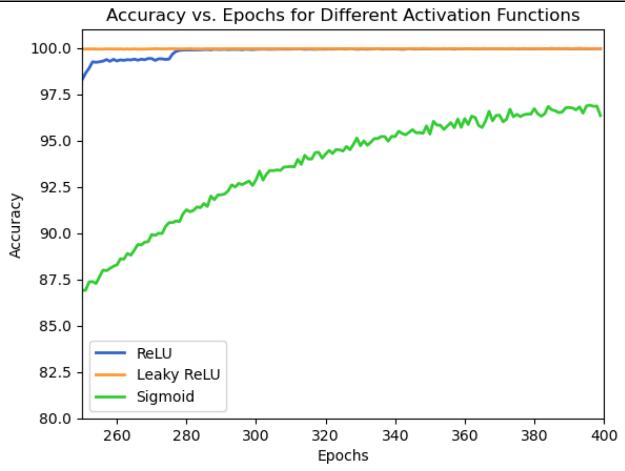
Bilag 5

TRAIN: Gennemsnit af 3 forskellige seeds: se bilag 7,8,9

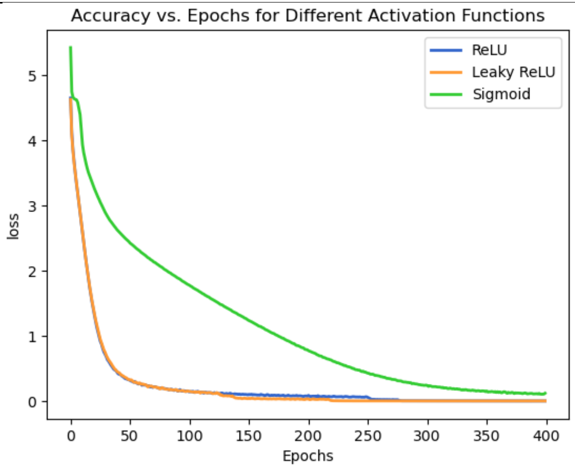
Accuracy



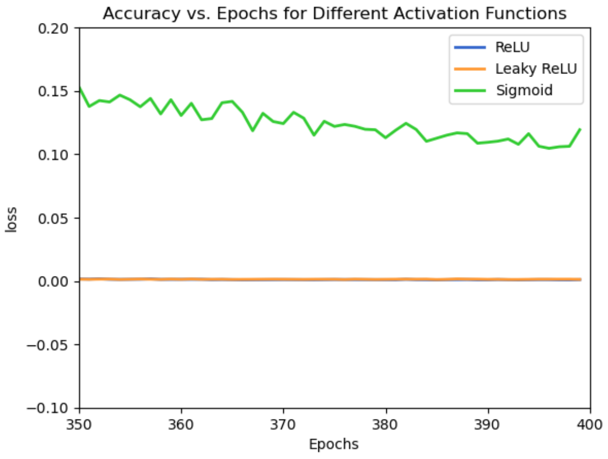
Accuracy zoomet ind



Loss

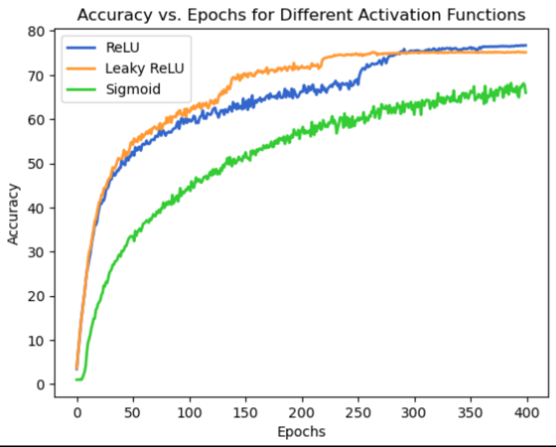
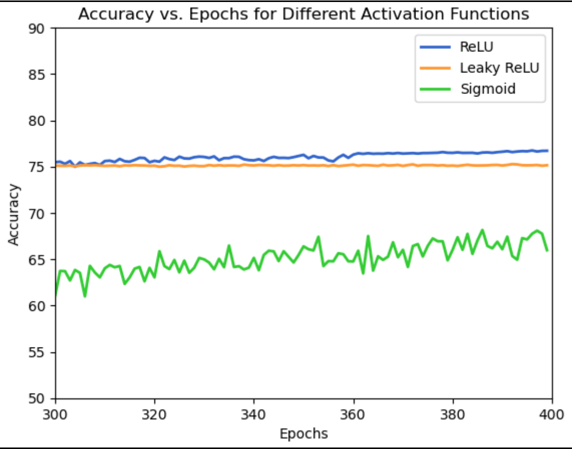
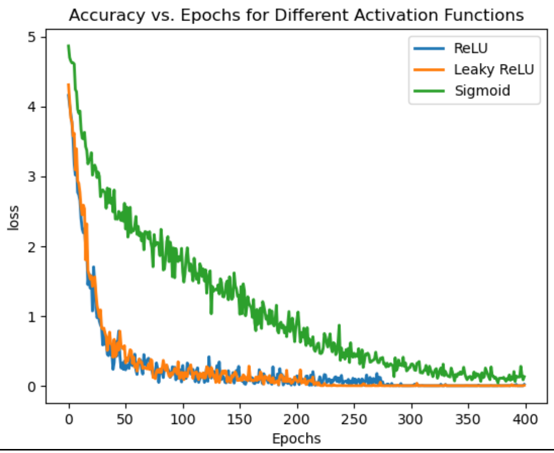
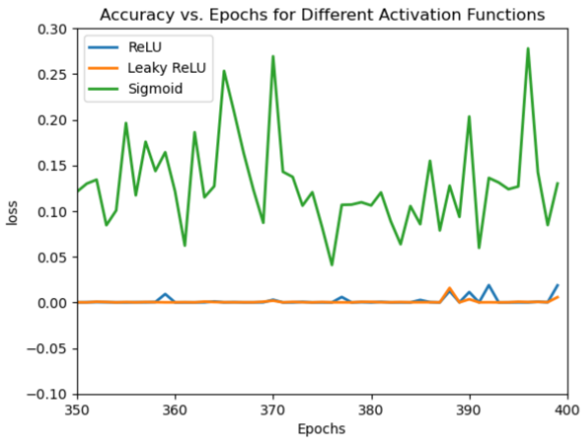


Loss zoomet ind

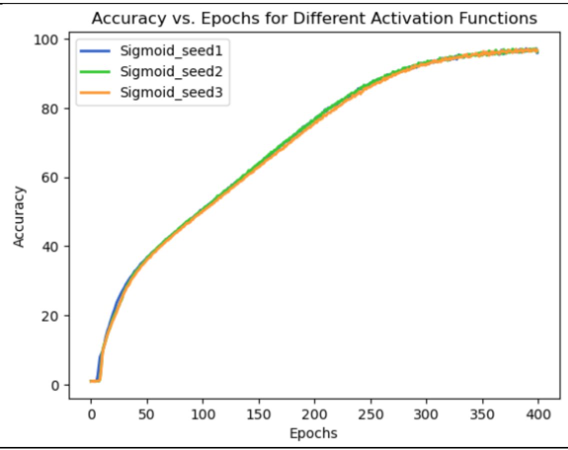
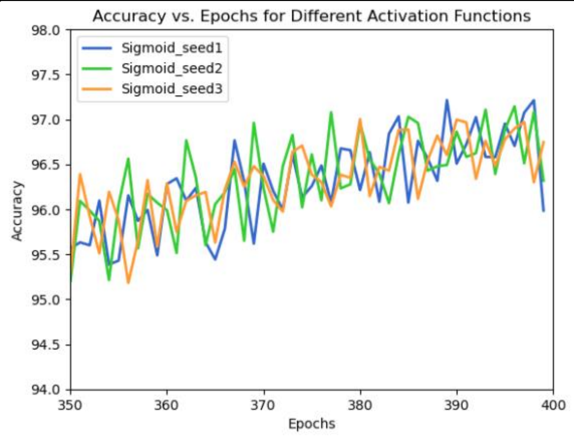
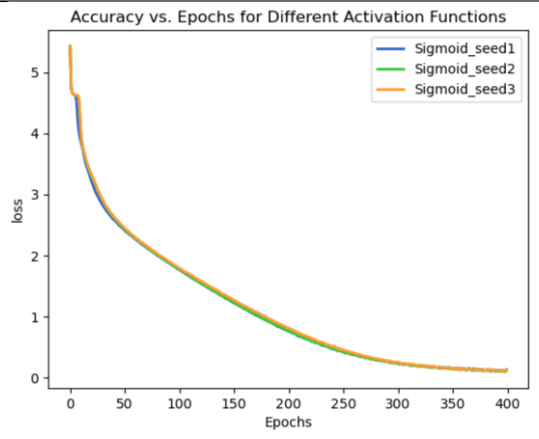
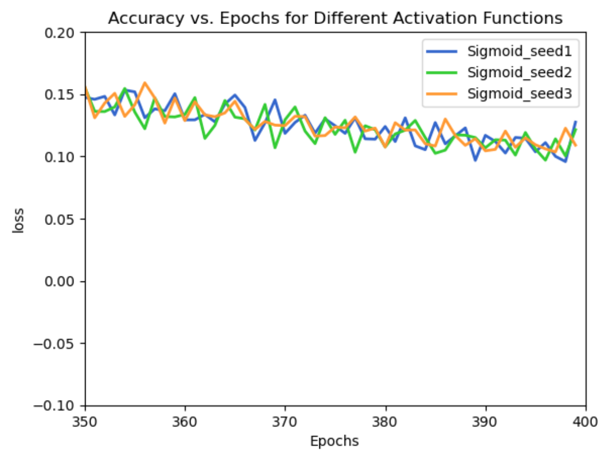


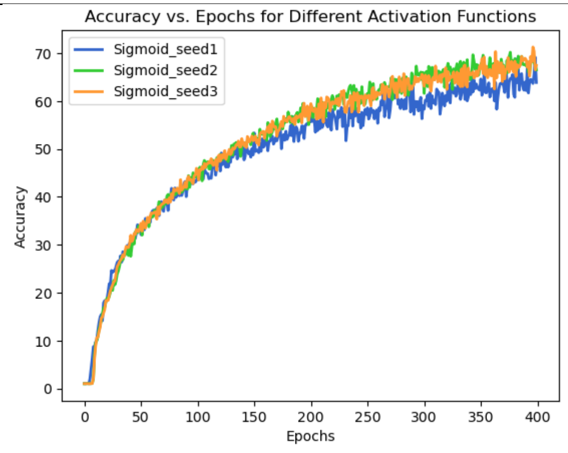
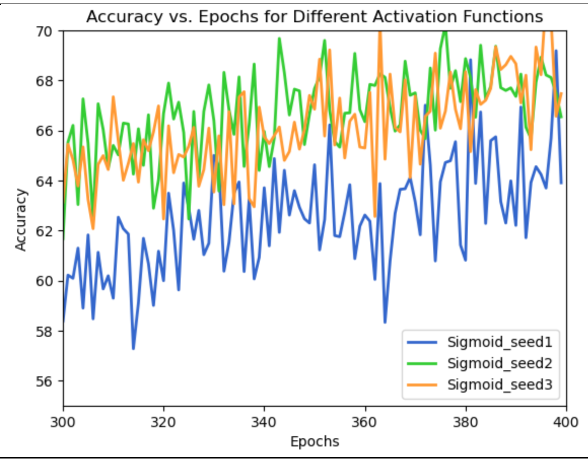
Bilag 6

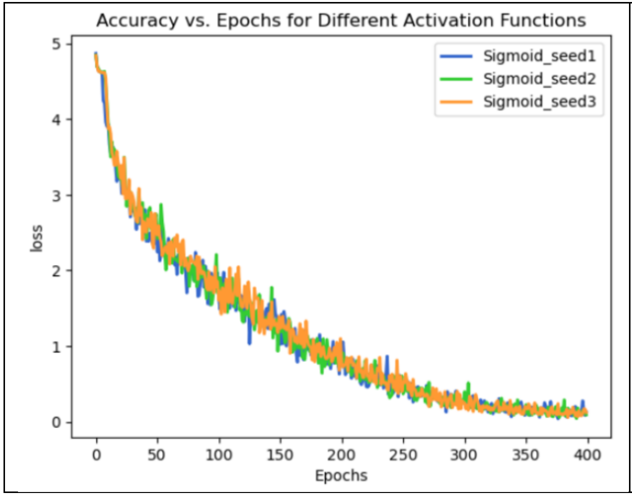
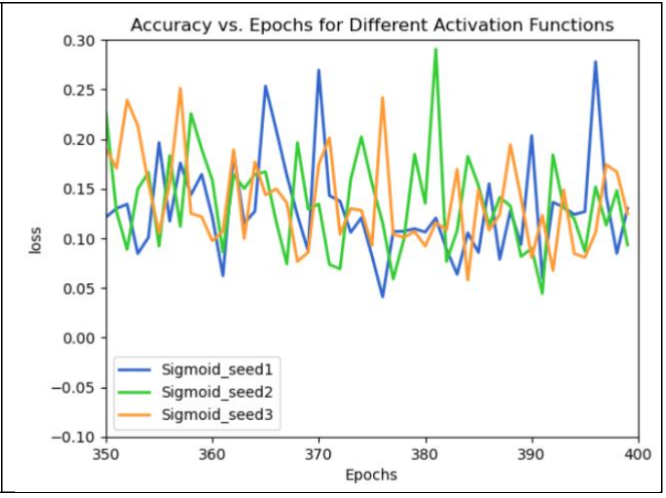
TEST: Gennemsnit af 3 forskellige seeds: se bilag 7,8,9

Accuracy	Accuracy zoomet ind
	
Loss	Loss zoomet ind
	

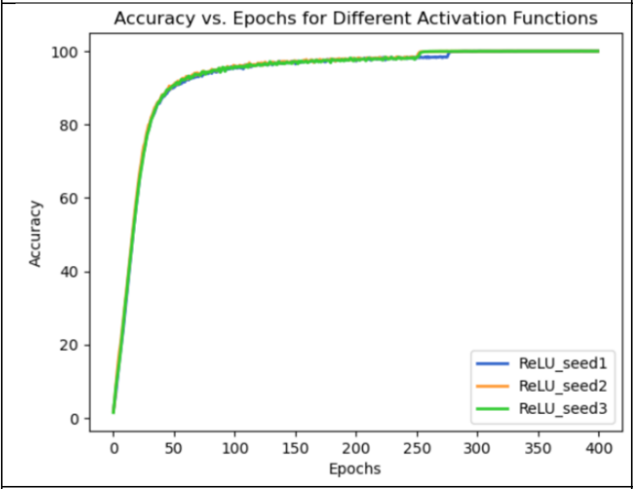
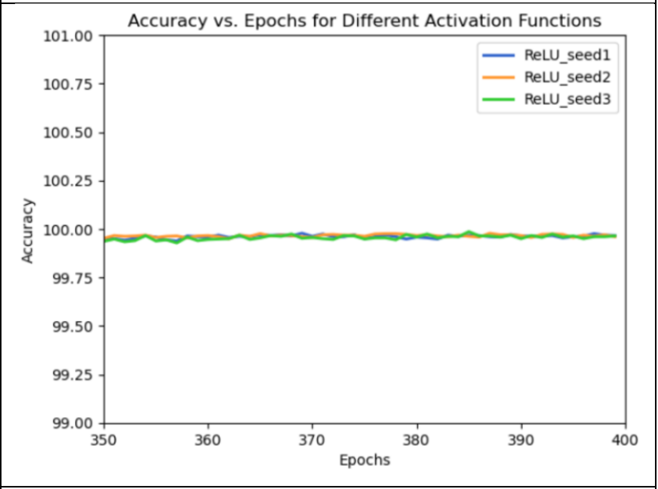
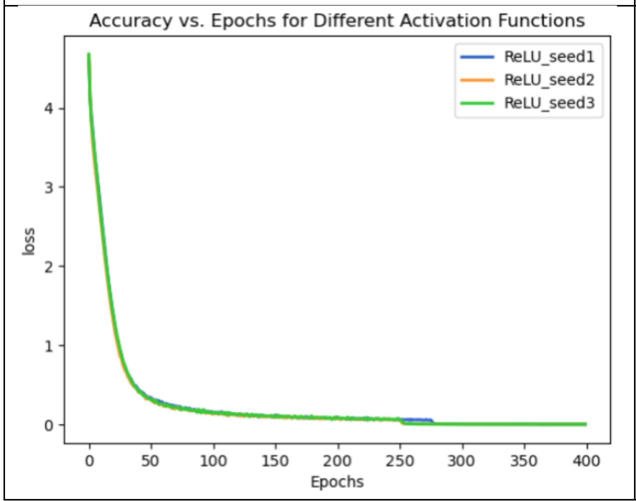
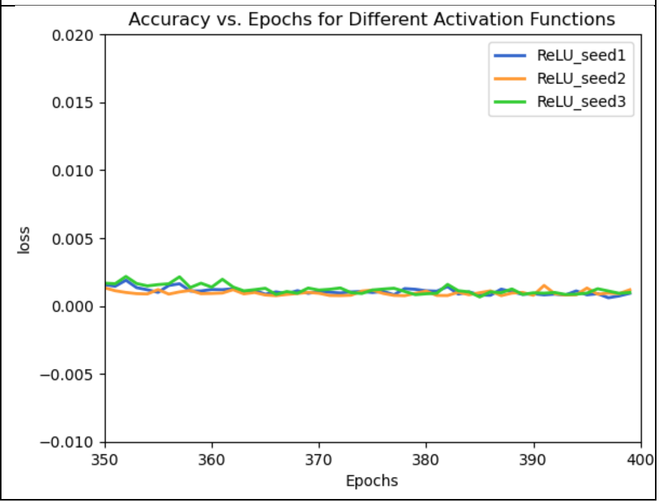
Bilag 7: Sigmoid med forskellige seeds. Både train og test

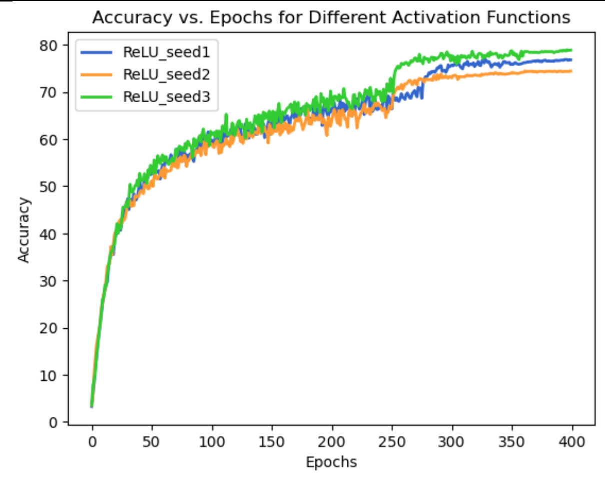
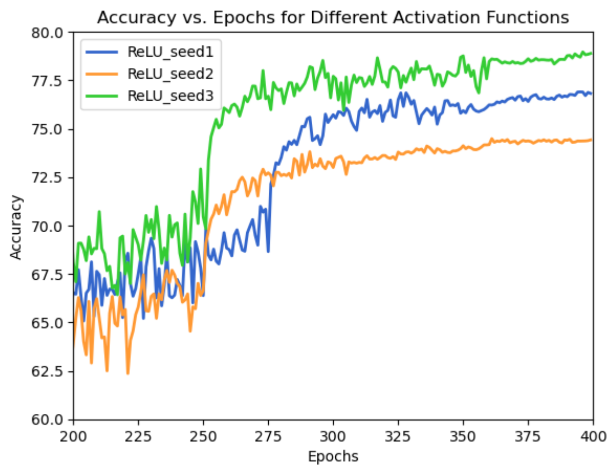
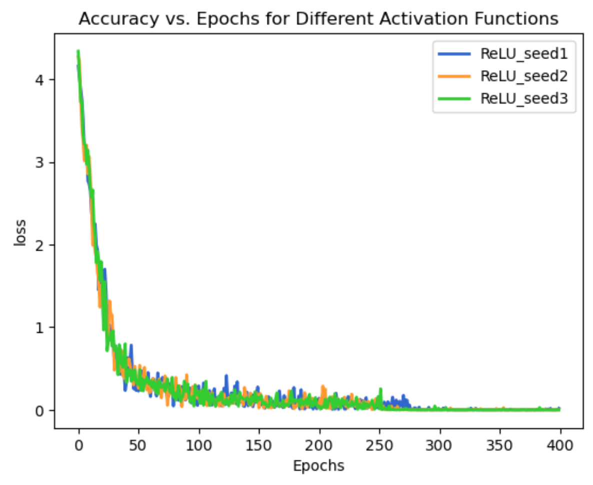
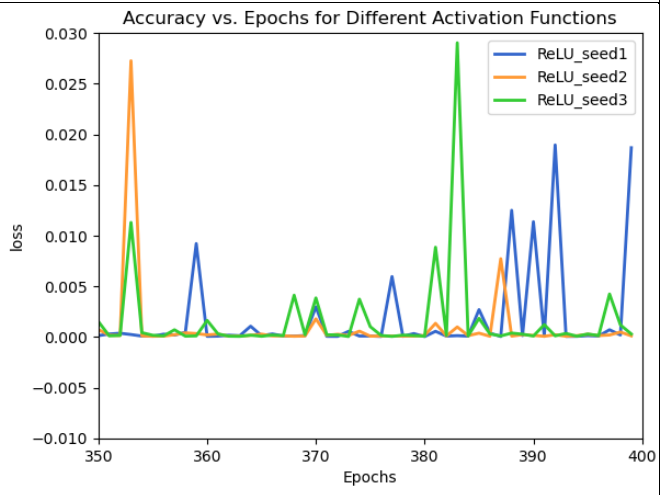
Train acc	Train acc zoomed
	
Train loss	**Train loss zoomed**
	

Test acc	Test acc zoomed
	
Test loss	**Test loss zoomed**

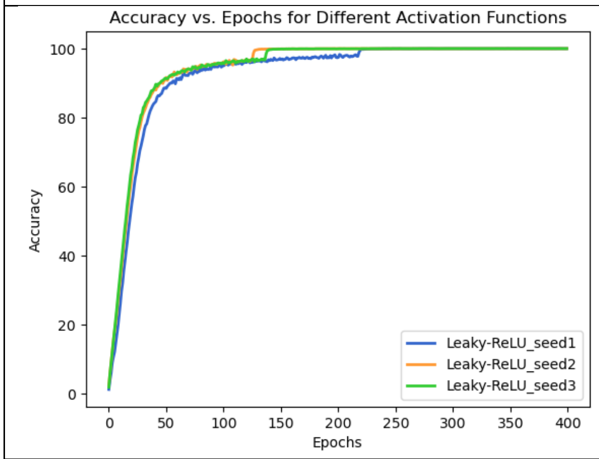
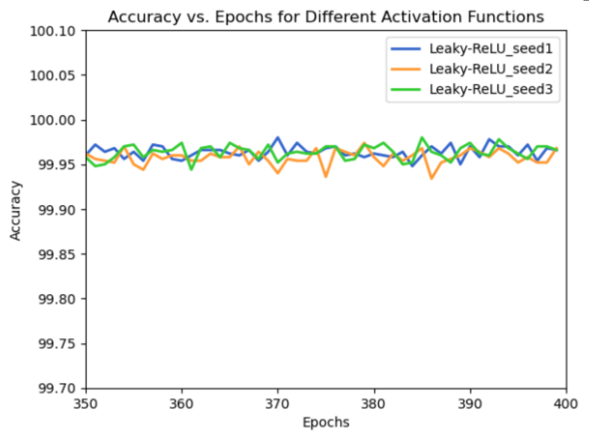
	

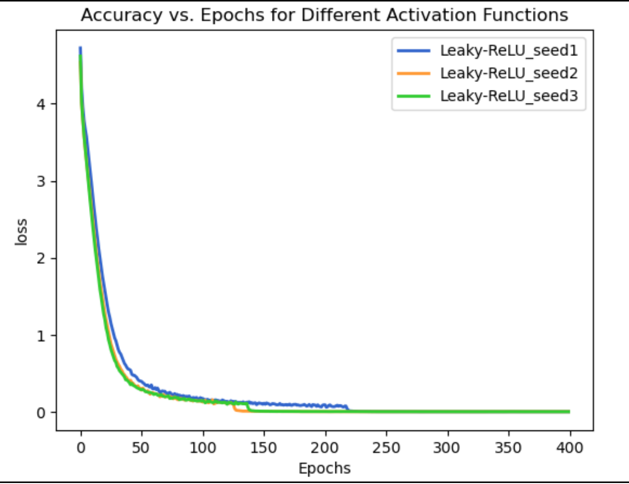
Bilag 8: ReLU forskellige seeds. Både train og test

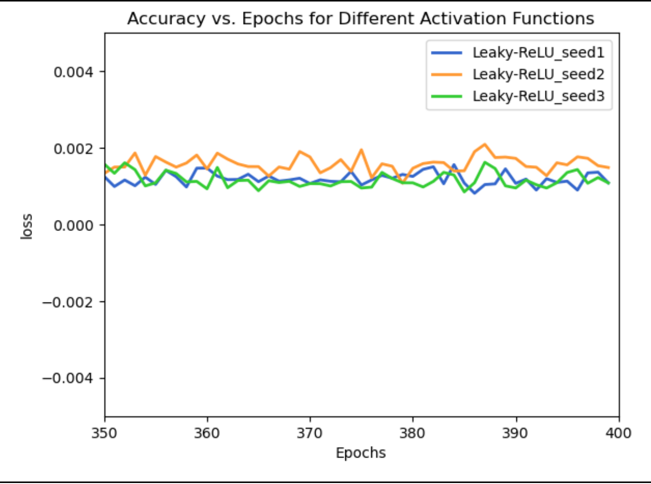
Train acc	Train acc zoomed
	
Train lost	**Train lost zoomed**
	

Test acc	Test acc zoomed
	
Test lost	**Test lost zoomed**
	

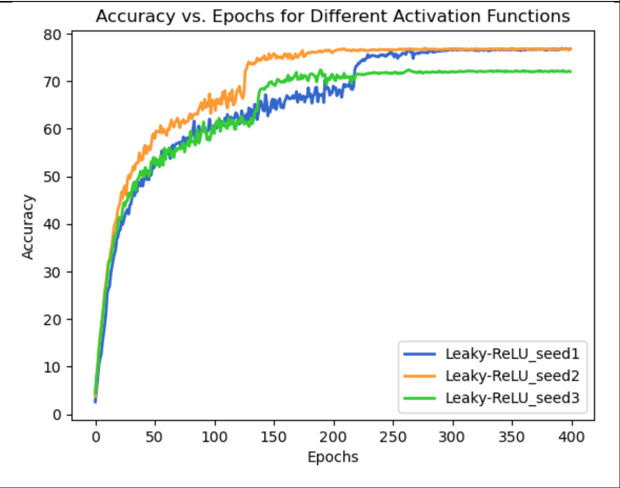
Bilag 9: Leaky-ReLU forskellige seeds. Både train og test

Train acc	Train acc zoomed
	
Train lost	**Train lost zoomed**

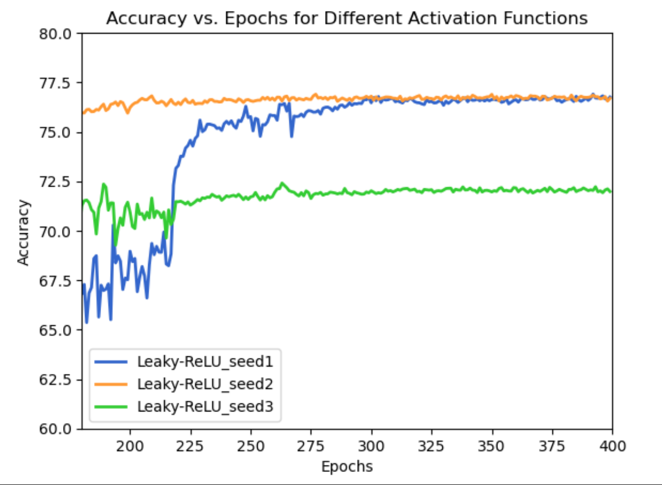




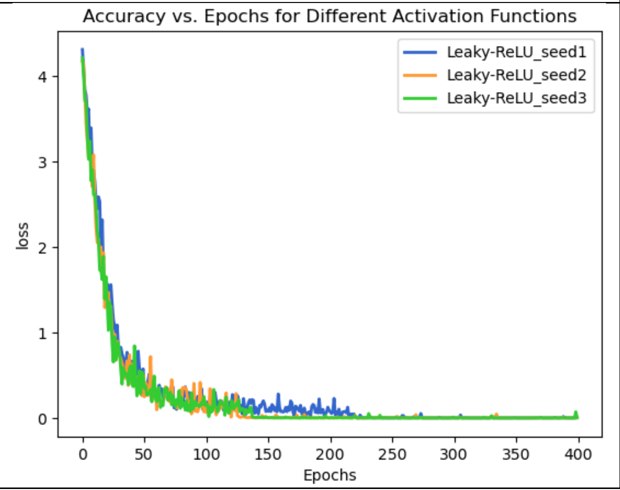
Test acc



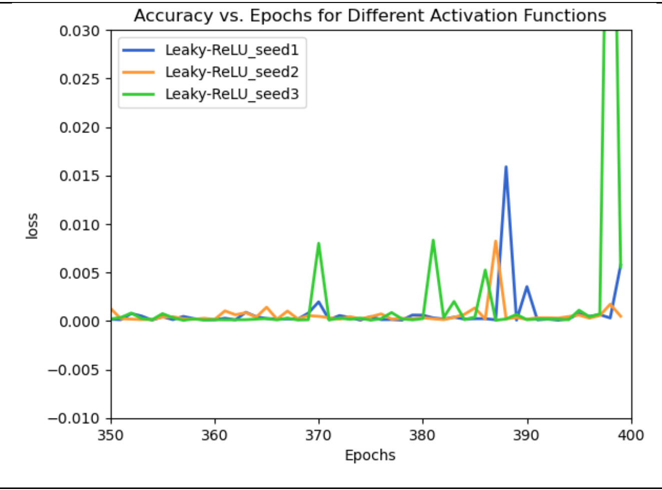
Test acc zoomed



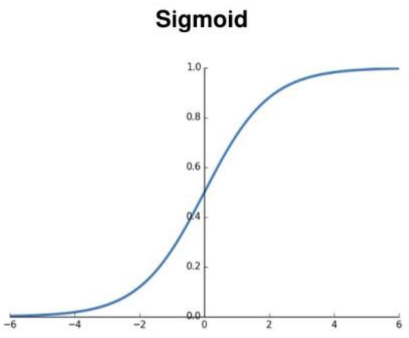
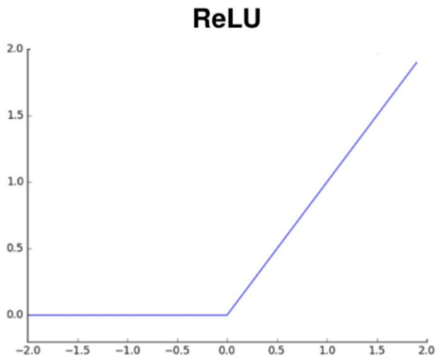
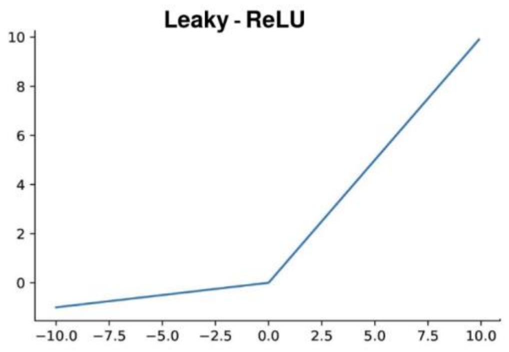
Test lost



Test lost zoomed

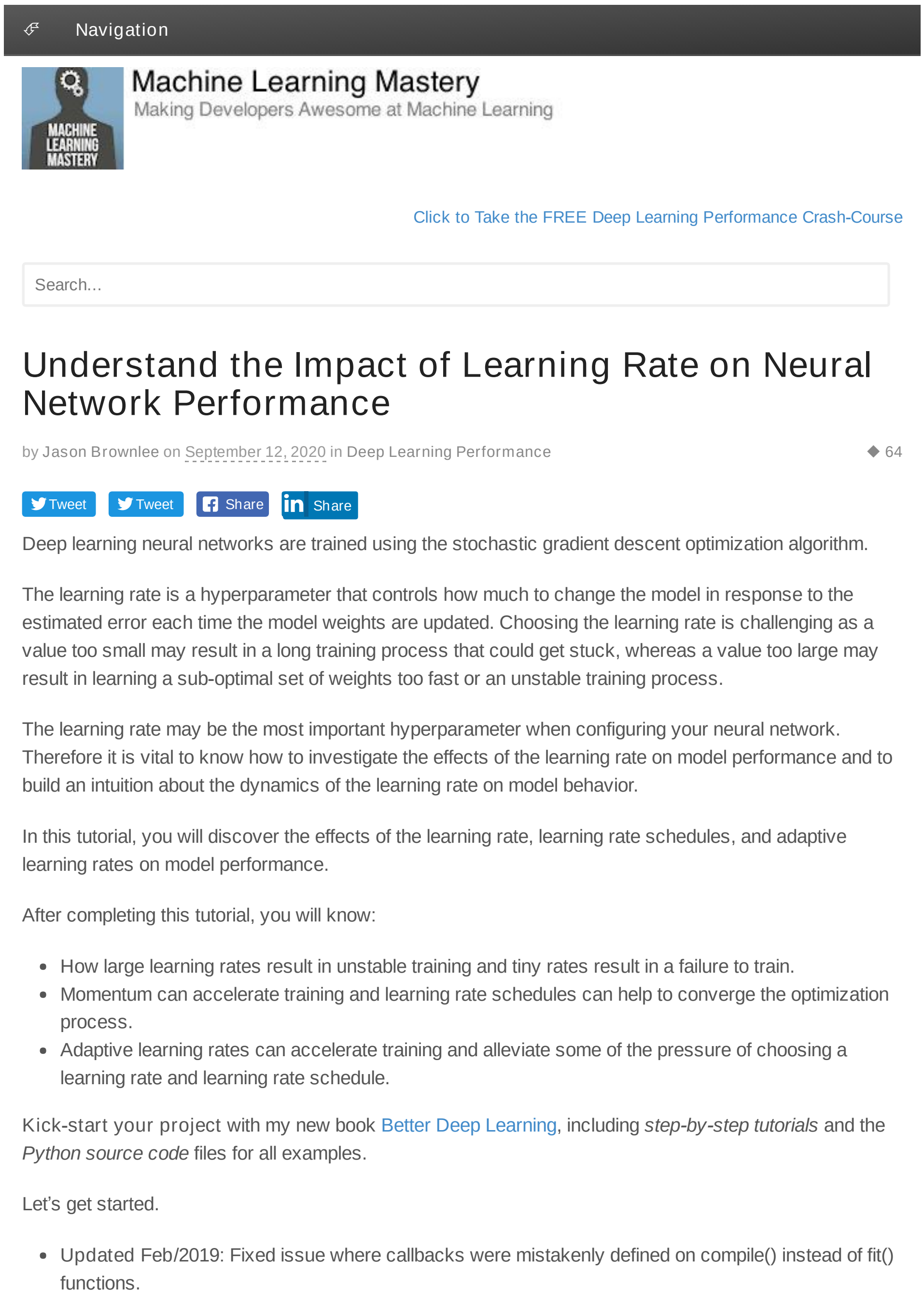


Bilag 10: Sigmoid, ReLU og Leaky-ReLU

$\boldsymbol{\sigma(z) = \frac{1}{1+e^{-z}}}$	
$ReLU(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$	
$Leaky\_Relu(z) = \begin{cases} z, & z > 0 \\ z \cdot a, & z \leq 0 \end{cases}$	

Bilagsside websides

Webside 1:
Brownlee, J. (12. september 2020). *machinelearningmastery*. Hentet fra machine learning mastery: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/


Navigation
Machine Learning Mastery
Making Developers Awesome at Machine Learning
Click to Take the FREE Deep Learning Performance Crash-Course
Search...
Tweet
Tweet
Share
Share
64

Understand the Impact of Learning Rate on Neural Network Performance

by Jason Brownlee on September 12, 2020 in Deep Learning Performance

Deep learning neural networks are trained using the stochastic gradient descent optimization algorithm.

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

The learning rate may be the most important hyperparameter when configuring your neural network. Therefore it is vital to know how to investigate the effects of the learning rate on model performance and to build an intuition about the dynamics of the learning rate on model behavior.

In this tutorial, you will discover the effects of the learning rate, learning rate schedules, and adaptive learning rates on model performance.

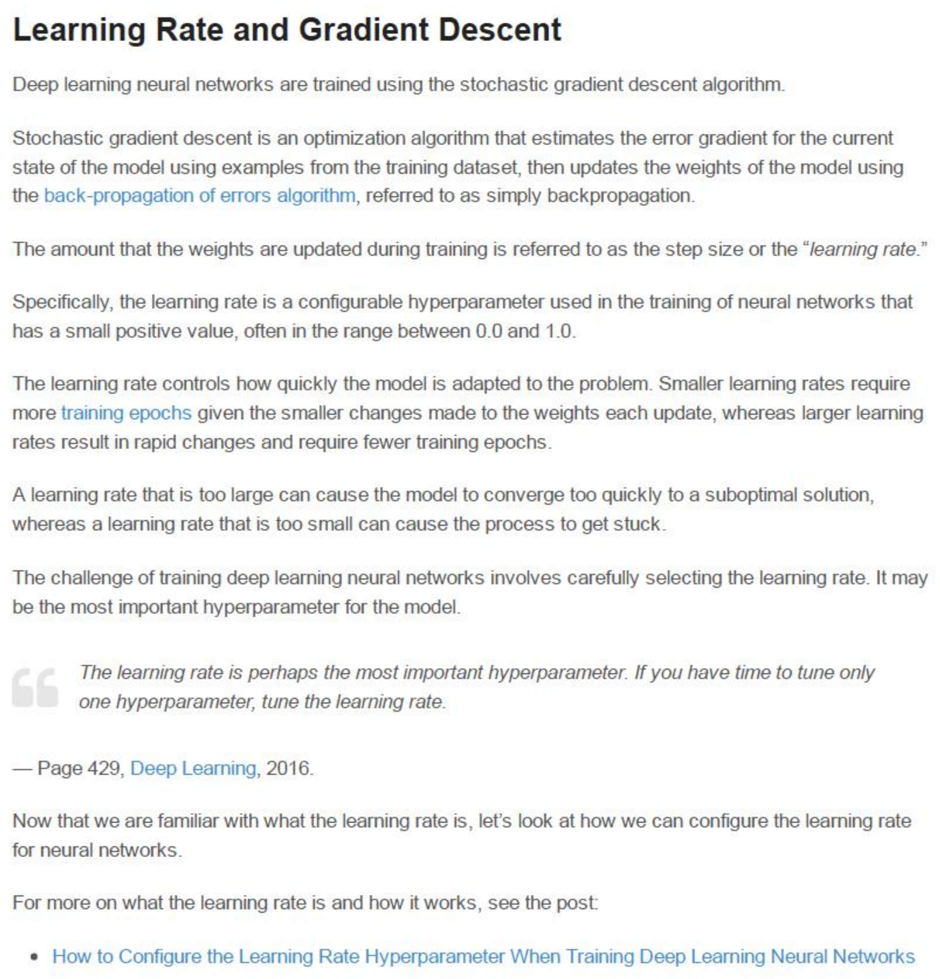
After completing this tutorial, you will know:

- How large learning rates result in unstable training and tiny rates result in a failure to train.
- Momentum can accelerate training and learning rate schedules can help to converge the optimization process.
- Adaptive learning rates can accelerate training and alleviate some of the pressure of choosing a learning rate and learning rate schedule.

Kick-start your project with my new book Better Deep Learning, including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.

- Updated Feb/2019: Fixed issue where callbacks were mistakenly defined on compile() instead of fit() functions.



Learning Rate and Gradient Descent

Deep learning neural networks are trained using the stochastic gradient descent algorithm.

Stochastic gradient descent is an optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset, then updates the weights of the model using the back-propagation of errors algorithm, referred to as simply backpropagation.

The amount that the weights are updated during training is referred to as the step size or the "*learning rate.*"

Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.

The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs.

A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck.

The challenge of training deep learning neural networks involves carefully selecting the learning rate. It may be the most important hyperparameter for the model.

> *The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate.*

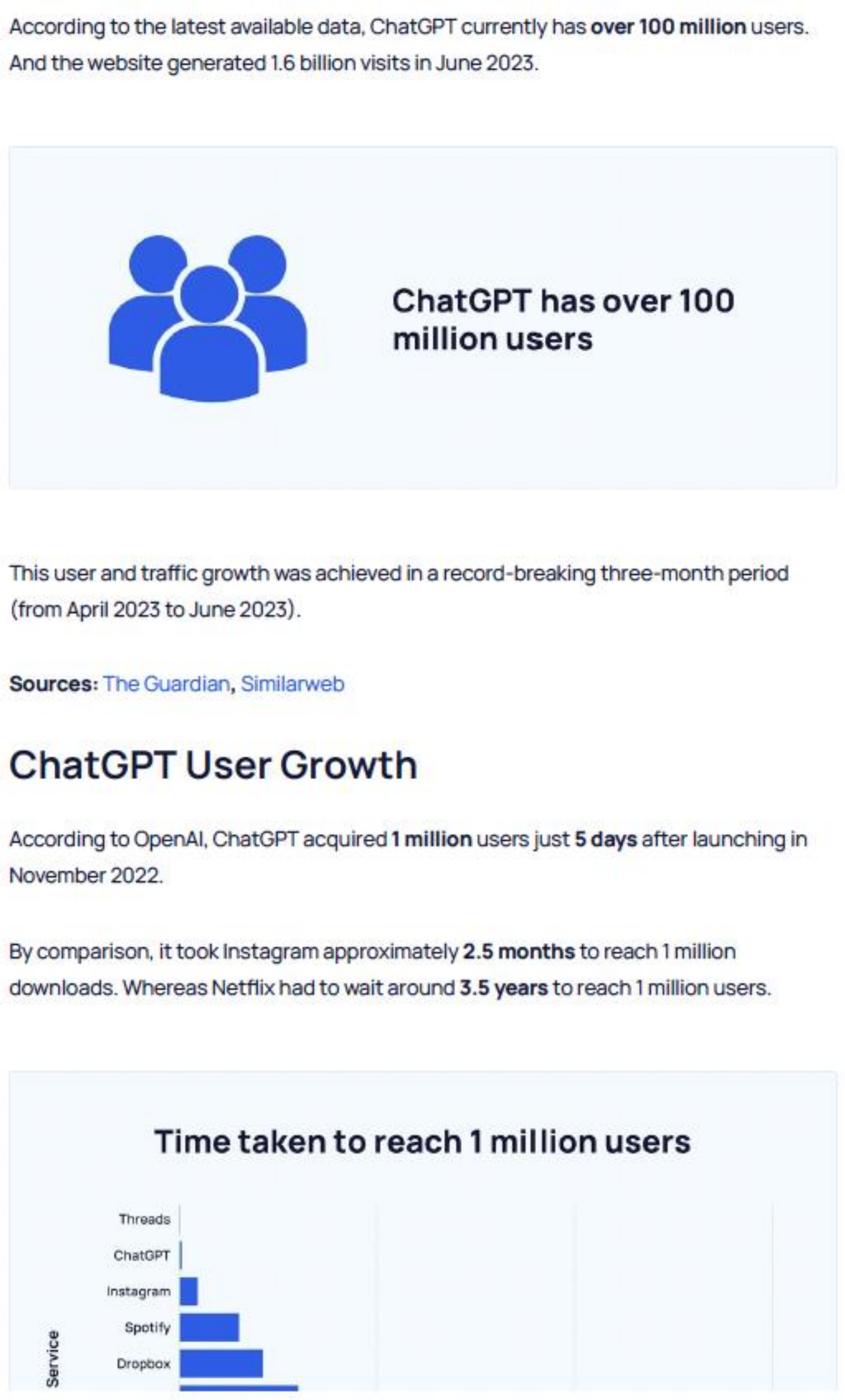
— Page 429, Deep Learning, 2016.

Now that we are familiar with what the learning rate is, let's look at how we can configure the learning rate for neural networks.

For more on what the learning rate is and how it works, see the post:

- How to Configure the Learning Rate Hyperparameter When Training Deep Learning Neural Networks

Webside 2:
Duarte, F. (13. july 2023). *explodingtopics*. Hentet fra https://explodingtopics.com/blog/chatgpt-users



According to the latest available data, ChatGPT currently has **over 100 million** users. And the website generated 1.6 billion visits in June 2023.

ChatGPT has over 100 million users

This user and traffic growth was achieved in a record-breaking three-month period (from April 2023 to June 2023).

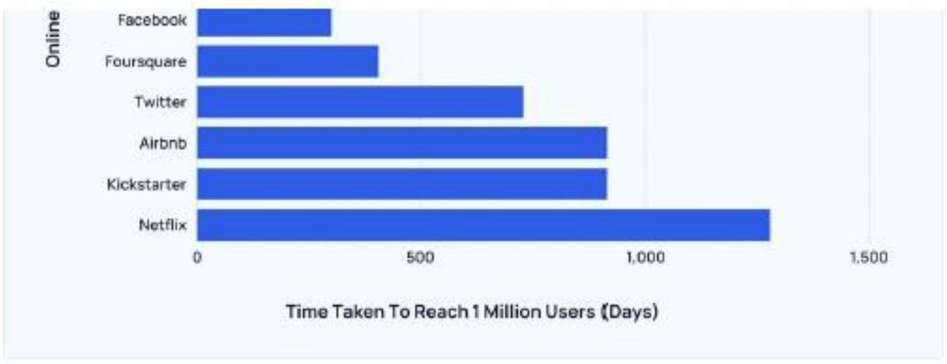
Sources: The Guardian, Similarweb

ChatGPT User Growth

According to OpenAI, ChatGPT acquired **1 million** users just **5 days** after launching in November 2022.

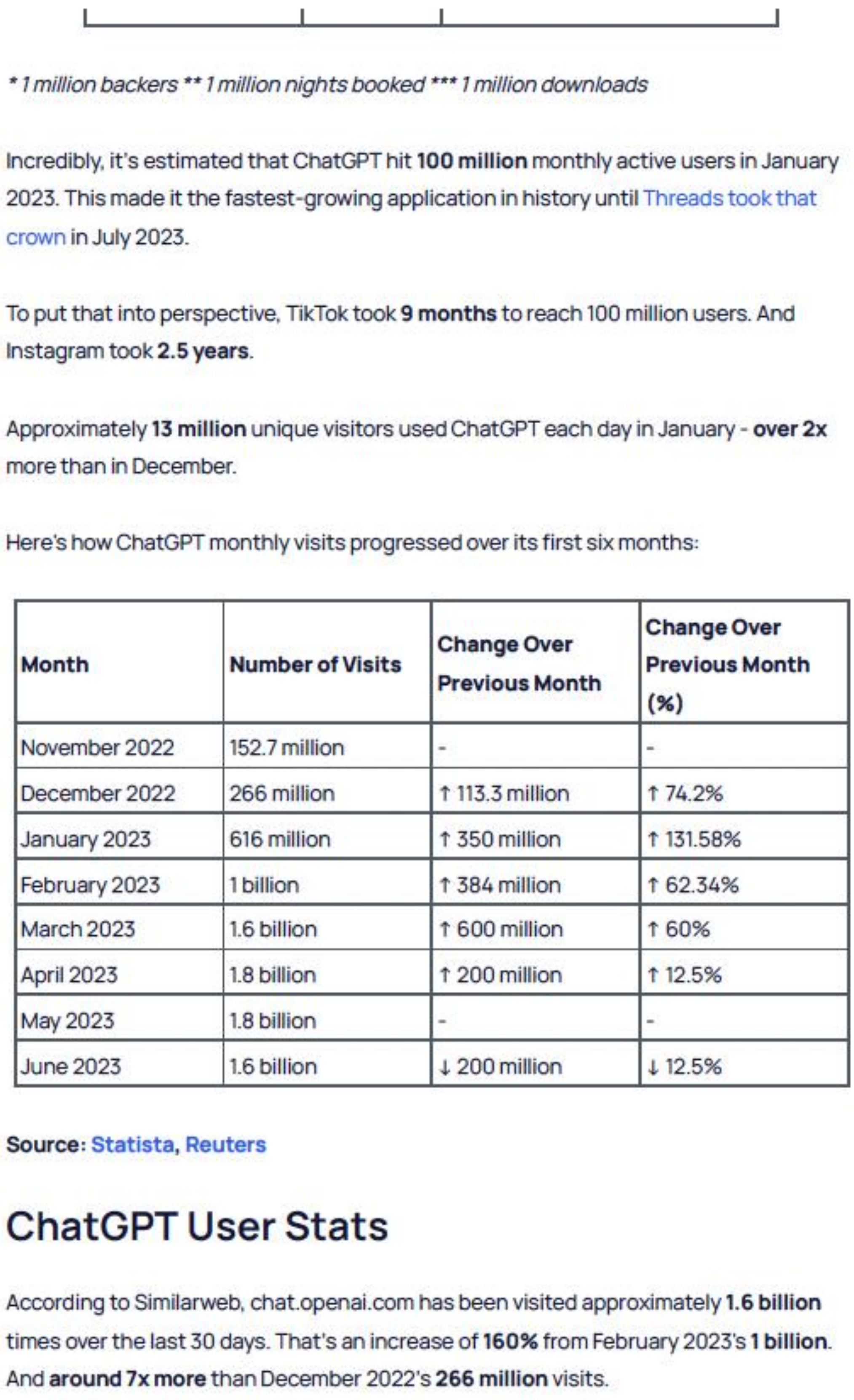
By comparison, it took Instagram approximately **2.5 months** to reach 1 million downloads. Whereas Netflix had to wait around **3.5 years** to reach 1 million users.

Time taken to reach 1 million users



Here's a breakdown of the approximate time taken to reach 1 million users for various online services:

Online Service	**Launch Year**	**Time Taken to Reach 1 Million Users**
Threads	2023	1 hour
ChatGPT	2022	5 days
Instagram***	2010	2.5 months
Spotify	2008	5 months
Dropbox	2008	7 months
Facebook	2004	10 months
Foursquare***	2009	13 months
Twitter	2006	2 years
Airbnb**	2008	2.5 years
Kickstarter*	2009	2.5 years
Netflix	1999	3.5 years



** 1 million backers ** 1 million nights booked *** 1 million downloads*

Incredibly, it's estimated that ChatGPT hit **100 million** monthly active users in January 2023. This made it the fastest-growing application in history until Threads took that crown in July 2023.

To put that into perspective, TikTok took **9 months** to reach 100 million users. And Instagram took **2.5 years**.

Approximately **13 million** unique visitors used ChatGPT each day in January - **over 2x** more than in December.

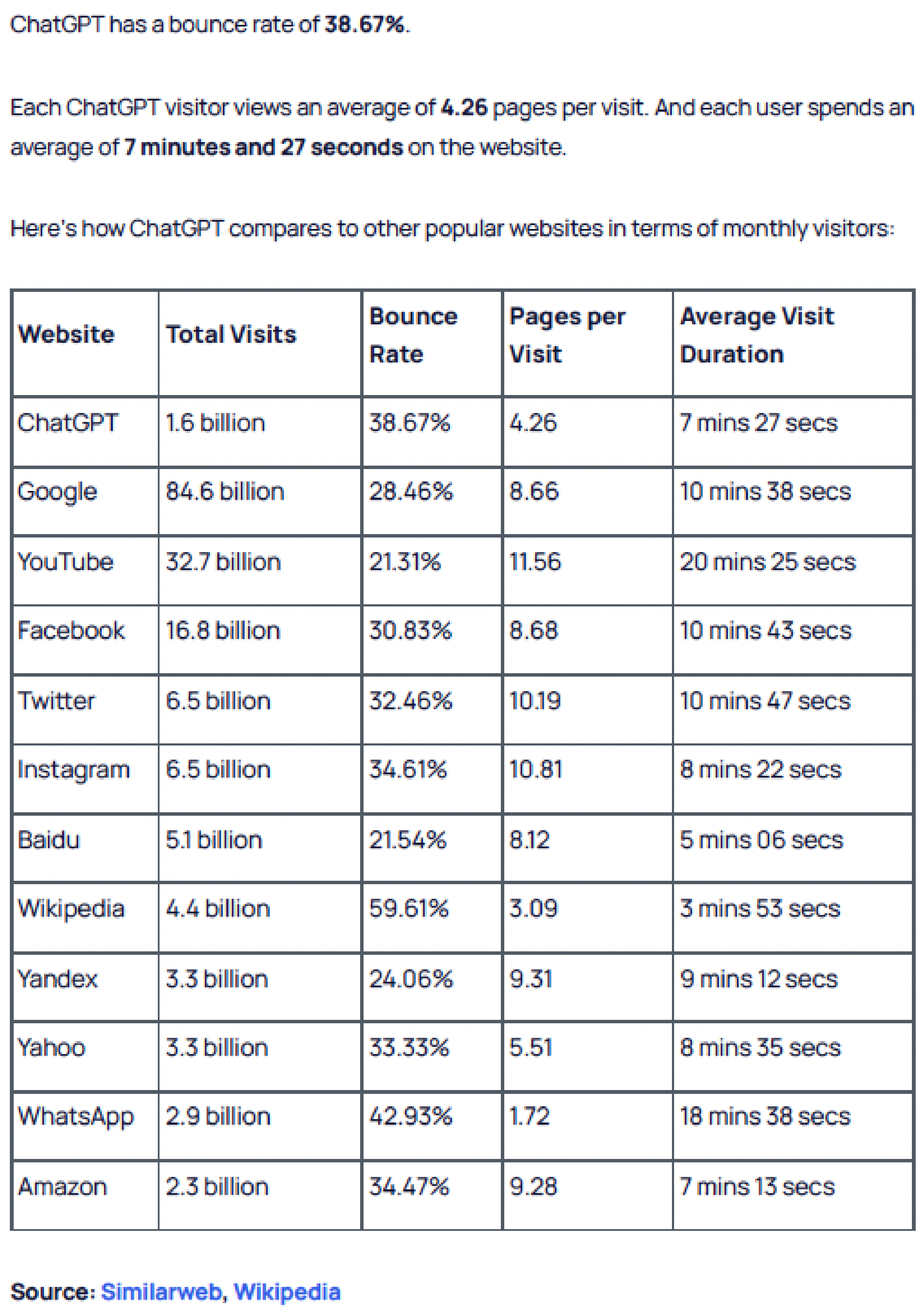
Here's how ChatGPT monthly visits progressed over its first six months:

Month	**Number of Visits**	**Change Over Previous Month**	**Change Over Previous Month (%)**
November 2022	152.7 million	-	-
December 2022	266 million	↑ 113.3 million	↑ 74.2%
January 2023	616 million	↑ 350 million	↑ 131.58%
February 2023	1 billion	↑ 384 million	↑ 62.34%
March 2023	1.6 billion	↑ 600 million	↑ 60%
April 2023	1.8 billion	↑ 200 million	↑ 12.5%
May 2023	1.8 billion	-	-
June 2023	1.6 billion	↓ 200 million	↓ 12.5%

Source: Statista, Reuters

ChatGPT User Stats

According to Similarweb, chat.openai.com has been visited approximately **1.6 billion** times over the last 30 days. That's an increase of **160%** from February 2023's **1 billion**. And **around 7x more** than December 2022's **266 million** visits.



ChatGPT has a bounce rate of **38.67%**.

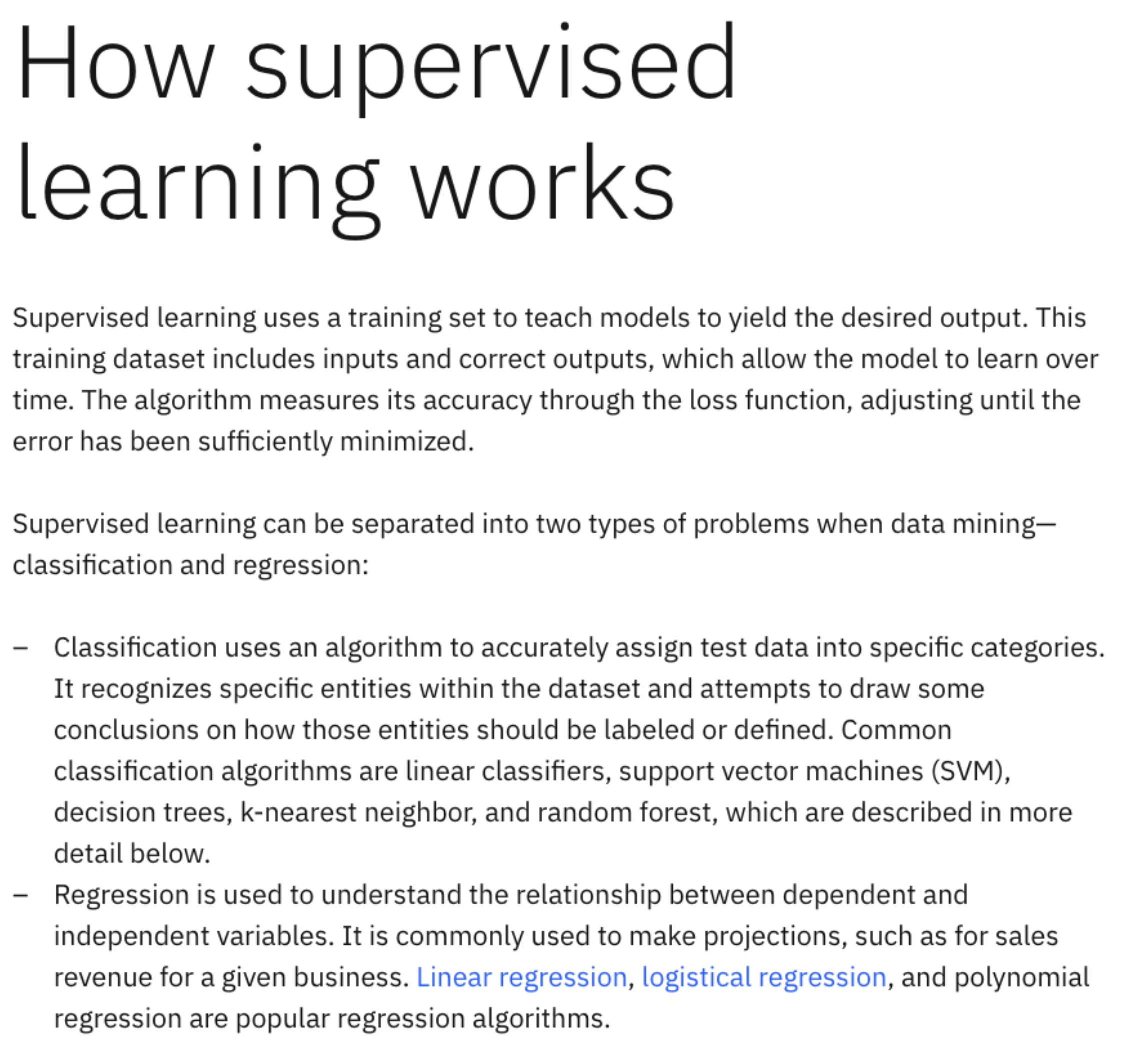
Each ChatGPT visitor views an average of **4.26** pages per visit. And each user spends an average of **7 minutes and 27 seconds** on the website.

Here's how ChatGPT compares to other popular websites in terms of monthly visitors:

Website	**Total Visits**	**Bounce Rate**	**Pages per Visit**	**Average Visit Duration**
ChatGPT	1.6 billion	38.67%	4.26	7 mins 27 secs
Google	84.6 billion	28.46%	8.66	10 mins 38 secs
YouTube	32.7 billion	21.31%	11.56	20 mins 25 secs
Facebook	16.8 billion	30.83%	8.68	10 mins 43 secs
Twitter	6.5 billion	32.46%	10.19	10 mins 47 secs
Instagram	6.5 billion	34.61%	10.81	8 mins 22 secs
Baidu	5.1 billion	21.54%	8.12	5 mins 06 secs
Wikipedia	4.4 billion	59.61%	3.09	3 mins 53 secs
Yandex	3.3 billion	24.06%	9.31	9 mins 12 secs
Yahoo	3.3 billion	33.33%	5.51	8 mins 35 secs
WhatsApp	2.9 billion	42.93%	1.72	18 mins 38 secs
Amazon	2.3 billion	34.47%	9.28	7 mins 13 secs

Source: Similarweb, Wikipedia

Webside 3:
IBM. (u.d.). *IBM*. Hentet fra https://www.ibm.com/topics/supervised-learning

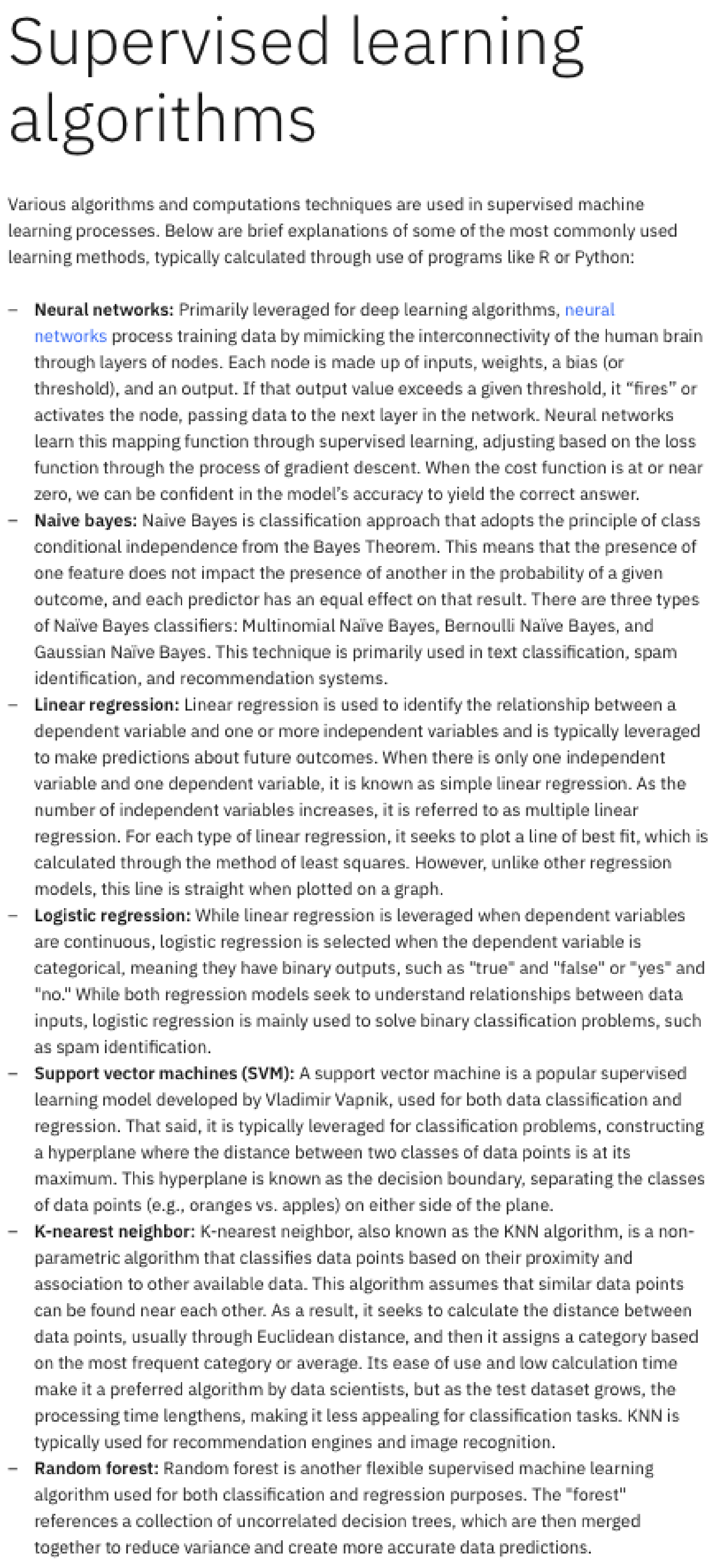


How supervised learning works

Supervised learning uses a training set to teach models to yield the desired output. This training dataset includes inputs and correct outputs, which allow the model to learn over time. The algorithm measures its accuracy through the loss function, adjusting until the error has been sufficiently minimized.

Supervised learning can be separated into two types of problems when data mining—classification and regression:

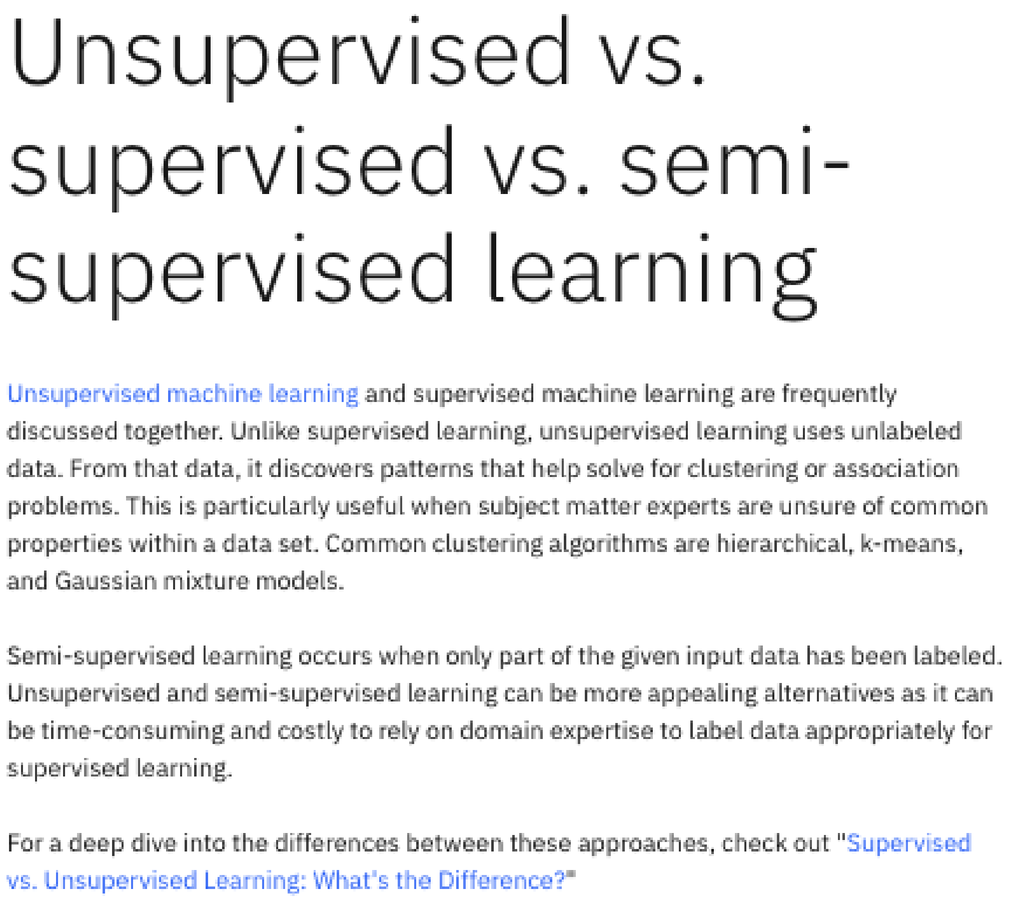
– Classification uses an algorithm to accurately assign test data into specific categories. It recognizes specific entities within the dataset and attempts to draw some conclusions on how those entities should be labeled or defined. Common classification algorithms are linear classifiers, support vector machines (SVM), decision trees, k-nearest neighbor, and random forest, which are described in more detail below.
– Regression is used to understand the relationship between dependent and independent variables. It is commonly used to make projections, such as for sales revenue for a given business. Linear regression, logistical regression, and polynomial regression are popular regression algorithms.



Supervised learning algorithms

Various algorithms and computations techniques are used in supervised machine learning processes. Below are brief explanations of some of the most commonly used learning methods, typically calculated through use of programs like R or Python:

- **Neural networks:** Primarily leveraged for deep learning algorithms, neural networks process training data by mimicking the interconnectivity of the human brain through layers of nodes. Each node is made up of inputs, weights, a bias (or threshold), and an output. If that output value exceeds a given threshold, it "fires" or activates the node, passing data to the next layer in the network. Neural networks learn this mapping function through supervised learning, adjusting based on the loss function through the process of gradient descent. When the cost function is at or near zero, we can be confident in the model's accuracy to yield the correct answer.
- **Naive bayes:** Naive Bayes is classification approach that adopts the principle of class conditional independence from the Bayes Theorem. This means that the presence of one feature does not impact the presence of another in the probability of a given outcome, and each predictor has an equal effect on that result. There are three types of Naïve Bayes classifiers: Multinomial Naïve Bayes, Bernoulli Naïve Bayes, and Gaussian Naïve Bayes. This technique is primarily used in text classification, spam identification, and recommendation systems.
- **Linear regression:** Linear regression is used to identify the relationship between a dependent variable and one or more independent variables and is typically leveraged to make predictions about future outcomes. When there is only one independent variable and one dependent variable, it is known as simple linear regression. As the number of independent variables increases, it is referred to as multiple linear regression. For each type of linear regression, it seeks to plot a line of best fit, which is calculated through the method of least squares. However, unlike other regression models, this line is straight when plotted on a graph.
- **Logistic regression:** While linear regression is leveraged when dependent variables are continuous, logistic regression is selected when the dependent variable is categorical, meaning they have binary outputs, such as "true" and "false" or "yes" and "no." While both regression models seek to understand relationships between data inputs, logistic regression is mainly used to solve binary classification problems, such as spam identification.
- **Support vector machines (SVM):** A support vector machine is a popular supervised learning model developed by Vladimir Vapnik, used for both data classification and regression. That said, it is typically leveraged for classification problems, constructing a hyperplane where the distance between two classes of data points is at its maximum. This hyperplane is known as the decision boundary, separating the classes of data points (e.g., oranges vs. apples) on either side of the plane.
- **K-nearest neighbor:** K-nearest neighbor, also known as the KNN algorithm, is a non-parametric algorithm that classifies data points based on their proximity and association to other available data. This algorithm assumes that similar data points can be found near each other. As a result, it seeks to calculate the distance between data points, usually through Euclidean distance, and then it assigns a category based on the most frequent category or average. Its ease of use and low calculation time make it a preferred algorithm by data scientists, but as the test dataset grows, the processing time lengthens, making it less appealing for classification tasks. KNN is typically used for recommendation engines and image recognition.
- **Random forest:** Random forest is another flexible supervised machine learning algorithm used for both classification and regression purposes. The "forest" references a collection of uncorrelated decision trees, which are then merged together to reduce variance and create more accurate data predictions.

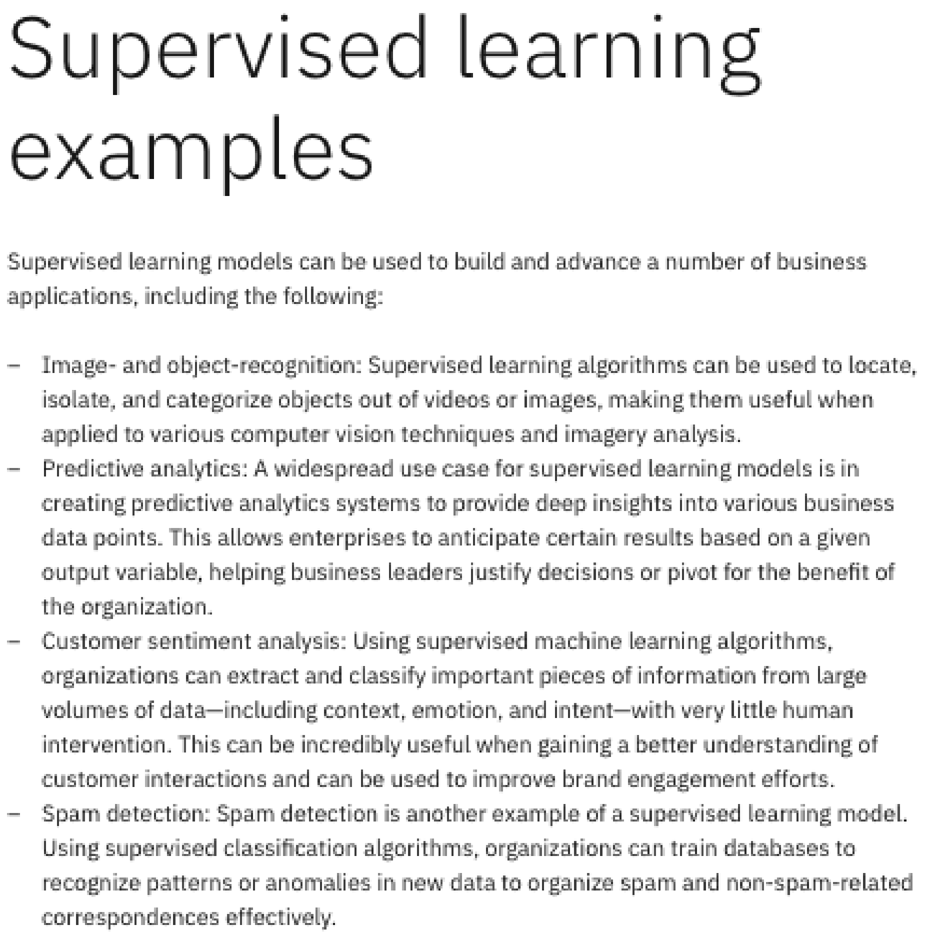


Unsupervised vs. supervised vs. semi-supervised learning

Unsupervised machine learning and supervised machine learning are frequently discussed together. Unlike supervised learning, unsupervised learning uses unlabeled data. From that data, it discovers patterns that help solve for clustering or association problems. This is particularly useful when subject matter experts are unsure of common properties within a data set. Common clustering algorithms are hierarchical, k-means, and Gaussian mixture models.

Semi-supervised learning occurs when only part of the given input data has been labeled. Unsupervised and semi-supervised learning can be more appealing alternatives as it can be time-consuming and costly to rely on domain expertise to label data appropriately for supervised learning.

For a deep dive into the differences between these approaches, check out "Supervised vs. Unsupervised Learning: What's the Difference?"



Supervised learning examples

Supervised learning models can be used to build and advance a number of business applications, including the following:

- Image- and object-recognition: Supervised learning algorithms can be used to locate, isolate, and categorize objects out of videos or images, making them useful when applied to various computer vision techniques and imagery analysis.
- Predictive analytics: A widespread use case for supervised learning models is in creating predictive analytics systems to provide deep insights into various business data points. This allows enterprises to anticipate certain results based on a given output variable, helping business leaders justify decisions or pivot for the benefit of the organization.
- Customer sentiment analysis: Using supervised machine learning algorithms, organizations can extract and classify important pieces of information from large volumes of data—including context, emotion, and intent—with very little human intervention. This can be incredibly useful when gaining a better understanding of customer interactions and can be used to improve brand engagement efforts.
- Spam detection: Spam detection is another example of a supervised learning model. Using supervised classification algorithms, organizations can train databases to recognize patterns or anomalies in new data to organize spam and non-spam-related correspondences effectively.

Webside 4:
Izmailov, P., Garipov, T., & Wilson, A. G. (2018). *Visualizing Mode Connectivity*. Hentet fra izmailovpavel.github: https://izmailovpavel.github.io/curves_blogpost/

Visualizing Mode Connectivity

Blogpost by Pavel Izmailov, Timur Garipov and Andrew Gordon Wilson; visualizations in collaboration Javier Ideami.

[Code, Paper]

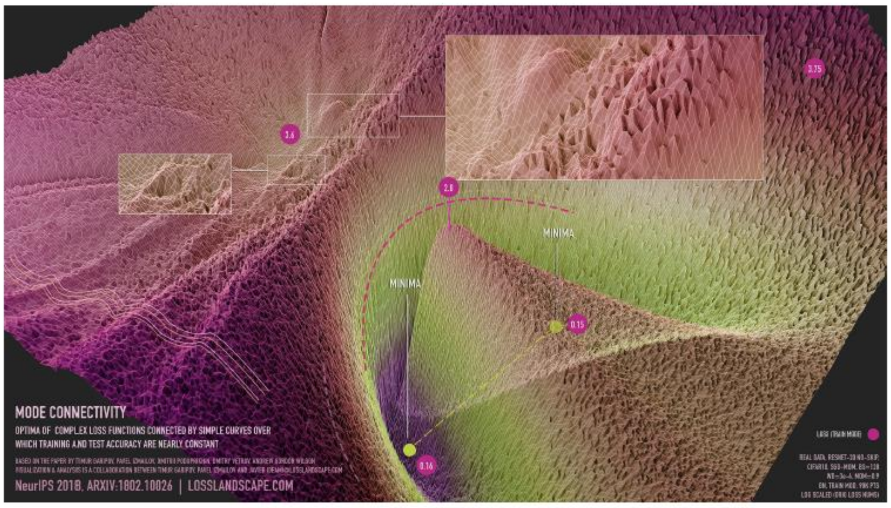

3.75
3.6
2.8
MINIMA
MINIMA
0.15
0.16
MODE CONNECTIVITY
OPTIMA OF COMPLEX LOSS FUNCTIONS CONNECTED BY SIMPLE CURVES OVER WHICH TRAINING AND TEST ACCURACY ARE NEARLY CONSTANT
NeurIPS 2018, ARXIV:1802.10026 | LOSSLANDSCAPE.COM

Figure 1: visualization of mode connectivity for ResNet-20 with no skip connections on CIFAR-10 dataset. The visualization is created in collaboration with Javier Ideami (https://losslandscape.com/).

Understanding generalization in deep neural networks is a great open question. Neural networks are trained by minimizing loss surfaces that are highly multimodal, with many settings of parameters that achieve no training loss but poor generalization. By understanding the geometric properties of these loss surfaces we can begin to resolve these questions and build more effective training procedures. Indeed, the local smoothness and convexity of loss surfaces is used for analyzing convergence of SGD and other optimizers [e. g. 12]. Recently, stochastic weight averaging [5] was proposed to find flatter regions of the loss, leading to better generalization.

The shape of the surface also has great implications for Bayesian approaches in deep learning. With a Bayesian approach, we not only want to find a single point that optimizes a risk, but rather to integrate over a loss surface to form a Bayesian model average. The geometric properties of the loss surface, rather than the specific locations of optima, therefore greatly influences the predictive distribution in a Bayesian procedure. Accordingly, recent approaches have exploited the geometry of the SGD trajectory for scalable and high performing Bayesian inference procedures [6, 7].

We still know very little about the properties of these loss surfaces. New discoveries are being made, showing topological behaviour that is highly distinct to neural networks. In this blogpost we describe mode connectivity, a surprising property of modern neural net loss landscapes presented in our NeurIPS 2018 paper. Our exposition in this post focuses on obtaining intuition through *visualization*.

Typically, the local optima of deep neural networks are imagined as isolated basins, as in the left panel of Figure 2. In this figure, we visualize the high dimensional loss surface in the plane formed from all affine combinations of three independently trained networks. In the next section we describe the details of the visualization procedure. This intuition comes from the following experiment: if we train two networks of the same architecture, we get two different local optima in the parameter space; the loss along the line segment connecting the two solutions blows up between the optima, reaching values attained by completely untrained networks at initialization. Surprisingly, the optima are actually not isolated. We can find a curved path between them, such that the loss is effectively constant along the path. We refer to this phenomenon as mode connectivity. These curved paths can be very simple, such as those shown in the middle and right panels of Figure 2. These paths are also easy to find, as we explain below.

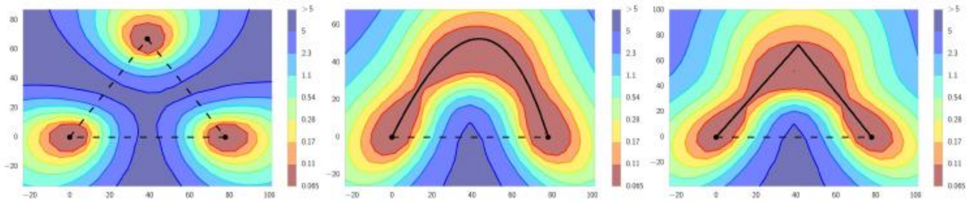


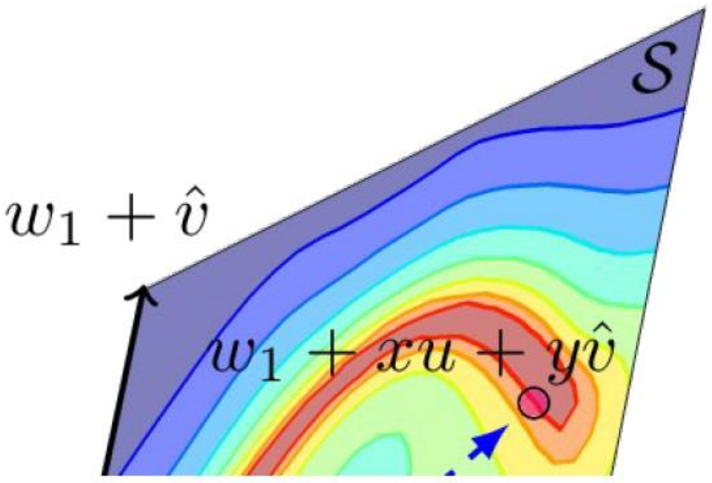
Figure 2: Loss surface of ResNet-164 on CIFAR-100. **Left**: three optima for independently trained networks; **Middle** and **Right**: A quadratic

Bezier curve, and a polygonal chain with one bend, connecting the lower two optima on the left panel along a path of near-constant loss.

Mode connectivity has been shown to hold very generally. In [1, 2] mode connectivity is demonstrated for multiple state-of-the-art image classification architectures and some recurrent architectures on text data. In [3] the authors show that it is possible to connect optima trained with different optimizers and hyper-parameters, such as batch sizes, weight decay, learning rate schedule and data augmentation strategy. In [4] the authors that mode connectivity holds for policy optimization algorithms in deep reinforcement learning.

How to Visualize Loss Surfaces?

Visualization helps us analyze and build intuition about complex objects. Visualizations based on dimensionality reduction can reveal interpretable structure, leading to new scientific insights; for example, in [11] t-SNE visualizations were used to discover new sub-types of retinal cells. Here, we study the properties of loss functions of deep neural networks, which depend on millions (or sometimes even billions!) of parameters. We cannot directly visualize a million-dimensional surface. However, we can look at a 2D slice of the loss function, and if this slice is chosen carefully, it can provide insights about the structure of the landscape.



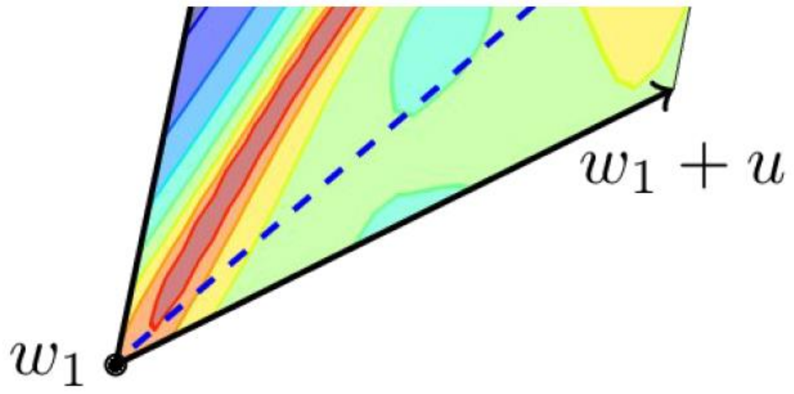


Figure 3: Illustration of the loss surface visualization procedure. We pick a 2D plane in the parameter space of a neural network, construct a coordinate system in the plane and define a grid in this coordinate system. Then, we evaluate loss for each point in the grid and visualize the result.

Three points in the parameter space always define a unique 2D plane that passes through these points. Suppose we have three points w_1, w_2, w_3 in the weight space. These can, for example, be the vectors of parameters (all weights and biases concatenated into a single vector) of three independently trained networks, as in the left panel of Figure 2. We can construct a 2-dimensional plane passing through w_1, w_2, w_3 as follows. We define a basis for the plane to be $u = (w_2 - w_1), v = (w_3 - w_1)$ and the shift vector to be w_1. We can orthogonalize the basis by switching to $v = v - cos(u, v)u$, where $cos(u, v)$ is the cosine of the angle between vectors u and v. We can then define a Cartesian coordinate system in the plane and map (x, y) coordinates to the points in the original parameter space using the formula $w(x, y) = w_1 + u \cdot x + v \cdot y$. Now we can construct a grid in the coordinate system and evaluate the loss function for the network corresponding to each of the points in the grid, and visualize the results. Figure 3 illustrates the visualization process.

Finding Paths between Modes

The method for finding a path of low loss between a pair of optima is intuitively very simple: we parameterize the path and minimize the average loss along the path with respect to its parameters. Formally, suppose we have two optima w_1 and w_2 of the loss function. We then define a path connecting them as $\phi(t)$, a mapping from the segment $[0, 1]$ to the parameter space, such that $\phi(0) = w_1, \phi(1) = w_2$. For example, we can use a quadratic Bezier curve: $\phi(t) = (1-t)^2 w_1 + 2t(1-t)\theta + t^2 w_2$, shown with a black line in the middle panel of Figure 2. Here is the parameter of the curve, which has the same dimensionality and structure as the weight vectors w_1 and w_2. We train to minimize the average loss over the curve. Specifically, we minimize

$$L(\theta) = \int_0^1 Loss(\phi(t))dt = E_{t \sim U[0,1]} Loss((t)).$$

with respect to θ, where $Loss()$ is the standard loss function used for training the networks w_1, w_2, such as cross-entropy loss for classification.

We can compute stochastic gradients of $L(\theta)$ with respect to θ efficiently. To do so, we sample a point t uniformly on $[0, 1]$, and then compute the gradient of $Loss(\phi(t))$ with respect to θ using the chain rule:

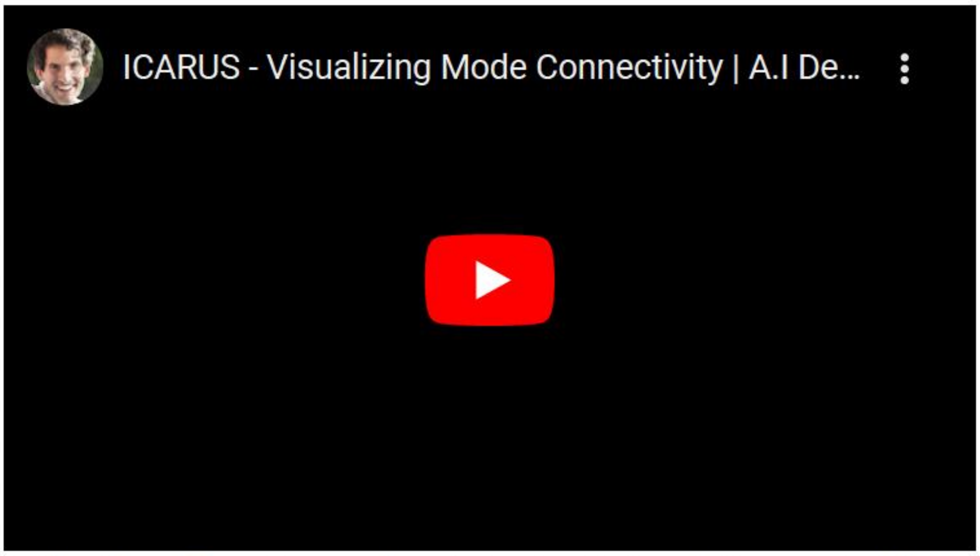
$$\frac{\partial Loss((\phi(t))}{\partial \theta} = \frac{\partial Loss(\phi(t))}{\partial \phi(t)} \frac{\partial \phi(t)}{\partial \theta}.$$

Using this stochastic gradient estimate, we minimize $L(\theta)$ with standard SGD.

For the quadratic Bezier curve, the whole path $\phi(t)$ lies in a 2-dimensional subspace of the parameter space. We can visualize the loss function in this plane throughout training, using the visualization procedure described in the previous section.

High-Resolution Visualizations of Mode Connectivity

In collaboration with Javier Ideami we have recently produced high-resolution visualizations of the loss surfaces in the planes containing mode connectivity. We created a video showing the training process of a quadratic Bezier curve connecting a pair of local optima for ResNet-20 with no skip connections on CIFAR-10:



In the video, the optima go from being isolated and disconnected for the randomly initialized curve to being connected by a tunnel of low loss, as in Figure 1.

We have also created static visualizations of ResNet-20 on the FastAI Imagenette dataset at an even higher resolution of 1000x1000. We show these visualizations in Figures 4, 5, 6.

Webside 5:
Jaadi, Z. (29. marts 2023). *builtin*. Hentet fra https://builtin.com/data-science/step-step-explanation-principal-component-analysis

DATA SCIENCE EXPERT CONTRIBUTORS

A Step-by-Step Explanation of Principal Component Analysis (PCA)

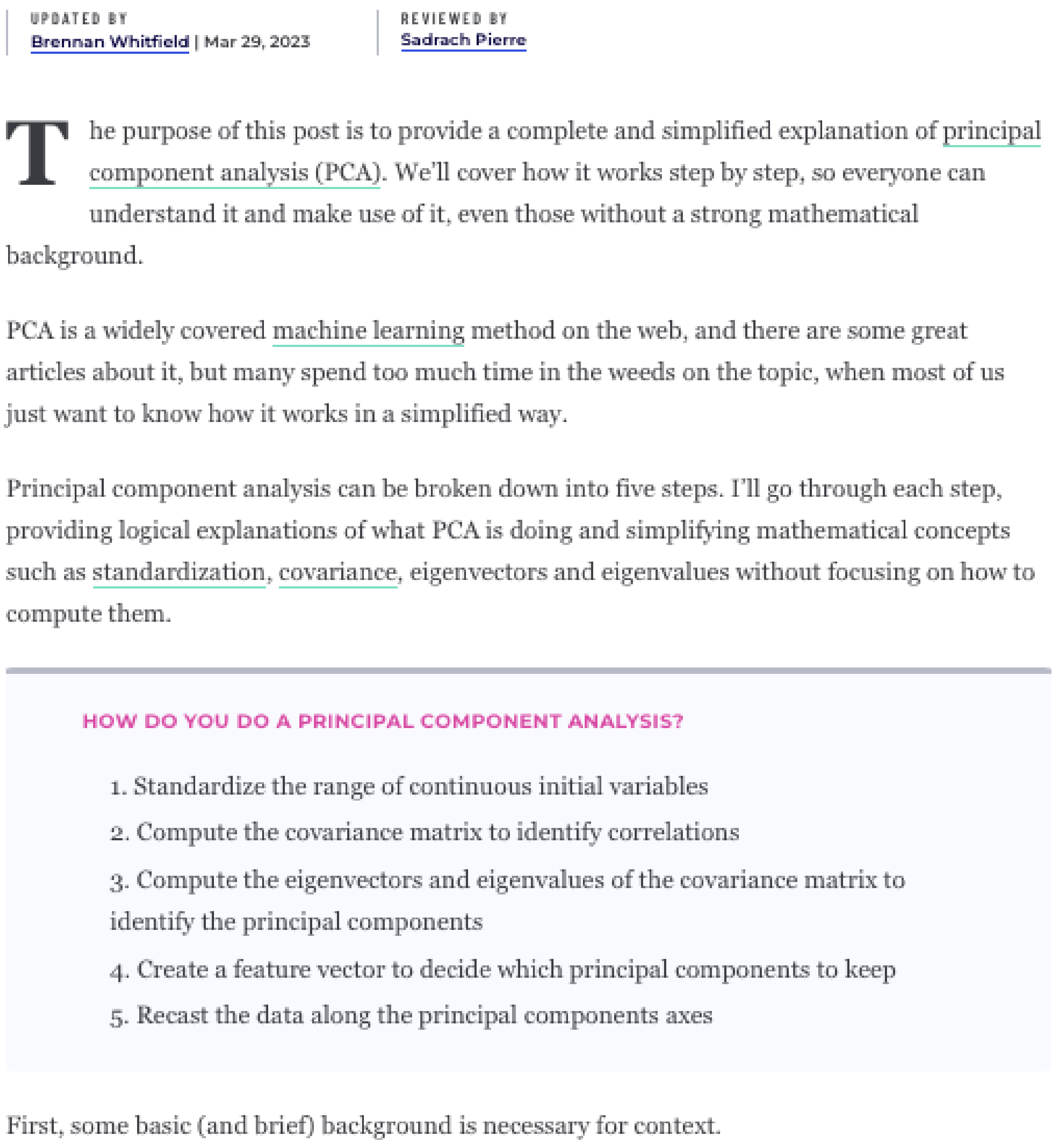
Learn how to use a PCA when working with large data sets.

Written by Zakaria Jaadi

Image: Shutterstock / Built In

UPDATED BY
Brennan Whitfield | Mar 29, 2023

REVIEWED BY
Sadrach Pierre



UPDATED BY
Brennan Whitfield | Mar 29, 2023

REVIEWED BY
Sadrach Pierre

The purpose of this post is to provide a complete and simplified explanation of principal component analysis (PCA). We'll cover how it works step by step, so everyone can understand it and make use of it, even those without a strong mathematical background.

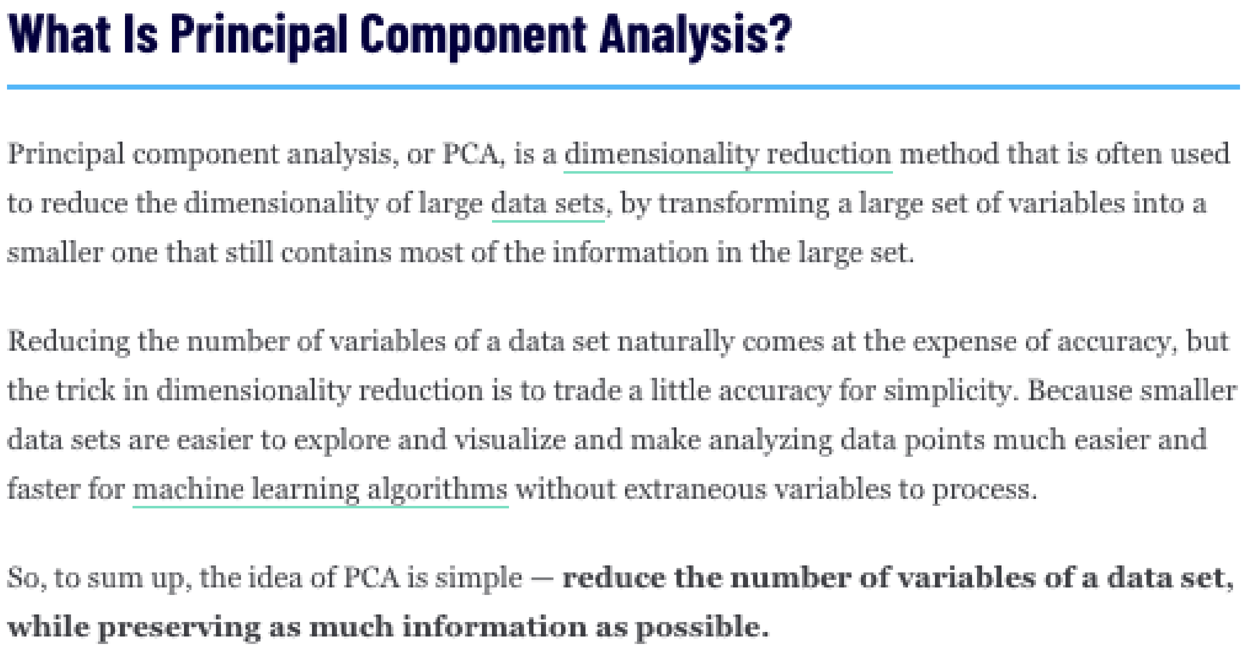
PCA is a widely covered machine learning method on the web, and there are some great articles about it, but many spend too much time in the weeds on the topic, when most of us just want to know how it works in a simplified way.

Principal component analysis can be broken down into five steps. I'll go through each step, providing logical explanations of what PCA is doing and simplifying mathematical concepts such as standardization, covariance, eigenvectors and eigenvalues without focusing on how to compute them.

HOW DO YOU DO A PRINCIPAL COMPONENT ANALYSIS?

1. Standardize the range of continuous initial variables
2. Compute the covariance matrix to identify correlations
3. Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components
4. Create a feature vector to decide which principal components to keep
5. Recast the data along the principal components axes

First, some basic (and brief) background is necessary for context.

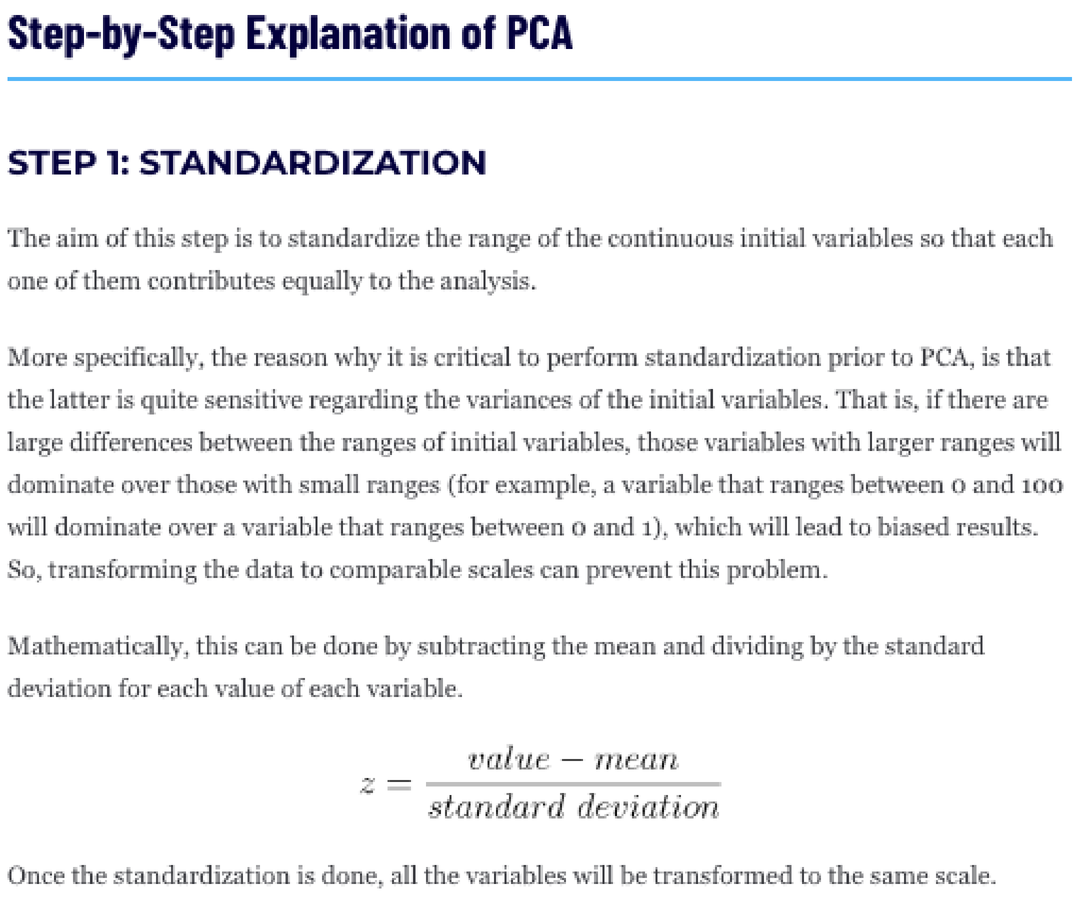


What Is Principal Component Analysis?

Principal component analysis, or PCA, is a dimensionality reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set.

Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity. Because smaller data sets are easier to explore and visualize and make analyzing data points much easier and faster for machine learning algorithms without extraneous variables to process.

So, to sum up, the idea of PCA is simple — **reduce the number of variables of a data set, while preserving as much information as possible.**



Step-by-Step Explanation of PCA

STEP 1: STANDARDIZATION

The aim of this step is to standardize the range of the continuous initial variables so that each one of them contributes equally to the analysis.

More specifically, the reason why it is critical to perform standardization prior to PCA, is that the latter is quite sensitive regarding the variances of the initial variables. That is, if there are large differences between the ranges of initial variables, those variables with larger ranges will dominate over those with small ranges (for example, a variable that ranges between 0 and 100 will dominate over a variable that ranges between 0 and 1), which will lead to biased results. So, transforming the data to comparable scales can prevent this problem.

Mathematically, this can be done by subtracting the mean and dividing by the standard deviation for each value of each variable.

$$z = \frac{value - mean}{standard\ deviation}$$

Once the standardization is done, all the variables will be transformed to the same scale.

STEP 2: COVARIANCE MATRIX COMPUTATION

The aim of this step is to understand how the variables of the input data set are varying from the mean with respect to each other, or in other words, to see if there is any relationship between them. Because sometimes, variables are highly correlated in such a way that they contain redundant information. So, in order to identify these correlations, we compute the covariance matrix.

The covariance matrix is a $p \times p$ symmetric matrix (where p is the number of dimensions) that has as entries the covariances associated with all possible pairs of the initial variables. For example, for a 3-dimensional data set with 3 variables x, y, and z, the covariance matrix is a 3×3 data matrix of this from:

$$\begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{bmatrix}$$

Covariance Matrix for 3-Dimensional Data.

Since the covariance of a variable with itself is its variance (Cov(a,a)=Var(a)), in the main diagonal (Top left to bottom right) we actually have the variances of each initial variable. And since the covariance is commutative (Cov(a,b)=Cov(b,a)), the entries of the covariance matrix are symmetric with respect to the main diagonal, which means that the upper and the lower triangular portions are equal.

What do the covariances that we have as entries of the matrix tell us about the correlations between the variables?

It's actually the sign of the covariance that matters:

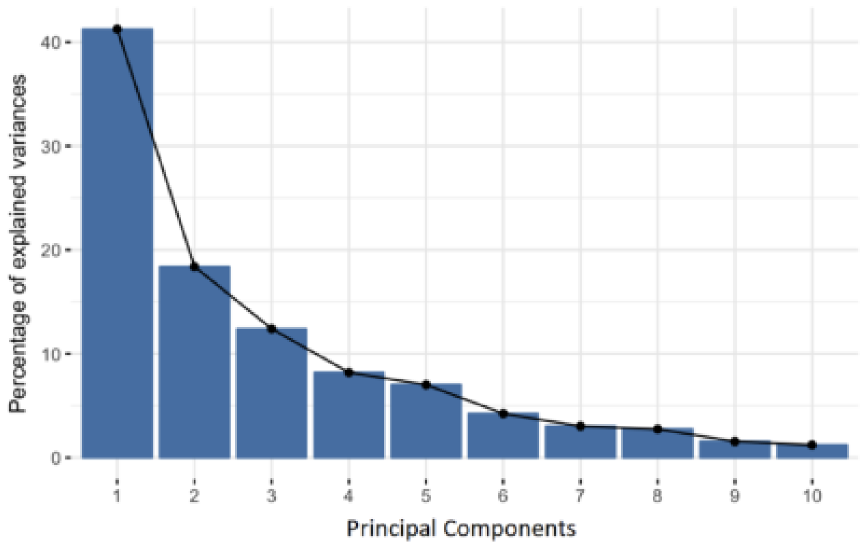
- If positive then: the two variables increase or decrease together (correlated)
- If negative then: one increases when the other decreases (Inversely correlated)

Now that we know that the covariance matrix is not more than a table that summarizes the correlations between all the possible pairs of variables, let's move to the next step.

STEP 3: COMPUTE THE EIGENVECTORS AND EIGENVALUES OF THE COVARIANCE MATRIX TO IDENTIFY THE PRINCIPAL COMPONENTS

Eigenvectors and eigenvalues are the linear algebra concepts that we need to compute from the covariance matrix in order to determine the ***principal components*** of the data. Before getting to the explanation of these concepts, let's first understand what do we mean by principal components.

Principal components are new variables that are constructed as linear combinations or mixtures of the initial variables. These combinations are done in such a way that the new variables (i.e., principal components) are uncorrelated and most of the information within the initial variables is squeezed or compressed into the first components. So, the idea is 10-dimensional data gives you 10 principal components, but PCA tries to put maximum possible information in the first component, then maximum remaining information in the second and so on, until having something like shown in the scree plot below.



Percentage of Variance (Information) for each by PC.

Organizing information in principal components this way, will allow you to reduce dimensionality without losing much information, and this by discarding the components with low information and considering the remaining components as your new variables.

An important thing to realize here is that the principal components are less interpretable and don't have any real meaning since they are constructed as linear combinations of the initial variables.

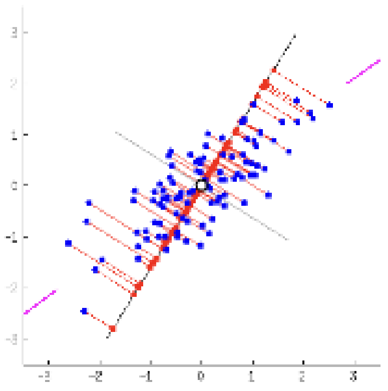
Geometrically speaking, principal components represent the directions of the data that explain a **maximal amount of variance**, that is to say, the lines that capture most information of the data. The relationship between variance and information here, is that, the larger the variance carried by a line, the larger the dispersion of the data points along it, and the larger the dispersion along a line, the more information it has. To put all this simply, just think of principal components as new axes that provide the best angle to see and evaluate the data, so that the differences between the observations are better visible.

HIRING NOW

View All Remote Data Science Jobs

How PCA Constructs the Principal Components

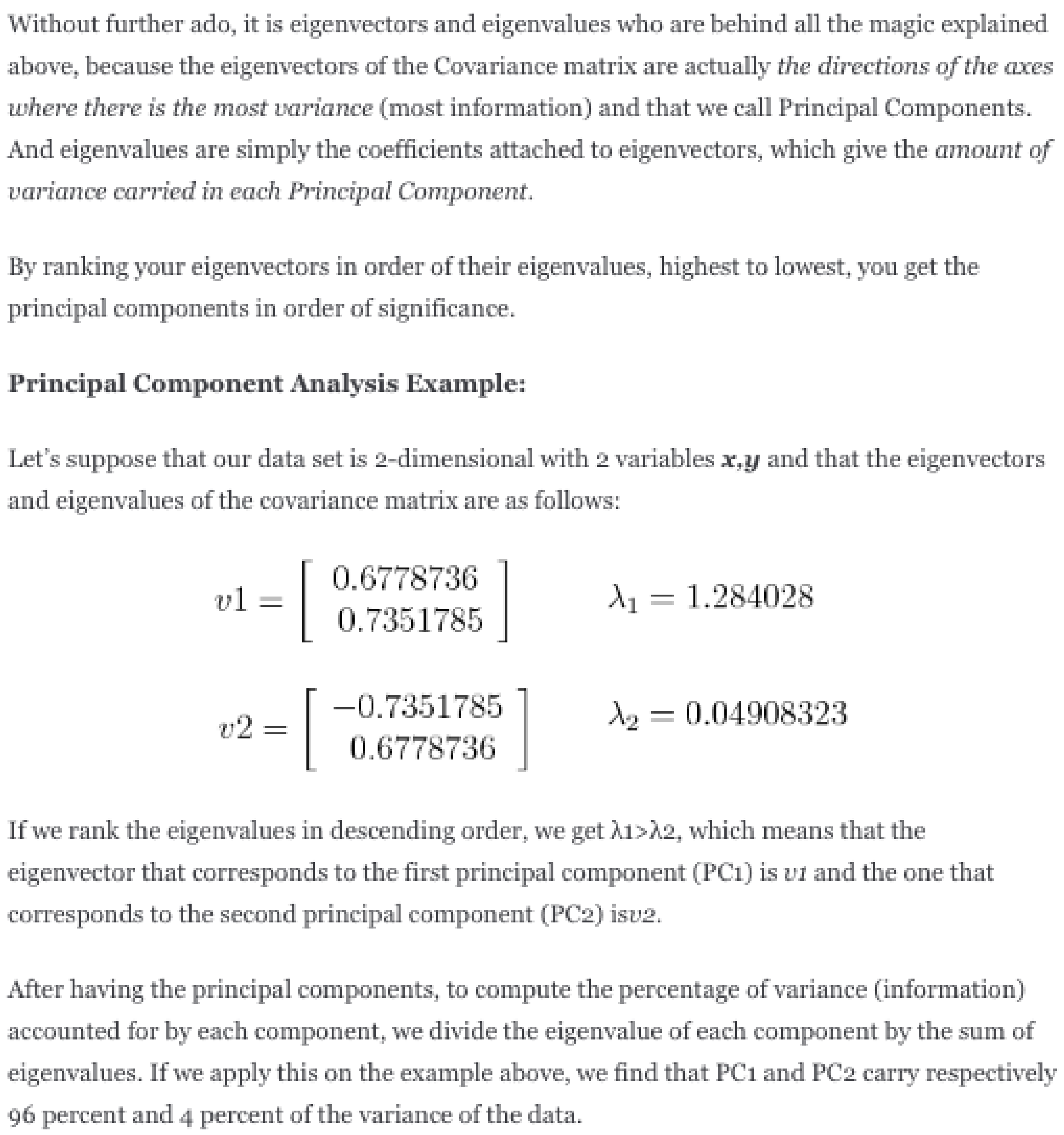
As there are as many principal components as there are variables in the data, principal components are constructed in such a manner that the first principal component accounts for the **largest possible variance** in the data set. For example, let's assume that the scatter plot of our data set is as shown below, can we guess the first principal component ? Yes, it's approximately the line that matches the purple marks because it goes through the origin and it's the line in which the projection of the points (red dots) is the most spread out. Or mathematically speaking, it's the line that maximizes the variance (the average of the squared distances from the projected points (red dots) to the origin).



The second principal component is calculated in the same way, with the condition that it is uncorrelated with (i.e., perpendicular to) the first principal component and that it accounts for the next highest variance.

This continues until a total of p principal components have been calculated, equal to the original number of variables.

Now that we understand what we mean by principal components, let's go back to eigenvectors and eigenvalues. What you first need to know about them is that they always come in pairs, so that every eigenvector has an eigenvalue. And their number is equal to the number of dimensions of the data. For example, for a 3-dimensional data set, there are 3 variables, therefore there are 3 eigenvectors with 3 corresponding eigenvalues.



Without further ado, it is eigenvectors and eigenvalues who are behind all the magic explained above, because the eigenvectors of the Covariance matrix are actually *the directions of the axes where there is the most variance* (most information) and that we call Principal Components. And eigenvalues are simply the coefficients attached to eigenvectors, which give the *amount of variance carried in each Principal Component*.

By ranking your eigenvectors in order of their eigenvalues, highest to lowest, you get the principal components in order of significance.

Principal Component Analysis Example:

Let's suppose that our data set is 2-dimensional with 2 variables ***x*,*y*** and that the eigenvectors and eigenvalues of the covariance matrix are as follows:

$$v1 = \begin{bmatrix} 0.6778736 \\ 0.7351785 \end{bmatrix} \qquad \lambda_1 = 1.284028$$

$$v2 = \begin{bmatrix} -0.7351785 \\ 0.6778736 \end{bmatrix} \qquad \lambda_2 = 0.04908323$$

If we rank the eigenvalues in descending order, we get λ1>λ2, which means that the eigenvector that corresponds to the first principal component (PC1) is *v1* and the one that corresponds to the second principal component (PC2) is*v2*.

After having the principal components, to compute the percentage of variance (information) accounted for by each component, we divide the eigenvalue of each component by the sum of eigenvalues. If we apply this on the example above, we find that PC1 and PC2 carry respectively 96 percent and 4 percent of the variance of the data.

STEP 4: FEATURE VECTOR

As we saw in the previous step, computing the eigenvectors and ordering them by their eigenvalues in descending order, allow us to find the principal components in order of significance. In this step, what we do is, to choose whether to keep all these components or discard those of lesser significance (of low eigenvalues), and form with the remaining ones a matrix of vectors that we call *Feature vector*.

So, the feature vector is simply a matrix that has as columns the eigenvectors of the components that we decide to keep. This makes it the first step towards dimensionality reduction, because if we choose to keep only $\boldsymbol{p}$ eigenvectors (components) out of $\boldsymbol{n}$, the final data set will have only $\boldsymbol{p}$ dimensions.

Principal Component Analysis Example:

Continuing with the example from the previous step, we can either form a feature vector with both of the eigenvectors $v1$ and $v2$:

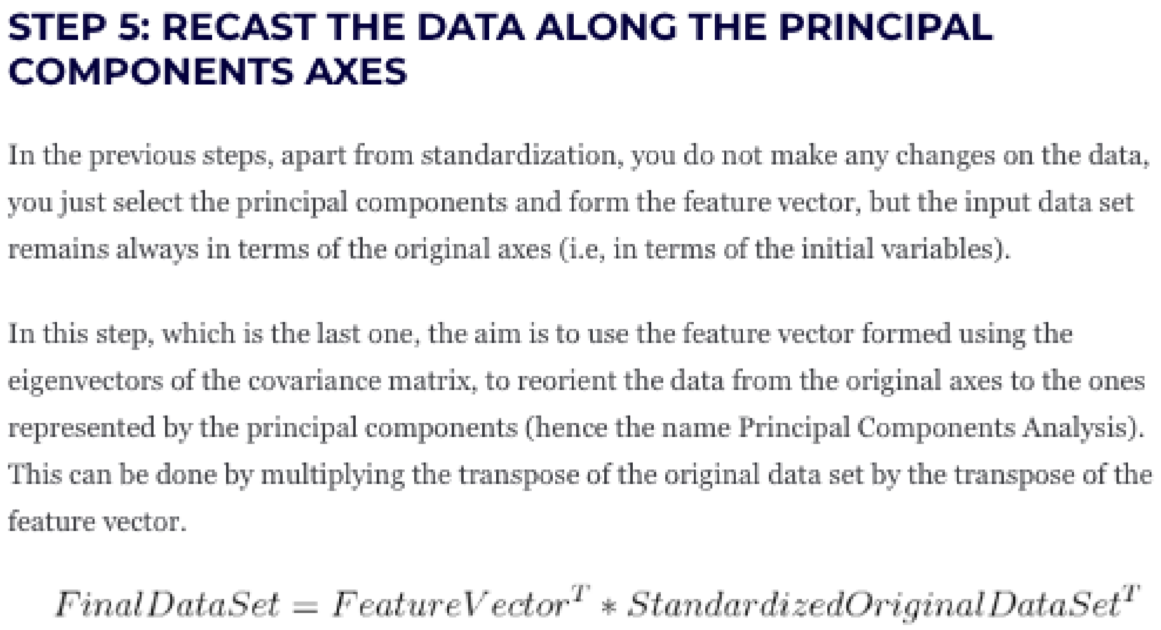
$$\begin{bmatrix} 0.6778736 & -0.7351785 \\ 0.7351785 & 0.6778736 \end{bmatrix}$$

Or discard the eigenvector $v2$, which is the one of lesser significance, and form a feature vector with $v1$ only:

$$\begin{bmatrix} 0.6778736 \\ 0.7351785 \end{bmatrix}$$

Discarding the eigenvector $v2$ will reduce dimensionality by 1, and will consequently cause a loss of information in the final data set. But given that $v2$ was carrying only 4 percent of the information, the loss will be therefore not important and we will still have 96 percent of the information that is carried by $v1$.

So, as we saw in the example, it's up to you to choose whether to keep all the components or discard the ones of lesser significance, depending on what you are looking for. Because if you just want to describe your data in terms of new variables (principal components) that are uncorrelated without seeking to reduce dimensionality, leaving out lesser significant components is not needed.



STEP 5: RECAST THE DATA ALONG THE PRINCIPAL COMPONENTS AXES

In the previous steps, apart from standardization, you do not make any changes on the data, you just select the principal components and form the feature vector, but the input data set remains always in terms of the original axes (i.e, in terms of the initial variables).

In this step, which is the last one, the aim is to use the feature vector formed using the eigenvectors of the covariance matrix, to reorient the data from the original axes to the ones represented by the principal components (hence the name Principal Components Analysis). This can be done by multiplying the transpose of the original data set by the transpose of the feature vector.

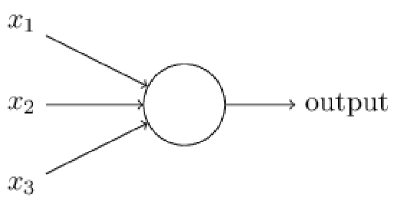
$$FinalDataSet = FeatureVector^T * StandardizedOriginalDataSet^T$$

Webside 6:
Nielsen, M. (December 2019). *neuralnetworksanddeeplearning*. Hentet fra http://neuralnetworksanddeeplearning.com/chap1.html

Perceptrons

What is a neural network? To get started, I'll explain a type of artificial neuron called a *perceptron*. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neurons - in this book, and in much modern work on neural networks, the main neuron model used is one called the *sigmoid neuron*. We'll get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand perceptrons.

So how do perceptrons work? A perceptron takes several binary inputs, $x_1, x_2, \ldots$, and produces a single binary output:



In the example shown the perceptron has three inputs, x_1, x_2, x_3. In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He introduced *weights*, $w_1, w_2, \ldots$, real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold value*. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases} \tag{1}$$

That's all there is to how a perceptron works!

That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence. Let me give an example. It's not a very realistic example, but it's easy to understand, and we'll soon get to more realistic examples. Suppose the weekend is coming up, and you've heard that there's going to be a cheese festival in your city. You like cheese, and are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

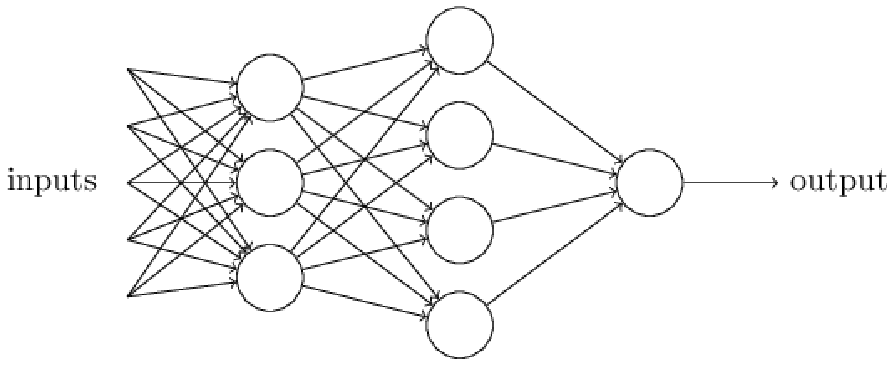
1. Is the weather good?
2. Does your boyfriend or girlfriend want to accompany you?
3. Is the festival near public transit? (You don't own a car).

We can represent these three factors by corresponding binary variables x_1, x_2, and x_3. For instance, we'd have $x_1 = 1$ if the weather is good, and $x_1 = 0$ if the weather is bad. Similarly, $x_2 = 1$ if your boyfriend or girlfriend wants to go, and $x_2 = 0$ if not. And similarly again for x_3 and public transit.

Now, suppose you absolutely adore cheese, so much so that you're happy to go to the festival even if your boyfriend or girlfriend is uninterested and the festival is hard to get to. But perhaps you really loathe bad weather, and there's no way you'd go to the festival if the weather is bad. You can use perceptrons to model this kind of decision-making. One way to do this is to choose a weight $w_1 = 6$ for the weather, and $w_2 = 2$ and $w_3 = 2$ for the other conditions. The larger value of w_1 indicates that the weather matters a lot to you, much more than whether your boyfriend or girlfriend joins you, or the nearness of public transit. Finally, suppose you choose a threshold of 5 for the perceptron. With these choices, the perceptron implements the desired decision-making model, outputting 1 whenever the weather is good, and 0 whenever the weather is bad. It makes no difference to the output whether your boyfriend or girlfriend wants to go, or whether public transit is nearby.

By varying the weights and the threshold, we can get different models of decision-making. For example, suppose we instead chose a threshold of 3. Then the perceptron would decide that you should go to the festival whenever the weather was good *or* when both the festival was near public transit *and* your boyfriend or girlfriend was willing to join you. In other words, it'd be a different model of decision-making. Dropping the threshold means you're more willing to go to the festival.

Obviously, the perceptron isn't a complete model of human decision-making! But what the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions:



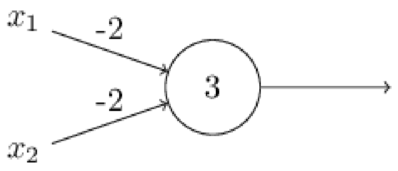
In this network, the first column of perceptrons - what we'll call the first *layer* of perceptrons - is making three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

Let's simplify the way we describe perceptrons. The condition $\sum_j w_j x_j >$ threshold is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\sum_j w_j x_j$ as a dot product, $w \cdot x \equiv \sum_j w_j x_j$, where w and x are vectors whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's *bias*, $b \equiv -$threshold. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{2}$$

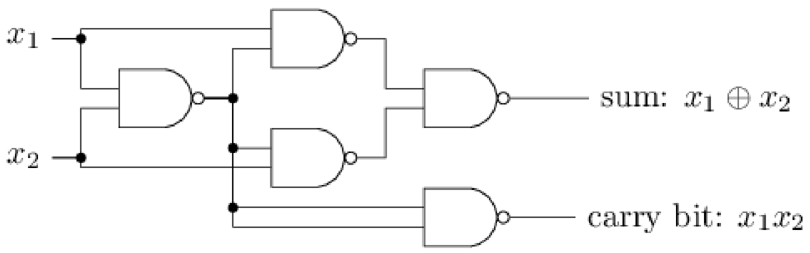
You can think of the bias as a measure of how easy it is to get the perceptron to output a 1. Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to *fire*. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1. Obviously, introducing the bias is only a small change in how we describe perceptrons, but we'll see later that it leads to further notational simplifications. Because of this, in the remainder of the book we won't use the threshold, we'll always use the bias.

I've described perceptrons as a method for weighing evidence to make decisions. Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as `AND`, `OR`, and `NAND`. For example, suppose we have a perceptron with two inputs, each with weight -2, and an overall bias of 3. Here's our perceptron:



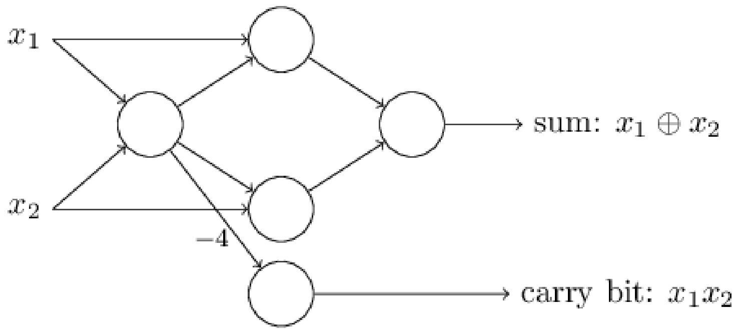
Then we see that input 00 produces output 1, since $(-2) * 0 + (-2) * 0 + 3 = 3$ is positive. Here, I've introduced the $*$ symbol to make the multiplications explicit. Similar calculations show that the inputs 01 and 10 produce output 1. But the input 11 produces output 0, since $(-2) * 1 + (-2) * 1 + 3 = -1$ is negative. And so our perceptron implements a NAND gate!

The NAND example shows that we can use perceptrons to compute simple logical functions. In fact, we can use networks of perceptrons to compute *any* logical function at all. The reason is that the NAND gate is universal for computation, that is, we can build any computation up out of NAND gates. For example, we can use NAND gates to build a circuit which adds two bits, x_1 and x_2. This requires computing the bitwise sum, $x_1 \oplus x_2$, as well as a carry bit which is set to 1 when both x_1 and x_2 are 1, i.e., the carry bit is just the bitwise product $x_1 x_2$:

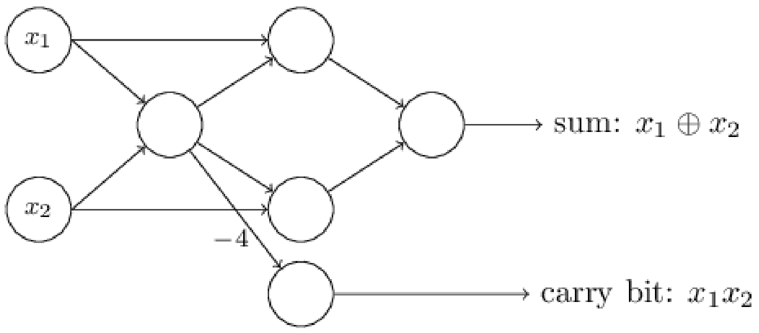


To get an equivalent network of perceptrons we replace all the NAND gates by perceptrons with two inputs, each with weight -2, and an overall bias of 3. Here's the resulting network. Note that I've moved the perceptron corresponding to the bottom right NAND gate a little, just to make it easier to draw the arrows on the diagram:

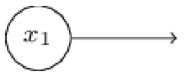
One notable aspect of this network of perceptrons is that the output from the leftmost perceptron is used twice as input to the bottommost perceptron. When I defined the perceptron model I didn't say whether this kind of double-output-to-the-same-place was allowed. Actually, it doesn't much matter. If we don't want to allow this kind of thing, then it's possible to simply merge the two lines, into a single connection with a weight of -4 instead of two connections with -2 weights. (If you don't find this obvious, you should stop and prove to yourself that this is equivalent.) With that change, the network looks as follows, with all unmarked weights equal to -2, all biases equal to 3, and a single weight of -4, as marked:



Up to now I've been drawing inputs like x_1 and x_2 as variables floating to the left of the network of perceptrons. In fact, it's conventional to draw an extra layer of perceptrons - the *input layer* - to encode the inputs:



This notation for input perceptrons, in which we have an output, but no inputs,



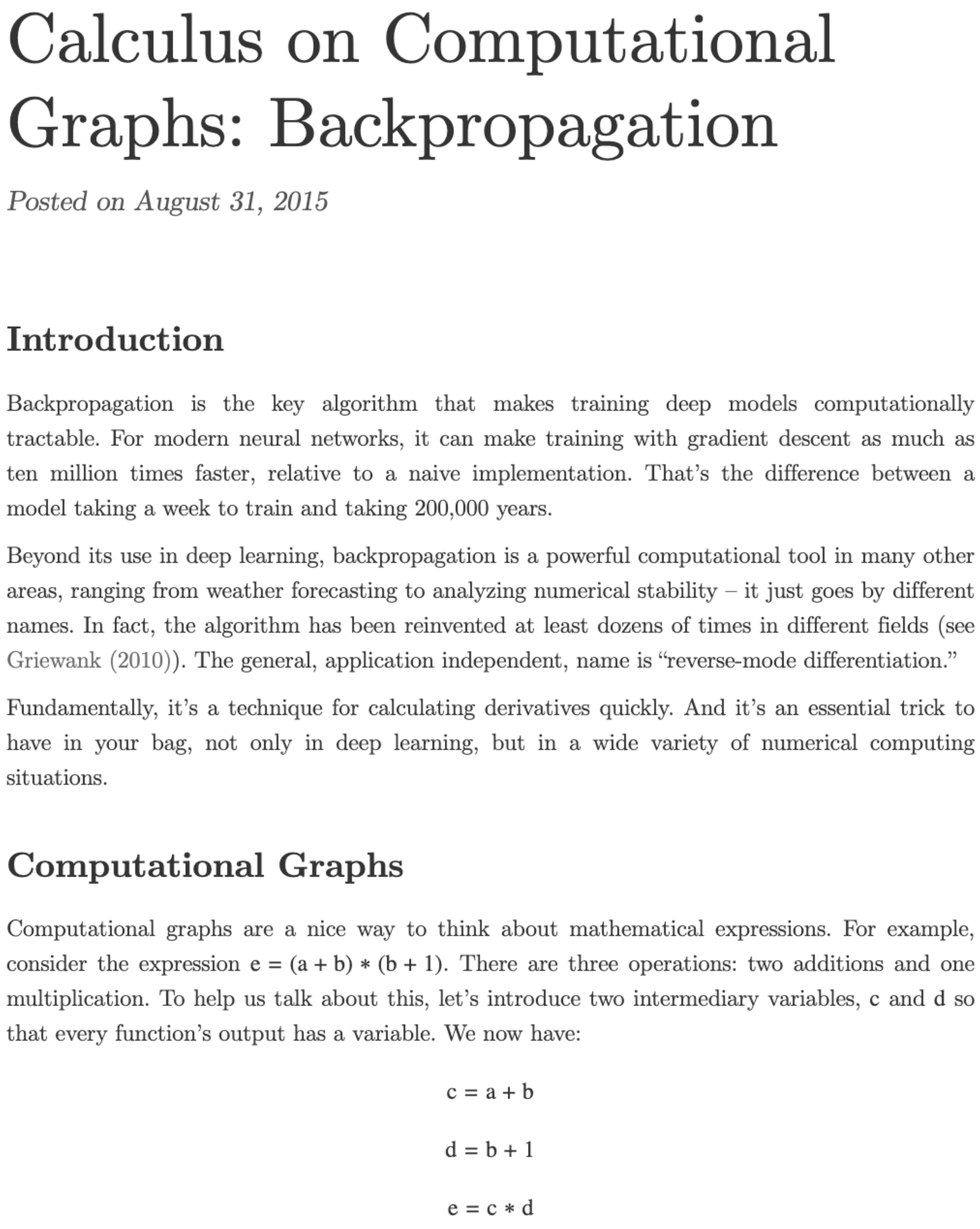
is a shorthand. It doesn't actually mean a perceptron with no inputs. To see this, suppose we did have a perceptron with no inputs. Then the weighted sum $\sum_j w_j x_j$ would always be zero, and so the perceptron would output 1 if $b > 0$, and 0 if $b \leq 0$. That is, the perceptron would simply output a fixed value, not the desired value (x_1, in the example above). It's better to think of the input perceptrons as not really being perceptrons at all, but rather special units which are simply defined to output the desired values, $x_1, x_2, \ldots$.

The adder example demonstrates how a network of perceptrons can be used to simulate a circuit containing many NAND gates. And because NAND gates are universal for computation, it follows that perceptrons are also universal for computation.

The computational universality of perceptrons is simultaneously reassuring and disappointing. It's reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it's also disappointing, because it makes it seem as though perceptrons are merely a new type of NAND gate. That's hardly big news!

However, the situation is better than this view suggests. It turns out that we can devise *learning algorithms* which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without direct intervention by a programmer. These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates. Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit.

Webside 7:
olah, c. (31. august 2015). *colah's blog* . Hentet fra http://colah.github.io/posts/2015-08-Backprop/



Calculus on Computational Graphs: Backpropagation

Posted on August 31, 2015

Introduction

Backpropagation is the key algorithm that makes training deep models computationally tractable. For modern neural networks, it can make training with gradient descent as much as ten million times faster, relative to a naive implementation. That's the difference between a model taking a week to train and taking 200,000 years.

Beyond its use in deep learning, backpropagation is a powerful computational tool in many other areas, ranging from weather forecasting to analyzing numerical stability – it just goes by different names. In fact, the algorithm has been reinvented at least dozens of times in different fields (see Griewank (2010)). The general, application independent, name is "reverse-mode differentiation."

Fundamentally, it's a technique for calculating derivatives quickly. And it's an essential trick to have in your bag, not only in deep learning, but in a wide variety of numerical computing situations.

Computational Graphs

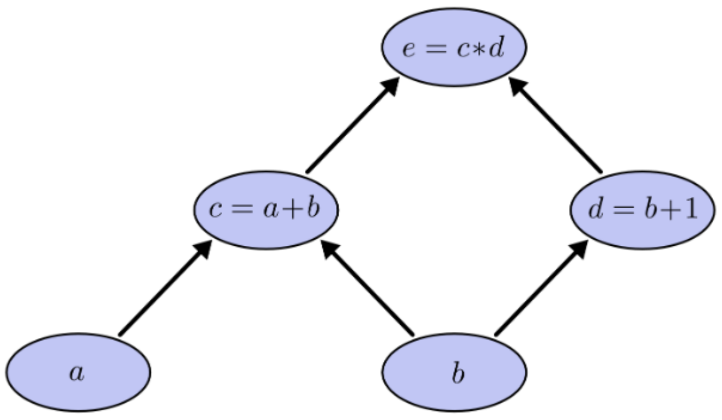
Computational graphs are a nice way to think about mathematical expressions. For example, consider the expression $e = (a + b) * (b + 1)$. There are three operations: two additions and one multiplication. To help us talk about this, let's introduce two intermediary variables, c and d so that every function's output has a variable. We now have:

$$c = a + b$$

$$d = b + 1$$

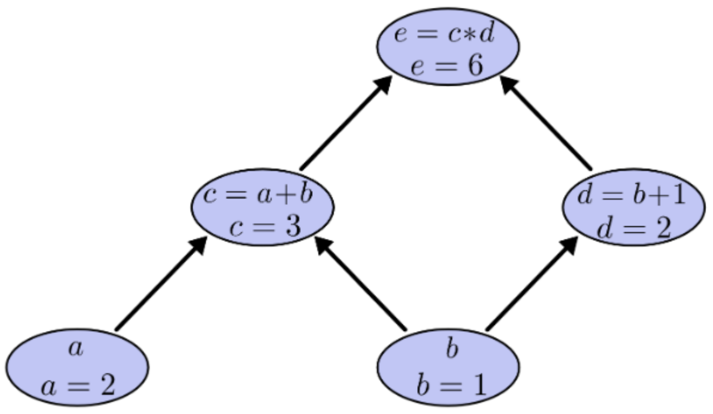
$$e = c * d$$

To create a computational graph, we make each of these operations, along with the input variables, into nodes. When one node's value is the input to another node, an arrow goes from one to another.



These sorts of graphs come up all the time in computer science, especially in talking about functional programs. They are very closely related to the notions of dependency graphs and call graphs. They're also the core abstraction behind the popular deep learning framework Theano.

We can evaluate the expression by setting the input variables to certain values and computing nodes up through the graph. For example, let's set a = 2 and b = 1:



The expression evaluates to 6.

Derivatives on Computational Graphs

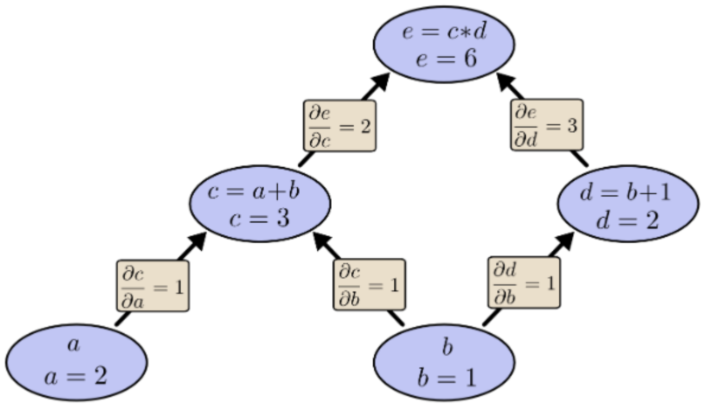
If one wants to understand derivatives in a computational graph, the key is to understand derivatives on the edges. If a directly affects c, then we want to know how it affects c. If a changes a little bit, how does c change? We call this the partial derivative of c with respect to a.

To evaluate the partial derivatives in this graph, we need the sum rule and the product rule:

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$

Below, the graph has the derivative on each edge labeled.


$e = c*d$
$e = 6$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$c = a+b$
$c = 3$
$d = b+1$
$d = 2$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
a
$a = 2$
b
$b = 1$

What if we want to understand how nodes that aren't directly connected affect each other? Let's consider how e is affected by a. If we change a at a speed of 1, c also changes at a speed of 1. In turn, c changing at a speed of 1 causes e to change at a speed of 2. So e changes at a rate of $1 * 2$ with respect to a.

The general rule is to sum over all possible paths from one node to the other, multiplying the derivatives on each edge of the path together. For example, to get the derivative of e with respect to b we get:

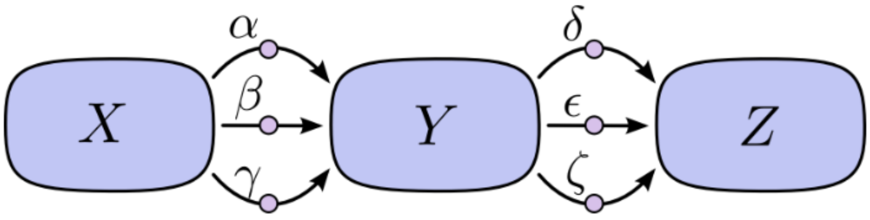
$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

This accounts for how b affects e through c and also how it affects it through d.

This general "sum over paths" rule is just a different way of thinking about the multivariate chain rule.

Factoring Paths

The problem with just "summing over the paths" is that it's very easy to get a combinatorial explosion in the number of possible paths.



In the above diagram, there are three paths from X to Y, and a further three paths from Y to Z. If we want to get the derivative $\frac{\partial Z}{\partial X}$ by summing over all paths, we need to sum over $3 * 3 = 9$ paths:

$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

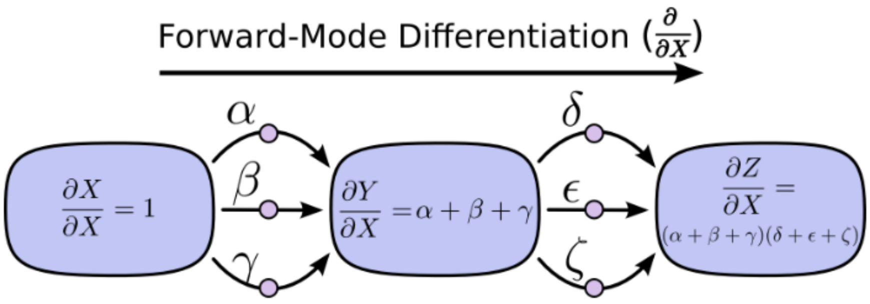
The above only has nine paths, but it would be easy to have the number of paths to grow exponentially as the graph becomes more complicated.

Instead of just naively summing over the paths, it would be much better to factor them:

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$$

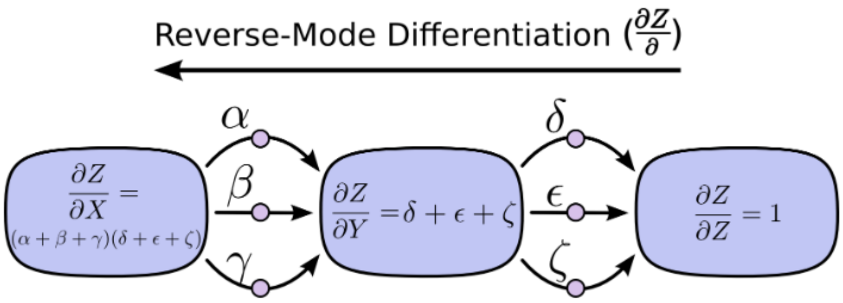
This is where "forward-mode differentiation" and "reverse-mode differentiation" come in. They're algorithms for efficiently computing the sum by factoring the paths. Instead of summing over all of the paths explicitly, they compute the same sum more efficiently by merging paths back together at every node. In fact, both algorithms touch each edge exactly once!

Forward-mode differentiation starts at an input to the graph and moves towards the end. At every node, it sums all the paths feeding in. Each of those paths represents one way in which the input affects that node. By adding them up, we get the total way in which the node is affected by the input, it's derivative.


Forward-Mode Differentiation $(\frac{\partial}{\partial X})$
$\frac{\partial X}{\partial X} = 1$
$\frac{\partial Y}{\partial X} = \alpha + \beta + \gamma$
$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$
α
β
γ
δ
ϵ
ζ

Though you probably didn't think of it in terms of graphs, forward-mode differentiation is very similar to what you implicitly learned to do if you took an introduction to calculus class.

Reverse-mode differentiation, on the other hand, starts at an output of the graph and moves towards the beginning. At each node, it merges all paths which originated at that node.

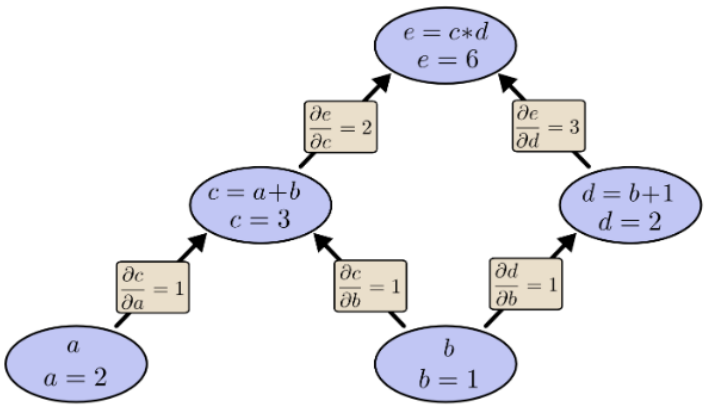

Reverse-Mode Differentiation $(\frac{\partial Z}{\partial})$
α
β
γ
δ
ϵ
ζ
$\frac{\partial Z}{\partial X} = (\alpha+\beta+\gamma)(\delta+\epsilon+\zeta)$
$\frac{\partial Z}{\partial Y} = \delta+\epsilon+\zeta$
$\frac{\partial Z}{\partial Z} = 1$

Forward-mode differentiation tracks how one input affects every node. Reverse-mode differentiation tracks how every node affects one output. That is, forward-mode differentiation applies the operator $\frac{\partial}{\partial X}$ to every node, while reverse mode differentiation applies the operator $\frac{\partial Z}{\partial}$ to every node.[1]

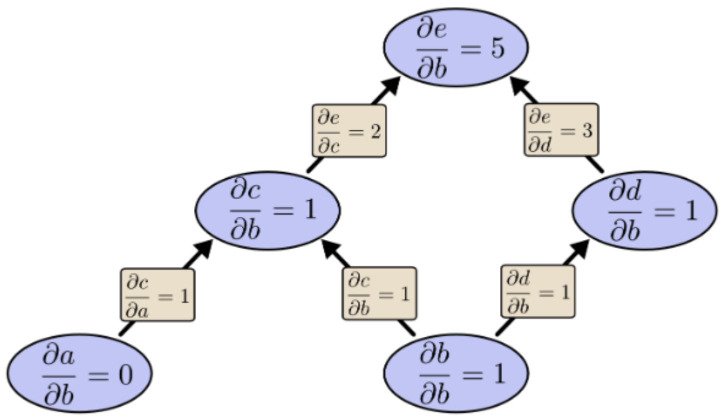
Computational Victories

At this point, you might wonder why anyone would care about reverse-mode differentiation. It looks like a strange way of doing the same thing as the forward-mode. Is there some advantage?

Let's consider our original example again:

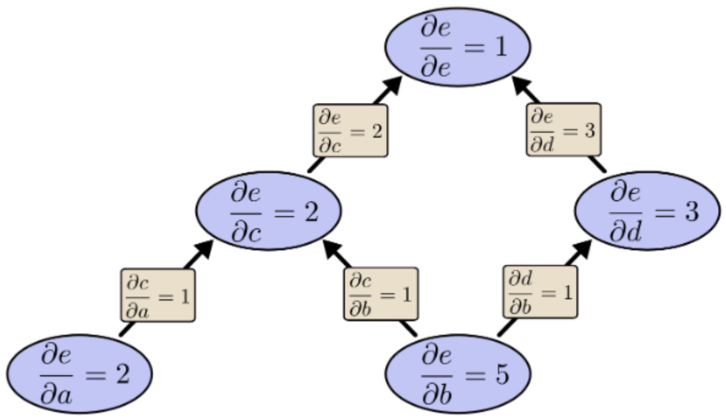

$e = c*d$
$e = 6$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$c = a+b$
$c = 3$
$d = b+1$
$d = 2$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
a
$a = 2$
b
$b = 1$

We can use forward-mode differentiation from b up. This gives us the derivative of every node with respect to b.


$\frac{\partial e}{\partial b} = 5$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial a}{\partial b} = 0$
$\frac{\partial b}{\partial b} = 1$

We've computed $\frac{\partial e}{\partial b}$, the derivative of our output with respect to one of our inputs.

What if we do reverse-mode differentiation from e down? This gives us the derivative of e with respect to every node:


$\frac{\partial e}{\partial e} = 1$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial e}{\partial a} = 2$
$\frac{\partial e}{\partial b} = 5$

When I say that reverse-mode differentiation gives us the derivative of e with respect to every node, I really do mean *every node*. We get both $\frac{\partial e}{\partial a}$ and $\frac{\partial e}{\partial b}$, the derivatives of e with respect to both inputs. Forward-mode differentiation gave us the derivative of our output with respect to a single input, but reverse-mode differentiation gives us all of them.

For this graph, that's only a factor of two speed up, but imagine a function with a million inputs and one output. Forward-mode differentiation would require us to go through the graph a million times to get the derivatives. Reverse-mode differentiation can get them all in one fell swoop! A speed up of a factor of a million is pretty nice!

For this graph, that's only a factor of two speed up, but imagine a function with a million inputs and one output. Forward-mode differentiation would require us to go through the graph a million times to get the derivatives. Reverse-mode differentiation can get them all in one fell swoop! A speed up of a factor of a million is pretty nice!

When training neural networks, we think of the cost (a value describing how bad a neural network performs) as a function of the parameters (numbers describing how the network behaves). We want to calculate the derivatives of the cost with respect to all the parameters, for use in gradient descent. Now, there's often millions, or even tens of millions of parameters in a neural network. So, reverse-mode differentiation, called backpropagation in the context of neural networks, gives us a massive speed up!

(Are there any cases where forward-mode differentiation makes more sense? Yes, there are! Where the reverse-mode gives the derivatives of one output with respect to all inputs, the forward-mode gives us the derivatives of all outputs with respect to one input. If one has a function with lots of outputs, forward-mode differentiation can be much, much, much faster.)

Isn't This Trivial?

When I first understood what backpropagation was, my reaction was: "Oh, that's just the chain rule! How did it take us so long to figure out?" I'm not the only one who's had that reaction. It's true that if you ask "is there a smart way to calculate derivatives in feedforward neural networks?" the answer isn't that difficult.

But I think it was much more difficult than it might seem. You see, at the time backpropagation was invented, people weren't very focused on the feedforward neural networks that we study. It also wasn't obvious that derivatives were the right way to train them. Those are only obvious once you realize you can quickly calculate derivatives. There was a circular dependency.

Worse, it would be very easy to write off any piece of the circular dependency as impossible on casual thought. Training neural networks with derivatives? Surely you'd just get stuck in local minima. And obviously it would be expensive to compute all those derivatives. It's only because we know this approach works that we don't immediately start listing reasons it's likely not to.

That's the benefit of hindsight. Once you've framed the question, the hardest work is already done.

Conclusion

Derivatives are cheaper than you think. That's the main lesson to take away from this post. In fact, they're unintuitively cheap, and us silly humans have had to repeatedly rediscover this fact. That's an important thing to understand in deep learning. It's also a really useful thing to know in other fields, and only more so if it isn't common knowledge.

Are there other lessons? I think there are.

Backpropagation is also a useful lens for understanding how derivatives flow through a model. This can be extremely helpful in reasoning about why some models are difficult to optimize. The classic example of this is the problem of vanishing gradients in recurrent neural networks.

Finally, I claim there is a broad algorithmic lesson to take away from these techniques. Backpropagation and forward-mode differentiation use a powerful pair of tricks (linearization and dynamic programming) to compute derivatives more efficiently than one might think possible. If you really understand these techniques, you can use them to efficiently calculate several other interesting expressions involving derivatives. We'll explore this in a later blog post.

This post gives a very abstract treatment of backpropagation. I strongly recommend reading Michael Nielsen's chapter on it for an excellent discussion, more concretely focused on neural networks.

Acknowledgments

Thank you to Greg Corrado, Jon Shlens, Samy Bengio and Anelia Angelova for taking the time to proofread this post.

Thanks also to Dario Amodei, Michael Nielsen and Yoshua Bengio for discussion of approaches to explaining backpropagation. Also thanks to all those who tolerated me practicing explaining backpropagation in talks and seminar series!

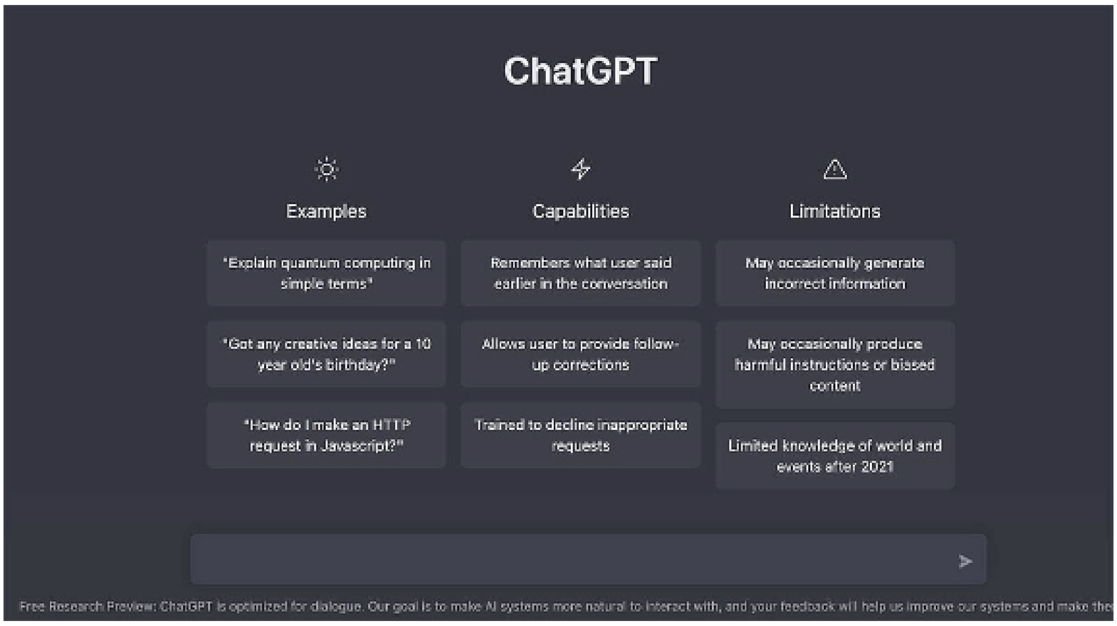
Webside 8:
Simonsen, J. F. (20. juli 2023). *Somejuan* . Hentet fra https://somejuan.dk/blog/hvad-er-chatgpt-og-hvordan-bruger-du-ai-chatbotten/ ¨

ChatGPT er blevet et internetfænomen på ingen tid. Den imponerer alle med dens *AI-genereret* indhold. Det er så hypet, at du ikke kan undgå at forholde dig til det.

Det har udviklet sig fra at være endnu en chatbot på nettet til at være et kæmpe fremskridt i den næste æra af innovation.

Det er dog ikke alle, der helt forstår hvad der sker, og måske spørger du dig selv: Hvorfor alt det larm omkring ChatGPT? Er det ikke bare en chatbot?

Læs videre og bliv klogere på hvad ChatGPT er.


ChatGPT

Examples	Capabilities	Limitations
"Explain quantum computing in simple terms"	Remembers what user said earlier in the conversation	May occasionally generate incorrect information
"Got any creative ideas for a 10 year old's birthday?"	Allows user to provide follow-up corrections	May occasionally produce harmful instructions or biased content
"How do I make an HTTP request in Javascript?"	Trained to decline inappropriate requests	Limited knowledge of world and events after 2021

Free Research Preview: ChatGPT is optimized for dialogue. Our goal is to make AI systems more natural to interact with, and your feedback will help us improve our systems and make the

Hvad er ChatGPT?

ChatGPT er en “conversional AI chatbot”. Med andre ord, en chatbot som svarer på dine forespørgsler, også kaldet prompts, og som simulerer menneskelig intelligens. På det simpleste niveau , så betyder det, at du kan stille et spørgsmål, og den vil svare dig – samtidigt med at den bliver “klogere” over tid.

Men lad os lige blive enige om hvad jeg mener med AI her…

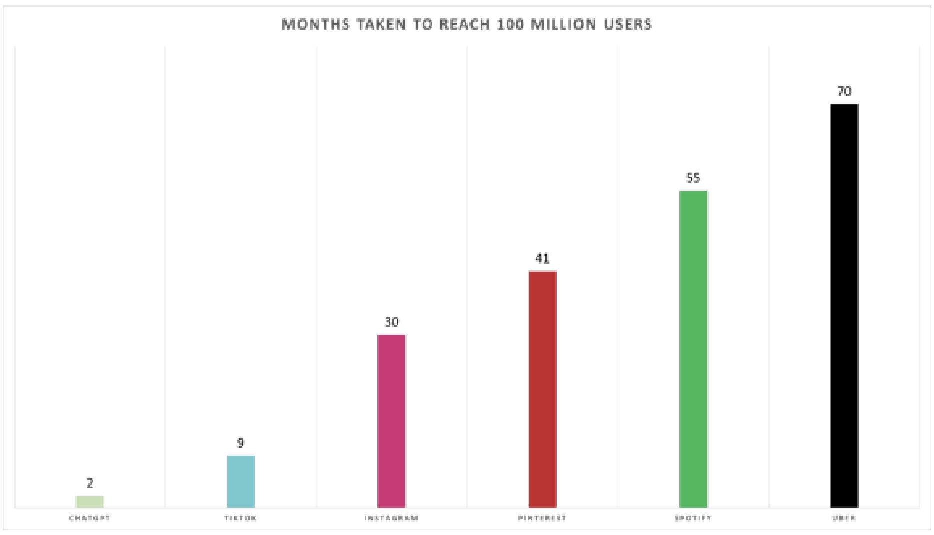
AI er en forkortelse for kunstig intelligens og henviser til simulering af menneskelig intelligens i maskiner, der er programmeret til at tænke og lære som mennesker.

ChatGPT blev først lanceret som en prototype for offentligheden i november 2022 og voksede hurtigt til over 100 millioner brugere i januar 2023, hvilket gør den til den hurtigst teknologissoftware, der nogensinde er blevet lavet.

ChatGPT fungerer ved at bruge en form for kunstig intelligens, der kaldes ‘machine learning’. Dataen, som den bruger til at svare, stammer fra store mængder tekst fra internettet og andre kilder, så chatbotten kan lære at genkende mønstre og sammenhænge mellem ord og sætninger.

Det er dog vigtigt at pointere at den nuværende version af ChatGPT (GPT-3.5) har som sådan ikke adgang til internettet, dens viden er begrænset til begivenheder som skete før 2021.

Overvej følgende: Det tog ChatGPT 2 måneder at nå over 100 millioner aktive brugere.



Kilde: USB/Yahoo Finance

Webside 9:
TARANOVICH, S. (27. april 2019). *EDN*. Hentet fra https://www.edn.com/how-the-sigmoid-function-is-used-in-ai/

How the sigmoid function is used in AI

APRIL 27, 2019
BY **STEVE TARANOVICH**

When is the last time you used a sigmoid function?

The sigmoid logistic function was introduced in a series of three papers by Pierre François Verhulst between 1838 and 1847, who devised it as a model of population growth by adjusting the exponential growth model, under the guidance of Adolphe Quetelet (**Figure 1**).

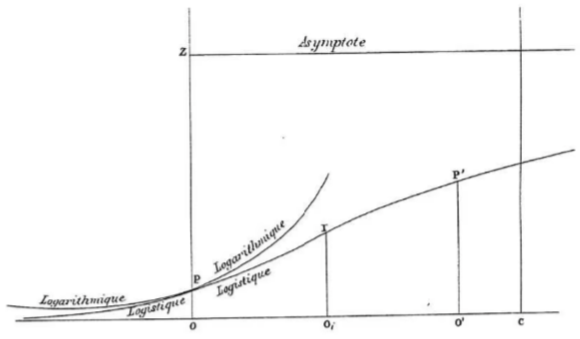


Figure 1 Verhulst's logistic curve, contrasted with a logarithmic curve

For a 1963 example of how a sigmoid function and curve can be used, see Reference 1. An electro-optical instrument measures the capacity of red blood cell membranes as their internal pressure increases due to the diffusion of water into the cells via a gradual deceasing of salt concentration of the fluid surrounding the cell. This test yields an osmotic fragility curve. A direct sigmoid curve or a derivative curve can then be used to fit the data and then recorded.

In today's modern world of artificial intelligence (AI), the sigmoid function is used in artificial neural networks (Reference 6) to determine the relationships between biological and artificial neural networks.

See how the sigmoid function can also be used in machine learning (ML) in a data center. In Reference 7, section 2.2.2 on Forward Propagation, the activation function mimics the biological neuron firing within a network by mapping the nodal input values to an output in the range (0,1). It is given by the sigmoidal logistic function.

Steve Taranovich is a senior technical editor at EDN *with 45 years of experience in the electronics industry.*

Webside 10:
Wang, C.-F. (8. jan 2019). *towardsdatascience*. Hentet fra https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

The Vanishing Gradient Problem

The Problem, Its Causes, Its Significance, and Its Solutions

Chi-Feng Wang · Follow
Published in Towards Data Science · 3 min read · Jan 8, 2019

2.8K 12

Title Image // Source

The problem:

As more layers using certain activation functions are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train.

Why:

Certain activation functions, like the sigmoid function, squishes a large input space into a small input space between 0 and 1. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small.

[illegible] // Source

The problem:

As more layers using certain activation functions are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train.

Why:

Certain activation functions, like the sigmoid function, squishes a large input space into a small input space between 0 and 1. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small.

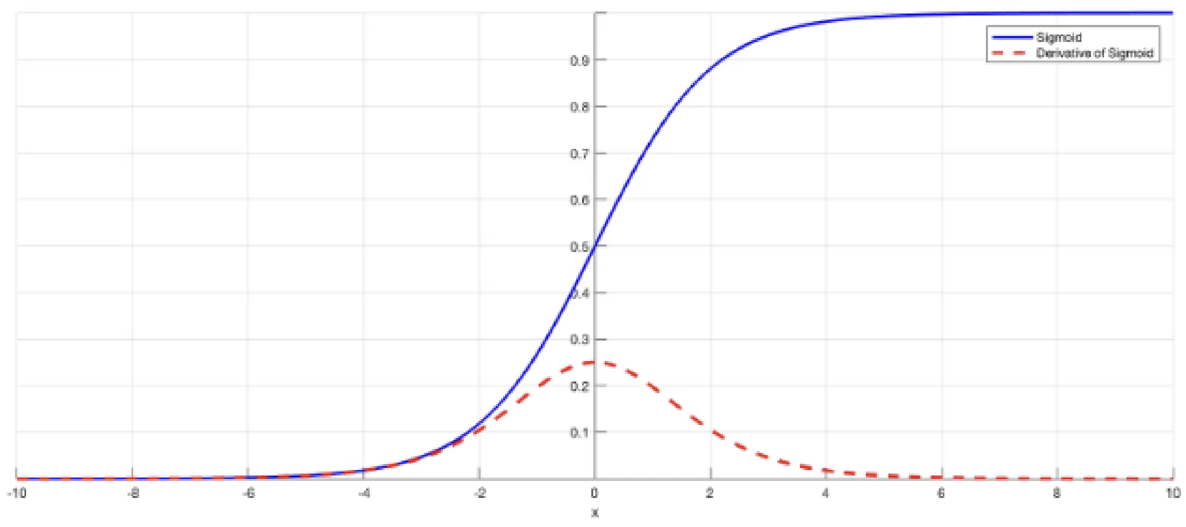
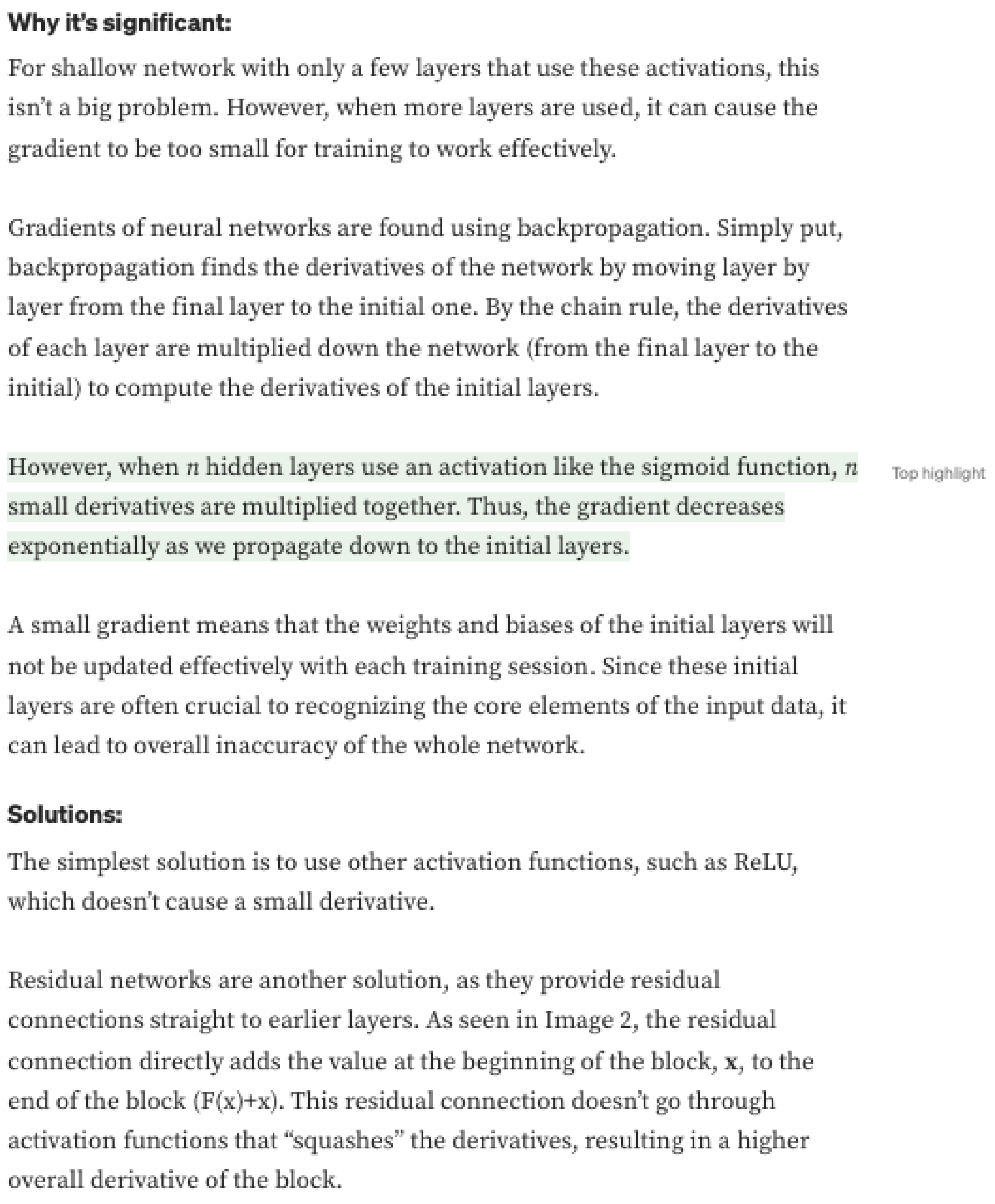


Image 1: The sigmoid function and its derivative // Source

As an example, Image 1 is the sigmoid function and its derivative. Note how when the inputs of the sigmoid function becomes larger or smaller (when |x| becomes bigger), the derivative becomes close to zero.



Why it's significant:

For shallow network with only a few layers that use these activations, this isn't a big problem. However, when more layers are used, it can cause the gradient to be too small for training to work effectively.

Gradients of neural networks are found using backpropagation. Simply put, backpropagation finds the derivatives of the network by moving layer by layer from the final layer to the initial one. By the chain rule, the derivatives of each layer are multiplied down the network (from the final layer to the initial) to compute the derivatives of the initial layers.

However, when n hidden layers use an activation like the sigmoid function, n small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers.

Top highlight

A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network.

Solutions:

The simplest solution is to use other activation functions, such as ReLU, which doesn't cause a small derivative.

Residual networks are another solution, as they provide residual connections straight to earlier layers. As seen in Image 2, the residual connection directly adds the value at the beginning of the block, **x**, to the end of the block (F(x)+x). This residual connection doesn't go through activation functions that "squashes" the derivatives, resulting in a higher overall derivative of the block.

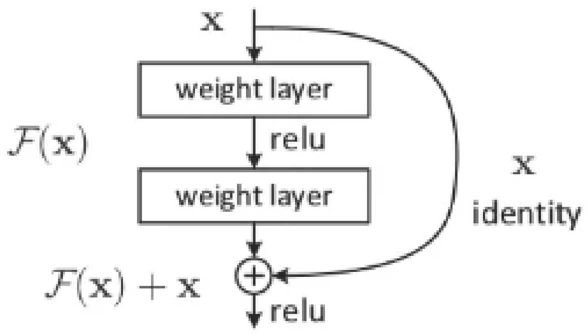

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Image 2: A residual block

Finally, batch normalization layers can also resolve the issue. As stated before, the problem arises when a large input space is mapped to a small one, causing the derivatives to disappear. In Image 1, this is most clearly seen at when |x| is big. Batch normalization reduces this problem by simply normalizing the input so |x| doesn't reach the outer edges of the sigmoid function. As seen in Image 3, it normalizes the input so that most of it falls in the green region, where the derivative isn't too small.

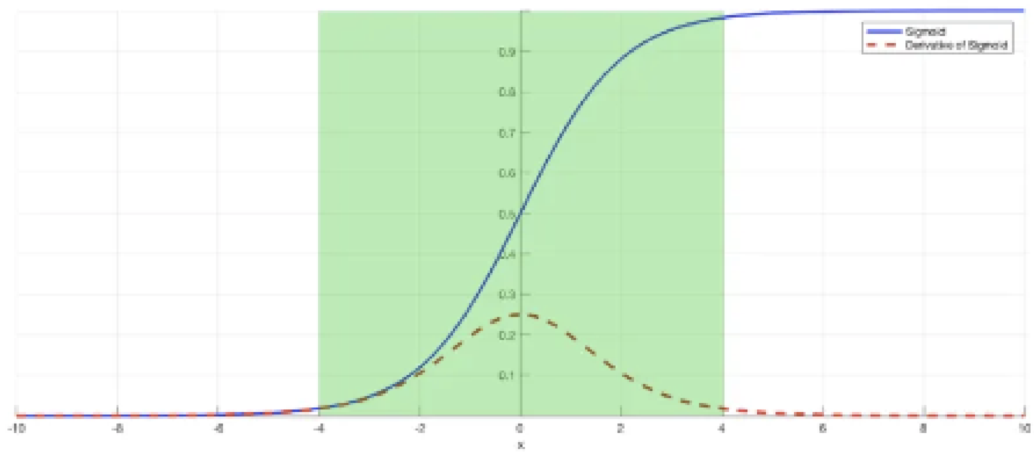
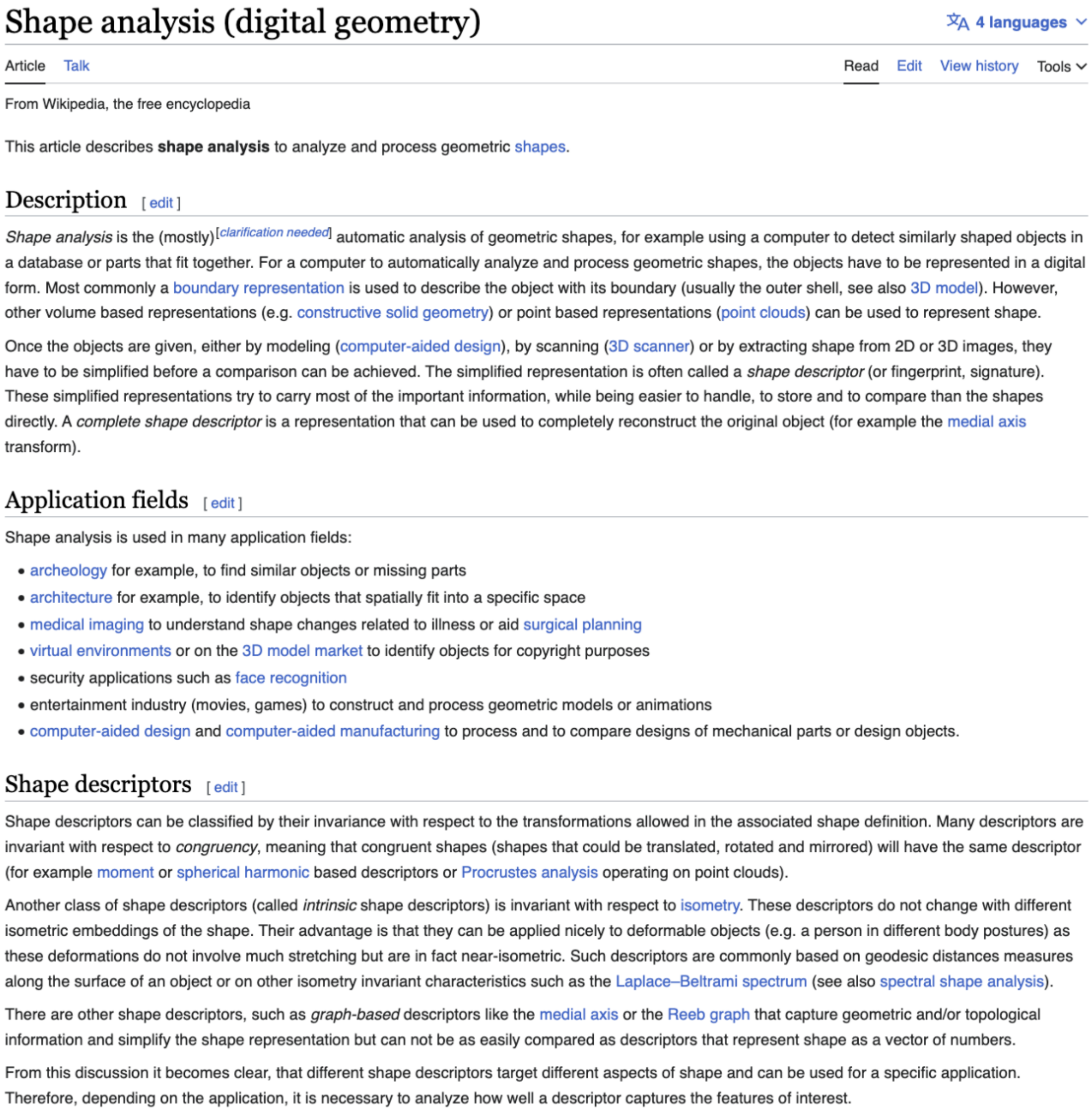


Image 3: Sigmoid function with restricted inputs

. . .

Do leave a comment below if you have any questions or suggestions :)

Webside 11:
wikipedia. (u.d.). Hentet fra Shape analysis (digital geometry): https://en.wikipedia.org/wiki/Shape_analysis_(digital_geometry)



Shape analysis (digital geometry)

4 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

This article describes **shape analysis** to analyze and process geometric shapes.

Description [edit]

Shape analysis is the (mostly)[clarification needed] automatic analysis of geometric shapes, for example using a computer to detect similarly shaped objects in a database or parts that fit together. For a computer to automatically analyze and process geometric shapes, the objects have to be represented in a digital form. Most commonly a boundary representation is used to describe the object with its boundary (usually the outer shell, see also 3D model). However, other volume based representations (e.g. constructive solid geometry) or point based representations (point clouds) can be used to represent shape.

Once the objects are given, either by modeling (computer-aided design), by scanning (3D scanner) or by extracting shape from 2D or 3D images, they have to be simplified before a comparison can be achieved. The simplified representation is often called a *shape descriptor* (or fingerprint, signature). These simplified representations try to carry most of the important information, while being easier to handle, to store and to compare than the shapes directly. A *complete shape descriptor* is a representation that can be used to completely reconstruct the original object (for example the medial axis transform).

Application fields [edit]

Shape analysis is used in many application fields:

- archeology for example, to find similar objects or missing parts
- architecture for example, to identify objects that spatially fit into a specific space
- medical imaging to understand shape changes related to illness or aid surgical planning
- virtual environments or on the 3D model market to identify objects for copyright purposes
- security applications such as face recognition
- entertainment industry (movies, games) to construct and process geometric models or animations
- computer-aided design and computer-aided manufacturing to process and to compare designs of mechanical parts or design objects.

Shape descriptors [edit]

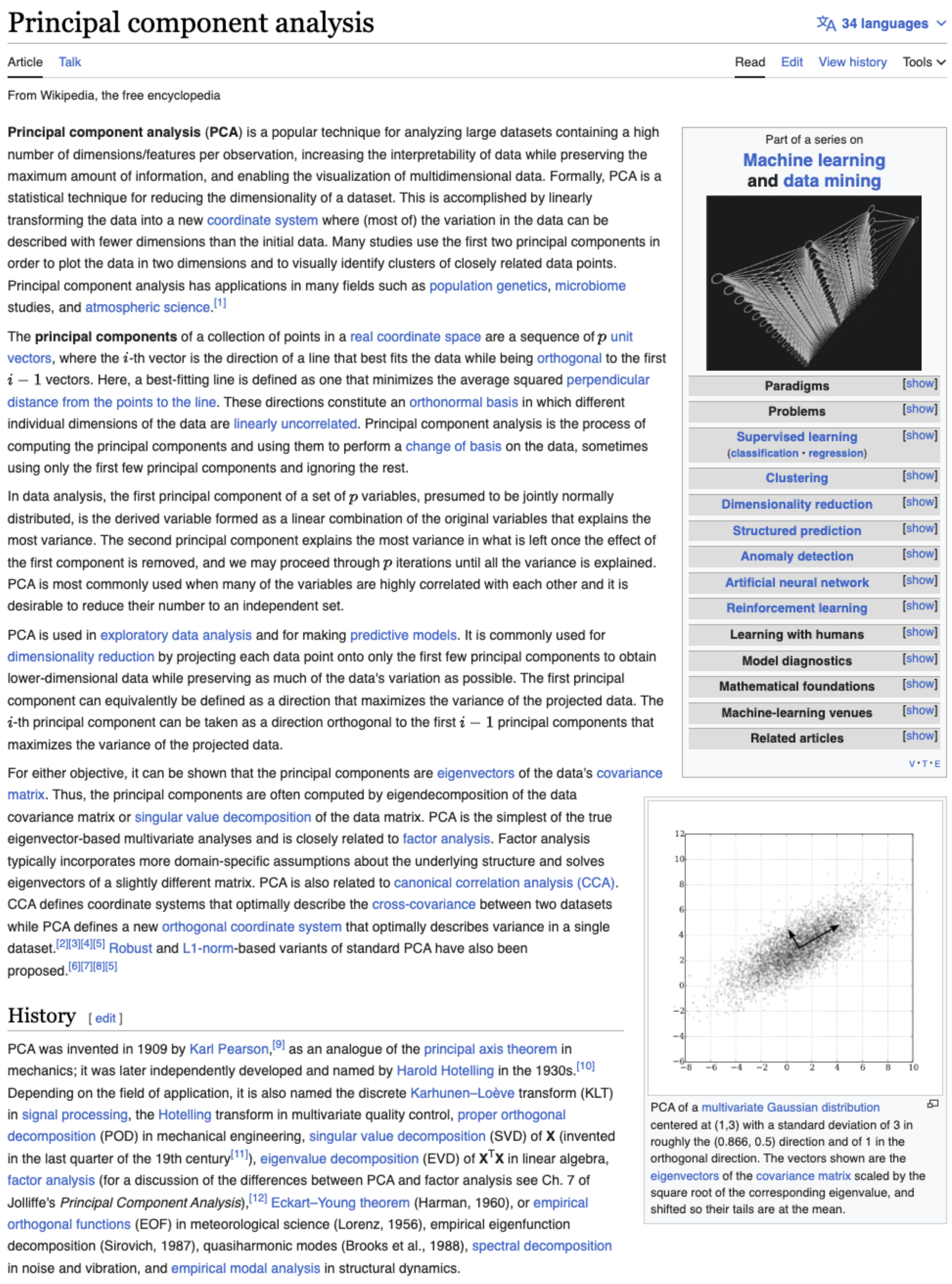
Shape descriptors can be classified by their invariance with respect to the transformations allowed in the associated shape definition. Many descriptors are invariant with respect to *congruency*, meaning that congruent shapes (shapes that could be translated, rotated and mirrored) will have the same descriptor (for example moment or spherical harmonic based descriptors or Procrustes analysis operating on point clouds).

Another class of shape descriptors (called *intrinsic* shape descriptors) is invariant with respect to isometry. These descriptors do not change with different isometric embeddings of the shape. Their advantage is that they can be applied nicely to deformable objects (e.g. a person in different body postures) as these deformations do not involve much stretching but are in fact near-isometric. Such descriptors are commonly based on geodesic distances measures along the surface of an object or on other isometry invariant characteristics such as the Laplace–Beltrami spectrum (see also spectral shape analysis).

There are other shape descriptors, such as *graph-based* descriptors like the medial axis or the Reeb graph that capture geometric and/or topological information and simplify the shape representation but can not be as easily compared as descriptors that represent shape as a vector of numbers.

From this discussion it becomes clear, that different shape descriptors target different aspects of shape and can be used for a specific application. Therefore, depending on the application, it is necessary to analyze how well a descriptor captures the features of interest.

Webside 12:
wikipedia . (u.d.). Hentet fra Level-set method: https://en.wikipedia.org/wiki/Level-set_method*wikipidea* . (u.d.). Hentet fra Principal component analysis: https://en.wikipedia.org/wiki/Principal_component_analysis



Principal component analysis

34 languages

Article Talk · Read Edit View history Tools

From Wikipedia, the free encyclopedia

Principal component analysis (**PCA**) is a popular technique for analyzing large datasets containing a high number of dimensions/features per observation, increasing the interpretability of data while preserving the maximum amount of information, and enabling the visualization of multidimensional data. Formally, PCA is a statistical technique for reducing the dimensionality of a dataset. This is accomplished by linearly transforming the data into a new coordinate system where (most of) the variation in the data can be described with fewer dimensions than the initial data. Many studies use the first two principal components in order to plot the data in two dimensions and to visually identify clusters of closely related data points. Principal component analysis has applications in many fields such as population genetics, microbiome studies, and atmospheric science.[1]

The **principal components** of a collection of points in a real coordinate space are a sequence of p unit vectors, where the i-th vector is the direction of a line that best fits the data while being orthogonal to the first $i-1$ vectors. Here, a best-fitting line is defined as one that minimizes the average squared perpendicular distance from the points to the line. These directions constitute an orthonormal basis in which different individual dimensions of the data are linearly uncorrelated. Principal component analysis is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal components and ignoring the rest.

In data analysis, the first principal component of a set of p variables, presumed to be jointly normally distributed, is the derived variable formed as a linear combination of the original variables that explains the most variance. The second principal component explains the most variance in what is left once the effect of the first component is removed, and we may proceed through p iterations until all the variance is explained. PCA is most commonly used when many of the variables are highly correlated with each other and it is desirable to reduce their number to an independent set.

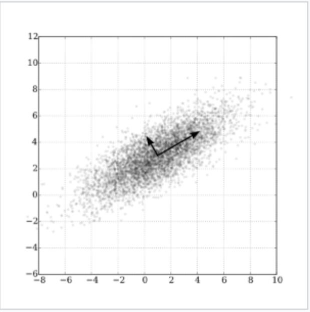
PCA is used in exploratory data analysis and for making predictive models. It is commonly used for dimensionality reduction by projecting each data point onto only the first few principal components to obtain lower-dimensional data while preserving as much of the data's variation as possible. The first principal component can equivalently be defined as a direction that maximizes the variance of the projected data. The i-th principal component can be taken as a direction orthogonal to the first $i-1$ principal components that maximizes the variance of the projected data.

For either objective, it can be shown that the principal components are eigenvectors of the data's covariance matrix. Thus, the principal components are often computed by eigendecomposition of the data covariance matrix or singular value decomposition of the data matrix. PCA is the simplest of the true eigenvector-based multivariate analyses and is closely related to factor analysis. Factor analysis typically incorporates more domain-specific assumptions about the underlying structure and solves eigenvectors of a slightly different matrix. PCA is also related to canonical correlation analysis (CCA). CCA defines coordinate systems that optimally describe the cross-covariance between two datasets while PCA defines a new orthogonal coordinate system that optimally describes variance in a single dataset.[2][3][4][5] Robust and L1-norm-based variants of standard PCA have also been proposed.[6][7][8][5]

Part of a series on
Machine learning and data mining

Paradigms	[show]
Problems	[show]
Supervised learning (classification • regression)	[show]
Clustering	[show]
Dimensionality reduction	[show]
Structured prediction	[show]
Anomaly detection	[show]
Artificial neural network	[show]
Reinforcement learning	[show]
Learning with humans	[show]
Model diagnostics	[show]
Mathematical foundations	[show]
Machine-learning venues	[show]
Related articles	[show]

V • T • E



PCA of a multivariate Gaussian distribution centered at (1,3) with a standard deviation of 3 in roughly the (0.866, 0.5) direction and of 1 in the orthogonal direction. The vectors shown are the eigenvectors of the covariance matrix scaled by the square root of the corresponding eigenvalue, and shifted so their tails are at the mean.

History [edit]

PCA was invented in 1909 by Karl Pearson,[9] as an analogue of the principal axis theorem in mechanics; it was later independently developed and named by Harold Hotelling in the 1930s.[10] Depending on the field of application, it is also named the discrete Karhunen–Loève transform (KLT) in signal processing, the Hotelling transform in multivariate quality control, proper orthogonal decomposition (POD) in mechanical engineering, singular value decomposition (SVD) of $\mathbf{X}$ (invented in the last quarter of the 19th century[11]), eigenvalue decomposition (EVD) of $\mathbf{X}^T\mathbf{X}$ in linear algebra, factor analysis (for a discussion of the differences between PCA and factor analysis see Ch. 7 of Jolliffe's *Principal Component Analysis*),[12] Eckart–Young theorem (Harman, 1960), or empirical orthogonal functions (EOF) in meteorological science (Lorenz, 1956), empirical eigenfunction decomposition (Sirovich, 1987), quasiharmonic modes (Brooks et al., 1988), spectral decomposition in noise and vibration, and empirical modal analysis in structural dynamics.

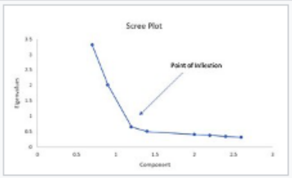
Intuition [edit]

PCA can be thought of as fitting a *p*-dimensional ellipsoid to the data, where each axis of the ellipsoid represents a principal component. If some axis of the ellipsoid is small, then the variance along that axis is also small.

To find the axes of the ellipsoid, we must first center the values of each variable in the dataset on 0 by subtracting the mean of the variable's observed values from each of those values. These transformed values are used instead of the original observed values for each of the variables. Then, we compute the covariance matrix of the data and calculate the eigenvalues and corresponding eigenvectors of this covariance matrix. Then we must normalize each of the orthogonal eigenvectors to turn them into unit vectors. Once this is done, each of the mutually-orthogonal unit eigenvectors can be interpreted as an axis of the ellipsoid fitted to the data. This choice of basis will transform the covariance matrix into a diagonalized form, in which the diagonal elements represent the variance of each axis. The proportion of the variance that each eigenvector represents can be calculated by dividing the eigenvalue corresponding to that eigenvector by the sum of all eigenvalues.

Biplots and scree plots (degree of explained variance) are used to explain findings of the PCA.

Details [edit]



The above picture is of a scree plot that is meant to help interpret the PCA and decide how many components to retain. The start of the bend in the line (point of inflexion) should indicate how many components are retained, hence in this example, three factors should be retained.

PCA is defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some scalar projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.[12]

Consider an $n \times p$ data matrix, **X**, with column-wise zero empirical mean (the sample mean of each column has been shifted to zero), where each of the *n* rows represents a different repetition of the experiment, and each of the *p* columns gives a particular kind of feature (say, the results from a particular sensor).

Mathematically, the transformation is defined by a set of size l of *p*-dimensional vectors of weights or coefficients $\mathbf{w}_{(k)} = (w_1, \dots, w_p)_{(k)}$ that map each row vector $\mathbf{x}_{(i)}$ of **X** to a new vector of principal component *scores* $\mathbf{t}_{(i)} = (t_1, \dots, t_l)_{(i)}$, given by

$$t_{k(i)} = \mathbf{x}_{(i)} \cdot \mathbf{w}_{(k)} \qquad \text{for} \qquad i = 1, \dots, n \qquad k = 1, \dots, l$$

in such a way that the individual variables $t_1, \dots, t_l$ of **t** considered over the data set successively inherit the maximum possible variance from **X**, with each coefficient vector **w** constrained to be a unit vector (where l is usually selected to be strictly less than p to reduce dimensionality).

First component [edit]

In order to maximize variance, the first weight vector $\mathbf{w}_{(1)}$ thus has to satisfy

$$\mathbf{w}_{(1)} = \arg\max_{\Vert \mathbf{w} \Vert = 1} \left\{ \sum_i (t_1)^2_{(i)} \right\} = \arg\max_{\Vert \mathbf{w} \Vert = 1} \left\{ \sum_i \left(\mathbf{x}_{(i)} \cdot \mathbf{w} \right)^2 \right\}$$

Equivalently, writing this in matrix form gives

$$\mathbf{w}_{(1)} = \arg\max_{\Vert \mathbf{w} \Vert = 1} \left\{ \Vert \mathbf{X}\mathbf{w} \Vert^2 \right\} = \arg\max_{\Vert \mathbf{w} \Vert = 1} \left\{ \mathbf{w}^\mathsf{T} \mathbf{X}^\mathsf{T} \mathbf{X} \mathbf{w} \right\}$$

Since $\mathbf{w}_{(1)}$ has been defined to be a unit vector, it equivalently also satisfies

$$\mathbf{w}_{(1)} = \arg\max \left\{ \frac{\mathbf{w}^\mathsf{T} \mathbf{X}^\mathsf{T} \mathbf{X} \mathbf{w}}{\mathbf{w}^\mathsf{T} \mathbf{w}} \right\}$$

The quantity to be maximised can be recognised as a Rayleigh quotient. A standard result for a positive semidefinite matrix such as $\mathbf{X}^\mathsf{T}\mathbf{X}$ is that the quotient's maximum possible value is the largest eigenvalue of the matrix, which occurs when ***w*** is the corresponding eigenvector.

With $\mathbf{w}_{(1)}$ found, the first principal component of a data vector $\mathbf{x}_{(i)}$ can then be given as a score $t_{1(i)} = \mathbf{x}_{(i)} \cdot \mathbf{w}_{(1)}$ in the transformed co-ordinates, or as the corresponding vector in the original variables, $\{\mathbf{x}_{(i)} \cdot \mathbf{w}_{(1)}\} \mathbf{w}_{(1)}$.

Further components [edit]

The *k*-th component can be found by subtracting the first *k* – 1 principal components from **X**:

$$\hat{\mathbf{X}}_k = \mathbf{X} - \sum_{s=1}^{k-1} \mathbf{X} \mathbf{w}_{(s)} \mathbf{w}_{(s)}^\mathsf{T}$$

and then finding the weight vector which extracts the maximum variance from this new data matrix

$$\mathbf{w}_{(k)} = \mathop{\arg\max}_{\Vert \mathbf{w} \Vert = 1} \left\{ \Vert \hat{\mathbf{X}}_k \mathbf{w} \Vert^2 \right\} = \arg\max \left\{ \frac{\mathbf{w}^\mathsf{T} \hat{\mathbf{X}}_k^\mathsf{T} \hat{\mathbf{X}}_k \mathbf{w}}{\mathbf{w}^T \mathbf{w}} \right\}$$

It turns out that this gives the remaining eigenvectors of $\mathbf{X}^\mathsf{T}\mathbf{X}$, with the maximum values for the quantity in brackets given by their corresponding eigenvalues. Thus the weight vectors are eigenvectors of $\mathbf{X}^\mathsf{T}\mathbf{X}$.

The *k*-th principal component of a data vector $\mathbf{x}_{(i)}$ can therefore be given as a score $t_{k(i)} = \mathbf{x}_{(i)} \cdot \mathbf{w}_{(k)}$ in the transformed coordinates, or as the corresponding vector in the space of the original variables, $\{\mathbf{x}_{(i)} \cdot \mathbf{w}_{(k)}\} \mathbf{w}_{(k)}$, where $\mathbf{w}_{(k)}$ is the *k*th eigenvector of $\mathbf{X}^\mathsf{T}\mathbf{X}$.

The full principal components decomposition of **X** can therefore be given as

$$\mathbf{T} = \mathbf{X}\mathbf{W}$$

where **W** is a *p*-by-*p* matrix of weights whose columns are the eigenvectors of $\mathbf{X}^\mathsf{T}\mathbf{X}$. The transpose of **W** is sometimes called the whitening or sphering transformation. Columns of **W** multiplied by the square root of corresponding eigenvalues, that is, eigenvectors scaled up by the variances, are called *loadings* in PCA or in Factor analysis.

Covariances [edit]

$\mathbf{X}^\mathsf{T}\mathbf{X}$ itself can be recognized as proportional to the empirical sample covariance matrix of the dataset $\mathbf{X}^\mathsf{T}$.[12]:30–31

The sample covariance *Q* between two of the different principal components over the dataset is given by:

$$\begin{aligned} Q(\mathrm{PC}_{(j)}, \mathrm{PC}_{(k)}) &\propto (\mathbf{X}\mathbf{w}_{(j)})^\mathsf{T}(\mathbf{X}\mathbf{w}_{(k)}) \\ &= \mathbf{w}_{(j)}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w}_{(k)} \\ &= \mathbf{w}_{(j)}^\mathsf{T}\lambda_{(k)}\mathbf{w}_{(k)} \\ &= \lambda_{(k)}\mathbf{w}_{(j)}^\mathsf{T}\mathbf{w}_{(k)} \end{aligned}$$

where the eigenvalue property of $\mathbf{w}_{(k)}$ has been used to move from line 2 to line 3. However eigenvectors $\mathbf{w}_{(j)}$ and $\mathbf{w}_{(k)}$ corresponding to eigenvalues of a symmetric matrix are orthogonal (if the eigenvalues are different), or can be orthogonalised (if the vectors happen to share an equal repeated value). The product in the final line is therefore zero; there is no sample covariance between different principal components over the dataset.

Another way to characterise the principal components transformation is therefore as the transformation to coordinates which diagonalise the empirical sample covariance matrix.

In matrix form, the empirical covariance matrix for the original variables can be written

$$\mathbf{Q} \propto \mathbf{X}^\mathsf{T}\mathbf{X} = \mathbf{W}\mathbf{\Lambda}\mathbf{W}^\mathsf{T}$$

The empirical covariance matrix between the principal components becomes

$$\mathbf{W}^\mathsf{T}\mathbf{Q}\mathbf{W} \propto \mathbf{W}^\mathsf{T}\mathbf{W}\,\mathbf{\Lambda}\,\mathbf{W}^\mathsf{T}\mathbf{W} = \mathbf{\Lambda}$$

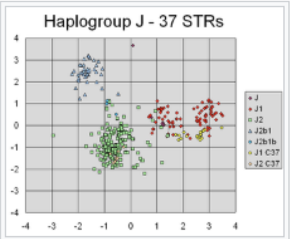
where **Λ** is the diagonal matrix of eigenvalues $\lambda_{(k)}$ of $\mathbf{X}^\mathsf{T}\mathbf{X}$. $\lambda_{(k)}$ is equal to the sum of the squares over the dataset associated with each component *k*, that is, $\lambda_{(k)} = \Sigma_i t_k^2{}_{(i)} = \Sigma_i (\mathbf{x}_{(i)} \cdot \mathbf{w}_{(k)})^2$.

Dimensionality reduction [edit]

The transformation **T** = **X W** maps a data vector $\mathbf{x}_{(i)}$ from an original space of *p* variables to a new space of *p* variables which are uncorrelated over the dataset. However, not all the principal components need to be kept. Keeping only the first *L* principal components, produced by using only the first *L* eigenvectors, gives the truncated transformation

$$\mathbf{T}_L = \mathbf{X}\mathbf{W}_L$$

where the matrix $\mathbf{T}_L$ now has *n* rows but only *L* columns. In other words, PCA learns a linear transformation $t = W_L^T x, x \in \mathbb{R}^p, t \in \mathbb{R}^L$, where the columns of $p \times L$ matrix W_L form an orthogonal basis for the *L* features (the components of representation *t*) that are decorrelated.[13] By construction, of all the transformed data matrices with only *L* columns, this score matrix maximises the variance in the original data that has been preserved, while minimising the total squared reconstruction error $\|\mathbf{T}\mathbf{W}^T - \mathbf{T}_L\mathbf{W}_L^T\|_2^2$ or $\|\mathbf{X} - \mathbf{X}_L\|_2^2$.

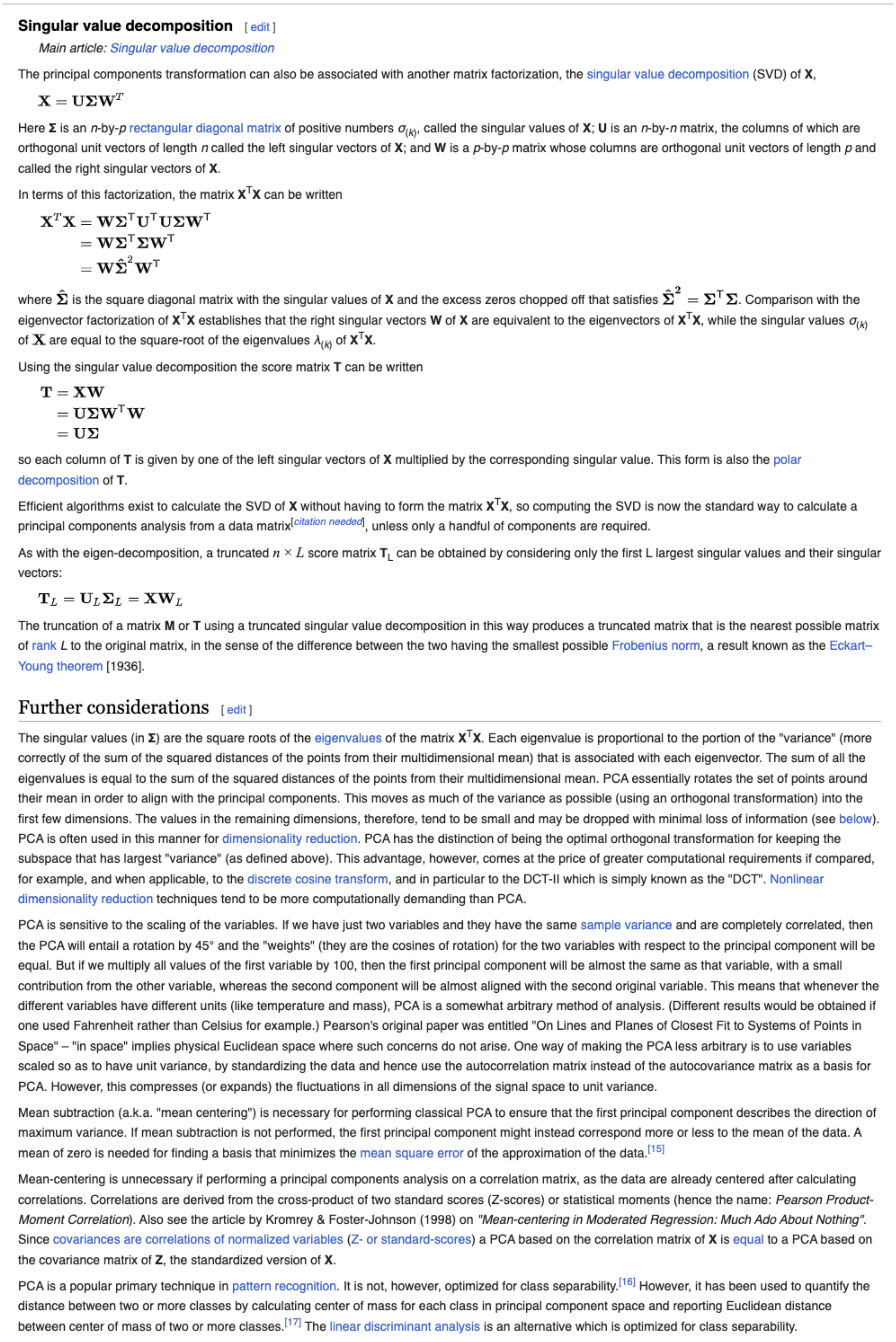


A principal components analysis scatterplot of Y-STR haplotypes calculated from repeat-count values for 37 Y-chromosomal STR markers from 354 individuals.
PCA has successfully found linear combinations of the markers that separate out different clusters corresponding to different lines of individuals' Y-chromosomal genetic descent.

Such dimensionality reduction can be a very useful step for visualising and processing high-dimensional datasets, while still retaining as much of the variance in the dataset as possible. For example, selecting *L* = 2 and keeping only the first two principal components finds the two-dimensional plane through the high-dimensional dataset in which the data is most spread out, so if the data contains clusters these too may be most spread out, and therefore most visible to be plotted out in a two-dimensional diagram; whereas if two directions through the data (or two of the original variables) are chosen at random, the clusters may be much less spread apart from each other, and may in fact be much more likely to substantially overlay each other, making them indistinguishable.

Similarly, in regression analysis, the larger the number of explanatory variables allowed, the greater is the chance of overfitting the model, producing conclusions that fail to generalise to other datasets. One approach, especially when there are strong correlations between different possible explanatory variables, is to reduce them to a few principal components and then run the regression against them, a method called principal component regression.

Dimensionality reduction may also be appropriate when the variables in a dataset are noisy. If each column of the dataset contains independent identically distributed Gaussian noise, then the columns of **T** will also contain similarly identically distributed Gaussian noise (such a distribution is invariant under the effects of the matrix **W**, which can be thought of as a high-dimensional rotation of the co-ordinate axes). However, with more of the total variance concentrated in the first few principal components compared to the same noise variance, the proportionate effect of the noise is less—the first few components achieve a higher signal-to-noise ratio. PCA thus can have the effect of concentrating much of the signal into the first few principal components, which can usefully be captured by dimensionality reduction; while the later principal components may be dominated by noise, and so disposed of without great loss. If the dataset is not too large, the significance of the principal components can be tested using parametric bootstrap, as an aid in determining how many principal components to retain.[14]



Singular value decomposition [edit]

Main article: Singular value decomposition

The principal components transformation can also be associated with another matrix factorization, the singular value decomposition (SVD) of **X**,

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{W}^T$$

Here **Σ** is an *n*-by-*p* rectangular diagonal matrix of positive numbers $\sigma_{(k)}$, called the singular values of **X**; **U** is an *n*-by-*n* matrix, the columns of which are orthogonal unit vectors of length *n* called the left singular vectors of **X**; and **W** is a *p*-by-*p* matrix whose columns are orthogonal unit vectors of length *p* and called the right singular vectors of **X**.

In terms of this factorization, the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ can be written

$$\begin{aligned}\mathbf{X}^T\mathbf{X} &= \mathbf{W}\mathbf{\Sigma}^\mathsf{T}\mathbf{U}^\mathsf{T}\mathbf{U}\mathbf{\Sigma}\mathbf{W}^\mathsf{T}\\ &= \mathbf{W}\mathbf{\Sigma}^\mathsf{T}\mathbf{\Sigma}\mathbf{W}^\mathsf{T}\\ &= \mathbf{W}\mathbf{\hat{\Sigma}}^2\mathbf{W}^\mathsf{T}\end{aligned}$$

where $\mathbf{\hat{\Sigma}}$ is the square diagonal matrix with the singular values of **X** and the excess zeros chopped off that satisfies $\mathbf{\hat{\Sigma}^2} = \mathbf{\Sigma}^\mathsf{T}\mathbf{\Sigma}$. Comparison with the eigenvector factorization of $\mathbf{X}^\mathsf{T}\mathbf{X}$ establishes that the right singular vectors **W** of **X** are equivalent to the eigenvectors of $\mathbf{X}^\mathsf{T}\mathbf{X}$, while the singular values $\sigma_{(k)}$ of $\mathbf{X}$ are equal to the square-root of the eigenvalues $\lambda_{(k)}$ of $\mathbf{X}^\mathsf{T}\mathbf{X}$.

Using the singular value decomposition the score matrix **T** can be written

$$\begin{aligned}\mathbf{T} &= \mathbf{X}\mathbf{W}\\ &= \mathbf{U}\mathbf{\Sigma}\mathbf{W}^\mathsf{T}\mathbf{W}\\ &= \mathbf{U}\mathbf{\Sigma}\end{aligned}$$

so each column of **T** is given by one of the left singular vectors of **X** multiplied by the corresponding singular value. This form is also the polar decomposition of **T**.

Efficient algorithms exist to calculate the SVD of **X** without having to form the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$, so computing the SVD is now the standard way to calculate a principal components analysis from a data matrix[*citation needed*], unless only a handful of components are required.

As with the eigen-decomposition, a truncated $n \times L$ score matrix $\mathbf{T}_L$ can be obtained by considering only the first L largest singular values and their singular vectors:

$$\mathbf{T}_L = \mathbf{U}_L\mathbf{\Sigma}_L = \mathbf{X}\mathbf{W}_L$$

The truncation of a matrix **M** or **T** using a truncated singular value decomposition in this way produces a truncated matrix that is the nearest possible matrix of rank *L* to the original matrix, in the sense of the difference between the two having the smallest possible Frobenius norm, a result known as the Eckart–Young theorem [1936].

Further considerations [edit]

The singular values (in **Σ**) are the square roots of the eigenvalues of the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$. Each eigenvalue is proportional to the portion of the "variance" (more correctly of the sum of the squared distances of the points from their multidimensional mean) that is associated with each eigenvector. The sum of all the eigenvalues is equal to the sum of the squared distances of the points from their multidimensional mean. PCA essentially rotates the set of points around their mean in order to align with the principal components. This moves as much of the variance as possible (using an orthogonal transformation) into the first few dimensions. The values in the remaining dimensions, therefore, tend to be small and may be dropped with minimal loss of information (see below). PCA is often used in this manner for dimensionality reduction. PCA has the distinction of being the optimal orthogonal transformation for keeping the subspace that has largest "variance" (as defined above). This advantage, however, comes at the price of greater computational requirements if compared, for example, and when applicable, to the discrete cosine transform, and in particular to the DCT-II which is simply known as the "DCT". Nonlinear dimensionality reduction techniques tend to be more computationally demanding than PCA.

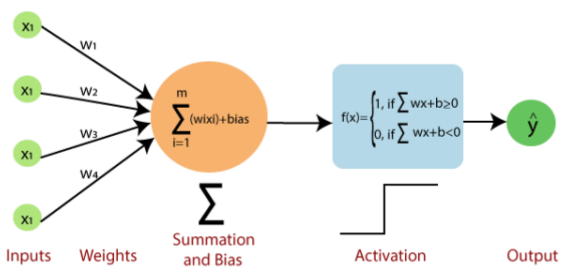
PCA is sensitive to the scaling of the variables. If we have just two variables and they have the same sample variance and are completely correlated, then the PCA will entail a rotation by 45° and the "weights" (they are the cosines of rotation) for the two variables with respect to the principal component will be equal. But if we multiply all values of the first variable by 100, then the first principal component will be almost the same as that variable, with a small contribution from the other variable, whereas the second component will be almost aligned with the second original variable. This means that whenever the different variables have different units (like temperature and mass), PCA is a somewhat arbitrary method of analysis. (Different results would be obtained if one used Fahrenheit rather than Celsius for example.) Pearson's original paper was entitled "On Lines and Planes of Closest Fit to Systems of Points in Space" – "in space" implies physical Euclidean space where such concerns do not arise. One way of making the PCA less arbitrary is to use variables scaled so as to have unit variance, by standardizing the data and hence use the autocorrelation matrix instead of the autocovariance matrix as a basis for PCA. However, this compresses (or expands) the fluctuations in all dimensions of the signal space to unit variance.

Mean subtraction (a.k.a. "mean centering") is necessary for performing classical PCA to ensure that the first principal component describes the direction of maximum variance. If mean subtraction is not performed, the first principal component might instead correspond more or less to the mean of the data. A mean of zero is needed for finding a basis that minimizes the mean square error of the approximation of the data.[15]

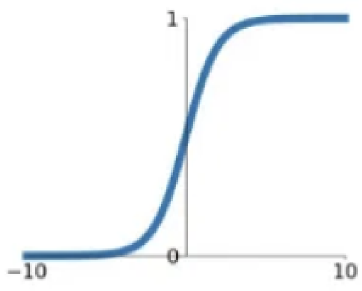
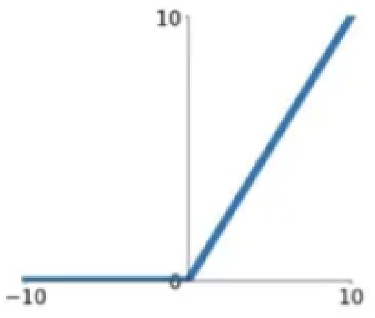
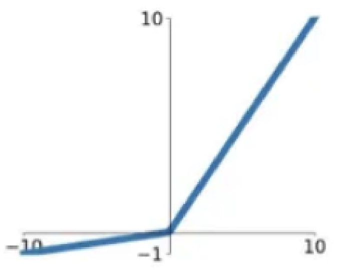
Mean-centering is unnecessary if performing a principal components analysis on a correlation matrix, as the data are already centered after calculating correlations. Correlations are derived from the cross-product of two standard scores (Z-scores) or statistical moments (hence the name: *Pearson Product-Moment Correlation*). Also see the article by Kromrey & Foster-Johnson (1998) on *"Mean-centering in Moderated Regression: Much Ado About Nothing"*. Since covariances are correlations of normalized variables (Z- or standard-scores) a PCA based on the correlation matrix of **X** is equal to a PCA based on the covariance matrix of **Z**, the standardized version of **X**.

PCA is a popular primary technique in pattern recognition. It is not, however, optimized for class separability.[16] However, it has been used to quantify the distance between two or more classes by calculating center of mass for each class in principal component space and reporting Euclidean distance between center of mass of two or more classes.[17] The linear discriminant analysis is an alternative which is optimized for class separability.

Figurliste:


X1
X1
X1
X1
W1
W2
W3
W4
$\sum_{i=1}^{m}$(wixi)+bias
f(x)= 1, if ∑ wx+b≥0 ; 0, if ∑ wx+b<0
ŷ
Inputs
Weights
Summation and Bias
Activation
Output

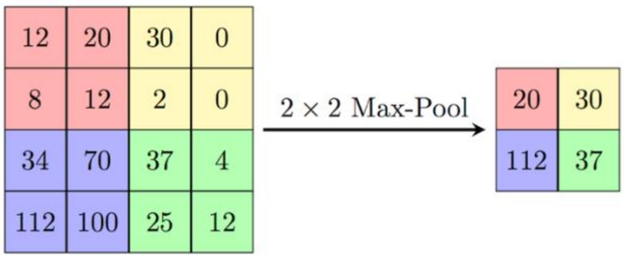
javatpoint. (u.d.). Hentet fra https://www.javatpoint.com/single-layer-perceptron-in-tensorflow

$\boldsymbol{\sigma(z) = \frac{1}{1+e^{-z}}}$	
$ReLU(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$	
$Leaky\_Relu(z) = \begin{cases} z, & z > 0 \\ z \cdot a, & z \leq 0 \end{cases}$	

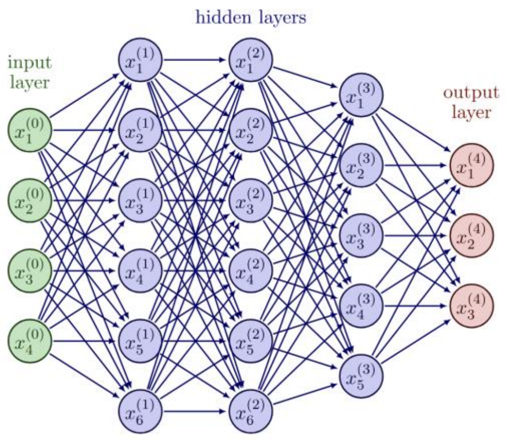
Jadon, S. (16. marts 2018). *Introduction to Different Activation Functions for Deep Learning*. Hentet fra medium.com: https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

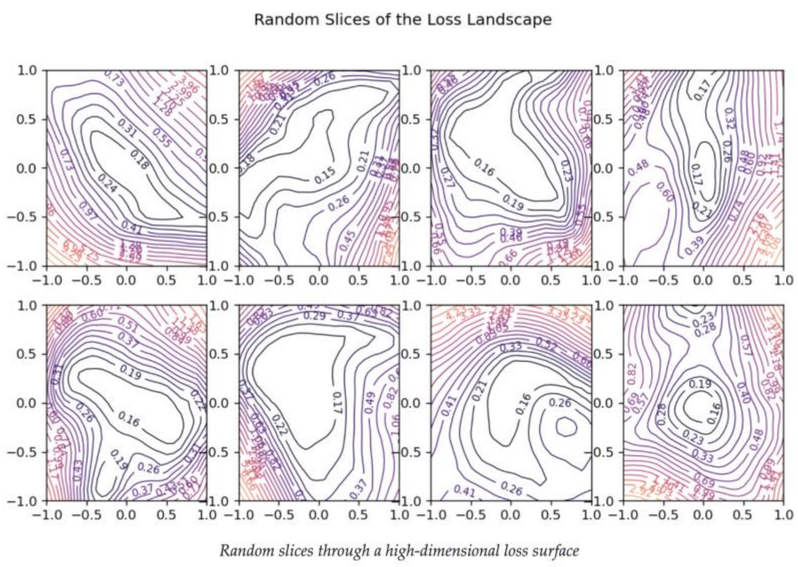
Dot Product. (u.d.). Hentet fra wikipedia: https://en.wikipedia.org/wiki/Dot_product



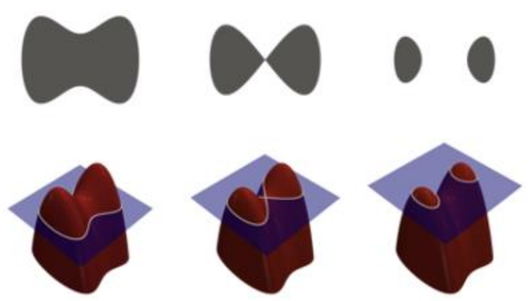
Max Pooling . (u.d.). Hentet fra paperswithcode: https://paperswithcode.com/method/max-pooling



the universal approximation theorem. (26. marts 2023). Hentet fra deep-mind: https://www.deep-mind.org/2023/03/26/the-universal-approximation-theorem/



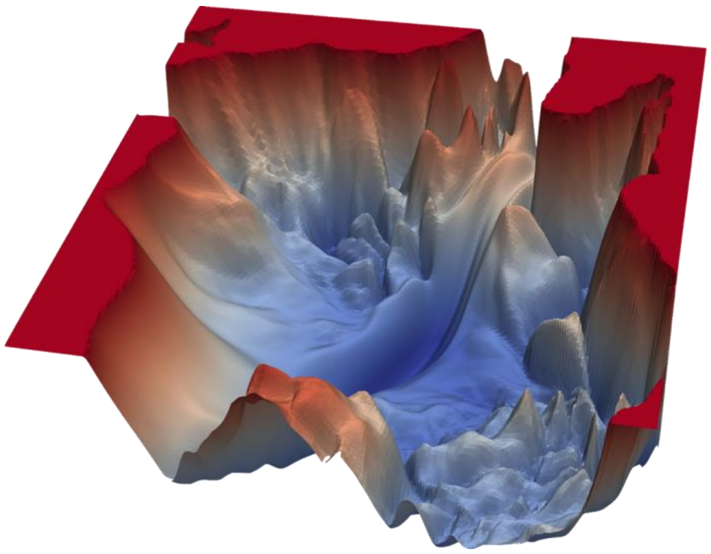
Visualizing the Loss Landscape of a Neural Network. (30. december 2020). Hentet fra Math for Machines: https://mathformachines.com/posts/visualizing-the-loss-landscape/



wikipedia . (u.d.). Hentet fra Level-set method: https://en.wikipedia.org/wiki/Level-set_method

$$H(t,p) = -\sum_{s \in S} t(s).\log(p(s))$$

Shah, D. (26. januar 2023). *Cross Entropy Loss: Intro, Applications, Code*. Hentet fra v7labs : https://www.v7labs.com/blog/cross-entropy-loss-guide



Li, H., Xu, Z., Taylor, G., Studer, C., & Goldstein, T. (7. november 2018). Visualizing the Loss Landscape of Neural Nets. *cornell university*, s. 1-18.